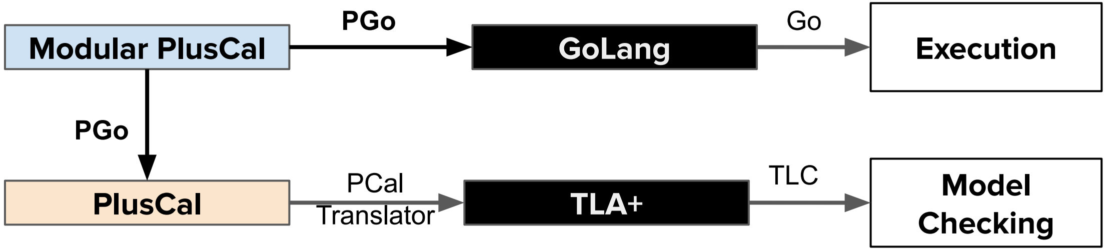
Compiling Distributed System Models into Implementations with PGo

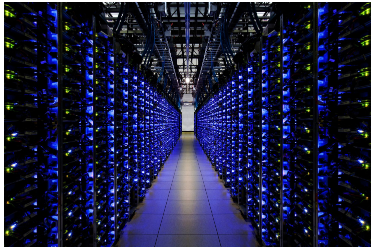
Finn Hackett, Shayan Hosseini,
Ivan Beschastnikh
Ruchit Palrecha, Yennis Ye,
Renato Costa, Matthew Do



https://github.com/DistCompiler/pgo

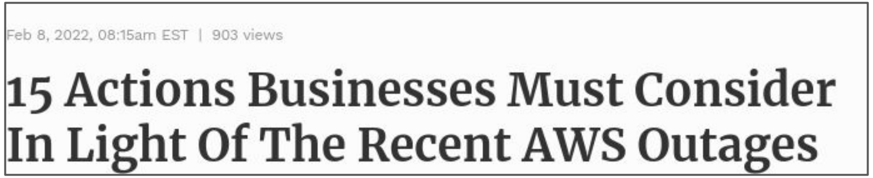
Motivation

- Distributed systems are widely deployed
- Despite this fact, writing correct distributed systems is **hard**
  - Asynchronous network
  - Crashes
  - Network delays, partial failures…
- Systems deployed in production **often have bugs**



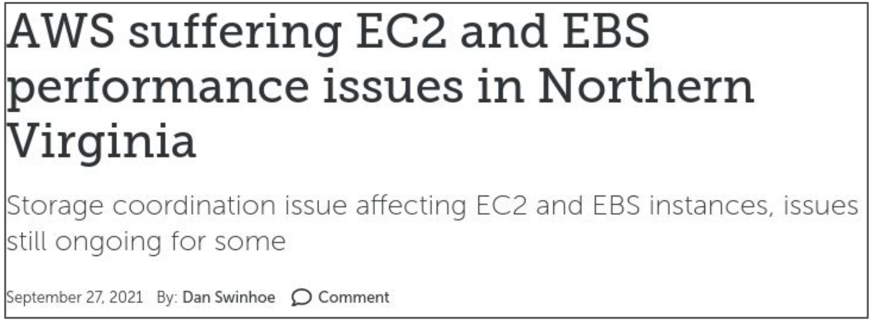
Google data center, Douglas County, Georgia

Bugs in Distributed Systems



Feb 8, 2022, 08:15am EST | 903 views

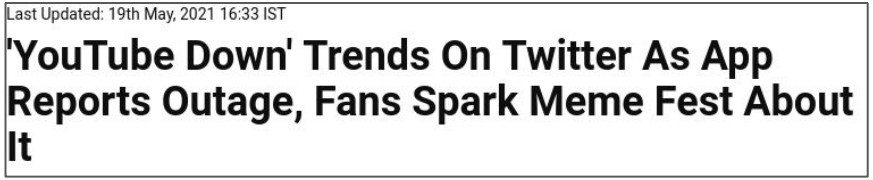
15 Actions Businesses Must Consider In Light Of The Recent AWS Outages



AWS suffering EC2 and EBS performance issues in Northern Virginia

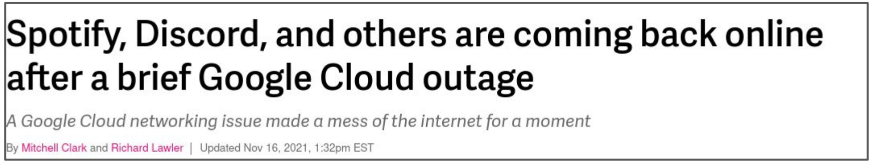
Storage coordination issue affecting EC2 and EBS instances, issues still ongoing for some

September 27, 2021 By: Dan Swinhoe Comment



Last Updated: 19th May, 2021 16:33 IST

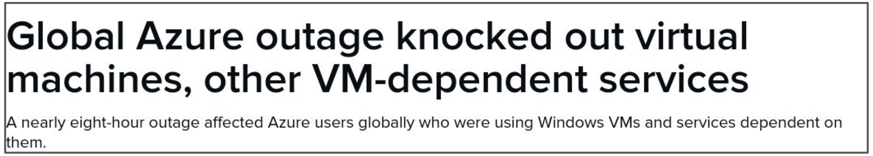
'YouTube Down' Trends On Twitter As App Reports Outage, Fans Spark Meme Fest About It



Spotify, Discord, and others are coming back online after a brief Google Cloud outage

A Google Cloud networking issue made a mess of the internet for a moment

By Mitchell Clark and Richard Lawler | Updated Nov 16, 2021, 1:32pm EST



Global Azure outage knocked out virtual machines, other VM-dependent services

A nearly eight-hour outage affected Azure users globally who were using Windows VMs and services dependent on them.

[1] Mark Cavage. There's Just No Getting around It: You're Building a Distributed System. Queue 11, 4, Pages 30, April 2013
[2] Mitchell Clark, Richard Lawler. Spotify, Discord, and others are coming back online after a brief Google Cloud outage. The Verge, Nov. 2021
[3] Greeshma Nayak. 'YouTube Down' Trends On Twitter As App Reports Outage, Fans Spark Meme Fest About It. RepublicWorld, May 2021
[4] Mary Jo foley. Global Azure outage knocked out virtual machines, other VM-dependent services. ZDNet, October 2021
[5] Dan Swinhoe. AWS suffering EC2 and EBS performance issues in Northern Virginia. Data Centre Dynamics Ltd, September 2021
[6] Forbes Technology Council. 15 Actions Businesses Must Consider In Light Of The Recent AWS Outages. Forbes, February 2022

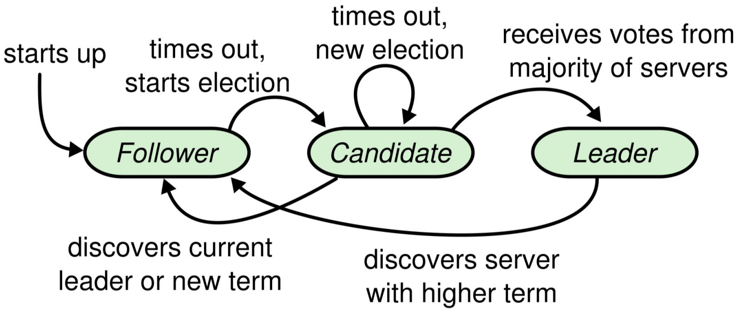
Protocol Descriptions Are **Not** Enough

- ➔ Distributed protocols typically have **edge cases**
  - ◆ Many of which may lack a **precise definition** of expected behavior
- ➔ Difficult to **correspond** final implementation with high-level protocol description
- ➔ Production implementations resort to **ad-hoc error handling** [1, 2, 3, 4]

