# Reconciling the Model-Implementation Duality in PGo

by

Shayan Hosseini

B.Sc., University of Tehran, 2020

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

**Master of Science**

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Computer Science)

The University of British Columbia
(Vancouver)

November 2023

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

**Reconciling the Model-Implementation Duality in PGo**

submitted by **Shayan Hosseini** in partial fulfillment of the requirements for the degree of **Master of Science** in **Computer Science**.

**Examining Committee:**

Ivan Beschastnikh, Associate Professor, Computer Science, UBC
*Supervisor*

Norman C. Hutchinson, Associate Professor, Computer Science, UBC
*Supervisory Committee Member*

# Abstract

Distributed systems are difficult to design and implement correctly, leading academia and industry to explore using formal methods to address these complexities. In previous work, we presented PGo, a framework for building verified distributed system implementations. PGo compiles distributed system models into executable programs.

Using PGo to build systems, we face a new paradigm where PGo models serve dual roles as both models and programs. Models and programs are inherently different. Models are designed for ease of reasoning and provide a simplified representation of the system, while programs prioritize efficiency and performance for execution on the hardware.

This work addresses the duality problem inherent in PGo models, where they must serve both as models and programs. We analyze various aspects of the duality we faced while building distributed systems using PGo. We propose techniques that reconcile the model-implementation duality. Additionally, we introduce a framework for constructing modular systems using PGo. Modularity is essential for building real-world distributed systems, as distributed systems are rarely implemented as monolithic systems.

Our evaluation demonstrates that despite the duality problem, we can satisfy the requirements of both the model and implementation sides in complex systems. We used PGo to model, compile, and evaluate several distributed systems, including key-value stores based on Raft and primary-backup protocols, as well as Conflict-free Replicated Data Types. Our Raft-based key-value store with three nodes is model checked and has 41% higher throughput than similar verified systems.

# Lay Summary

Designing and implementing distributed systems accurately is challenging. In a prior work, we presented the PGo framework for building correct distributed systems. PGo compiles specifications of distributed systems into programs. When using PGo, we encounter a new challenge: the specification we create must serve as both a simplified representation for reasoning (model) and an efficient program for execution (implementation). This study addresses this duality inherent in PGo specification. Our evaluation demonstrates that despite the duality, we can satisfy the requirements of both the model and implementation sides in complex distributed systems.

# Preface

This thesis is based on my contributions to the following publication [29] and includes its text and material:

- "Compiling Distributed System Models with PGo".
  Finn Hackett, Shayan Hosseini, Renato Costa, Matthew Do, Ivan Beschastnikh. *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2023.*

I led the research and development of building distributed systems using the PGo compiler toolchain and evaluated them. In Chapter 11, Finn Hackett evaluated related work for comparison. Chapter 12 is based on Finn Hackett's writing of related work in the PGo paper [29]. This work has been done under the supervision of Ivan Beschastnikh, and he provided comments and feedback on the writing.

# Table of Contents

# List of Tables

# List of Figures

# Acknowledgments

I had the privilege of working with Ivan Beschastnikh throughout my Master's program. Ivan's support has been an invaluable asset to my growth, and I have gleaned a wealth of knowledge from him. I extend my gratitude to Finn Hackett, my primary collaborator during this period. We spent countless hours discussing ideas and implementing cool stuff together. I am grateful to Norm Hutchinson for serving on my thesis committee.

My time as a member of the UBC Systopia lab during my Master's was truly remarkable. I am sincerely thankful to all the Systopians who became my friends and are awesome people.

I also extend my thanks to Behnaz Arzani, who generously hosted me at Microsoft Research during the summer of 2022. This experience gave me the unique opportunity to engage in world-class research in the industry.

My heartfelt thanks go to my parents and my younger brother. Their unwavering support has paved the way for my achievements, and I am profoundly grateful for everything they have done for me.

This research is supported in part by the NSERC Discovery Grants (RGPIN-202005203, RGPIN-2014-04870), by AWS through an Amazon Research Award, by an Azure Education Grant, and by the Institute for Computing, Information and Cognitive Systems (ICICS) at UBC.

# Chapter 1

# Introduction

*If an army, that sheddeth the blood of lovers, grief raise*
*Content together are I and the Saki; and up its foundation, we cast*

Hafiz

Distributed systems are a key part of the modern software stack. They support important applications such as banking, communication, and social networks. Despite decades of research and industry experience, building correct distributed systems is still notoriously hard. As a consequence, systems in production often have bugs leading to degraded performance [23], outages [25, 67], and data loss [24]. For example, Roblox, an online gaming service with more than 100 million users, experienced a 73 hours outage in November 2021 [64] due to bugs in Consul [31], a distributed key-value store.

In recent years, developers started using formal methods to ensure the correctness of their systems [57]. Typically, developers create models of their systems and use the available tools to verify the correctness of a model. Then, they translate the model into an actual implementation, which is a manual and error-prone task. This approach leaves the final implementation

unverified, creating a gap between the verified model and the implementation.

PGo [29] bridges the model-implementation gap by automatically compiling a model of a distributed system into a program in the Go programming language. PGo introduces the MPCal modeling language, which is built on top of PlusCal/TLA$^+$ family of languages [43, 44]. For verification, PGo leverages model checking to verify the correctness of MPCal models. On the implementation side, PGo compiles MPCal models into Go programs. Users have to write a few lines of configuration code to make the compiled Go program executable.

Building distributed systems using PGo helps developers with the model-implementation gap. However, it introduces a new paradigm where MPCal code serves both as a model and a program. Modeling and implementation have different purposes and concerns. Models are designed to facilitate reasoning and provide a simplified representation of the system under consideration. They aim to capture the essential aspects of the system while abstracting away unnecessary details. By reducing complexity, models enable easier analysis and reasoning about system behavior and properties. On the contrary, programs are intended for execution on hardware, where efficiency and performance are key concerns. Programs operate at a lower level of abstraction, taking into account specific hardware and implementation considerations. While models focus on conceptual understanding and verification, programs prioritize practical execution and optimal utilization of hardware resources. Thus, models and programs serve different purposes and operate at different levels of abstraction to fulfill their respective goals.

Similarly, modeling languages and programming languages serve distinct purposes and exhibit differences in their design and features. Programming languages prioritize aspects such as performance, efficiency, and ease of programming. On the other hand, modeling languages are specifically designed to facilitate the expression and analysis of systems. As a result, typically, we have different languages for writing programs and models.

MPCal code is a model itself but since it is compiled into a program, it has to fulfill the needs of a program, as well. This introduces a *duality* in

the MPCal code where it has two purposes, as a model and as a program. When using PGo to build systems, the development process begins with constructing the model, which is then compiled into an executable program. As a result, considerations and constraints from the program side impact modeling decisions. In this work, we explore the model-program duality in PGo and discuss how the two sides with different concerns can be reconciled.

Managing complexity is a crucial aspect when developing large-scale systems. Modularity plays a fundamental role in the managing of complexity. Mainly, modularity is vital for verified systems to manage the verification cost [1, 15]. By decomposing the system into smaller, interconnected modules, we can effectively handle the intricacies of system design and verification. This work introduces a framework for constructing modular systems using PGo.

Throughout this work, we present RaftStore as an example to address the challenges arising from the duality of building a complex distributed system. RaftStore is a distributed key-value store database based on the Raft consensus protocol [59]. By employing RaftStore as a running example, we demonstrate how we tackled the duality concerns as a practical showcase for the concepts and techniques discussed in this work. We built a modular version of RaftStore, named RaftStoreMod, consisting of consensus and key-value store components. We use RaftStoreMod to demonstrate the modular design in an intricate distributed system.

In summary, the contributions of this work are:

- We propose techniques to reconcile the duality between modeling land and implementation land using the provided PGo primitives. We rely on RaftStore as a running example to show how two sides with different concerns can be reconciled.

- We show how to build modular verified systems using PGo. This design approach allows us to reduce the trusted-computing base, model complexity, and model-checking cost, while building larger scale systems.

- Our evaluation shows that despite the duality problem, we are able

to build high performance and fault-tolerant systems while model-checking them within reasonable bounds.

# Chapter 2

# PGo Background

*In the end, it is a social process that determines whether mathematicians feel confident about a theorem.*

Richard A. De Millo, Richard J. Lipton and Alan J. Perlis

PGo is a compiler toolchain that automatically translates MPCal models to TLA$^+$ for model-checking. In addition, it compiles MPCal models to the Go programming language for execution.

Figure 2.1 shows the PGo workflow. The left side of the workflow is the verification side, where users can model check their MPCal models. The right side of the workflow is the implementation side, where the MPCal model is compiled into a Go program. Note that we use the words *program* and *implementation* interchangeably in this work. Next, we review the necessary information to understand PGo and the different pieces in the workflow. We use a distributed lock service as a running example throughout the chapter. The lock service has a central server that manages a lock. Clients request to acquire or release the lock through the network.

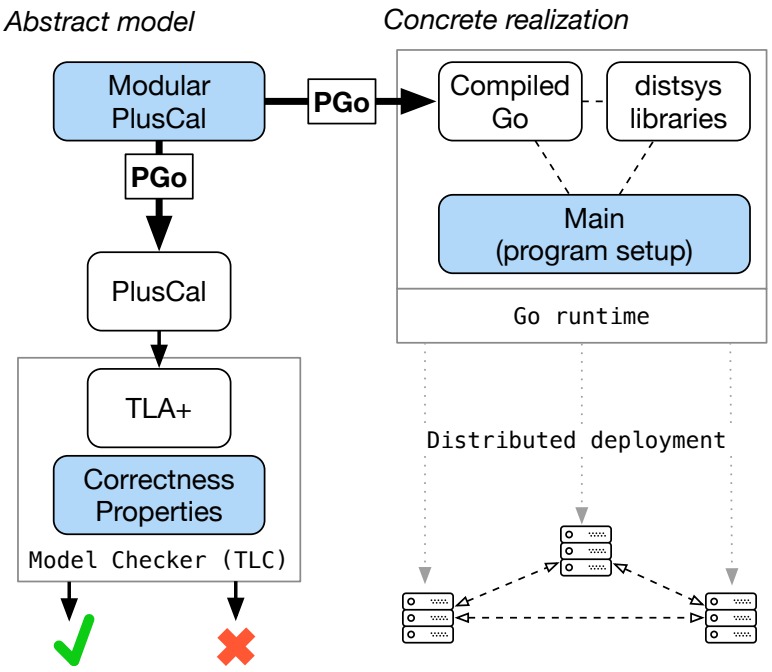



**Figure 2.1:** PGo workflow. The shaded blue components must be provided by the user.

## 2.1 Model checking

**Model checking** is a technique used to verify whether a given *model* of a system meets a specific *specification*. By exhaustively examining all possible states of the system model, a model checker ensures that the properties defined in the specification hold true in each state. Often, the model checking process is confined within a finite search space, limiting the scope of exploration. If a model checker detects a state that violates a property, it generates a *counterexample*, which represents the execution trace leading to that state.

Specifications are expressed as system properties. Properties can be categorized as either *safety* or *liveness* properties [2, 40]. Safety properties focus on preventing undesirable events or states from occurring within the system, ensuring that nothing bad happens. On the other hand, liveness

$$\nexists i, j \in ClientSet : i \neq j \wedge state_i = HasLock \\ \wedge state_j = HasLock$$

**(a)** Safety property: there are no two clients that hold the lock at the same time (mutual exclusion).

$$\forall i \in ClientSet : \Box(state_i = WaitLock \\ \implies \Diamond(state_i = HasLock))$$

**(b)** Liveness property: always ($\Box$) every client that waits for the lock, eventually ($\Diamond$) holds the lock. Note that satisfying the safety property above **does not** guarantee this fact, since a lock server that never grants the lock to any client trivially satisfies the safety requirement.

**Figure 2.2:** Example safety and liveness properties of a lock service. *ClientSet* is the set of all clients in the system, and $state_i$ is the $i$-th client state.

properties emphasize the guarantee that the system eventually reaches a desirable state or accomplishes some good outcome.

Figure 2.2 lists a safety and a liveness property for the lock service.

## 2.2 TLA$^+$

**TLA$^+$** [43] is a language that employs a declarative approach to model systems based on set-theory and first-order logic [61]. The TLC model checker [78] can verify that a TLA$^+$ model adheres to both safety and liveness specifications within bounded limits. A TLA$^+$ model is made of several predicates. Predicates define the modeled system's *initial state* and the *transition relation* describing how the system progresses with new states. The transitions are atomic steps in the system.

For example, Figure 2.3 shows our lock server modeled in TLA$^+$. The initial state is defined in the *Init* predicate, and the *Next* predicate defines the transition relations at each step. Possible transitions to the next state

are either *serverReceive* or *serverRespond*. A variable without prime (e.g., *msg*) represents the value in the old state. A variable with prime (e.g., $msg'$) represents the value in the new state. We modeled the network by a set of unbounded arrays, *network*. The lock server has a queue denoted by the variable $q$. The first client in the queue holds the lock, and the rest of the clients in the queue are waiting for the lock. In the *serverReceive* predicate, the server receives a request and stores it for processing if there is no other request pending. During *serverRespond* the server processes the received message. For lock requests, the server grants the lock to the requesting client if the queue is empty, and it adds that client to the end of its queue. For unlock requests, the server pops the requesting client from the head of its queue and grants the lock to the next client waiting in the queue, if any.

## 2.3 PlusCal

**PlusCal** [44] is a modeling language for specifying algorithms. PlusCal code can be translated to TLA$^+$. Within PlusCal, system's behavior is defined procedurally, with different processes having their behavior defined by statements and interacting using control flow structures such as *while* loops and *if* statements. To convert PlusCal specifications into TLA$^+$, users can utilize a PlusCal translator, which enables them to use the TLC model checker on PlusCal models.

A PlusCal spec is comprised of one or more *processes*. Each process runs sequentially, and different processes can run concurrently. PlusCal processes consist of *labels*. Each label contains a block of statements that define an atomic step in the model. PlusCal provides a C-like `goto` statement that allows jumping between different labels. We also refer to labels as *critical sections*. During compilation to TLA$^+$, each block of labels in PlusCal is converted into a TLA$^+$ transition. Designers must consider the trade-off presented by labels, as incorporating more labels enables increased concurrency between labels in different processes, resulting in a more realistic model. However, this can lead to an exponential increase in the state space and its associated costs.

```
1   In the initial state, no message in the network and the
2   server's queue is empty.
3   Init  ≜  ∧ network = [n ∈ NodeSet ↦ ⟨⟩]
4            ∧ q = ⟨⟩
5            ∧ msg = Nil

6   Next  ≜  Possible transitions to the next state.
7         ∨ serverReceive
8         ∨ serverRespond

9   serverReceive  ≜  Receving a request from clients
10        ∧ msg = Nil
11        Ensure that the server's buffer is not empty.
12        ∧ Len(network[ServerId]) > 0
13        Receiving a message by reading the first element
14        in the server's network buffer.
15        ∧ msg′ = Head(network[ServerId])
16        After reading the message, we can remove it from the buffer.
17        ∧ network′[ServerId] = Tail(network[ServerId])

18  serverRespond  ≜  Handling the request and responding back
19        ∨ ∧ msg.type = LockMsg
20          ∧ q = ⟨⟩ ⇒
21              Sending a message to msg.from by adding it
22              to the end of recipient's network buffer.
23            LET dst ≜ msg.from
24            IN  network′[dst] = Append(network[dst], GrantMsg)
25          ∧ q′ = Append(q, msg.from)
26          ∧ msg′ = Nil
27        ∨ ∧ msg.type = UnlockMsg
28          ∧ q′ = Tail(q)
29          ∧ q′ ≠ ⟨⟩ ⇒
30              Sending a message to Head(q′)
31            LET dst ≜ Head(q′)
32            IN  network′[dst] = Append(network[dst], GrantMsg)
33          ∧ msg′ = Nil
```

**Figure 2.3:** Lock server specification in TLA$^+$.

Figure 2.4 shows our lock server modeled in PlusCal. It has the same semantics as the TLA$^+$ model in Figure 2.3. Besides the usual control flow statements, PlusCal has two other essential statements that do not typically exist in programming languages: (1) The `await` statement has the form `await cond`, where `cond` is a boolean expression. `await` blocks the process execution until `cond` becomes true. For example, in line 11 of Figure 2.4, the server blocks until its network queue becomes non-empty. (2) The `either` statement has the form `either {clause1} or {clause2}`, where `clause1` and `clause2` are statements. `either` nondeterministically executes one of the executable clauses. The `either` statement is executable if at least one of the clauses is executable; otherwise, the process blocks.