[1] Ding Yuan et al. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems. OSDI 14
[2] Tanakorn Leesatapornwongsa at al. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. ASPLOS 16
[3] Jie Lu et al. CrashTuner: detecting crash-recovery bugs in cloud systems via meta-info analysis. SOSP 19
[4] Yu Gao et al. An empirical study on crash recovery bugs in large-scale distributed systems. FSE 2018

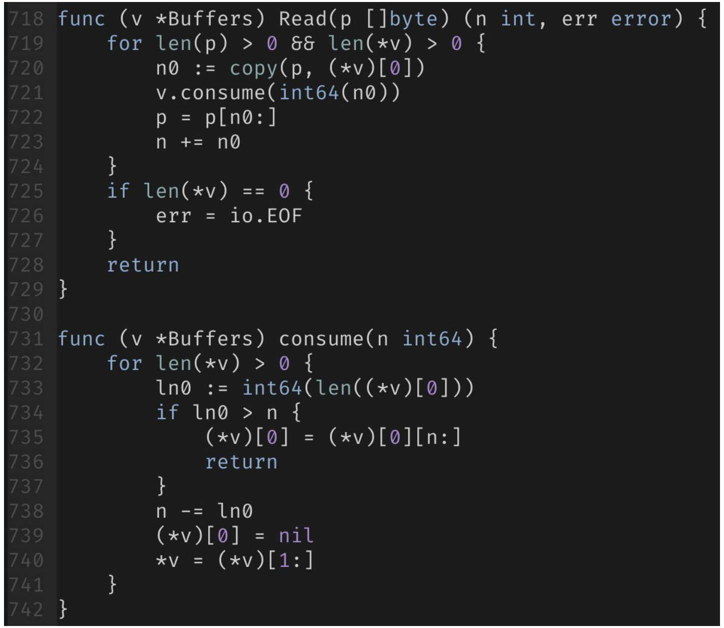
Key problem: Gap between design and implementation

Design


starts up
times out, starts election
times out, new election
receives votes from majority of servers
Follower
Candidate
Leader
discovers current leader or new term
discovers server with higher term

gap

Implementation



```
func (v *Buffers) Read(p []byte) (n int, err error) {
    for len(p) > 0 && len(*v) > 0 {
        n0 := copy(p, (*v)[0])
        v.consume(int64(n0))
        p = p[n0:]
        n += n0
    }
    if len(*v) == 0 {
        err = io.EOF
    }
    return
}

func (v *Buffers) consume(n int64) {
    for len(*v) > 0 {
        ln0 := int64(len((*v)[0]))
        if ln0 > n {
            (*v)[0] = (*v)[0][n:]
            return
        }
        n -= ln0
        (*v)[0] = nil
        *v = (*v)[1:]
    }
}
```

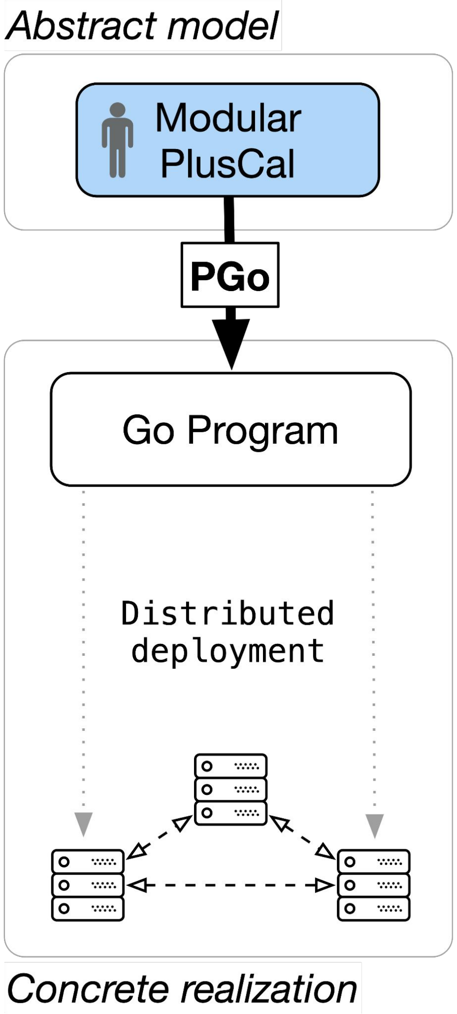
[1] Raft protocol server states; Diego Ongaro el al. In search of an understandable consensus algorithm. Usenix ATC 14

PGo: a Tool for Spec2Code

- ➔ We should automate implementation generation
- ➔ PGo generates Go code from MPCal
- ➔ MPCal is a superset / cousin of PlusCal
- ➔ Things PGo might help you with:
  - ◆ Prototyping something that runs from your TLA+/PlusCal
  - ◆ Code generation for core protocol logic
  - ◆ Having a specific relationship between spec and implementation; an opportunity for tracing and other instrumentation

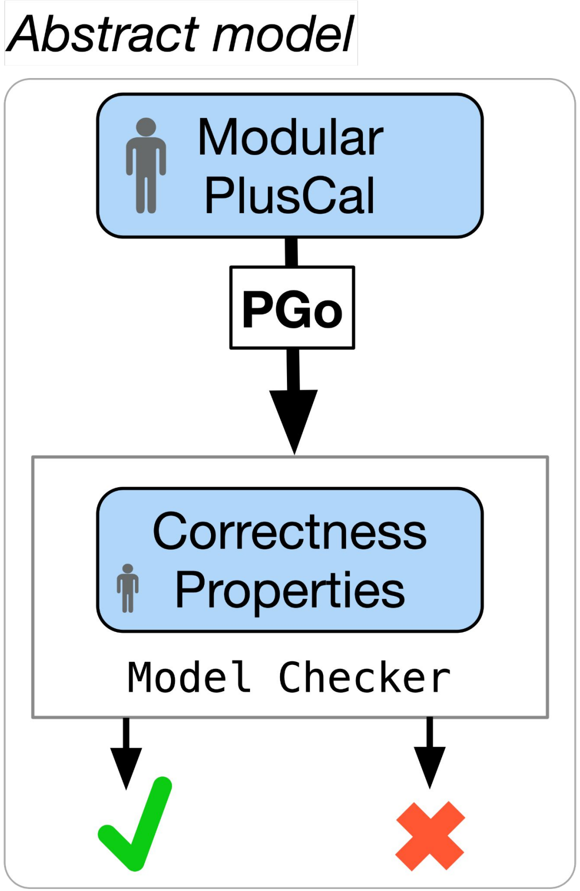
PGo: Generating an implementation

- PGo is a **compiler** from **models** in Modular PlusCal (MPCal) to **implementations** in Go
- Capable of generating **concurrent** and **distributed** systems from MPCal specifications

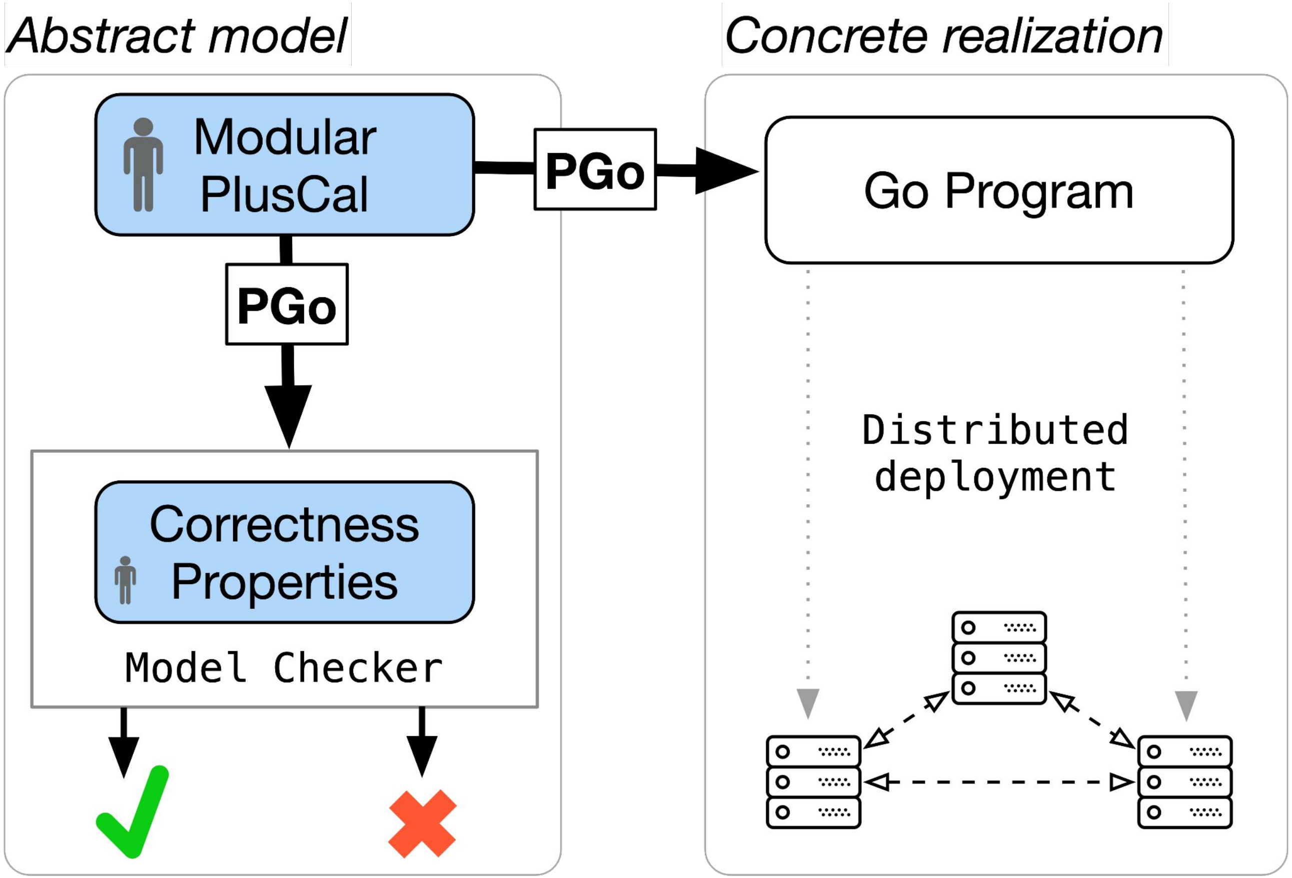


PGo: Model checking

- ➔ Modular PlusCal models can be model checked
- ➔ Users define their desired properties for the model
- ➔ Properties can be checked with the TLC [1] or Apalache [2] model checkers, or the TLAPS proof assistant [3]



[1] Lamport, L. TLA+ Tools. https://lamport.azurewebsites.net/tla/tools.html
[2] Igor Konnov et al. TLA+ model checking made symbolic. OOPSLA 19
[3] TLA+ Proof System. https://tla.msr-inria.inria.fr/tlaps/content/Home.html



Abstract model

Modular PlusCal

PGo

Correctness Properties

Model Checker

✓

✗

Concrete realization

PGo

Go Program

Distributed deployment

… déjà vu?

- ➔ We were here in 2019, also talking about PGo. What gives?

- ➔ In 2019, our example was a producer-consumer toy
- ➔ We rewrote PGo in Scala, with a -20k change in LOC
- ➔ PGo's improved in several ways:
    - ◆ More and bigger systems (e.g Raft, CRDTs, failure recovery…)
    - ◆ Better performance
    - ◆ Modular verification (connecting multiple specs)
    - ◆ Implementation tracing (runtime analysis of generated code)

Outline

Problem description and motivation

➢ **PGo recap**

Raft Implementation

Performance improvements and challenges

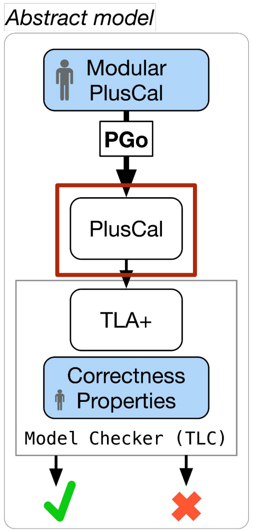
Modular verification

Implementation tracing

Conclusion

PlusCal overview

- ➔ An algorithm description language that can be compiled to TLA+.
- ➔ PlusCal makes it easier to specify systems in a procedural style.



Abstract model

Modular PlusCal

PGo

PlusCal

TLA+

Correctness Properties

Model Checker (TLC)

✓ ✗

Process definition

```
process (p ∈ Procs) {
transfer:
    if (aliceSavings >= amount) {
        aliceSavings := aliceSavings - amount;
        bobSavings := bobSavings + amount;
    };
}
```

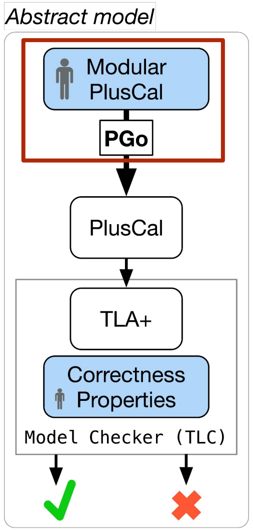
PlusCal

`transfer` is a **label**. PlusCal labels are translated to TLA+ transitions.

Our language: Modular PlusCal (MPCal)

- *Goal: automatically compile models into implementations.*

- Automatic translation between TLA+ or PlusCal models and implementations is impractical.
- Our approach: a new language on top of PlusCal



Problem: How to implement PlusCal code?

PlusCal

```
variables network = <<>>;
...
readMessage: \* blocking read from the network
    await Len(network[self]) > 0;
    msg := Head(network[self]);
    network := [network EXCEPT ![self] = Tail(network[self])];
```

This algorithm is **not abstract enough**

Almost all this code is for the **model checker**

```
// blocking read from the network
_, err = network.Read(netRead)
if err != nil {
    return err
}
msg := netRead
```

Go

We **model** a network read, but this implementation **does not do that**

Invent a new kind of macro: *archetype*

```
archetype AServer(ref network[_], ...)
...
readMessage:
    msg := network[self];
```

MPCal

Archetypes are **parameterised** by an **abstraction** over the environment.