PlusCal mixes the details of the system and its environment such that it is not possible to distinguish them. For example, in Figure 2.4, the lock server logic, which is the definition of the system, is mixed with its environment, the buffered network.

## 2.4 MPCal

**MPCal** is a modeling language built on top of PlusCal. MPCal provides extra abstractions to separate the system definition and its environment. Having this separation allows the PGo compiler to compile MPCal models into implementations. PGo compiles MPCal code into PlusCal, where users can use the TLC model checker for verification. MPCal adds three additional abstractions to PlusCal: (1) archetype definitions, (2) archetype instantiations, and (3) mapping macros.

Archetypes are similar to PlusCal processes, but they only contain system definitions. An archetype does not have access to global variables similar to a PlusCal process. An archetype only can interact with environment instances, which are the parameters that an archetype takes. Archetype parameters are called *resources*. Resources describe archetypes' environment dependencies. The definition of an archetype contains all the necessary information for PGo to create a functional system, including the interfacing points where the system definition interacts with its environment.

```
variables network = [id \in NodeSet |-> <<>>];

\* fair keyword specifies assumption of fair scheduling
fair process (Server = 1)
variables msg, q = <<>>;
{
serverLoop:
  while (TRUE) {
  serverReceive:
    \* blocks the process until the there is some message available
    await Len(network[self]) > 0;
    msg := Head(network[self]);
    network[self] := Tail(network[self]);
  serverRespond:
    if (msg.type = LockMsg) {
      \* if q is empty
      if (q = <<>>) {
        network[msg.from] := Append(network[msg.from], GrantMsg);
      };
      q := Append(q, msg.from);
    } else if (msg.type = UnlockMsg) {
      q := Tail(q);
      \* if q is not empty (/= is the not equals operator)
      if (q /= <<>>) {
        network[Head(q)] := Append(network[Head(q)], GrantMsg);
      };
    };
  };
}
```

**Figure 2.4:** Lock server specification in PlusCal.

Archetypes access environment through their resources and *mapping macros* define how resources are working. *Mapping macros* define the behavior of resources (e.g. buffered network) through a simple read/write API. PGo does not compile mapping macros into implementation since they contain verification-specific parts of the model, such as semantics of an abstract environment.

Archetype instantiations provide existing archetypes with the right set of resources and connect resources to mapping macros to define the behavior of resources.

Figure 2.5 shows the lock server modeled in MPCal. Archetype `AServer`

```
archetype AServer(ref network[_])
variables msg, q = <<>>;
{
serverLoop:
  while (TRUE) {
  serverReceive:
    msg := network[self];
  serverRespond:
    if (msg.type = LockMsg) {
      if (q = <<>>) {
        network[msg.from] := GrantMsg;
      };
      q := Append(q, msg.from);
    } else if (msg.type = UnlockMsg) {
      q := Tail(q);
      if (q /= <<>>) {
        network[Head(q)] := GrantMsg;
      };
    };
  };
}

mapping macro ReliableFIFOLink {
  read {
    await Len($variable) > 0;
    with (readMsg = Head($variable)) {
      $variable := Tail($variable);
      yield readMsg;
    };
  }
  write {
    yield Append($variable, $value);
  }
}

variables network = [id \in NodeSet |-> <<>>];

fair process (Server = 1) == instance AServer(ref network[_])
  mapping network[_] via ReliableFIFOLink;
```

**Figure 2.5:** MPCal specification of the lock server, corresponding to Figure 2.4.

is defined on line 1, and it has a resource, `ref network[_]`[1]. The server receives a message from the network in line 7 by reading from the `network` parameter. The behavior of this read operation is defined in the read section of the `ReliableFIFOLink` mapping macro, starting on line 24. Then, in the `serverReceive` label, the server processes the received message. We send a message by writing to the `network` variable. Similar to the read case, the behavior of network write is defined in the write section of the `ReliableFIFOLink` mapping macro.

Mapping macros define the behavior of read/write operations on resources by injecting code wherever operations are used. These can be arbitrary PlusCal code, which the PGo compiler inserts into an instantiated archetype definition's body. Each mapping macro has a `read` and a `write` section that defines the behavior of read and write operations, respectively. Expressions in these sections have access to two special parameters. `$variable` is the underlying state variable to operate on, and `$value` is the value written by the caller (an archetype). In addition, the `yield(_)` statement yields the computed output of its expression as the result of the operation.

Archetype `AServer` is instantiated on line 1 of Figure 2.5 with `ref network[_]` as its parameter. The `network` is a global variable defined on line 36. The statement `mapping network[_] via ReliableFIFOLink` on line 39 declares that the behavior of `network` resource is being defined by the `ReliableFIFOLink` mapping macro.

## 2.5 PGo

Using the PGo compiler, the MPCal code can be translated to PlusCal, and then it can be translated to TLA$^+$. Users can express the properties of Figure 2.2 in TLA$^+$ and use the TLC model checker to verify the model against these properties.

PGo compiler translates archetypes into implementations in Go. PGo ensures that the compiled Go program has the same semantics as its corre-

[1]The `ref` keyword means this parameter is a passed by reference. The `[_]` syntax means indexed access and restricts accessing `network` resource to be only indexed.

sponding MPCal. The *distsys* Go library is responsible for providing such guarantees. An essential part of distsys is offering atomicity guarantees for each label while maximizing concurrency. distsys executes each archetype in a main loop that provides the necessary semantics of critical section and control flow.

Figure 2.6 shows the `serverReceive` critical section of Figure 2.5 compiled into Go. This is almost a direct translation from MPCal to Go. PGo compiles each archetype into a set of critical sections, and the distsys's main loop is responsible for correctly executing them. The `serverReceive` critical section acquires access to resources `msg` and `network` in lines 6-7. Note that it treats both archetype parameters and local variables as resources; however, the access calls are slightly different. The critical section body has an archetype interface as input `iface` that allows the archetype to access its resources through read and write calls, as demonstrated in lines 11 and 15. These correspond to operations on line 7 of Figure 2.5. Line 19 is the implicit jump at the end of the `serverReceive` label where it finishes, and the program goes to the next label, `serverRespond`.

Resource implementations have to follow the interface defined in Figure 2.7. Inspired by the two-phase commit protocol, this API allows the distsys library to provide correct linearizability semantics for critical sections. distsys main loop starts executing a critical section by applying the changes temporarily. Then it performs the pre-commit stage across all parties participating in a critical section. The main loop proceeds to the commit stage if `PreCommit` does not return any error for any of the resources. In the commit stage, each resource commits the critical section changes. If at least one of the resources involved in the critical section fails in the pre-commit stage, the main loop aborts the critical section by calling `Abort` on all resources and tries to execute the critical section again. `Commit` always supposed to succeed and this assumption might cause liveness issues, such as blocking the process execution forever. In addition, distsys provides a library of commonly used resources such as networking and storage.

The user configures the generated Go code with the resource implementations. Figure 2.8 shows the code that bootstraps the compiled `AServer`

```
distsys.MPCalCriticalSection{
  Name: "AServer.serverReceive",
  Body: func(iface distsys.ArchetypeInterface) error {
    var err error // setup
    _ = err
    msg := iface.RequireArchetypeResource("AServer.msg")
    network, err := iface.RequireArchetypeResourceRef("AServer.network")
    if err != nil {
      return err
    } // read network[self]
    networkRead, err := iface.Read(network, []tla.Value{iface.Self()})
    if err != nil {
      return err
    } // msg := <value>
    err = iface.Write(msg, nil, networkRead)
    if err != nil {
      return err
    }
    return iface.Goto("AServer.serverRespond")
  },
},
```

**Figure 2.6:** The `serverReceive` label from the archetype definition in Figure 2.5 compiled to Go.

```
type ArchetypeResource interface {
  Abort()
  PreCommit() error
  Commit()
  ReadValue() (tla.Value, error)
  WriteValue(value tla.Value) error
}
```

**Figure 2.7:** Go archetype resource interface definition.

archetype. The user begins by creating an `MPCalContext` object for each archetype. `MPCalContext` contains the main loop for its given archetype. It requires an archetype name, an archetype instance, and a list of archetype resources. In this case, we provided `TCPMailboxes` for the `network` resource. Lines 7 to 16 configure the `TCPMailboxes` with the right set of network addresses. Finally, the user runs the instantiated archetype in line 18.

```
ctxServer := distsys.NewMPCalContext(
  serverId,
  AServer,
  distsys.EnsureArchetypeRefParam(
    "network",
    resources.NewTCPMailboxes(func(v tla.Value)
      (resources.MailboxKind, string) {
      addr := addressMap[v.AsNumber()]
      var kind resources.MailboxKind
      if v.Equal(serverSelf) {
        kind = resources.MailboxesLocal
      } else {
        kind = resources.MailboxesRemote
      }
      return kind, addr
    })),
)
err := ctxServer.Run()
```

**Figure 2.8:** Bootstrapping the compiled `AServer` archetype in Go. Archetype `AServer` is instantiated with appropriate resource implementations.

## 2.6 Assumptions

As described earlier, PGo has two workflows, verification and implementation. In the verification workflow, the user trusts the MPCal to PlusCal translation process provided by the PGo compiler. Moreover, users trust PlusCal to TLA$^+$ translation and the TLC model checker. TLC assumes the user has provided the right properties in the given specification. On the implementation side, users trust the MPCal compilation with PGo, the distsys library, including all the resource implementations. Also, users have to trust the Go runtime and the underlying operating system.

# Chapter 3

# Model-Implementation Duality

*Aren't programs simpler than logical formulas? The answer is no.*

Leslie Lamport

PGo offers two distinct workflows, namely verification and implementation, which were previously discussed in Chapter 2. In each of these workflows, the MPCal code is utilized as the source code, which is subsequently compiled into different target languages to serve both as a model and a program. We define this as *model-implementation duality* in MPCal code, where the same code is employed to address two different concerns. This section discusses the similarities and differences between models and programs, and provides additional insight into the duality problem.

## 3.1 Models and Programs

A model and a program both involve describing a system in an abstract formal language. Models and programs are defined based on abstractions; however, their abstraction level is different. Models and programs serve

**Table 3.1:** Comparison of models and programs across different aspects.

| | **Program** | **Model** |
|---|---|---|
| **Purpose** | Execution on hardware | Reasoning by human/tools |
| **Non-functional requirement** | Correctness | Correctness |
| **Abstraction level** | Concrete | Abstract |
| **Performance measure** | Execution time, resource usage | Degree and cost of reasoning |
| **Complexity** | High | Low |
| **Verification cost** | High | Low |
| **Language** | Closer to hardware | Closer to math |

different purposes. The primary objective of a program is to be executed on a hardware platform, such as a CPU. In contrast, models are not necessarily designed for execution. The primary goal of a model is to enable reasoning about a system. These distinct purposes result in different abstraction levels and requirements for models and programs, while an essential requirement for both models and programs is correctness. Table 3.1 summarizes the comparison of models and programs across different aspects.

### 3.1.1 Models

Developers create models to aid in the analysis and reasoning of their systems. This reasoning can be performed either manually by humans or automatically by tools. For example, Unified Modeling Language (UML) [4] is a modeling language intended to visualize software systems primarily for human analysis. In contrast, the TLA$^+$ modeling language is based on logic and enables automated reasoning about models through tools such as model checking, automated theorem proving, and randomized testing.

Models are designed to facilitate reasoning, and as such, they tend to be simpler than programs and abstract away details as much as possible. Typically, models are only a few hundred lines of code, making them easy to modify. In addition, models focus on one specific concern and abstract

away unrelated ones. For example, consider a model of a replication protocol where nodes communicate through the network. Such a model would focus solely on the replication protocol and would abstract away other system components, such as the network. To achieve this, the model would only use a simple abstract representation of the network that has the necessary guarantees for the replication protocol.

Models often utilize math because it offers well-defined semantics for various constructs, making reasoning easier [42]. Models tend to use mathematical objects like sets and vectors which are used instead of complex objects and data structures. Unlike programs, models are not necessarily executable. Models typically adopt a declarative approach and are designed to specify the *what* rather than the *how*. Consequently, a model may not define a single method of computation, but instead they may declare classes of computation [45]. An example of this is the `either` statement in MPCal, which indicates that one of several possibilities may occur without specifying which.

### 3.1.2 Programs

Programs are designed to be executed on hardware and, as such, they are more concrete than models, with abstractions that are closer to the underlying hardware. Programs operate in real environments and need to implement the full stack of the underlying system they rely on. For instance, a networked system needs to have the complete network stack implementation in order to interact with it. Due to the level of abstraction, programs are often large. It is not unusual to have programs consisting of hundreds of thousands of lines of code.

A vital requirement for programs is execution performance, which includes factors such as execution time and resource usage (e.g., CPU, memory, and disk). To meet performance and efficiency requirements, programs often become increasingly complex. This complexity, combined with their large size, can make them difficult to change over time.

The complexity of programs presents a significant challenge in verifying

their correctness. As a result, developers often rely on techniques with weaker guarantees, such as testing, to analyze program correctness.

### 3.1.3 Languages

The distinct goals and concerns associated with models and programs necessitate the use of different languages to express them. Modeling languages aim to enhance the understandability of models and enable verification of their correctness.

Different modeling languages are tailored for specific types of analysis. Unified Modeling Language (UML) and Systems Modeling Language (SysML) [22] are designed to visualize software systems, making them easier to understand. Alloy [36], on the other hand, is a declarative modeling language based on first-order logic that can be used to express complex structural constraints and behavior in software systems. Meanwhile, TLA$^+$ and PlusCal are well-suited for specifying concurrent and distributed systems. Isar [70] and Gallina [68] are specification languages for the Isabelle [70] and Coq [69] automated theorem provers, respectively, and are particularly effective for writing proofs.

Programming languages prioritize execution optimization and ease of writing large programs. Each programming language offers its unique set of features. For instance, C is a low-level programming language that enables high performance, while a language such as Java provides garbage collection, which removes the memory management burden from programmers. However, such convenience comes at the cost of performance.

## 3.2 Duality

We discussed that models and programs have different purposes and levels of abstraction. When the same code serves both as a model and a program, it has to satisfy two different set of goals and even has to operate in two different levels of abstraction, where models are abstract and programs and implementations are concrete. We call this model-implementation duality.

The PGo workflow begins with the construction of a system model in

MPCal. As the model is compiled into a program, the requirements and considerations of the program side begin to impact the model. This integration of program-specific concerns into the model introduces duality challenges. The main idea behind MPCal is to provide abstractions that effectively separate the concerns of the model and the program. However, effectively utilizing these abstractions to build complex systems and navigating through the duality can be challenging.

Archetypes serve as a common system definition that remains consistent between the model and the compiled program in PGo. As a result, the PGo compiler compiles archetypes into the Go code. On the other hand, mapping macros define the system environment, where the model and programs operate at different levels of abstraction. Hence, users provide resource implementation for mapping macros. The first design consideration revolves around determining the appropriate division between archetypes and mapping macros in terms of what functionality should be included in each. Archetypes must be able to compile into executable programs. Writing a model that does not compile into a meaningful program is easy. We discuss an example illustrating this point in Chapter 6 in the context of fault-tolerant systems. When it comes to mapping macros, it is crucial to provide resource implementations with similar guarantees. Another point to consider is that overloading mapping macros with excessive responsibilities can increase the trusted computing base (TCB) size, as the resource implementations may not be fully verified.

Models and programs operate at different levels of abstraction, leading to fundamental differences between them. For example, there is no notion of time in an MPCal model, while programs need timing guarantees for various reasons. Programs require efficient IO and memory access to be performant, while in MPCal models, we only have access to high-level immutable data types. Duality reflects programs' performance optimization concerns on the model side.

Similarly, implementation-side requirements can introduce complexity into models. For instance, programs aiming for improved performance often rely on concurrency. However, a highly concurrent system increases com-

plexity, enlarges the state space, and escalates the cost of model checking. This presents a trade-off in terms of reflecting performance optimization requirements in the model while maintaining a manageable level of reasoning complexity.

In the next chapter, we present an overview of RaftStore, a distributed key-value store we built using PGo. We use RaftStore as a running example in the rest of the work, where we more precisely discuss different aspects of duality.

# Chapter 4

# RaftStore

In this work, we leverage PGo to develop a distributed key-value store called RaftStore, which is built upon the Raft consensus protocol. RaftStore serves as our primary example throughout this work. This chapter provides an overview of RaftStore, starting with a concise review of the Raft consensus protocol. Subsequently, we delve into the architecture of RaftStore, highlighting its extended functionalities, including the integration of client interaction and a key-value store layer.