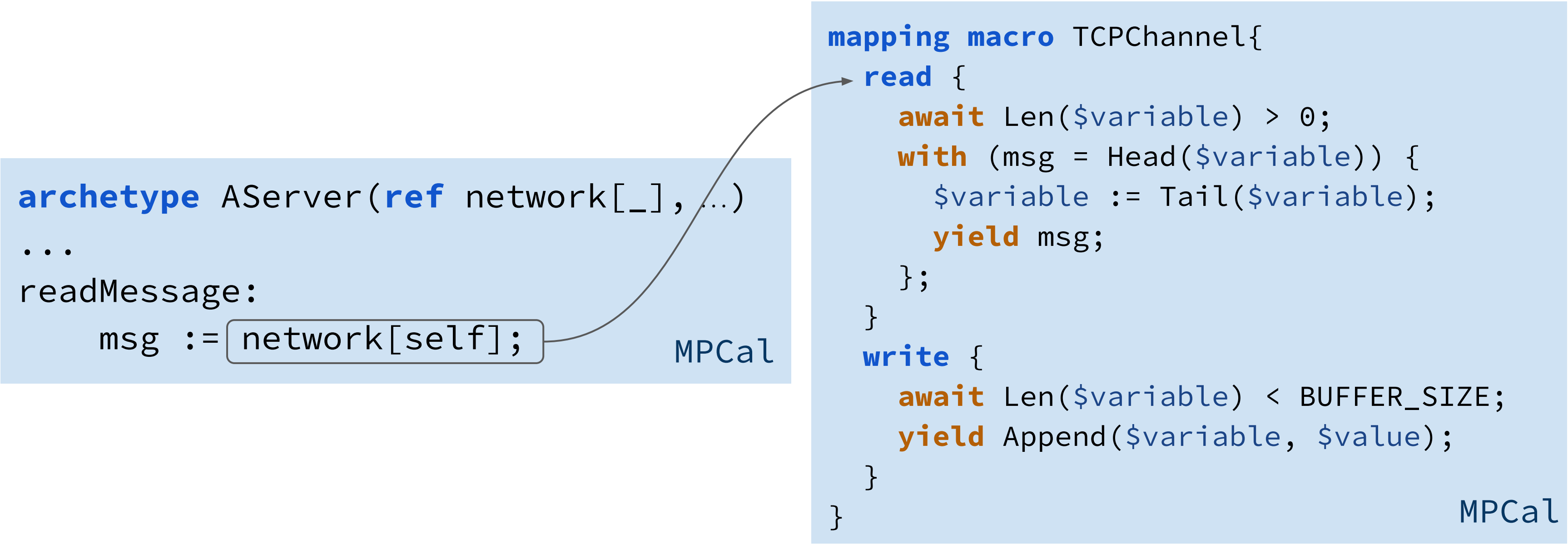
Complex network semantics can become a **variable read** or **write**

Any number of model checker and environment behaviors should be defined elsewhere, because archetypes only contain the system definition.

```
netRead, err := Read(network, self)
if err != nil { ... }
msg := netRead
```

Go

Invent a new kind of macro: *mapping macro*



```
archetype AServer(ref network[_], ...)
...
readMessage:
    msg := network[self];
```

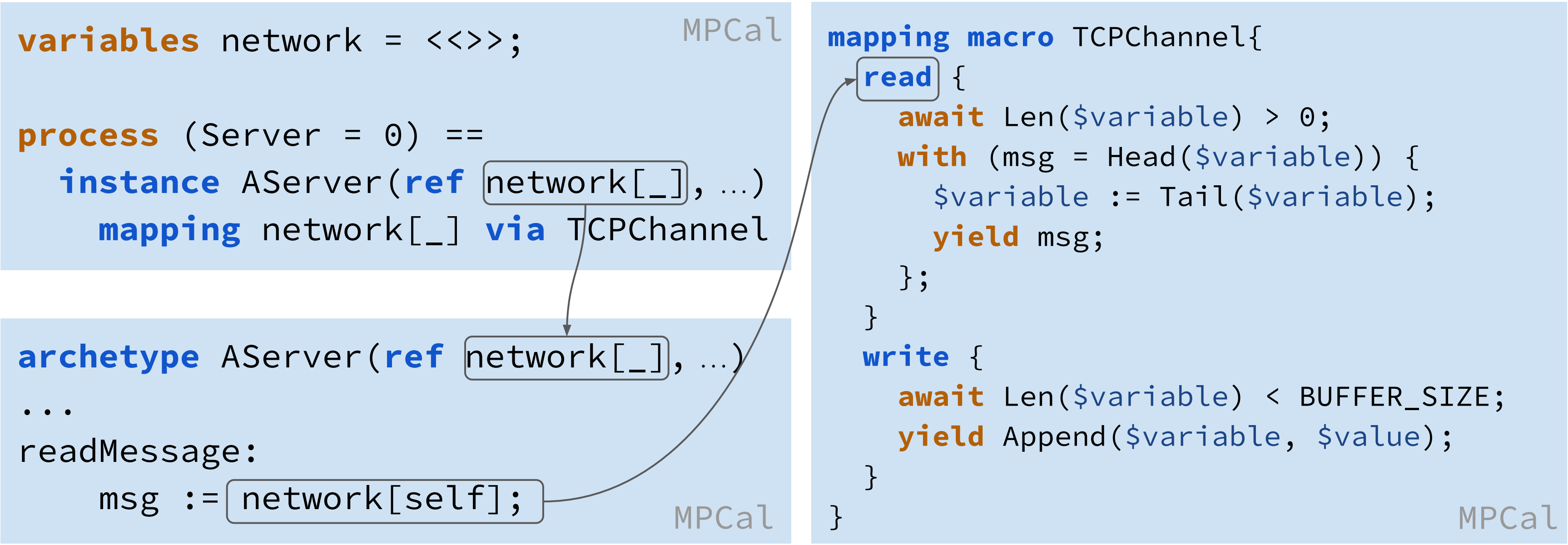
MPCal

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

MPCal

Modular PlusCal Language overview

- ➔ **Archetypes**: only contain the **system definition**
- ➔ **Mapping Macros**: define behavior of the **environment**
- ➔ **Instances**: configures abstract environment for model checking



```
variables network = <<>>;

process (Server = 0) ==
  instance AServer(ref network[_], ...)
    mapping network[_] via TCPChannel
```

```
archetype AServer(ref network[_], ...)
...
readMessage:
    msg := network[self];
```

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

Linking Abstractions and Concrete Implementations

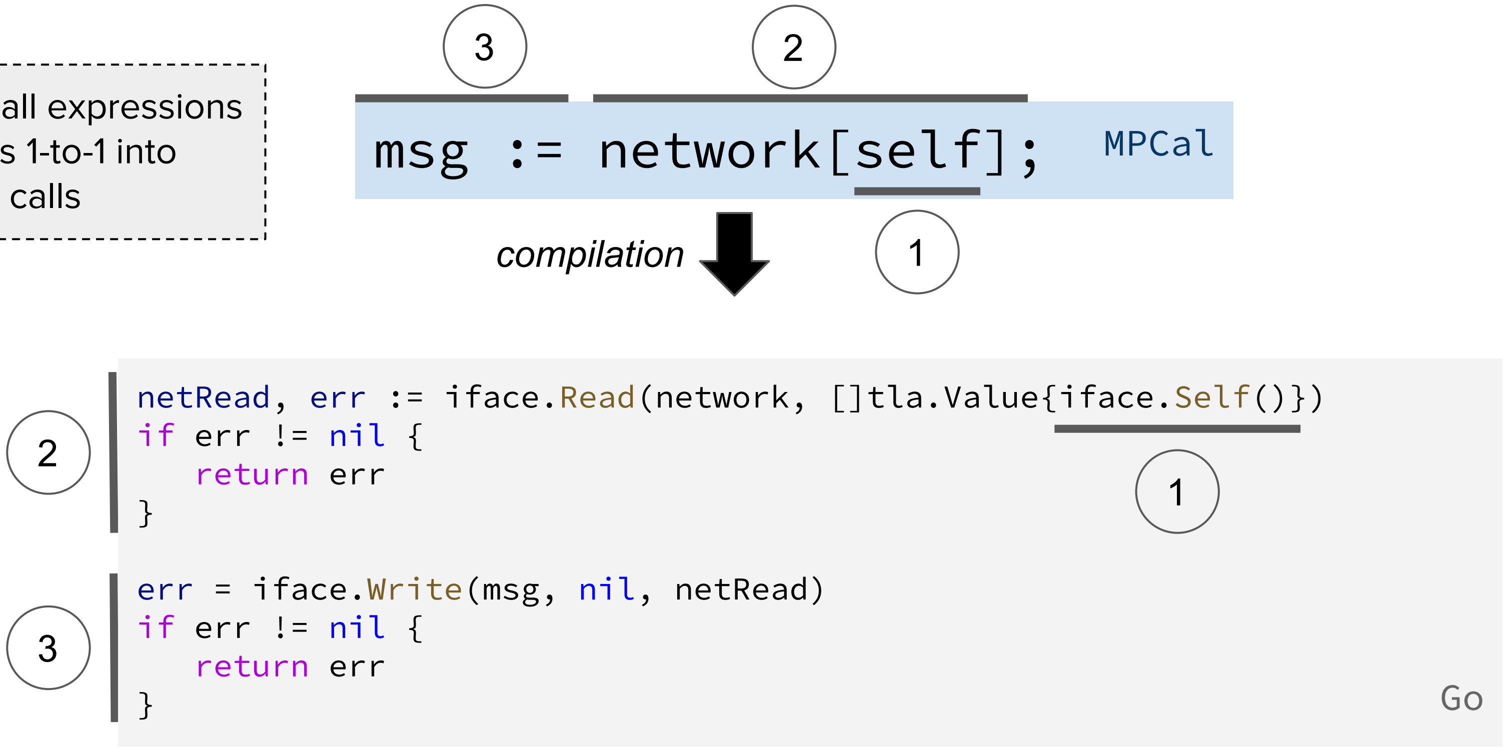
- ➔ PGo is not aware of the **concrete representation** of abstract resources passed to archetypes
- ➔ Instead, we define a **contract** that valid implementations must follow
  - ◆ Should support **diverse implementations**
  - ◆ Should allow exploration of **non-deterministic program flow**
  - ◆ We represent this contract as a **Go API** and a **state machine**
  - ◆ Ideally have simple, **bug-averse compilation process**

Defining our Objective

- **Goal**: every execution of the resulting system can be mapped to an accepted **behavior** of the spec (**refinement**)
- Environment modeled **abstractly** in Modular PlusCal needs an **implementation** in Go with **matching semantics**
- Need to understand how to do this **safely**

One-to-one Compilation of MPCal Code

PGo compiles all expressions and statements 1-to-1 into runtime library calls



```
msg := network[self];
```

MPCal

```
netRead, err := iface.Read(network, []tla.Value{iface.Self()})
if err != nil {
   return err
}

err = iface.Write(msg, nil, netRead)
if err != nil {
   return err
}
```

Go

An MPCal Server Example

```
archetype AClient(ref network[_],
                  ref paths, ref out)
{
  mkRequest:
    with(path = paths) {
      network[SERVER_ID] := [
        client_id |-> self,
        path |-> path
      ];
    };

  rcvResponse:
    out := network[self];
    goto mkRequest;
}
```

MPCal

```
archetype AServer(ref network[_],
                  ref file_system[_])
variable msg;
{
  readMessage:
    msg := network[self];

  sendPage:
    network[msg.client_id] :=
                      file_system[msg.path];
    goto readMessage;
}
```

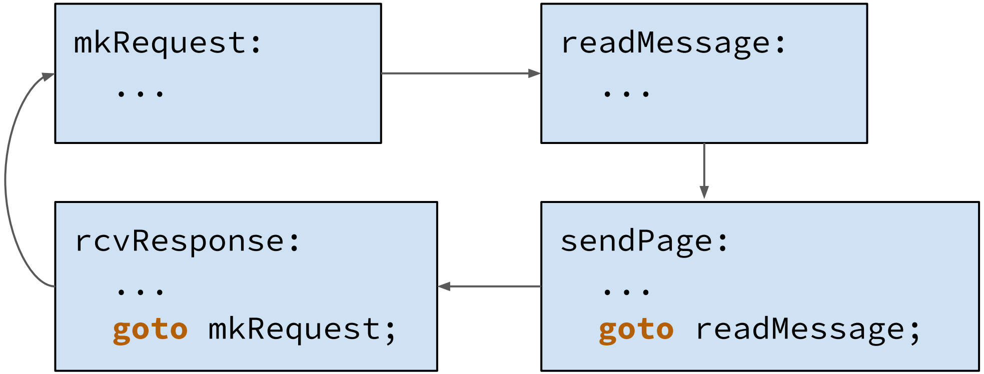
MPCal

Labels Define Atomic Steps

- ➔ Either **all of a step** is taken, or **none of it**
- ➔ Most programming languages do not work like this
- ➔ Many I/O interactions do not work like this

More complex models feature non-deterministic branching:

```
step:
  either { (* option A ... *) }
  or { (* option B ... *) };
  ...
```



```
mkRequest:
  ...
```

```
readMessage:
  ...
```

```
sendPage:
  ...
  goto readMessage;
```

```
rcvResponse:
  ...
  goto mkRequest;
```

These concepts form our **primary implementation challenge**

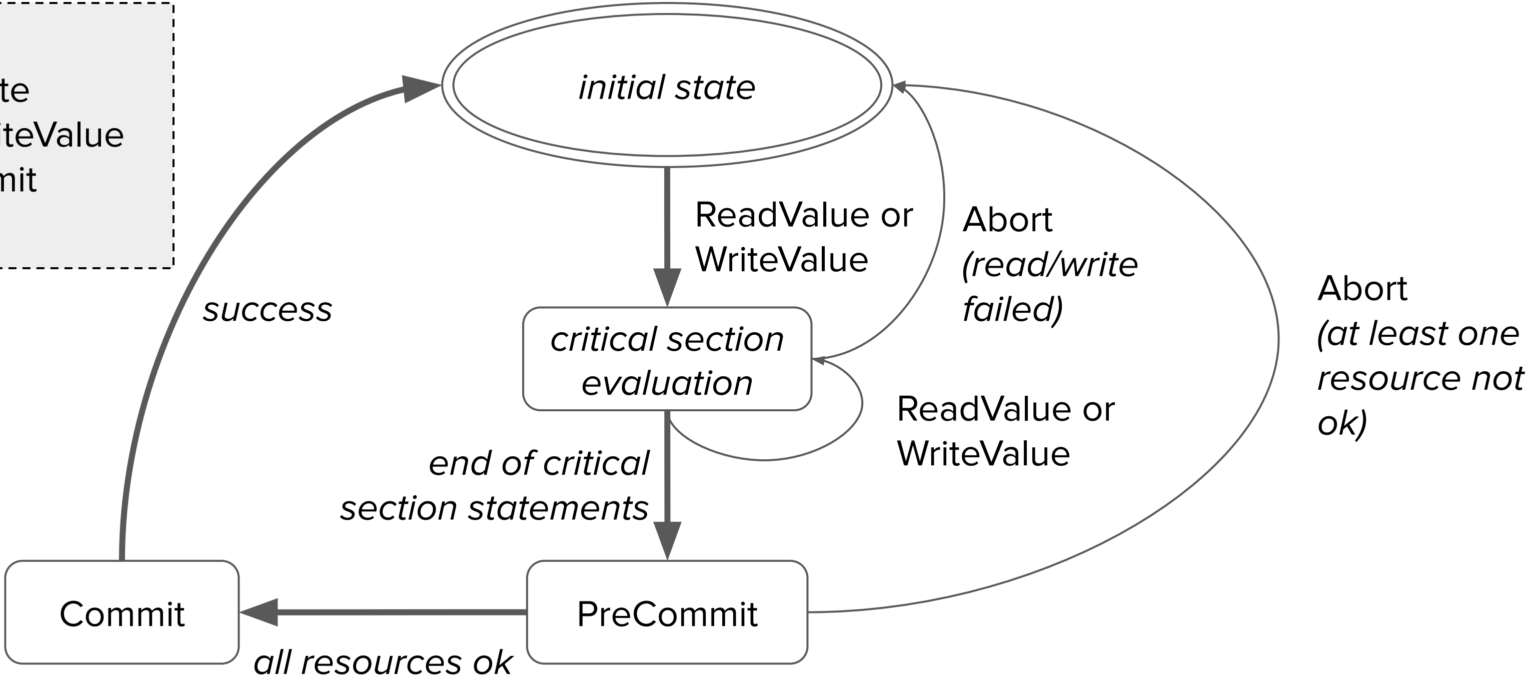
Executing an Atomic Step in Go

- Compiled **archetypes** perform a **local consensus step** between resource implementations
  - Steps in an archetype may be executed **concurrently** with steps from other archetypes, *as long as resource implementations consider it safe*
- Overview of the execution model of a **single step (2PC like)**:
  - Execute all **statements** in order, **exploring non-determinism** (may spuriously abort and restart atomic action)
  - **Pre-commit** changes to all resources used, seeking consensus
  - If all resources allow, **commit**, otherwise **abort and retry** step