## 4.1 Raft Basics

Consensus algorithms are commonly formulated as a state machine replication problem, which can be solved using a replicated log. In this approach, each server maintains a log, with each entry representing a command for execution. The consensus algorithm ensures that the log remains consistent among all servers containing the same set of commands in the same order.

Raft consensus protocol works by electing a *distinguished leader*. The elected leader replicates the commands it receives from clients to other servers and tells them when it is safe to apply log entries to their state machines. A Raft cluster consists of multiple servers, each of which can be in one of three states: *follower*, *candidate*, or *leader*. Under normal operation, there is one leader, and all other servers are followers. A follower server

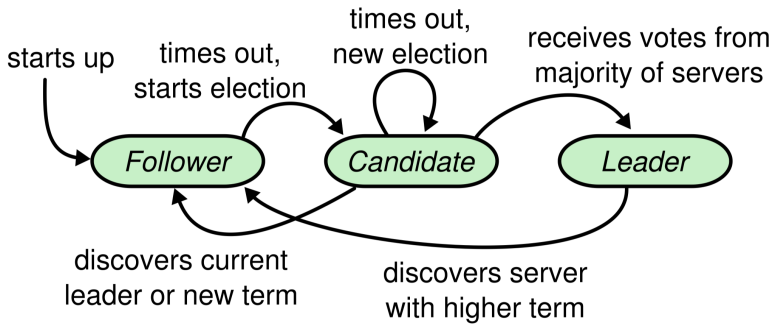



**Figure 4.1:** Raft servers' state transitions. This figure is taken from the Raft paper [59].

recognizes the leader and replicates its log entries. The leader server is responsible for accepting new client requests and replicating them across the followers. In the candidate state, a server is not a follower nor a leader and is running an election round to become the new leader. Figure 4.1 depicts the transitions between different server states.

Raft divides time into terms. Terms act as a logical clock [41] and each server stores its monotonically increasing clock in its `currentTerm` variable. Raft guarantees there is at most one leader at any given term.

We can divide the Raft protocol into two parts: (1) leader election: servers select a new leader for the cluster when no leader exists or the previous one has failed. (2) log replication: the elected leader replicates new log entries to the other servers and notifies them when they can apply a log entry to their state machine.

### 4.1.1 Leader Election

Initially, all the servers in a Raft cluster start in the follower state. Servers rely on heartbeats to initiate a leader election round. A follower remains in the follower state if it receives period heartbeats. A leader periodically sends heartbeat messages to followers. A follower times out if it does not receive a heartbeat after a while. Then, it increases its `currentTerm` by one, transitions into the candidate state, and starts a new election. A candidate node sends `RequestVote` messages to all other servers to run an election

round. A candidate wins an election if it receives votes from at least a majority of nodes. A candidate might become a follower if it receives a heartbeat message from another server. After some time, if none of these happen, the candidate times out and starts another election round.

### 4.1.2 Log Replication

After a leader has been elected in a Raft cluster, it begins to process client requests. The leader adds each request to its log and then replicates it to other servers in the cluster through `AppendEntries` messages. These messages also serve as heartbeats for leader election. Raft guarantees that all *committed* log entries are durable and will eventually be executed by all available servers' state machines. An entry is considered committed once the leader has successfully replicated it on a majority of the servers in the cluster. Each Raft server stores the index of the latest log entry known to be committed in a local variable `commitIndex`, which increases monotonically.

## 4.2 Design Overview

RaftStore is a distributed key-value store that uses the Raft protocol to replicate data among servers. RaftStore has a cluster of servers. Servers implement the Raft protocol with a key-value dictionary as their state machine. Clients interact with the cluster by sending requests to the leader. Client interaction is implemented as an extension of the Raft protocol [58, 59]. Figure 4.2 shows RaftStore in a normal operation, where one server is the leader and interacts with the clients.

RaftStore MPCal model has several archetypes and mapping macros. Figure 4.3 shows a partial architecture of RaftStore MPCal model, including some of its archetypes and mapping macros. Two IO resources are being used in a server: `net` is the network, and `log` is the Raft log that gets persisted on disk. Servers have other resources such as failure detectors that they use for fault-tolerance and several timers such as `leaderTimeout` for leader election. In addition, a Raft server has several variables that are shared among its archetypes. These variables include the server state

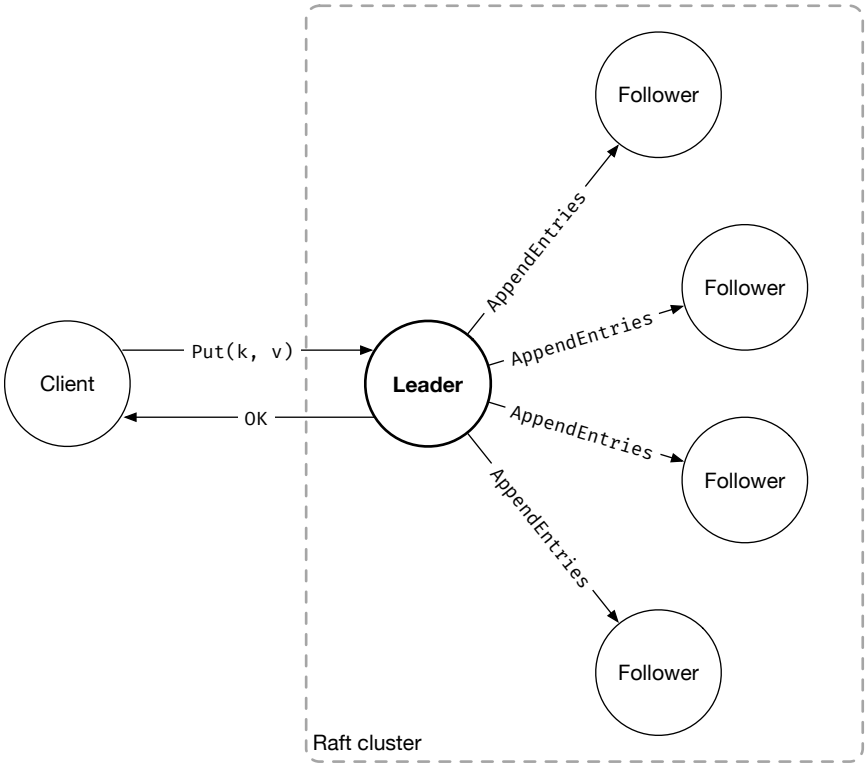



**Figure 4.2:** RaftStore architecture in a normal operation: (1) Leader receives a request from a client. (2) Replicates the new entry to followers by sending `AppendEntries`. (3) After receiving a majority of responses from followers, leader replies back to the client.

(`state`), current term (`currentTerm`), commit index (`commitIndex`), and candidate ID that the server is voted for in the current term (`votedFor`).

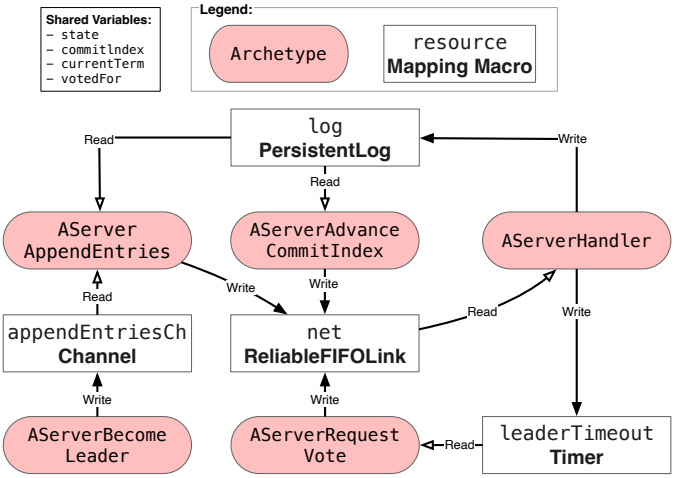
A RaftStore server has several archetypes that run concurrently. These archetypes are:

- `AServerHandler` handles the incoming messages in a server.
- `AServerRequestVote` starts a new election round in case of a leader timeout.
- `AServerBecomeLeader` detects if the current server has a quorum of

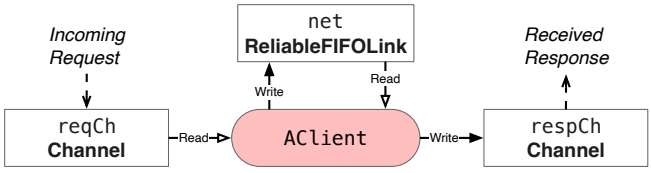
votes and then promotes itself to be the leader.

- If the current server is the leader, `AServerAppendEntries` broadcasts new entries.
- `AServerAdvanceCommitIndex` increases the commit index and replies back to the clients.

The RaftStore client consists of one archetype. The user-facing archetype `AClient` relays input requests from a channel (`reqCh`) to instances of `AServer` via `ReliableFIFOLink`. It passes corresponding responses to the user via the channel (`respCh`).




**(a)** Architecture of MPCal Raft server




**(b)** Architecture of MPCal Raft client

**Figure 4.3:** Partial architecture of RaftStore MPCal model. Arrows show the interaction between archetypes and mapping macros. The direction of each read/write arrow denotes the direction of data flow.

## 4.3 Checking Correctness

We specified five properties of the Raft protocol in the RaftStore MPCal model. These properties are:

- **Election Safety:** in every term there is at most one elected leader.
- **Leader Append-Only:** a leader only appends new entries to its log, without changing or deleting the log entries.
- **Log Matching:** If there exists an entry with the same index and term in two logs, it indicates that the logs are identical for all entries until the specified index.
- **Leader Completeness:** When a log entry is committed during a term, it implies that the same entry will be found in the logs of the leaders of subsequent higher-numbered terms.
- **State Machine Safety:** Once a server has successfully applied a log entry to its state machine at a specific index, it guarantees that no other server will apply a different log entry for that same index.

We used the TLC model checker for the RaftStore model with the properties above. In the model checking mode, TLC explores all the possible states the model allows using breadth-first search by default. We bounded this search by the number of servers, clients, failures, election rounds, and the number of entries committed to the log.

The Raft protocol may not always maintain liveness in an asynchronous network, as the FLP result indicates [19]. However, in scenarios that do not reach the worst-case conditions, Raft is designed to be capable of electing a leader and making progress. The inherent randomness observed in real-world environments enables consensus systems to function effectively despite the inherent impossibility of achieving consensus.

We have specified two liveness properties for RaftStore:

- **Election Liveness:** At any given point, eventually a server will be elected as the leader.

- **Client Progress:** A client eventually receives a response for the request it had sent. Note that a client can retry and send a request multiple times.

These properties always get violated during model checking since TLC explores all possible states. For example, in the execution path where servers keep timing out before having a majority for election, the election liveness property will be violated. We used TLC simulation mode to ensure that RaftStore is live in a real environment. In simulation mode, TLC starts from a randomly selected initial state and chooses the next step randomly up to a limited number of steps. We expect our liveness properties to hold during simulation mode with enough steps.

# Chapter 5

# Performing IO

*Pessimists sound smart.*
*Optimists make money.*

Nat Friedman

Real-world applications often use input/output (IO) operations, such as networking and storage, to interact with their environment. However, the representation and treatment of IO in models and programs differ significantly. In models, IO is typically abstracted due to two main reasons. First, the model's primary focus is often not the IO operation itself. In addition, executing actual IO operations can be costly and time-consuming. As a result, models tend to simplify and abstract IO operations, making them more concise and straightforward. Conversely, programs must handle IO operations comprehensively, considering their full intricacies and complexities.

IO operations are often regarded as part of the system environment. In the context of RaftStore, we consider IO devices such as the network and persistent log to be elements of the system environment, distinct from the system's core definition, which encompasses the Raft protocol and the key-value store layer. In an MPCal model, IO operations can be abstracted using mapping macros. Each mapping macro corresponds to a resource implementation on the implementation side. These resource implementations must adhere to the resource API defined in Figure 2.7. Modeling various IO

interfaces using simple read/write APIs of mapping macros presents a challenge. Ensuring consistent guarantees across both the IO mapping macros and resource implementations can be difficult, particularly when striving for optimal performance.

Next, we will explore the process of constructing fundamental IO devices, such as networking and storage, using PGo.

## 5.1 Networking

The link abstraction serves as a representation of the network components in a distributed system. In this work, we assume a bidirectional link connects every pair of nodes, resulting in a topology that ensures complete connectivity among all nodes. However, various topologies may be employed in practical implementations to realize this abstraction, potentially utilizing routing algorithms [6].

Link abstractions must be defined to be modeled effectively in MPCal, and a corresponding resource implementation can be provided. Links resource implementations must have the same guarantees as the resource in the MPCal model.

### 5.1.1 Unordered Lossy Link

A simple link abstraction is an unordered lossy link that connects two nodes. Two parties can send messages to each other, but the unordered lossy link guarantees neither message delivery nor the order. Hence, node $A$ can send a message $m$ to node $B$, and $B$ might not receive $m$ or it might receive $m$ before another message $m'$ that $A$ has sent before $m$.

Figure 5.1 illustrates a model of an unordered lossy link in MPCal. Each node has a network queue, and `UnorderedLossyLink` utilizes it to model the link abstraction. In the *read* part of the `UnorderedLossyLink` mapping macro, we wait until some message is available in the network queue. We use the `with` statement to randomly select one of the messages from the queue, not only the first one. This modeling approach captures the unordered nature of the link. In the *write* section of the mapping macro, we employ the `either`

statement for non-deterministic behavior, allowing the message to either be added to the queue or discarded. This modeling strategy represents the lossy characteristic of the link.

It is important to note that in our simulation, we utilized nondeterminism to model message loss and out-of-order delivery. However, these aspects are inherent properties of the environment in the implementation. Therefore, the actual implementation does not need to simulate them explicitly. The unordered lossy link resource implementation makes its best effort to deliver messages, but it does not incorporate specific mechanisms to guarantee message delivery and ordering. Message loss and reordering are properties that arise naturally from the system environment during runtime.

To implement the `UnorderedLossyLink` mapping macro, a resource implementation must adhere to the `ArchetypeInterface` interface in Figure 2.7. In the absence of delivery and ordering guarantees we can use the UDP protocol for the implementation of this mapping macro, hence we refer to the implementation as *UDP mailboxes*. UDP mailboxes must handle two distinct cases: local mailbox, which serves received messages through the `ReadValue` method, and remote mailboxes, responsible for transmitting messages to other nodes via the `WriteValue` method. It is important to note that a local mailbox does not support the `WriteValue` method, while remote mailboxes do not support the `ReadValue` method.

A remote UDP mailbox is responsible for sending messages to a specific remote node. It buffers written values in a local buffer. In case of an abort, it discards the buffer content. In case of a commit, it sends the buffered messages through a UDP socket to its remote node without waiting for an ack. Note that we can skip the pre-commit phase since `UnorderedLossyLink` does not guarantee message delivery.

A local UDP mailbox is responsible for handling received messages from all nodes. It has a queue to store these received messages. When the `ReadValue` method is invoked, the local UDP mailbox removes the first message in the queue, returns it as the result, and stores it in a temporary buffer. If the execution of the critical section is successful, the local UDP mailbox discards the contents of the temporary buffer. However, in the event of an

abort, the messages in the temporary buffer are added back to the head of the queue. If no message is available during a `ReadValue` call, the local mailbox returns an error causing the critical section to abort, prompting the PGo runtime to execute the critical section again. This results in busy waiting instead of the blocking on archetypes' goroutines.

The primary challenge in building a resource implementation lies in managing MPCal's atomicity semantics. The `ArchetypeInterface` simplifies this process by reducing it to implementing a flexible interface. The key aspect here is that the interface does not have to be as expensive as a full two-phase commit protocol. Consequently, in the resource implementation discussed earlier, we could skip the pre-commit phase due to the weak guarantees of unordered lossy links.

The unordered lossy link has an efficient resource implementation. The cost of sending and receiving messages is the same as normal send and receive operations. Using the UDP protocol further avoids the intricacies associated with more complex protocols like TCP, leading to potential performance improvements in specific environments. However, it is important to note that model checking becomes expensive when using unordered lossy links due to the increased non-determinism in the model. As a result, we opted not to employ unordered lossy links in most of the systems built in MPCal, including RaftStore, prioritizing model checking efficiency for these scenarios.

### 5.1.2 Reliable FIFO Link

A reliable FIFO link guarantees both delivery and order to senders and receivers. Figure 5.2 shows a mapping macro that models this link in MPCal, the network model that we used in the lock server example in Chapter 2 and the RaftStore MPCal model.