Critical Section State Machine

Happy path

1. Initial state
2. Read/WriteValue
3. PreCommit
4. Commit


initial state
ReadValue or WriteValue
Abort (read/write failed)
Abort (at least one resource not ok)
success
critical section evaluation
ReadValue or WriteValue
end of critical section statements
Commit
PreCommit
all resources ok

Outline

Problem description and motivation

PGo recap

➢ **Raft Implementation**

Performance improvements and challenges

Modular verification

Implementation tracing

Conclusion

A partial list of specs that we wrote

- Raft, and Raft-based Key-Value Store
  - Based on a draft of the original TLA+ spec
- Non-monolithic Raft and Raft-based Key-Value Store
  - Separates Raft and KV logic
- Primary-Backup Replicated Key-Value Store
- Distributed Lock Service
- AWORSet CRDT (eventually-consistent distributed set)

More about our Raft[1] KV Store

- ➔ Supports GET, PUT
- ➔ All in Go, client library includes PGo-generated code
- ➔ Model checked in TLC w/ safety and liveness properties
- ➔ Resilient to server failures

- ➔ 930 lines of MPCal, 7 archetypes, 22 person days dev time
- ➔ Runs faster than other Spec2Code solutions we tested[2-4]

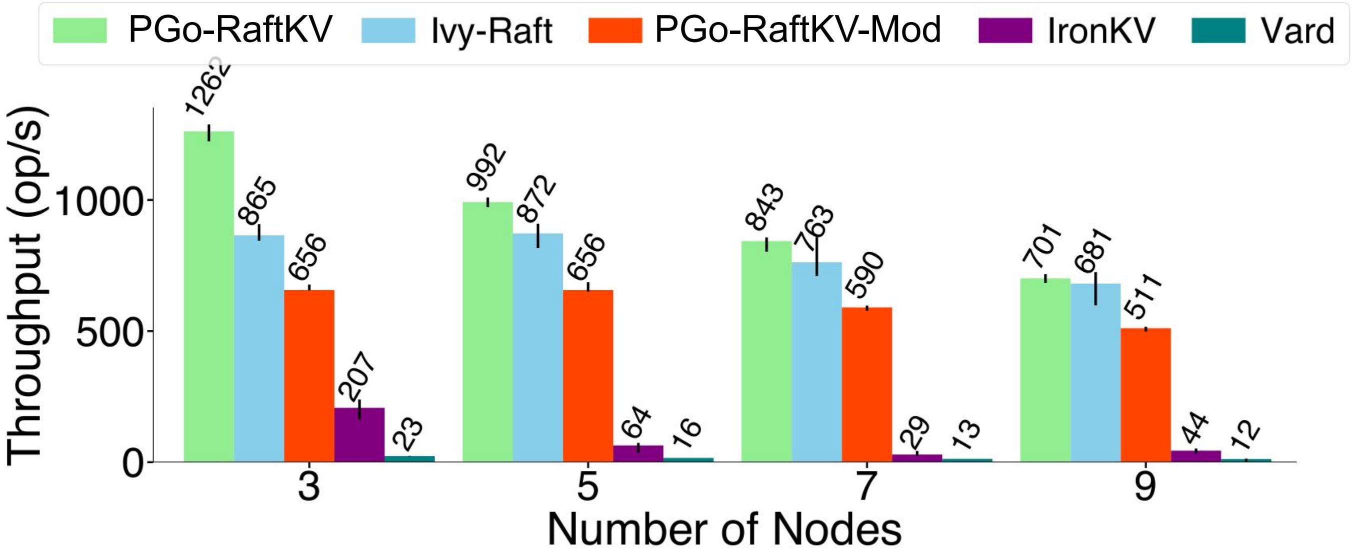
[1] Ongaro, Diego, and John Ousterhout. "In search of an understandable consensus algorithm." 2014 USENIX ATC’14.

[2] Jeffrey S Foster, Dan Grossman, Marcelo Taube, Giuliano Losa, Kenneth L McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, James R Wilcox, and Doug Woos. Modularity for decidability of deductive verification with applications to distributed systems. ACM SIGPLAN’18.

[3] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: a framework for implementing and formally verifying distributed systems. ACM SIGPLAN’15.

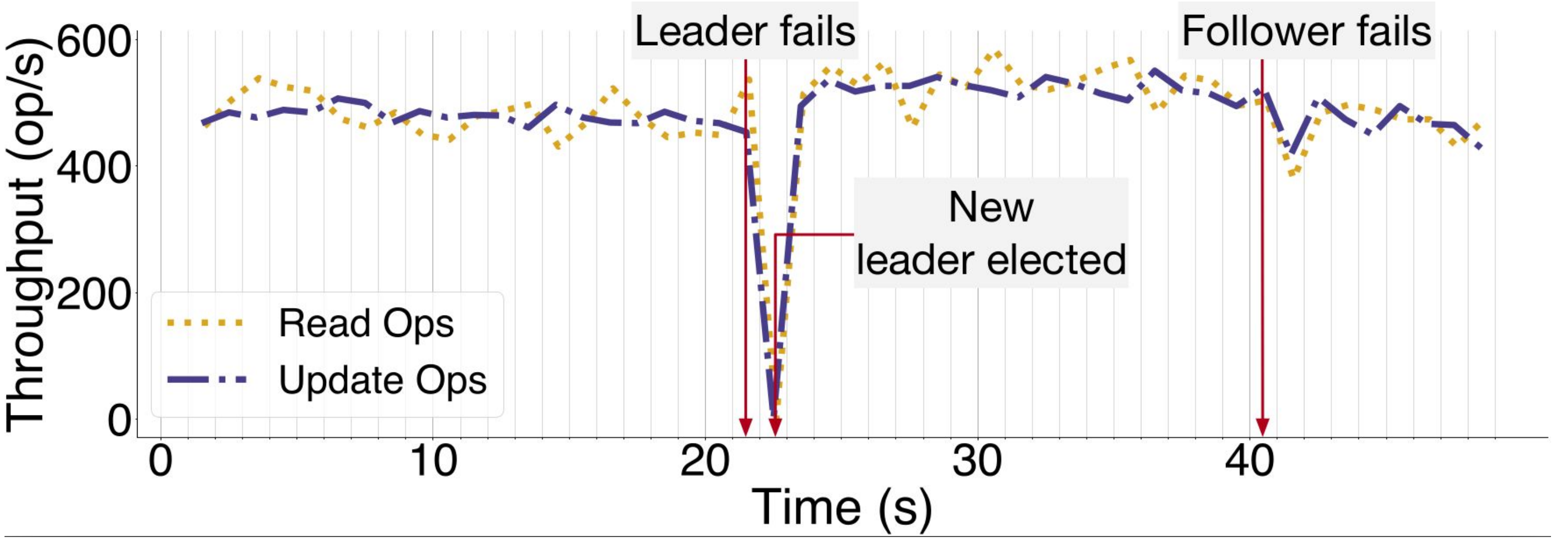
[4] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving Safety and Liveness of Practical Distributed Systems. Commun. ACM’17.

Comparison to other Spec2Code Raft KV Stores



- ➔ Faster than Ivy, IronKV, Vard (other Spec2Code tools)
- ➔ etcd scored 5,866-10,504 op/s, beating all Spec2Code

Graph of Failure Recovery in Action for PGo Raft KV



Outline

Problem description and motivation

PGo recap

Raft Implementation

➢ **Performance improvements and challenges**

Modular verification

Implementation tracing

Conclusion

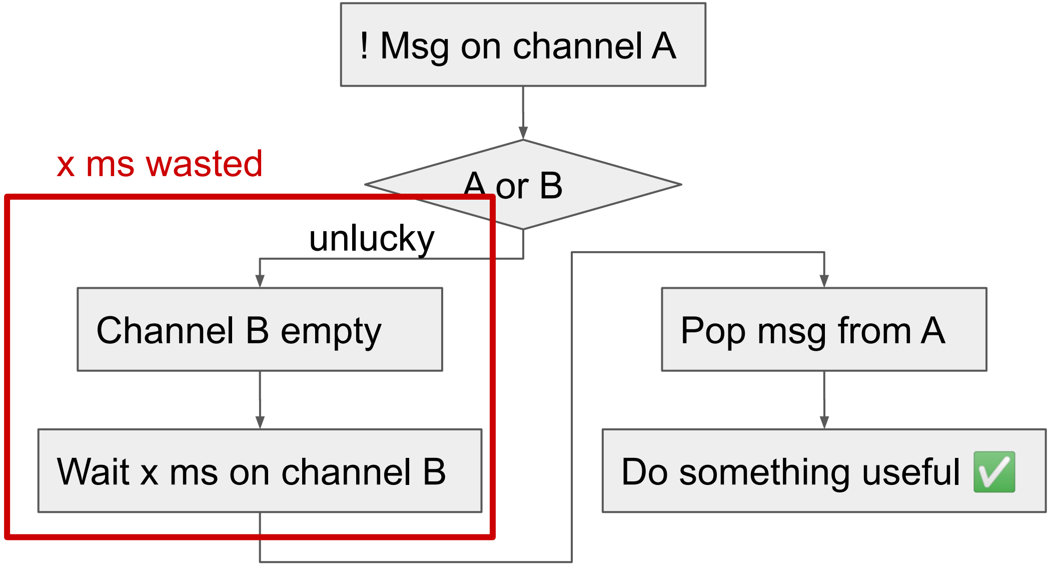
Systemic Performance Concerns

- ➔ With enough human effort, PGo-generated code can be fast enough for distributed systems
- ➔ Currently, it takes more effort than we’d like
- ➔ Key issues:
  - ◆ Non-deterministic branching can waste time
  - ◆ Waiting can waste CPU cycles

Non-Determinism Problems

- Choosing between I/O behaviors can waste time
- Branches are chosen at random, timeouts are serial

```
either {          MPCal
  // read channel A
} or {
  // read channel B
}
```



Await Problems

Await statements may cause busy loops

```
actionA:                     MPCal
await x = 3;
...

... // in another process
actionB:
x := 3;
```

- ➔ Action A may be repeatedly retried if x # 3, in a busywait
- ➔ If action B is also available, it may be starved of CPU cycles
- ➔ Functional, but not ideal

Opportunities for Performance Improvements

- **In progress:** more intelligent handling of non-determinism
  - Current exploration of non-deterministic branches is sequential and **only changes branch on timeout**
  - Ongoing work to **concurrently explore branches without waiting**
  - Possibility of implementing a more reactive evaluation model
- **Opportunity:** leverage static analysis and model checking to selectively remove unnecessary concurrency control

Outline

Problem description and motivation

PGo recap

Raft Implementation

Performance improvements and challenges

➢ **Modular verification**

Implementation tracing

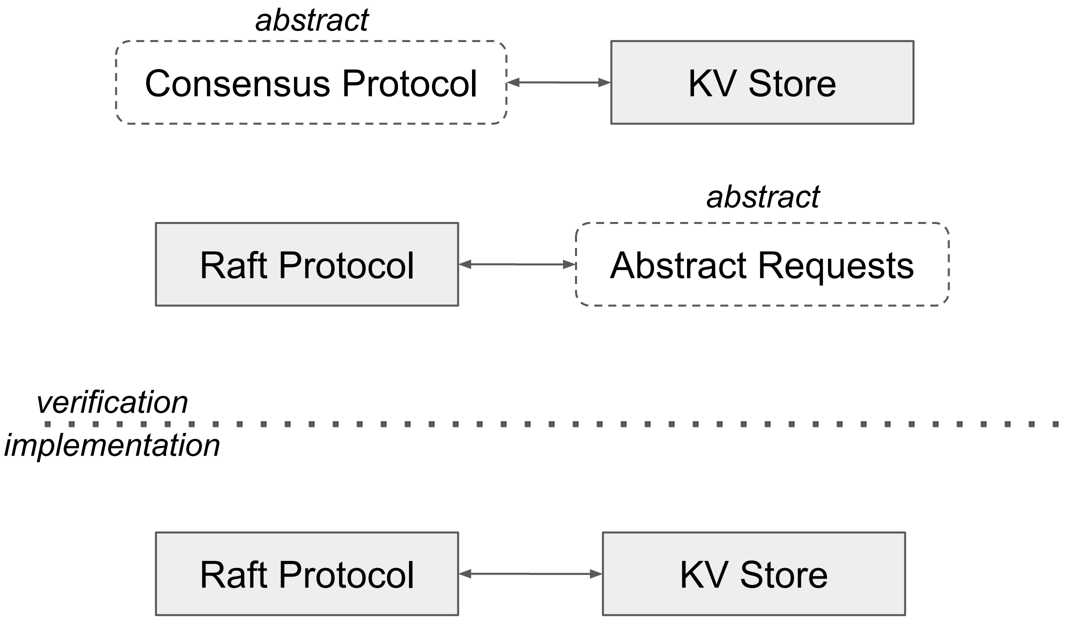
Conclusion

Modular Verification Support

- ➔ PGo can generate implementations for a variety of systems, **including dependencies of other MPCal**
- ➔ Any API can be expressed as **message-passing communication** with a PGo-generated system
- ➔ PGo provides general-purpose glue code
- ➔ This technique offers a **path away from handwritten dependency implementations**, when the implementation is complex and reliability is a priority

Example: Modular Raft KV Store

- Separately verify:
  - Raft protocol
  - KV Store
- Each specification models a simplified, generalized representative of the other



Discussion: Modular Raft KV Store

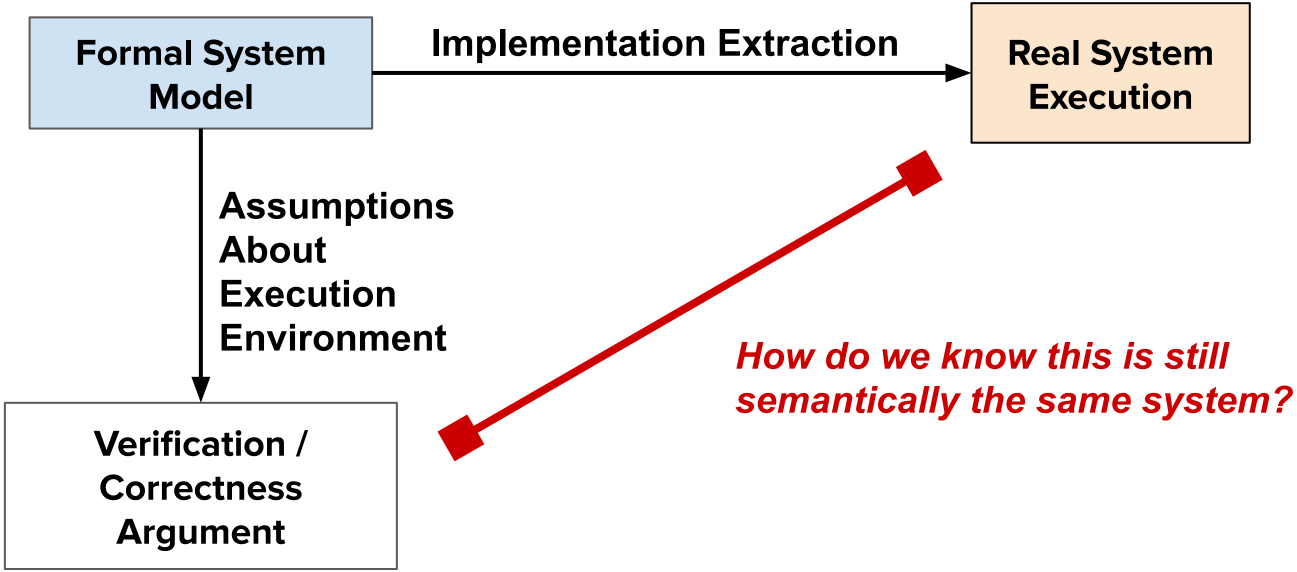
- ✔ Advantages
  - ◆ Raft model becomes re-usable
  - ◆ Smaller state space for TLC
- ❌ Disadvantages
  - ◆ Need to manually co-ordinate separate specifications
  - ◆ Code may be more complex