Reliable FIFO links combined with the semantics of MPCal labels are powerful as we can use them to implement atomic broadcast. The `AtomicBroadcast` archetype, depicted in Figure 5.3, transmits a message to two nodes within a label. Since labels execute atomically, either both nodes receive the mes-

```
mapping macro UnorderedLossyLink {
  read {
    await Len($variable) > 0;
    with (readMsg \in $variable) {
      $variable := $variable \ readMsg;
      yield readMsg;
    };
  }
  write {
    either {
      yield $variable;
    } or {
      yield $variable \cup {$value};
    };
  }
}
```

**Figure 5.1:** Unordered lossy link modeled in MPCal.

sage or none of them. This characteristic poses a challenge for the resource implementation of the `ReliableFIFOLink`, as the critical section can be as complex as a full consensus round [9, 11].

The resource implementation of `ReliableFIFOLink` utilizes the TCP protocol for reliable packet transmission over the network, referred to as *TCP mailboxes*. Similar to UDP mailboxes, TCP mailboxes handle local and remote mailboxes separately. The implementation of TCP mailboxes ensures that all nodes participating in a critical section either receive all messages in the same order or none of them receive anything. Thus, TCP mailboxes implement the full two-phase commit protocol [26] in the `ArchetypeInterface`. During the pre-commit phase, TCP mailboxes ensure that all participating nodes agree on the messages and their order before attempting to commit the critical section. If the pre-commit phase fails, TCP mailboxes abort the critical section. TCP mailboxes assume the commit phase always succeeds. This is based on the `ArchetypeInterface` assumptions and may causes liveness issues as mentioned in Chapter 2.

```
mapping macro ReliableFIFOLink {
  read {
    await Len($variable) > 0;
    with (readMsg = Head($variable)) {
      $variable := Tail($variable);
      yield readMsg;
    };
  }
  write {
    yield Append($variable, $value);
  }
}
```

**Figure 5.2:** Reliable FIFO link modeled in MPCal.

```
archetype AtomicBroadcast(ref network[_]) {
lbl:
  network[NodeA] := "hi";
  network[NodeB] := "hi";
}
```

**Figure 5.3:** Modeling atomic broadcast using reliable FIFO links.

### 5.1.3 Relaxed Reliable Link

Performing a full consensus round for every network send introduces significant overhead. However, in many cases, this additional cost can be avoided. For instance, consider the `serverReceive` operation presented in Figure 2.5 within Chapter 2, which involves a single network operation. In this scenario, employing TCP mailboxes that implement the complete two-phase commit protocol would result in unnecessary overhead. Since there is only one network send operation within the critical section, and the remaining operations are local and do not trigger an abort, we can optimize the process. After sending the message and receiving an acknowledgment for its delivery, we can be confident that the execution of the critical section will succeed. As a result, we can safely skip the pre-commit and commit phases.

PGo empowers users to mitigate the additional performance overhead associated with reliable FIFO links by leveraging extra domain knowledge to relax resource semantics. In the case of the `serverReceive` critical section,

a *relaxed resource* implementation can be employed. This implementation can simplify the pre-commit and commit phases, treating them as trivially successful and only sending messages without requiring coordination for failure cases. Our relaxed implementation of TCP mailboxes assumes that no abort will occur during execution. Consequently, if an abort does happen, the implementation will panic. To achieve the best performance, we used the relaxed version of TCP mailboxes in the implementation of RaftStore.

This approach of semantic weakening can be applied to various relevant scenarios with careful consideration. It provides a means to prioritize performance over heavier-weight correctness guarantees when necessary. However, it is important to note that these relaxations may increase the size of the trusted computing base as they deviate from the complete implementation of an MPCal resource definition. Nevertheless, through developer discipline and good practices, we have observed that these trade-offs can be effectively managed while still leveraging the benefits of protocol-level verification. We leave automated methods of ensuring the correctness of these relaxations to future work.

## 5.2 Storage

Storage is a crucial IO device for storing data. Storage devices offer various interfaces, including block, file system, log, and object. Effectively modeling these storage interfaces is essential to achieve high performance. We delve into the techniques and strategies employed in this chapter to ensure efficient modeling, enabling us to optimize performance with storage operations.

### 5.2.1 File System

The file system is a fundamental abstraction that facilitates working with storage systems. In our modeling approach, we represent the file system using a map data structure. Each element of the map will refer to a file, with keys and values being required to be string-typed, and keys being required to refer to valid paths (or create-able paths, if a key is written to before it is read). The file system model in MPCal is illustrated in Figure 5.4.

```
1  mapping macro FileSystem {
2    read { yield $variable; }
3    write { yield $value; }
4  }
5
6  variable fs = [id \in NodeSet |-> []];
```

**Figure 5.4:** A simple file system modeled in MPCal.

The resource implementation of the `FileSystem` mapping macro incorporates a buffering mechanism for written values, which temporarily stores them in memory. Upon invoking the `Commit` operation, the implementation atomically writes the buffered values to a specified location within the file system. Conversely, in the event of an abort, the in-memory buffer is simply discarded. Given our assumption that writing to the underlying file system is always possible, the pre-commit phase trivially succeeds. This design efficiently handles file system operations while maintaining the desired semantics and guarantees of the `FileSystem` mapping macro.

### 5.2.2 Persistent Log

RaftStore has a log that has to be persisted on disk. The log supports the following operations:

- Reading a range of elements within the log.
- Appending one or more elements to the end of the log.
- Removing one or more elements from the end of the log.

The file system abstraction does not align well with the requirements of a log, where elements are added or removed from the end. To address this, we have devised an alternative approach to model the log in a way that provides a streamlined interface within the MPCal specification. The `PersistentLog` mapping macro, depicted in Figure 5.5, embodies our log specification in MPCal. The `SubSeq` function there returns a subsequence of log between the given indexes. Leveraging a list as the underlying data

```
mapping macro PersistentLog {
  read {
    yield $variable;
  }

  write {
    if ($value.cmd = LogConcat) {
      \* \o is the concatenation operator
      yield $variable \o $value.entries;
    } else if ($value.cmd = LogPop) {
      yield SubSeq($variable, 1, Len($variable) - $value.cnt);
    };
  }
}
```

**Figure 5.5:** Raft persistent log modeled in MPCal.

```
log[i] := [
  cmd     |-> LogConcat,
  entries |-> entries
];

log[i] := [
  cmd |-> LogPop,
  cnt |-> count
];
```

**Figure 5.6:** Raft persistent log add and remove operations.

structure for the log, the `PersistentLog` mapping macro defines the semantics of read and write operations. In a read operation, the entire list is simply returned. However, the write operation is more intricate, as it requires different arguments to distinguish between addition and removal scenarios. The interaction between the archetypes and the log, involving the writing of a record with specific fields, is illustrated in Figure 5.6.

The resource implementation of `PersistentLog` leverages BadgerDB [51, 63] as the storage engine, which is an embedded high-performance key-value store. The pre-commit, commit, and abort phases in the resource implementation of the `PersistentLog` mapping macro closely resemble those of the file system resource implementation.

# Chapter 6

# Modeling and Dealing with Failures

> *A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.*
>
> Leslie Lamport

Failure is a critical aspect to consider when designing distributed systems. Failures are prevalent in real-world scenarios, and distributed systems deployed in a production environment must be able to tolerate failures.

In a production environment, failures can arise due to factors including software errors, hardware malfunctions, network outages, and power disruptions. However, failures do not inherently exist in models and users have to model them accurately.

PGo-based systems have a dual requirement that they must be able to handle failure both on the model side and implementation side. During model checking, the model itself must be capable of tolerating simulated failures, while the compiled Go program must be equipped to handle failures during execution.

```
mapping macro FaultyLink {
  read {
    \* same as the reliable FIFO link
  }
  write {
    either {
      yield Append($variable, $value);
    } or {
      yield $variable; \* silently drop message
    };
  }
}
```

**Figure 6.1:** Modeling a faulty network link in MPCal.

Fault tolerance is dependent on the system model and failure model. In this chapter, we assume an asynchronous computing model where there is no bound on message delays. We assume nodes fail with crash failure semantics [6] and/or network partitions might happen. This model is a realistic assumption about distributed systems running in data centers. We build RaftStore using this system model.

In the rest this chapter, we will describe how we can simulate failure behavior in the model and how we can handle them in the model and the compiled system.

## 6.1 Modeling Failure Behavior

Network faults in PGo can be represented using a `mapping macro` that incorporates weak guarantees. For instance, the fault model depicted in Figure 6.1 demonstrates a faulty network link. When a message is transmitted through this link, it may or may not be delivered. To capture this nondeterministic behavior, we employ the `either` statement, which allows us to express the two or more possible outcomes.

The idiom presented in Figure 6.2 offers a mechanism to simulate crash failures. In order to simulate a crash failure for the `AServer` archetype, a concurrent process named `ServerCrasher` is executed alongside it, and it is responsible for triggering the crash of the server. To prevent a failed

```
process ServerCrasher(serverId) {
serverCrash:
  netEnabled[serverId] := FALSE;
fdUpdate:
  fd[serverId] := TRUE;
}

archetype AServer(ref net[_], ref netEnabled[_]) {
lbl:
  \* beginning of a critical section that might fail
  \* \lnot is the logical negation operator
  if (\lnot netEnabled[self]) {
      await FALSE;
  };
}
```

**Figure 6.2:** Modeling crash failures in MPCal.

process from receiving any network messages, we use the toggle `netEnabled` to disable networking for the server. This ensures that the failed process is isolated from incoming network communication.

`ServerCrasher` turns off the networking of its corresponding server in the first label, `serverCrash`. In the `AServer` archetype, there exists a code snippet at the beginning of its critical sections that stops the server's execution if its networking is disabled. This is achieved by blocking the server forever using `await FALSE;` statement. The disabled networking serves as a reliable indication of a server failure. Moving on to the `fdUpdate` label of `ServerCrasher`, it updates the state of its failure detector, which is further discussed in Section 6.2.

We used PlusCal processes for `ServerCrasher` because we only want them for model checking, and there is no need to compile them to Go since, during runtime, failures manifest as a result of the environment. Similarly, we provide dummy resource implementation for `netEnabled` such that it does not interfere with normal execution of an archetype.

```
1  either {
2    network[id] := msg; \* some communication
3  } or {
4    \* handle failure...
5  };
```

**Figure 6.3:** Each branch will be taken non-deterministically at runtime. This makes the process to execute handle failure branch without even knowing that a failure has happened.

```
1  either {
2    network[id] := msg; \* some communication
3  } or {
4    await fd[id]; \* only run if failure detector reports <id> has failed
5    \* handle failure...
6  };
```

**Figure 6.4:** Handling failures with a failure detector in MPCal archetypes.

## 6.2 Handling Failures with Failure Detectors

Producing a model of failure handling from which PGo can generate a reasonable implementation can be subtle. For example, an `either` statement can be used to explore failures with model checking, but this expresses what *could happen*. Consider Figure 6.3, which over-approximates failure in a way that does not work in an implementation. During verification, this code means that either line 2 will execute successfully, or a failure will be handled on line 4. This is fine for verification, as it expresses what could happen, either due to some transient issue, or due to the failure of a peer. This type of pattern would allow an implementation to spontaneously handle a failure, *regardless of whether any failure was detected*. Since the PGo runtime schedules attempts at executing branches of an `either` statement using a round-robin strategy, this means that, while the system would technically implement the specification, roughly one in two network operations would directly jump to failure recovery without even attempting communication.

In the implementation, we want to attempt the network send operation

```
1  mapping macro PerfectFailureDetector {
2    read { yield $variable; }
3    write { yield $value; }
4  }
```

**Figure 6.5:** Perfect failure detector modeled in MPCal. A perfect failure detector guarantees strong completeness and strong accuracy.

initially and, if a timeout occurs, proceed to execute the failure handling code. However, MPCal has no notion of time, and we discuss this more in Chapter 7.

To handle failures, we use *failure detectors* to abstract time and eliminate unwanted executions. This approach is depicted in Figure 6.4. By employing a suitable implementation of the `fd` resource, the failure handling branch will only be executed when the `fd` resource indicates that the remote process has failed. If `fd[id]` returns `FALSE`, the execution will be rolled back, allowing the other branch to be attempted. This design enables having practical failure checks during runtime and parameterizes verification with different failure detectors.

Failure detectors are characterized by two properties, *completeness* and *accuracy* [9]. Strong completeness states that eventually every node that crashes is permanently suspected by every correct node. Strong accuracy states that no process is suspected before it crashes. A perfect failure detector is a failure detector that is both strongly complete and accurate. Figure 6.5 depicts a mapping macro in MPCal that represents a perfect failure detector. The resource `fd` is associated with this failure detector and its values are updated by the `ServerCrasher`, as illustrated in Figure 6.2. When a server fails, its networking is first disabled, and eventually its failure is detected by the failure detector. A failure detector eventually detects node failures by updating the `fd` resource in a separate label following the `serverCrash` label within the `ServerCrasher` process.

In an asynchronous environment in practice, we do not have any accuracy guarantee for failure detectors. Chandra and Toueg [9] showed that it is

```
mapping macro PracticalFailureDetector {
  read {
    if ($variable = FALSE) { \* process is alive
      \* no accuracy guarantee
      either { yield TRUE; } or { yield FALSE; };
    } else {
      yield $variable; \* strong completeness
    };
  }

  write { yield $value; }
}
```

**Figure 6.6:** Practical failure detector modeled in MPCal. Practical failure detector guarantees strong completeness.

possible to solve consensus in an asynchronous environment using a perfect failure detector. From the FLP result [19], we know solving consensus in an asynchronous network environment is impossible. Thus, having a perfect failure detector in an asynchronous network environment is impossible too.

We can have a failure detector in practice that guarantees strong completeness. This failure detector eventually detects crashed nodes as failed, but alive nodes might get detected as failed due to the lack of accuracy. Figure 6.6 captures this failure detector in MPCal, named practical failure detector.

We used failure detectors in RaftStore to detect failed nodes while sending messages to other nodes. We have tried both perfect and practical failure detectors in RaftStore. Using practical failure detectors gives us stronger guarantees as it is closer to the runtime environment; however, it makes the model checking process more expensive by adding more non-determinism.

A model with a failure detector is still a model, and the essential question is how close of an approximation one desires to the real runtime environment. Full fidelity is neither feasible nor desirable, considering a model's purpose is to be an abstract verification tool.

## 6.3 Resource Implementation

Failure detectors resource implementation consist of two abstractions: monitor and failure detector resource.

A monitor monitors archetypes execution. It provides an API for registering archetypes and it monitors the archetypes by wrapping them. A monitor provides the `IsAlive` API, which can be queried to find out whether a specific archetype is alive. At most one monitor should be defined in each OS process and it catches all archetypes' goroutines errors and panics. In the case of an error or a panic, the monitor responds false to `IsAlive` requests. The Monitor exposes the `IsAlive` API as a Remote Procedure Call (RPC). In the event of a complete failure of the OS process, subsequent calls to `IsAlive` times out. This timeout behavior serves as an indication of the failure of the queried archetype.

Failure detector is an archetype resource that provides guarantees of a failure detector in practice, as modeled in Figure 6.6. Each server is assigned an index, such that `fd[i]` represents the state of the failure detector for server `i`. The failure detector periodically initiates the `IsAlive` RPC to check the state of the corresponding servers. On a false response or a timeout of the RPC call, the failure detector marks the respective archetype as failed. An example scenario demonstrating the execution of the monitor and failure detector, where the monitor responds with false, is illustrated in Figure 6.7. Another scenario is depicted in Figure 6.8, where the monitor's OS process crashes, resulting in a timeout during the failure detector's call.

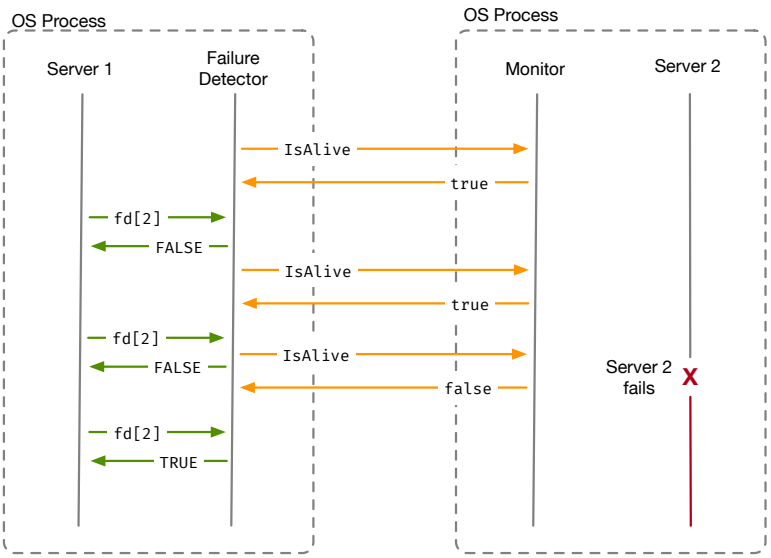



**Figure 6.7:** Failure detector and monitor execution example, when an archetype crashes and monitor replies false to the subsequent `IsAlive` request.

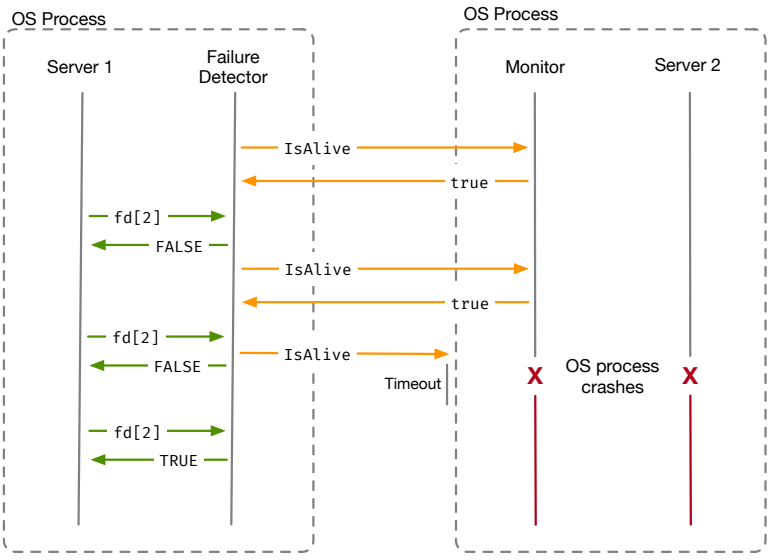



**Figure 6.8:** Failure detector and monitor execution example, in a case where the the entire OS process crashed and the failure detector detects that with a timeout.

# Chapter 7

# The Problem with Time

*Other times are just special cases of other universes.*

David Deutsch

Timing plays a pivotal role in the design of systems, finding application in numerous common patterns. Time-based mechanisms are vital for addressing challenges like timeouts, scheduling periodic activities, cache expiration, leases, and leader election. However, a key duality problem arises concerning time in the context of models and programs. While programs require timing for the aforementioned purposes, MPCal models lack the notion of time. This raises the fundamental question of how to construct models that lack the concept of time, while at the same time enabling their corresponding programs to use timing effectively.

In the context of failure detection, as outlined in Section 6.2, we proposed the abstraction of failure detectors as a means to abstract time. However, failure detectors are limited to the failure detection problem and we cannot solve all the problems that need timing with failure detectors. Because many of the problems that deal with timing, such as scheduling activities, do not necessarily involve failures.

On the implementation side timing problems can be effectively addressed by employing a straightforward timer abstraction. Algorithm 1 presents a

timer implementation that offers a viable solution. In the first line of the algorithm, a timer object $t$ is created to trigger after the specified timeout duration. This timer object includes a method called $t.Wait()$, which suspends the execution of the current process until the timer fires. By using such timer objects, we can solve the timing-related problems in programs. The duality problem arises where there is no notion of time in models. Hence, we cannot have such a timer object in MPCal models.

**Algorithm 1** Simple timer object

$t \leftarrow timer(timeout)$
$t.Wait()$ ▷ process blocks until $t$ fires

We propose an abstract timer on the model side and a concrete timer resource on the implementation side. We demonstrate that the execution semantics of the concrete timer resource aligns with the timer on the model side. We ensure that both sides consistently handle timing-related aspects while satisfying their concerns.

In an MPCal model, we represent a timer by a boolean value that serves as a resource passed to archetypes. The resource, denoted as `t`, is true when the timer has fired, and false otherwise. To illustrate this, consider the archetype `ANode` shown in Figure 7.1, where it is has the timer resource `t`. The archetype `ANode` will be blocked until the timer `t` fires, indicating $t$ is true.

The key question to address is when does `t` fire and what is its timeout value. On the model side, we assume that regardless of the timeout value, every time the archetype `ANode` executes, the timer `t` also fires. Essentially, this treats `t` as a dummy value on the model side that is *always true*.

The timer resource implementation provides a proper implementation of a timer object. It takes a timeout value and implements the timer object described in Algorithm 1 as an archetype resource. The implementation includes an internal timer object, and when this timer has fired, the `ReadValue` function returns true; otherwise, it returns false. In the `await` statement demonstrated in Figure 7.1, waiting on the timer implies that the execution of the critical section will be aborted if the timer resource's value is false.

```
1  \* archetype Node has access to timer t as a resource
2  archetype ANode(ref t) {
3  lbl:
4    await t; \* ANode blocks until t fires
5    \* some work
6  }
```

**Figure 7.1:** A timer modeled as a boolean resource in MPCal. Every time when archetype `ANode` gets executed, timer `t` is true, which means it has fired.

Consequently, the archetype will execute only when the timer value is true, indicating that its internal timer has fired.

On both the implementation side and the model sides, a critical section with a timer gets executed only if its timer is true. Therefore, the timer resource has the same semantics both on the model and implementation side. It is important to note that this is not a formal proof showing the strict equivalence, and particularly, we did not reason about implementation of the timer resource.

In RaftStore, we employ timers in two different places. First, the leader employs a timer to schedule the periodic sending of `AppendEntries` messages to its followers. This ensures that the followers receive regular updates from the leader. Second, each follower maintains a timer that triggers if it does not receive an `AppendEntries` message within a specified timeout period. Once this timeout occurs, the follower transitions into a candidate state and initiates a new round of leader election.

Note that instead of using failure detectors, we could handle failures by implementing timeouts using timers. However, this approach significantly increases the state space on the model side. It also reduces the flexibility of failure detection and leads to exploring the failure handling branch on the model side regardless of the receiving node's state. In contrast, using failure detectors, as discussed in Chapter 6, provides greater flexibility in handling failures and improves the efficiency of model checking.

# Chapter 8

# Achieving High Concurrency

*The wise use of leisure, it must be conceded, is a product of civilization and education.*

Bertrand Russell

In this chapter, we explore the process of building concurrent systems using PGo. Initially, we examine the duality challenges that arise during the development of such systems, considering both the model and implementation aspects. Subsequently, we present our experience in building RaftStore and how we addressed its need for a concurrent server implementation.

## 8.1 Concurrency and Duality

Efficiency and performance are essential requirements for a software implementation. High concurrency plays a significant role in attaining better performance. By leveraging high concurrency, software systems can effectively utilize available resources and maximize throughput. Concurrent execution allows multiple tasks to be executed simultaneously, minimizing idle time and maximizing the utilization of processing capabilities, resulting in improved throughput and latency in the system.

In the context of model-implementation duality in PGo, achieving high

performance, which often involves high concurrency, is primarily an implementation concern. However, when building systems using PGo and compiling models into programs, this requirement also influences the model side. As users develop an MPCal model, they must consider how this model compiles into Go.

Introducing a high level of concurrency in an MPCal model can lead to increased complexity and a larger state space. We can increase the concurrency level in an MPCal model by having more concurrently executing archetypes or by utilizing finer-grained labels. However, such expansions result in a larger state space, which in turn escalates the cost of model checking. As a result, a duality trade-off emerges, wherein a more concurrent model becomes more challenging to reason about while offering the potential for improved implementation concurrency. Striking a balance between these factors is crucial for users, as they must identify a point where the model checking cost remains feasible while achieving a desired level of implementation concurrency.

When building a highly concurrent system in PGo, users must consider the compilation semantics that influence the execution behavior. It is possible to have two different models that describe the same system; however, when compiled into a Go program, one may execute concurrently while the other does not. For instance, consider the system depicted in Figure 8.1, which comprises an archetype `SingleThread` having an `either` statement that non-deterministically executes either `TaskA` or `TaskB`. In contrast, Figure 8.2 models the same system using two archetypes, `ThreadA` or `ThreadB`, each executing `TaskA` and `TaskB` respectively. Both models, ultimately, compile to the same TLA$^+$ specification, illustrated in Figure 8.3, capturing the two possible transitions for the next step, `TaskA` and `TaskB`, in the system's state machine. The resulting PGo program from Figure 8.1 runs sequentially as a single archetype, while the program generated from Figure 8.2 comprises two archetypes capable of concurrent execution.

It is important to mention that high concurrency does not necessarily result in a high performance. In a highly concurrent program, problems such as resource contention and process coordination can be expensive. Some of

```
archetype SingleThread() {
lbl:
  while (TRUE) {
    either {
      \* TaskA
    } or {
      \* TaskB
    };
  };
}
```

**Figure 8.1:** Archetype `SingleThread` models a system which either does `TaskA` or `TaskB` at each step. PGo compiles this archetype into a sequential program.

```
archetype ThreadA() {
lbl:
  while (TRUE) {
    \* TaskA
  };
}

archetype ThreadB() {
lbl:
  while (TRUE) {
    \* TaskB
  };
}
```

**Figure 8.2:** Archetypes `ThreadA` and `ThreadB` model a system where either does `TaskA` or `TaskB` at each step. PGo compiles this archetype into a concurrent program, where two archetypes run concurrently.

$$Next \triangleq TaskA \lor TaskB$$

**Figure 8.3:** TLA$^+$ compilation of Figure 8.1 and Figure 8.2.

these problems actually have to be addressed on the model side, however, are not visible during model checking and they become apparent by running the compiled program.

In an MPCal model, archetypes typically represent individual nodes in the system, and each archetype runs sequentially. However, high-performance systems often involve multiple threads executing concurrently. To accommodate such scenarios in PGo-based systems, we adopted a new design approach to run multiple archetypes on the same node simultaneously. Although this pattern enables enhanced performance, it introduces new challenges, such as the need to share resources between these archetypes and coordinate their execution.

To facilitate communication between archetypes running on the same machine, we use channels inspired by the Go programming language and communicating sequential processes (CSP) [34]. In this design, senders can transmit messages through a channel, and the receiver will receive the message. If a channel happens to be empty, the receiver will be blocked until a message becomes available on the channel. This mechanism enables synchronization and coordination among concurrent archetypes.

Sharing variables among different archetypes works as expected without any additional work on the model side. However, on the implementation side, using unprotected local resources when different archetypes access them can lead to data inconsistency and race conditions. To address this, we introduced a shared resource that encapsulates a local resource with a read-/write lock. This approach allows multiple readers to access the resource simultaneously, while ensuring that only one writer can modify it at a time. The shared resources use timeout when acquiring the lock to prevent deadlocks. In case of a timeout, the execution of the critical section is aborted and the PGo runtime retries the execution.

## 8.2 How We Made RaftStore Highly Concurrent

In the initial version of RaftStore, the Raft server was represented by a single archetype. This resulted in a long `either` statement in the Raft server

archetype, with various tasks specified in each branch. During execution, the PGo runtime would nondeterministically select a branch to execute. If the execution failed, the runtime would abort and attempt the critical section again, choosing a different branch. The original model of the Raft server archetype is depicted on the left side of Figure 8.4.

After compiling the initial RaftStore version to Go, we noticed some problems. Specifically, we realized that leader election is not working due to the sequential execution of the Raft server. According to the Raft consensus algorithm, when a node becomes the leader, it should promptly notify the other nodes by sending them `AppendEntries` messages. However, the sequential execution caused a delay in sending these messages, leading to timeouts in other nodes, which triggered new election rounds repeatedly.

To address this problem, we separated the `AppendEntries` branch from the main Raft server archetype and created a new archetype dedicated solely to sending `AppendEntries` messages, if the current server is the leader. With this change, the `AppendEntries` logic can now execute concurrently with the rest of the Raft server, allowing for faster and more efficient communication between the nodes during leader election. This change made the leader election work in RaftStore.

To further improve RaftStore performance, we deleted the entire `either` statement from the server archetype and created separate archetypes for each branch. This architectural transformation allowed us to execute each task concurrently, resulting in a significant performance boost. The revised RaftStore server architecture is depicted on the right side of Figure 8.4. Each of these archetypes now corresponds to a specific task, allowing them to execute concurrently. Additionally, we introduced shared variables and channels for synchronization and data sharing among the archetypes.

Throughout the development of the RaftStore model, we adopted a deliberate strategy to minimize the number of labels in each archetype. This approach was crucial in ensuring that the model checking cost remains feasible and manageable. Additionally, we made the assumption that all server archetypes fate-share, thereby eliminating the possibility of partial failure among them.

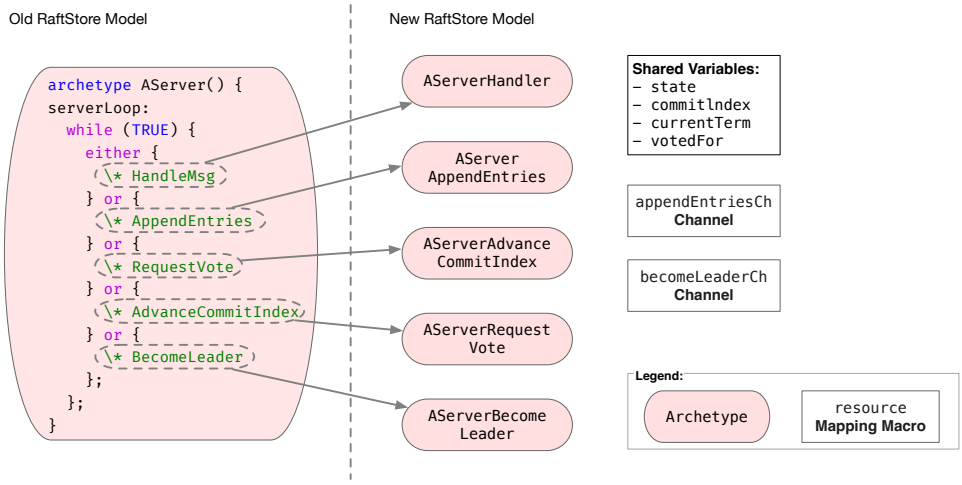



**Figure 8.4:** Architecture of RaftStore. The left side shows the initial version, which had one archetype. The right side shows the new architecture that enables concurrency. Arrows demonstrate how we create a distinct archetype from each branch of the `either` statement in the initial version.

One of the major challenges we encountered while transforming RaftStore into a concurrent system was the management of complexity. In addition to carefully designing the model to ensure that the model checking cost remains feasible, we had to address the issue of managing shared variables among the five server archetypes in RaftStore. To tackle this, we defined all the shared variables as resources and passed them to each archetype. However, this approach resulted in a significant amount of error-prone boilerplate code, both in the model and the implementation, to initialize the system. With more appropriate language features in MPCal, we could have mitigated these complexities and provided a more seamless developer experience. We consider these potential improvements as a topic for future work.

# Chapter 9

# Modular Design

*Make each program do one thing well.*

Douglas McIlroy

So far, we have been talking about systems that consist of a single model. This chapter explores a design approach where a system comprises several models. We call this a modular design.

Building systems modularly is an essential step in building large-scale systems. Modularity allows us to manage the complexity in large systems. This design approach makes models smaller and easier to reason about. It also reduces the state space of each component model, which makes it possible to scale MPCal models beyond the conventionally viable limits of model-checked systems.

The fundamental principle behind modular design is to divide the system into smaller, more manageable components. Let us consider a modular system denoted as $S$, having $n$ components, namely $c_1, \ldots, c_n$. Each component, such as $c_i$, interacts with a specific set of other components denoted as $M_i$. The model of component $c_i$ abstractly defines its interactions with each component $c_j \in M_i$ and establishes the communication protocol between $c_i$ and $c_j$. Moreover, component $c_j$ adheres to the communication contract specified by $c_i$ and defines compatible communication channels in its own

specification to ensure correct interaction.

We model check each component $c_i$ individually. This technique significantly reduces the model checking cost. Let $c_i$ be a component with at most $a_i$ enabled actions in each step, and we run the model checker for traces up to depth $d$. If we consider a monolithic system $L$ consisting of $c_1, \ldots, c_n$, the size of state space that the model checker needs to explore is

$$\left(\sum_{i=1}^{n} a_i\right)^d.$$

While the size of state space that model checker explores for a modular system $S$ consisting of $c_1, \ldots, c_n$ is

$$\sum_{i=1}^{n} {a_i}^d.$$

It is trivial to see that size of state space in system $S$ is smaller than $L$'s state space, knowing $a_i$ and $d$ values are positive.

We compile each module to Go and connect them on the implementation side with respect to the communication contract that is defined in each component's model.

Modularity in PGo-based systems offers the advantage of reducing the trusted computing base (TCB). We provide hand-written resource implementations in Go, which allows for maximum flexibility and the opportunity to perform various performance optimizations. However, such resource implementations contribute to the system's TCB. To mitigate this risk and minimize the TCB, it is essential to allow verification of resource implementations, particularly since these resources may encapsulate complex distributed protocols (e.g., distributed mailboxes). Users can define components and necessary communication channels in a way that one component uses the other component as a resource, and this way they can have a verified resource implementation.

At this time, it is not feasible to integrate different MPCal specifications in a way that ensures soundness during verification. The verification pro-

cess is limited to one specification at a time. Nevertheless, it is still possible to manually link MPCal specifications and align the semantics of different components through manual correspondence. When combining different specifications, users must ensure that the contracts between the components remain valid after compilation. We recognize this as a challenge that could benefit from further tooling automation, and we leave it as a topic for future investigation and improvement.

To demonstrate the modular design in PGo, we developed a modular version of RaftStore, called RaftStoreMod, which consists of two distinct components. The high level architecture of RaftStoreMod is shown in Figures 9.1 and 9.2. The first component is the Raft protocol, named RaftProto, represents the core Raft protocol without client interaction semantics. The second component is the key-value store system, called DistKV, which operates as a distributed key-value store, handling client requests and interacting with an abstract state machine replication component.

RaftProto and DistKV interact using two channels: accept and propose. Each RaftProto server has a pair of accept and propose channels. A RaftProto server accepts new requests through its propose channel and broadcasts committed entries through its accept channel.

DistKV is composed of multiple servers, each responsible for maintaining a local key-value database. These servers interact with an abstract state machine replication system to keep their local database in sync. Similar to RaftProto, each DistKV server possesses a pair of accept and propose channels. When a client request is received, a DistKV server forwards the request to the state machine replication component via its propose channel. Once the new entry is committed in the state machine replication component, the DistKV servers receive notifications through their accept channels, allowing them to update their local database and respond to the clients accordingly. By compiling each of these individual models and linking them in Go, we obtain the full implementation of RaftStoreMod.

In Chapter 11 we compare the runtime performance of RaftStoreMod store with several others, including RaftStore.

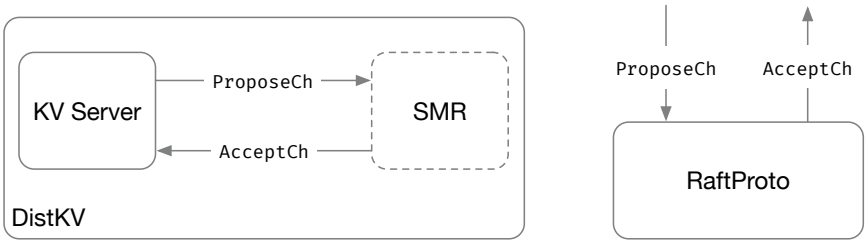



**Figure 9.1:** Architecture of RaftStoreMod components. The left side shows the high level architecture of DistKV. Each KV server interacts with an abstract state machine replication component (SMR). The right side shows the RaftProto component, where it communicates with its propose and accept channels.

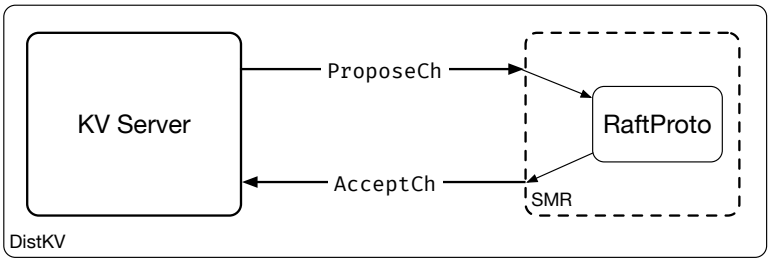



**Figure 9.2:** RaftStoreMod implementation architecture. On the implementation side, we use RaftProto to implement the abstract state machine replication (SMR) component of DistKV.

# Chapter 10

# Development Process

*Software engineering is what happens to programming when you add time and other programmers.*

Ross Cox

In this chapter, we outline our development process, which draws from our experience in building systems using PGo. Managing duality is an integral aspect of our development approach, as we strive to optimize for both model and program concerns. Balancing the requirements of the model and the implementation is a central challenge, and our process aims to find the most effective trade-offs to achieve a well-optimized and reliable system. We share instances of our experience using this process while building RaftStore.

Our development process of building systems using PGo is an iterative process of (1) model engineering and (2) implementation optimization.

Our system development process begins with model engineering. During this stage, our focus is on modeling concerns, disregarding implementation details. Our objective is to create an MPCal model of the system and perform model checking to verify its correctness without delving into the intricacies of implementation. To keep the model checking cost manageable, we start by modeling the simplest system, comprising the minimal number

of archetypes and labels. This approach ensures that we establish a solid foundation by confirming the correctness of a basic system through model checking.

Once we have our initial model-checked specification, we proceed with compiling the model into Go and crafting the necessary glue code to create the first version of the running system. At this stage, we begin to identify the requirements originating from the implementation side, some of which have implications on the model side. In cases where the model does not translate coherently into a program, alternative specifications must be considered. Additionally, due to the unique execution behavior of the implementation, certain adjustments might be required in the model. An example of this scenario is the RaftStore leader election, which we previously discussed in Chapter 8.

The subsequent step involves improving the performance of the implementation. During the implementation optimization phase, we identify bottlenecks in the implementation and strategize how to address them. Some problems can be resolved on the implementation side, such as using relaxed resources to get a performance boost. Some other problems require a different approach to modeling, for example, building a model that compiles into a concurrent program. Furthermore, certain bottlenecks may be inherent to the model itself, requiring the development of a more efficient model. For instance, in RaftStore, we have implemented Raft optimizations, such as batching [58], to improve performance. These protocol-level optimizations must be applied directly to the MPCal model to achieve the desired efficiency.

After identifying the implementation requirements, we return to the model engineering phase, where we incorporate these changes. Nevertheless, the primary focus remains on addressing the modeling concerns and ensuring the system can be model-checked. Once the adjustments are made, we compile the model into Go, and this iterative process continues until we achieve the desired system. Figure 10.1 demonstrates this iterative process.

We recommend that new users interested in building systems with PGo or similar frameworks follow this development process, commencing with

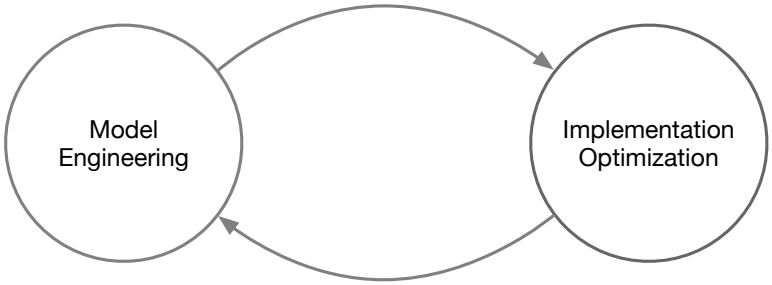



**Figure 10.1:** Development process of building systems using PGo where we face model-implementation duality. This is an iterative process of model engineering and implementation optimization.

the simplest model. As they progress from the model to a functioning implementation, they will inevitably encounter duality challenges. Our work addresses various duality challenges we faced while constructing complex distributed systems using PGo. We believe that other users will also confront similar challenges, and we encourage them to refer to this paper for guidance in addressing these issues.

# Chapter 11

# Evaluation

*The Eiffel Tower was built in 2 years and 2 months; that is, in 793 days. When completed in 1889, it became the tallest building in the world, a record it held for more than 40 years.*

Jill Jonnes

In our evaluation, we address three key questions: (1) To what extent were the requirements of each side (model and program) satisfied in the presence of duality? Specifically, we assess how effectively we managed to fulfill the requirements of both the model side and the program side. (2) What is the development effort involved in building systems using PGo, considering the challenges posed by duality? (3) What is the performance overhead incurred when building modular systems using PGo?

## 11.1 Evaluated Systems and Methodology

Table 11.1 lists the seven systems we have constructed using MPCal. The MPCal along with the compiled PlusCal, TLA$^+$, and compiled and glue Go code for these systems are available in our GitHub repository [28]. RaftStore and RaftStoreMod are consensus-based key-value stores that we discussed

in Chapters 4 and 9. RaftStoreMod is a modular composition of the pure Raft protocol specification (RaftProto) and DistKV as described in Chapter 9. PBStore is a primary-backup key-value store where the primary node synchronously replicates client requests to backup nodes. PGoCRDT is an add-wins observed removed set (AWORSet) state-based Conflict-free Replicated Data Type (CRDT) [3] that uses vector clocks for merging and conflict resolution. Lock service is the simple lock service system that we discussed in Chapter 2.

We conducted a performance evaluation of multiple systems listed in Table 11.1. Our experiments were conducted on Azure, with each system deployed across Ubuntu 20.04 `Standard_B8ms` VMs, utilizing default Azure Cloud routing. To ensure reliability, we made our best effort to fully reinitialize the server state between measurements, and each benchmarking scenario was repeated 5 times. Each scenario involved tens of thousands of operations and ran for an average duration of 10 minutes. To monitor network performance, we examined network interface metrics to ensure that no network connections were saturated. For reporting our results, we calculated the medians of the trials and represented the 10th and 90th percentiles using whiskers on bar graphs.

**Table 11.1:** Systems we developed using PGo. Our evaluation focuses on the bolded systems: (1) *RaftStoreMod*, which is a modular composition of *RaftProto* and *DistKV* (see Chapter 9), (2) monolithic *RaftStore*, (3) *PBStore*, and (4) *PGoCRDT*.

| System | Effort (person days) | Properties model checked | Checked # states | Checking time (m) | Archetype Count | MPCal SLOC | Glue Go SLOC |
|---|---|---|---|---|---|---|---|
| RaftProto | 22 | Five Raft properties [59] | $2.7 \times 10^9$ | 312 | 9 | 771 | 676 |
| DistKV | 3 | Client interaction, consistency | $2.6 \times 10^7$ | 4 | 3 | 256 | 383 |
| **RaftStoreMod** | 25 | - | - | - | - | - | 1059 |
| **RaftStore** | 25 | Client interaction plus Raft | $3.1 \times 10^9$ | 404 | 7 | 758 | 1099 |
| Lock service | 2 | Mutual exclusion and liveness | $4.6 \times 10^7$ | 73 | 2 | 67 | 87 |
| **PBStore** | 10 | Strong consistency | $4.5 \times 10^7$ | 235 | 4 | 420 | 270 |
| **PGoCRDT** | 10 | Convergence and termination | $5.8 \times 10^6$ | 3954 | 2 | 160 | 185 |

## 11.2 Development Effort

All MPCal specifications were written primarily by me, with some help and contributions from two other students. Table 11.1 lists their effort in person days. The most complex systems we have developed are RaftStore and RaftStoreMod. Table 11.1 also lists the number of archetypes and SLOC in each MPCal spec, and SLOC for the Go code we hand-wrote to bootstrap the generated Go implementation of each system.

Table 11.1 presents a comparison of the efforts required to build different systems. RaftStore was developed in less than one person-month, whereas the construction of a similar system in Ivy [21] demanded 3 person-months, Verdi [71] took 12 person-months, and IronFleet [32] required 18 person-months. These results are promising; however, it is important to acknowledge that all these figures, including ours, are based on anecdotal and self-reported data. Moreover, the efforts were undertaken by researchers who may not represent the average software developer. Future studies should focus on user-based evaluations to assess the usability of tools within this domain.

During the process of constructing these systems, we observed a recurring pattern of reusing mapping macros across different systems that share identical or similar assumptions regarding failures or the environment. Additionally, we have created and reused various implementations of commonly used resources, such as the network and the file system.

## 11.3 Model Checking Performance

Table 11.1 presents the properties we defined and verified, the number of states explored by the TLC model checker, and the time taken for TLC checking. Our experiments were conducted on a machine equipped with 64 CPU cores and 128GB of RAM.

TLC is an exhaustive explicit state model checker. It comprehensively explores the entire reachable state space provided to it before terminating. This approach offers more robust guarantees compared to heuristic state space sampling since it ensures the consideration of all reachable states.

However, due to its bounded nature, TLC requires a finite state space for exploration. As a result, we needed to impose restrictions on each system's state space during the verification process.

For the model checking of RaftStore and RaftProto, our configuration included three servers, a single round of election, and at most two entries committed to the log with the possibility of one node failure. For DistKV, the model checking setting comprised five servers and three clients, where the servers could process up to three client requests. For the lock service, we used a model checking configuration with eight servers. In the PBStore model, the configuration consisted of five servers and four clients. Similarly, for PGoCRDT, the model checking configuration included five servers.

A previous study has demonstrated that reproducing all failures in a complex distributed system can be achieved with a setup comprising at most three nodes [79]. This highlights the effectiveness of our bounded model checking setting in capturing bugs. Furthermore, by utilizing more powerful machines, we could potentially apply even looser bounds and check more states, as shown in Table 11.1.

## 11.4 Performance of Raft-Based KV Stores

We conduct a comparative analysis of our Raft-based KV stores against several verified KV stores: Vard [72], a KV store verified in Verdi [71]; IronKV [33], a KV store verified in Dafny; and Ivy-Raft [21], a KV store verified in Ivy. All the KV stores are Raft-based, except IronKV that implements MultiPaxos. These implementations are extracted in OCaml, C#, and C++ accordingly. To interact with the underlying platform, each implementation employs custom shim code, communicating via plain TCP or UDP. While IronKV supports SSL, we disable it for consistency with its original evaluation. RaftStore and Vard incorporate disk-based durability, whereas IronKV and Ivy-Raft do not. To assess the impact of disk-based durability, we re-run selected benchmarks with this feature disabled in our artifact, observing minimal changes in throughput. We also include benchmark results for etcd v3.5.4 [18] as a baseline, a widely used Raft-based KV

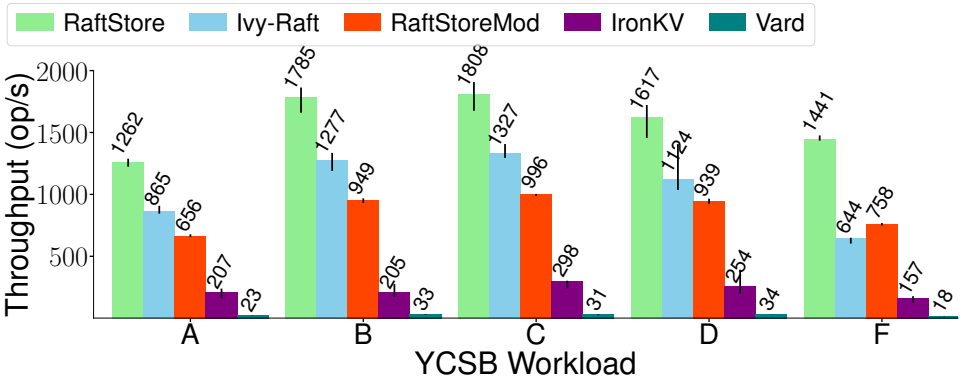



**Figure 11.1:** Throughput of RaftStore and RaftStoreMod as compared to various systems for a selection of standard YCSB workloads.

store implemented in Go.

We attempted to include Coyote [13] and StateRight [55] in our evaluation since they appeared to have Raft and Paxos prototypes, respectively. However, we discovered that these prototypes were not directly comparable with practical consensus implementations. The Raft prototype in Coyote was designed solely as a model checking target, as confirmed by the authors. Similarly, the authors of StateRight clarified that their Paxos prototype could only agree on a single value during an execution, necessitating a system reset process for a second value to be agreed upon.

We assess the performance of these KV stores using the Yahoo! Cloud Serving Benchmark (YCSB) suite [10], where we measure throughput and latency. Our evaluation comprises five YCSB workloads: (A) 50/50 read-/update Zipfian, (B) 95/5 read/update Zipfian, (C) read-only Zipfian, (D) 95/5 read/update latest (where the most recently inserted records are at the head of the Zipfian distribution), and (F) 50/50 read/read-modify-write (causally linked read/write) Zipfian. We exclude YCSB workload E as our systems do not support scans.

Figure 11.1 illustrates the performance comparison between RaftStore and RaftStoreMod, as well as other related work KV stores, across various YCSB workloads. All systems were deployed in 3-node clusters. To

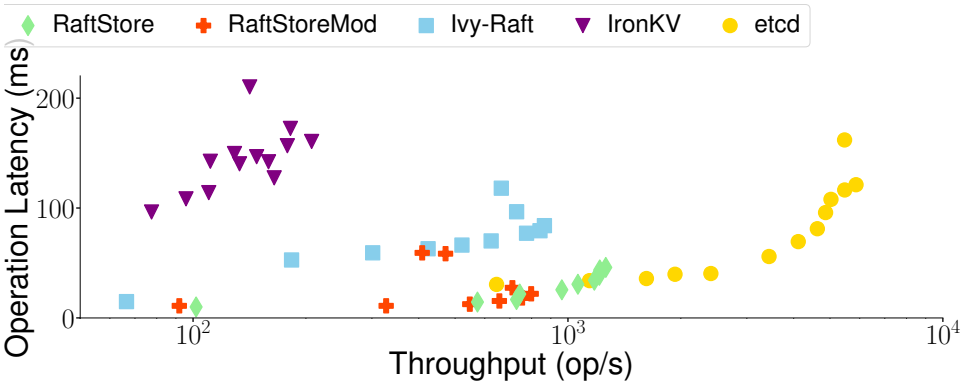



**Figure 11.2:** Latency-throughput data of Raft-based KV systems with varying number of concurrent clients.

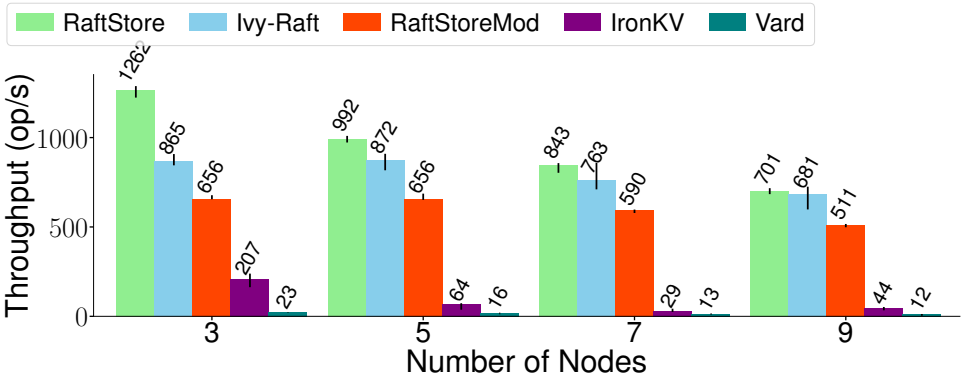



**Figure 11.3:** Scalability of Raft-based KV systems with varying cluster size.

obtain the peak possible throughput for each system and workload, we conducted repeated benchmarks with varying numbers of concurrent clients and recorded the highest achieved throughput.

Among all workloads, RaftStore had the highest throughput. It outperformed Ivy-Raft, the closest performing system, in terms of overall mean throughput by 41%.

This demonstrates the level of flexibility enabled by the model-implementation duality allows us to optimize I/O behavior and achieve an efficient multi-threaded implementation. Our optimization efforts encompass dividing the

MPCal model into multiple communicating processes, each dedicated to a specific task, thus enabling more concurrent processing compared to existing approaches. To achieve multi-threading, we had to modify and recompile the model, which we subsequently model checked to ensure its correctness. Additionally, we fine-tuned timeout values and the delay between log synchronization attempts between nodes to maximize runtime performance. These optimizations were performed on the Go side since our model does not account for physical time.

Furthermore, we note that RaftStoreMod has a lower performance compared to RaftStore, with a slightly lower maximum throughput than Ivy-Raft. This observation indicates that partitioning a system into two separate MPCal models might introduce some performance overhead. However, it is worth considering that the discrepancy in performance could also be attributed to our focus on fine-tuning the RaftStore implementation, which might have contributed to its superior performance.

All systems, including RaftStore, have considerably lower performance compared to the etcd baseline (not shown), with etcd achieving peak throughput ranging from 5,866 to 10,504 operations per second across all workloads. We attribute RaftStore's lower throughput to two main factors. First, etcd's architecture allows for significantly higher concurrency in processing client requests when compared to RaftStore. This difference arises due to a design distinction between etcd and the other evaluated Raft-based KV stores. Specifically, etcd implements a threaded extension of Raft [58], enabling greater concurrency than the core Raft specification. While RaftStore leverages more multi-threading than related systems, it remains rooted in the original Raft TLA$^+$ specification, without significant deviations from the core protocol specification. Second, RaftStore uses inherently less efficient immutable data structures within its compiled TLA$^+$ representation. Although these data structures offer asymptotically good performance, they are known to carry substantial overheads compared to mutable counterparts. Addressing these issues and approaching performance levels closer to production-grade tools like etcd is left for future research and improvements.

We analyze the relationship between latency and throughput for the

benchmarked systems in Figure 11.2. This analysis involves clusters of size 3, executing workload A, and presents the median throughput and client latency curves for different numbers of clients used in calculating maximum throughput (as shown in Figure 11.1). For clarity, Vard has been excluded from the plot; its maximum throughput was 31 op/s, and its minimum latency was 738ms. The comparison shows that, on the whole, RaftStore has 42% lower median latency than the lowest-latency related system, Ivy-Raft, while achieving latency similar to that of etcd. This improved latency in RaftStore can be attributed to our optimized multi-threaded implementation, which allow for internal data buffering and concurrent task execution when possible, rather than strictly adhering to the model's higher-level totally-ordered semantics.

Figure 11.3 demonstrates the scalability of each system concerning varying cluster sizes while utilizing workload A. Similar to Figure 11.1, we determined the number of concurrent clients that resulted in peak throughput for each system and cluster size. For consensus-based systems, it is expected that peak throughput will decrease as the cluster size increases due to the increased coordination work required. This effect is clearly shown in Figure 11.3.

Figure 11.4 depicts the fault tolerance capabilities of RaftStore during the execution of YCSB workload A with a cluster size of 5, displaying the throughput over time. The plot reveals a leader failure at approximately 22 seconds. Subsequently, after a timeout, the clients initiate the process of finding a new leader. Later, at around 41 seconds, we deliberately terminate a follower, which has minimal impact as the client communication with the leader remains unaltered.

## 11.5 Performance of Primary-Backup KV Stores

PBStore is a distributed key-value store that operates based on the primary-backup protocol. The PBStore architecture consists of a primary node responsible for synchronously replicating data to one or more backup nodes. We compared PBStore's performance with Redis, a widely-used key-value

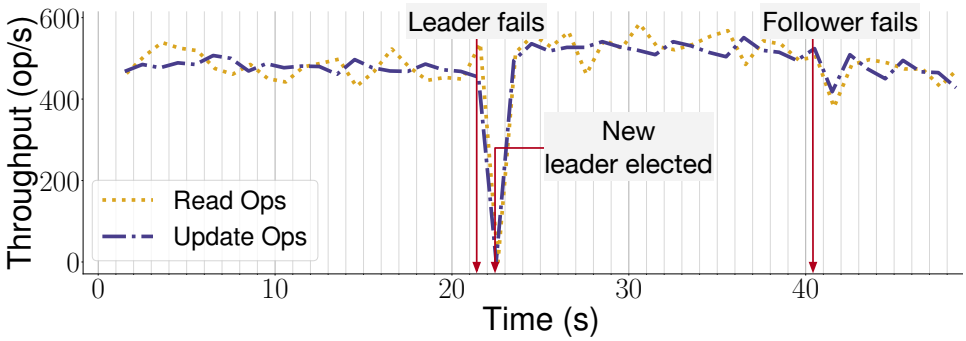



**Figure 11.4:** Throughput of RaftStore over time with three highlighted events: leader failure, new leader election, follower failure.

store written in C. We configured Redis's replication to utilize the primary-backup protocol in synchronous replication mode to closely match PBStore's behavior. Also, we attempted to evaluate Verdi's primary-backup system, but we could not proceed as we confirmed with the authors that they had not implemented the necessary runtime glue code for evaluation purposes, as it was solely used for verification purposes.

To perform the evaluation, we deployed both PBStore and Redis on three machines, with one serving as the primary and the other two as backups. We used the YCSB workload A to evaluate the systems' performance. The peak throughput of PBStore was found to be 340 op/s, while Redis demonstrated an impressive throughput of over 50,000 op/s.

The performance disparity between PBStore and Redis is attributed to a lack of protocol optimizations and tuning in PBStore. Specifically, PBStore lacks support for batching incoming requests for replication and sending replication requests to backups in parallel, which requires a more complex set of correctness properties and implementation semantics that we did not have sufficient time to implement and optimize.

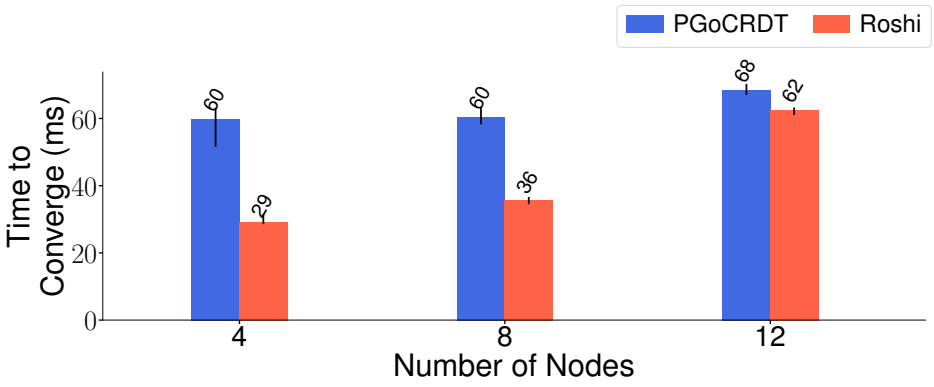



**Figure 11.5:** Convergence times for PGoCRDT and Roshi.

## 11.6 Performance of CRDT-based Systems

In scenarios where strict consistency is not required for shared state, commutative replicated data types (CRDTs) offer a reliable and efficient approach to represent such data. We evaluated our state-based CRDT set, PGoCRDT, and an open-source CRDT set developed by SoundCloud, called Roshi [5].

To compare these systems, we assessed the time it took for all nodes' states to converge to a consistent value, which we refer to as *convergence time*. During the experiment, each node participated in multiple rounds. In round $r$, a node $n$ added the pair $\langle r, n \rangle$ to its set and then waited until its set contained all pairs of the form $\langle r, i \rangle$, for every node $i$. For each round, we recorded the duration from when a node updated its local set until the specified condition was met. We repeated this process for a total of 100 rounds. It is important to note that both systems apply updates locally, and each node broadcasts its state every 50ms.

Figure 11.5 illustrates that Roshi outperforms PGoCRDT by up to 2× in terms of performance, but PGoCRDT scales more consistently. Although the performance gap is smaller compared to that between RaftStore and etcd, it is likely attributed to Roshi benefiting from more person-hours dedicated to tuning and optimization, as well as potential inefficiencies in the data structures utilized by PGo's compiled output.

# Chapter 12

# Related Work

## 12.1 Model-Checked DSLs

P [12, 14] presents a verifiable state machine model that shares similarities with MPCal, although it employs a lower-level language similar to C augmented with actor-like features. Mace [37, 38] introduces a model based on nested state machines, working as a DSL integrated with C++. Mace lacks some of the abstraction capabilities found in MPCal. StateRight [55] is a DSL designed for model checking, built in Rust. It represents distributed systems as state machines, resembling the approach taken by Mace. StateRight provides exhaustive model checking capabilities and leverages Rust's robust low-level safety assurances. Coyote [13], serves as an implementation model checker for unaltered C# code, optionally incorporating an actor-based DSL. ModP [15] is a programming system that enables compositional reasoning (assume-guarantee) of distributed systems. ModP does not have a model checker and employs systematic testing with weaker verification guarantees.

The duality problem also arises in this category of work. We believe MPCal provides more flexibility to developers, such that they can change the model to optimize the implementation while avoiding the state space explosion problem. Our evaluation shows we could build more performant systems while exploring a larger state space during model checking.

## 12.2 Automated Theorem-Proving

Techniques based on automated theorem proving provide a logical framework in which a system can be formally specified, formal proofs can be written about its properties, and an implementation can be automatically generated. Explicit proofs avoid the state space explosion inherent to model checking, and can provide strong guarantees about a given specification. These proofs can, however, be prohibitively difficult to write, requiring many person-years and measuring up to ten times the size of the system's specification [39]. Many efforts in this space aim to minimise proof effort for users, often by automating or generalising certain proof patterns into DSLs.

Verdi [71] and Adore [35] present Coq [69] libraries and support implementation extraction. Verdi primarily emphasizes relaxing assumptions through refinement, while Adore reduces proof effort through protocol abstraction. EventML[62] is designed for Nuprl rather than Coq and employs a logic based on causal ordering of events. PSync [17] facilitates semi-automated verification and operates under the assumption of a round-based program structure. Disel [65] presents a Coq DSL designed for creating and validating imperative specifications, utilizing a Hoare-style logic that offers straightforward composition of verified elements. Chapar [48], another Coq DSL, is specialized in specifying and verifying key-value stores and their corresponding clients. IronFleet [32] supplies tools that enable developers to demonstrate that practical implementations refine a high-level specification created in Dafny [47]. Ivy [60], DuoAI [77], DistAI [76], SWISS [30], and I4 [53] reduce the complexity associated with formulating inductive invariants for verification. It is important to note that Ivy serves as the verifier, while Ivy-Raft, a KV store [21], is a separate work by different authors. Sift [54] presents a methodology centered around proof decomposition, utilizing automated refinement. Armada [16] introduces a specification language similar to C for verified concurrent programs.

Building systems using automated theorem provers requires a significant effort for writing proofs [74], while implementation concerns tend to be a secondary concern. Reflecting implementation requirements on the system's

specification can be particularly difficult, as these adjustments can lead to extensive modifications in the proofs.

## 12.3 Model Checking Implementations

Prior research has extended the concept of exploring state spaces to actual system implementations [27, 46, 52, 56, 66, 75]. This pragmatic approach successfully addresses challenging specification problems [20]. However, this approach to model checking has inherent scalability limitations due to the heightened concurrency and larger state space inherent in system implementations compared to system models. The duality problem does not arise in these works since they do not have a model, and as a result, state space explosion becomes an essential problem.

## 12.4 Go Systems Tooling

Recent research has introduced tools for detecting and addressing concurrency problems in Go [49, 50], as well as verifying the correctness of Go code [7, 8, 73]. This body of work aligns with our framework, as it offers a complementary approach to enhance users' confidence in the Go code generated by PGo.

# Chapter 13

# Conclusion

In this work, we introduced the model-implementation duality and provided solutions to reconcile the modeling land and implementation land. Duality arises in different aspects and imposes trade-offs while building systems using PGo. We proposed a framework for building modular systems to manage the complexity and size of the trusted computing base. Our evaluation shows that in the presence of duality, we satisfied the requirements of both sides of the duality. We built several distributed systems and model-checked them with at least three nodes. Our systems perform at least 40% better than verified systems from related work and take at least 3× less time to construct.

Our experience with duality can offer valuable insights for shaping future efforts in building better frameworks for verified systems. Users can refer to our shared experience in this to address duality-related challenges. We believe that by addressing the duality challenge effectively, we can go towards having practical and verified distributed system implementations.

# Bibliography


[1] M. Abadi and L. Lamport. Conjoining specifications. *ACM Trans. Program. Lang. Syst.*, 17(3):507–535, may 1995. ISSN 0164-0925. doi:10.1145/203095.201069. URL https://doi.org/10.1145/203095.201069. → page 3

[2] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, Sep 1987. ISSN 1432-0452. → page 6

[3] A. Bieniusa, M. Zawirski, N. Preguiça, M. Shapiro, C. Baquero, V. Balegas, and S. Duarte. An optimized conflict-free replicated set. *arXiv preprint arXiv:1210.3368*, 2012. → page 64

[4] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005. ISBN 0321267974. → page 18

[5] P. Bourgon. Roshi: A CRDT system for timestamped events, May 2014. https://developers.soundcloud.com/blog/roshi-a-crdt-system-for-timestamped-events. → page 73

[6] C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011. → pages 31, 40

[7] T. Chajed, J. Tassarotti, F. M. Kaashoek, and N. Zeldovich. Verifying concurrent Go code in Coq with Goose. In *Proceedings of the International Workshop on Coq for Programming Languages (CoqPL)*, 2020. → page 76

[8] T. Chajed, J. Tassarotti, M. Theng, M. F. Kaashoek, and N. Zeldovich. Verifying the DaisyNFS concurrent and crash-safe file

system with sequential reasoning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022. → page 76

[9] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, Mar 1996. ISSN 0004-5411. → pages 34, 43

[10] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the ACM Symposium on Cloud computing (SoCC)*, 2010. → page 68

[11] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36 (4):372–421, dec 2004. ISSN 0360-0300. doi:10.1145/1041680.1041682. URL https://doi.org/10.1145/1041680.1041682. → page 34

[12] P. Deligiannis, A. F. Donaldson, J. Ketema, A. Lal, and P. Thomson. Asynchronous Programming, Analysis and Testing with State Machines. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2015. → page 74

[13] P. Deligiannis, N. Ganapathy, A. Lal, and S. Qadeer. Building Reliable Cloud Services Using Coyote Actors. In *Proceedings of the ACM Symposium on Cloud computing (SoCC)*, 2021. → pages 68, 74

[14] A. Desai, V. Gupta, E. Jackson, S. Qadeer, S. Rajamani, and D. Zufferey. P: Safe Asynchronous Event-driven Programming. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2013. → page 74

[15] A. Desai, A. Phanishayee, S. Qadeer, and S. A. Seshia. Compositional programming and testing of dynamic distributed systems. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018. doi:10.1145/3276529. URL https://doi.org/10.1145/3276529. → pages 3, 74

[16] A. F. Donaldson, E. Torlak, J. R. Lorch, Y. Chen, M. Kapritsos, B. Parno, S. Qadeer, U. Sharma, J. R. Wilcox, and X. Zhao. Armada: low-effort verification of high-performance concurrent programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2020. → page 75

[17] C. Drăgoi, T. A. Henzinger, and D. Zufferey. PSync: A Partially Synchronous Language for Fault-Tolerant Distributed Algorithms. In *Proceedings of the ACM on Programming Languages (POPL)*, 2016. → page 75

[18] etcd. etcd, 2021. https://etcd.io/. → page 67

[19] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2): 374–382, apr 1985. ISSN 0004-5411. doi:10.1145/3149.214121. URL https://doi.org/10.1145/3149.214121. → pages 28, 44

[20] P. Fonseca, K. Zhang, X. Wang, and A. Krishnamurthy. An Empirical Study on the Correctness of Formally Verified Distributed Systems. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2017. → page 76

[21] J. S. Foster, D. Grossman, M. Taube, G. Losa, K. L. McMillan, O. Padon, M. Sagiv, S. Shoham, J. R. Wilcox, and D. Woos. Modularity for decidability of deductive verification with applications to distributed systems. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2018. → pages 66, 67, 75

[22] S. Friedenthal, A. Moore, and R. Steiner. *A Practical Guide to SysML: Systems Modeling Language*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008. ISBN 9780080558363. → page 20

[23] GitHub. October 21 post-incident analysis, 2018. https://github.blog/2018-10-30-oct21-post-incident-analysis. → page 1

[24] GitLab. Postmortem of database outage of January 31, 2017. https://about.gitlab.com/2017/02/10/postmortem-of-database-outage-of-january-31/. → page 1

[25] Google. global: Elevated HTTP 500s errors for a small number of customers with load balancers on Traffic Director-managed backends, 2022. https://status.cloud.google.com/incidents/LuGcJVjNTeC5Sb9pSJ9o. → page 1

[26] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992. ISBN 1558601902. → page 34

[27] H. Guo, M. Wu, L. Zhou, G. Hu, J. Yang, and L. Zhang. Practical Software Model Checking via Dynamic Interface Reduction. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2011. → page 76

[28] F. Hackett, S. Hosseini, R. Costa, M. Do, and I. Beschastnikh. PGo, 2022. https://github.com/DistCompiler/pgo. → page 63

[29] F. Hackett, S. Hosseini, R. Costa, M. Do, and I. Beschastnikh. Compiling Distributed System Models with PGo. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 159–175, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399166. doi:10.1145/3575693.3575695. URL https://doi.org/10.1145/3575693.3575695. → pages v, 2

[30] T. Hance, M. Heule, R. Martins, and B. Parno. Finding Invariants of Distributed Systems: It's a Small (Enough) World After All. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2021. → page 75

[31] HashiCorp. Consul, 2023. https://www.consul.io/. → page 1

[32] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. IronFleet: Proving Safety and Liveness of Practical Distributed Systems. *Commun. ACM*, 60(7):83–92, 2017. ISSN 0001-0782. → pages 66, 75

[33] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. IronClad commit at which IronKV implementation was evaluated, 2022. https://github.com/microsoft/Ironclad/tree/bcb296737df6541c9542ad4e35499b347992f238. → page 67

[34] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, aug 1978. ISSN 0001-0782. doi:10.1145/359576.359585. URL https://doi.org/10.1145/359576.359585. → page 53

[35] W. Honoré, J.-Y. Shin, J. Kim, and Z. Shao. Adore: Atomic Distributed Objects with Certified Reconfiguration. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2022. → page 75

[36] D. Jackson. *Software Abstractions: logic, language, and analysis*. MIT press, 2012. → page 20

[37] C. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. Vahdat. Mace: Language Support for Building Distributed Systems. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007. → page 74

[38] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2007. → page 74

[39] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2009. → page 75

[40] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng.*, 3(2):125–143, Mar 1977. ISSN 0098-5589. → page 6

[41] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, jul 1978. ISSN 0001-0782. doi:10.1145/359545.359563. URL https://doi.org/10.1145/359545.359563. → page 24

[42] L. Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, May 1994. ISSN 0164-0925. → page 19

[43] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 032114306X. → pages 2, 7

[44] L. Lamport. The pluscal algorithm language. *Theoretical Aspects of Computing-ICTAC 2009, Martin Leucker and Carroll Morgan editors. Lecture Notes in Computer Science, number 5684, 36-60.*, Jan 2009. URL https://www.microsoft.com/en-us/research/publication/pluscal-algorithm-language/. → pages 2, 8

[45] L. Lamport. A PlusCal's User Manual. Online material, Aug 2018. → page 19

[46] T. Leesatapornwongsa, M. Hao, P. Joshi, J. F. Lukman, and H. S. Gunawi. SAMC: Semantic-aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014. → page 76

[47] R. Leino. Dafny: An automatic program verifier for functional correctness. In E. M. Clarke and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-17511-4. → page 75

[48] M. Lesani, C. J. Bell, and A. Chlipala. Chapar: certified causally consistent distributed key-value stores. *ACM SIGPLAN Notices*, 51 (1):357–370, 2016. ISSN 0362-1340. → page 75

[49] Z. Liu, S. Zhu, B. Qin, H. Chen, and L. Song. Automatically Detecting and Fixing Concurrency Bugs in Go Software Systems. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021. → page 76

[50] Z. Liu, S. Xia, Y. Liang, L. Song, and H. Hu. Who Goes First? Detecting Go Concurrency Bugs via Message Reordering. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022. → page 76

[51] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-Conscious Storage. *ACM Transactions on Storage (TOS)*, 13(1): 1–28, 2017. → page 38

[52] J. F. Lukman, H. Ke, C. A. Stuardo, R. O. Suminto, D. H. Kurniawan, D. Simon, S. Priambada, C. Tian, F. Ye, T. Leesatapornwongsa, A. Gupta, S. Lu, and H. S. Gunawi. FlyMC: Highly Scalable Testing of Complex Interleavings in Distributed Systems. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2019. → page 76

[53] H. Ma, A. Goel, J.-B. Jeannin, M. Kapritsos, B. Kasikci, and K. A. Sakallah. I4: Incremental Inference of Inductive Invariants for Verification of Distributed Protocols. In *Proceedings of the ACM*

*Symposium on Operating Systems Principles (SOSP)*, 2019. → page 75

[54] H. Ma, H. Ahmad, A. Goel, E. Goldweber, J.-B. Jeannin, M. Kapritsos, and B. Kasikci. Sift: Using Refinement-guided Automation to Verify Complex Distributed Systems. In *USENIX Annual Technical Conference (ATC)*, 2022. → page 75

[55] J. Nadal. Building Distributed Systems With Stateright. https://www.stateright.rs/. Accessed: 2022-10-27. → pages 68, 74

[56] L. Nelson, J. Bornholt, R. Gu, A. Baumann, E. Torlak, and X. Wang. Scaling symbolic evaluation for automated verification of systems code with Serval. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019. → page 76

[57] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. How Amazon Web Services Uses Formal Methods. *Commun. ACM*, 58(4):66–73, Mar 2015. → page 1

[58] D. Ongaro. *Consensus: Bridging theory and practice*. Stanford University, 2014. → pages 25, 61, 70

[59] D. Ongaro and J. Ousterhout. In Search of an Understandable Consensus Algorithm. In *USENIX Annual Technical Conference (ATC)*, 2014. → pages x, 3, 24, 25, 65

[60] O. Padon, K. L. McMillan, A. Panda, M. Sagiv, and S. Shoham. Ivy: Safety verification by interactive generalization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2016. → page 75

[61] A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the Annual Symposium on Foundations of Computer Science*, 1977. → page 7

[62] V. Rahli, D. Guaspari, M. Bickford, and R. L. Constable. Formal Specification, Verification, and Implementation of Fault-Tolerant Systems using EventML. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.*, 72, 2015. → page 75

[63] M. Rai Jain. Introducing Badger: A fast key-value store written purely in go, May 2017. https://dgraph.io/blog/post/badger/. → page 38

[64] Roblox. Roblox Return to Service 10/28-10/31 2021, 2022. https://blog.roblox.com/2022/01/roblox-return-to-service-10-28-10-31-2021/. → page 1

[65] I. Sergey, J. R. Wilcox, and Z. Tatlock. Programming and proving with distributed protocols, 2017. → page 75

[66] J. Simsa, R. Bryant, and G. Gibson. dBug: Systematic Evaluation of Distributed Systems. In *Proceedings of the 5th International Conference on Systems Software Verification (SSV)*, 2010. → page 76

[67] T. Strickx and J. Hartman. Cloudflare outage on June 21, 2022, 2022. https://blog.cloudflare.com/cloudflare-outage-on-june-21-2022/. → page 1

[68] G. D. Team. The Gallina specification language, 2023. https://coq.github.io/doc/v8.9/refman/language/gallina-specification-language.html. → page 20

[69] T. C. D. Team. The Coq Proof Assistant, version 8.9.0, 2019. https://web.archive.org/web/20190415015254/https://zenodo.org/record/2554024. → pages 20, 75

[70] M. Wenzel et al. The isabelle/isar reference manual, 2004. → page 20

[71] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. *ACM SIGPLAN Notices*, 50 (6):357–368, Jun 2015. ISSN 03621340. → pages 66, 67, 75

[72] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. verdi-raft commit at which Vard implementation was evaluated, 2022. https://github.com/uwplse/verdi-raft/tree/ea99a7453c30a0c11b904b36a3b4862fad28abe1. → page 67

[73] F. A. Wolf, L. Arquint, M. Clochard, W. Oortwijn, J. a. C. Pereira, and P. Müller. Gobra: Modular specification and verification of go programs. In *International Conference on Computer Aided Verification (CAV)*, 2021. → page 76

[74] D. Woos, J. R. Wilcox, S. Anton, Z. Tatlock, M. D. Ernst, and T. Anderson. Planning for change in a formal verification of the raft consensus protocol. In *Proceedings of the 5th ACM SIGPLAN*

*Conference on Certified Programs and Proofs*, CPP 2016, page 154–165, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341271. doi:10.1145/2854065.2854081. URL https://doi.org/10.1145/2854065.2854081. → page 75

[75] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2009. → page 76

[76] J. Yao, R. Tao, R. Gu, J. Nieh, S. Jana, and G. Ryan. DistAI: Data-Driven automated invariant learning for distributed protocols. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021. → page 75

[77] J. Yao, R. Tao, R. Gu, and J. Nieh. DuoAI: Fast, Automated Inference of Inductive Invariants for Verifying Distributed Protocols. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022. → page 75

[78] Y. Yu, P. Manolios, and L. Lamport. Model checking tla+ specifications. In L. Pierre and T. Kropf, editors, *Correct Hardware Design and Verification Methods*, pages 54–66, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN 978-3-540-48153-9. → page 7

[79] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 249–265, USA, 2014. USENIX Association. ISBN 9781931971164. → page 67

# Appendix A

# Supporting Materials

We include parts of the RaftStore MPCal model here.

```
\* A perfect failure detector. Can be replaced with other failure
\* detectors with different guarantees.
mapping macro PerfectFailureDetector {
  read {
    yield $variable;
  }

  write {
    yield $value;
  }
}

\* Persistent log mapping macro. It provides efficient add and
\* remove operations.
mapping macro PersistentLog {
  read {
    yield $variable;
  }

  write {
    if ($value.cmd = LogConcat) {
      yield $variable \o $value.entries;
    } else if ($value.cmd = LogPop) {
      yield SubSeq($variable, 1, Len($variable) - $value.cnt);
```

```
25      };
26    }
27  }
28
29  \* A helper for handling failure when sending a network message.
30  macro Send(net, dest, fd, m) {
31    either {
32      net[dest] := m;
33    } or {
34      await fd[dest];
35    };
36  }
37
38  archetype AServerHandler(...) {
39  serverLoop:
40    while (TRUE) {
41      m := net[srvId];
42    handleMsg:
43      if (m.mtype = RequestVoteRequest) {
44        UpdateTerm(self, m, currentTerm, state, votedFor, leader);
45
46        \* HandleRequestVoteRequest
47        with (
48          i = srvId, j = m.msource,
49          logOK = \/ m.mlastLogTerm > LastTerm(log[i])
50                  \/ /\ m.mlastLogTerm = LastTerm(log[i])
51                     /\ m.mlastLogIndex >= Len(log[i]),
52          grant = /\ m.mterm = currentTerm[i]
53                  /\ logOK
54                  /\ votedFor[self] \in {Nil, j}
55        ) {
56          assert m.mterm <= currentTerm[i];
57          if (grant) {
58            votedFor[i] := j;
59          };
60          Send(net, j, fd, [
61            mtype        |-> RequestVoteResponse,
62            mterm        |-> currentTerm[i],
63            mvoteGranted |-> grant,
64            msource      |-> i,
```

```
65            mdest        |-> j
66          ]);
67        };
68      } else if (m.mtype = RequestVoteResponse) {
69        \* HandleRequestVoteResponse
70      } else if (m.mtype = AppendEntriesRequest) {
71        \* HandleAppendEntriesRequest
72      } else if (m.mtype = AppendEntriesResponse) {
73        \* HandleAppendEntriesResponse
74      } else if (
75        \/ m.mtype = ClientPutRequest
76        \/ m.mtype = ClientGetRequest
77      ) {
78        \* HandleClientRequest
79      };
80    };
81  }
82
83  archetype AServerAppendEntries(...) {
84  serverAppendEntriesLoop:
85    while (appendEntriesCh[srvId]) {
86      await state[srvId] = Leader;
87      idx := 1;
88    appendEntriesLoop:
89      \* AppendEntries
90      while (
91        /\ state[srvId] = Leader
92        /\ idx <= NumServers
93      ) {
94        if (idx /= srvId) {
95          with (
96            prevLogIndex = nextIndex[srvId][idx] - 1,
97            prevLogTerm  = IF prevLogIndex > 0
98                           THEN log[srvId][prevLogIndex].term
99                           ELSE 0,
100           entries      = SubSeq(log[srvId], nextIndex[srvId][idx],
101                                 Len(log[srvId]))
102         ) {
103           \* Leader server sending new entries to follower servers
104           Send(net, idx, fd, [
```

```
105             mtype         |-> AppendEntriesRequest,
106             mterm         |-> currentTerm[srvId],
107             mprevLogIndex |-> prevLogIndex,
108             mprevLogTerm  |-> prevLogTerm,
109             mentries      |-> entries,
110             mcommitIndex  |-> commitIndex[srvId],
111             msource       |-> srvId,
112             mdest         |-> idx
113           ]);
114         };
115       };
116       idx := idx + 1;
117     };
118   };
119 }
120
121 archetype AClient(...) {
122 clientLoop:
123   while (TRUE) {
124     req := reqCh;
125   sndReq:
126     if (leader = Nil) {
127       with (srv \in ServerSet) {
128         leader := srv;
129       };
130     };
131     Send(net, leader, fd, [
132       mtype   |-> ClientPutRequest,
133       mcmd    |-> [
134         idx   |-> reqIdx,
135         type  |-> Put,
136         key   |-> req.key,
137         value |-> req.value
138       ],
139       msource |-> self,
140       mdest   |-> leader
141     ]);
142   rcvResp:
143     either {
144       resp := net[self];
```

```
145         leader := resp.mleaderHint;
146         if (\lnot resp.msuccess) {
147           goto sndReq;
148         } else {
149           respCh := resp;
150         };
151       } or {
152         await \/ /\ fd[leader]
153                  /\ netLen[self] = 0 \* no unread message
154                \/ timeout;            \* timeout injection
155         leader := Nil;
156         goto sndReq;
157       };
158     };
159   }
```