Idea: could address disadvantages with more automation

Outline

- Problem description and motivation
- PGo recap
- Raft Implementation
- Performance improvements and challenges
- Modular verification
- ➢ **Implementation tracing**
- Conclusion

Key problem: Model-Implementation Mismatch



Formal System Model
Implementation Extraction
Real System Execution
Assumptions About Execution Environment
Verification / Correctness Argument

How do we know this is still semantically the same system?

Potential Model-Implementation Mismatches

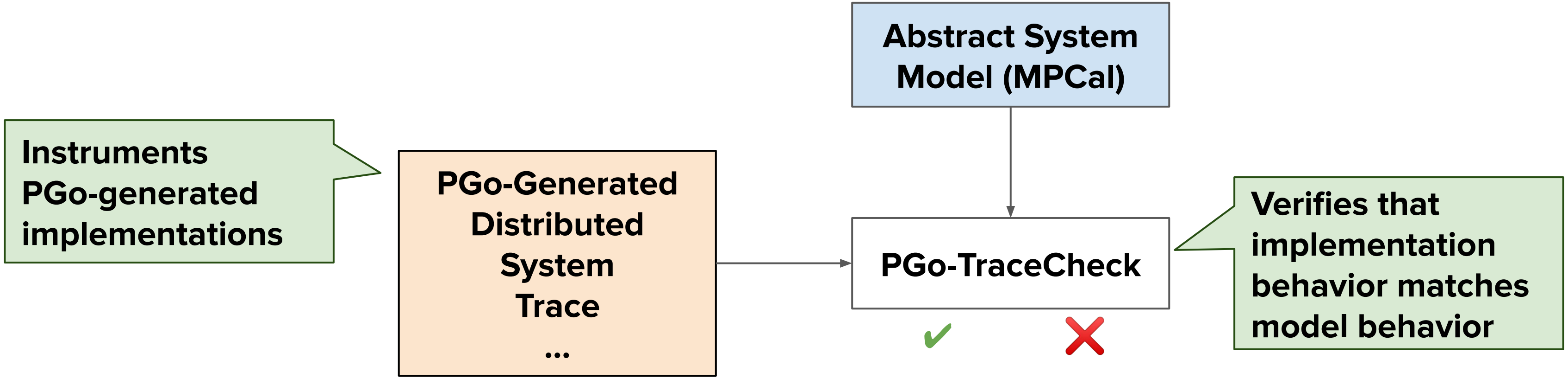
- Systems can be mis-configured
- Systems can be run in situations that do not match model assumptions
  - Model might assume an incorrect model of network communication e.g not accounting for packet size ceiling in UDP
  - Model might not account for certain failure scenarios
- Code generation can be buggy
- Glue code (between model and environment) can be buggy

Implementation Tracing Goals

- We could capture and analyze anything the system does if we trace the implementation…
- We want to cross-check **full system behavior**
  - Including implementation quirks
  - Including full configuration / deployment data

- So, try to holistically **trace implementation behavior**
- We should double check those traces **match the original MPCal spec**

Introducing PGo-TraceCheck



Instruments PGo-generated implementations

PGo-Generated Distributed System Trace
...

Abstract System Model (MPCal)

PGo-TraceCheck

✓ ✗

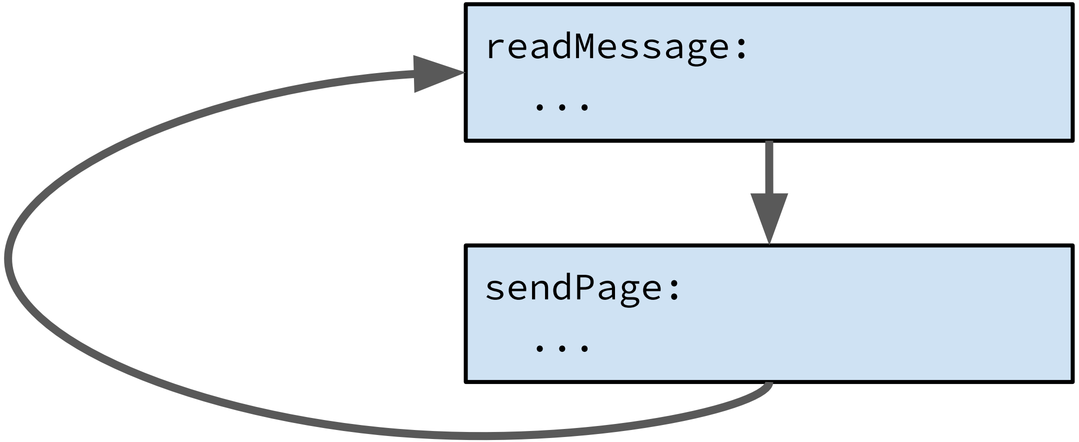
Verifies that implementation behavior matches model behavior

Project Challenges

- ➔ Understand how MPCal executes, especially the **relationship between MPCal model and implementation**
- ➔ Derive **expected behavior** from MPCal that can be compared with the implementation traces
- ➔ **Efficiently** compare implementation and model information

What to trace?

- All MPCal behavior is expressed as **atomic actions**
- Anything more precise than an action is **not modeled**
- So, only need to **record each critical section**

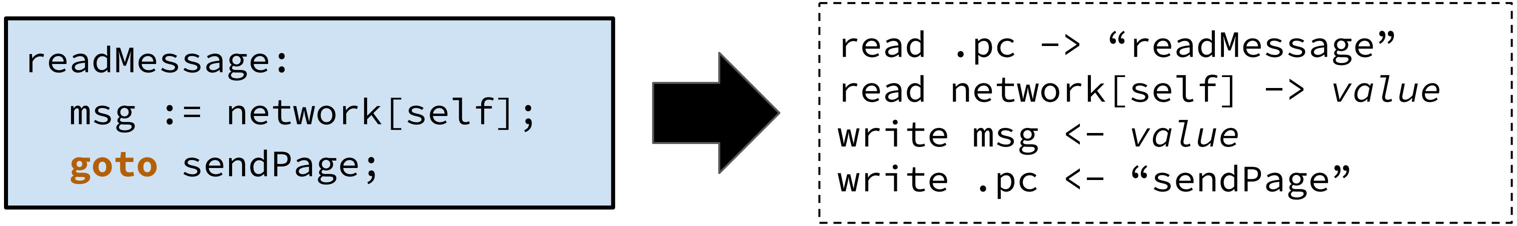


```
readMessage:
  ...
```

```
sendPage:
  ...
```

Tracing Critical Section Behavior

- MPCal communication occurs **only via side-effects**
- PGo-generated code **relies on real-world implementations** of environment features
- So, give up on inspecting e.g the network implementation, but **trace everything that goes into or out of it**.

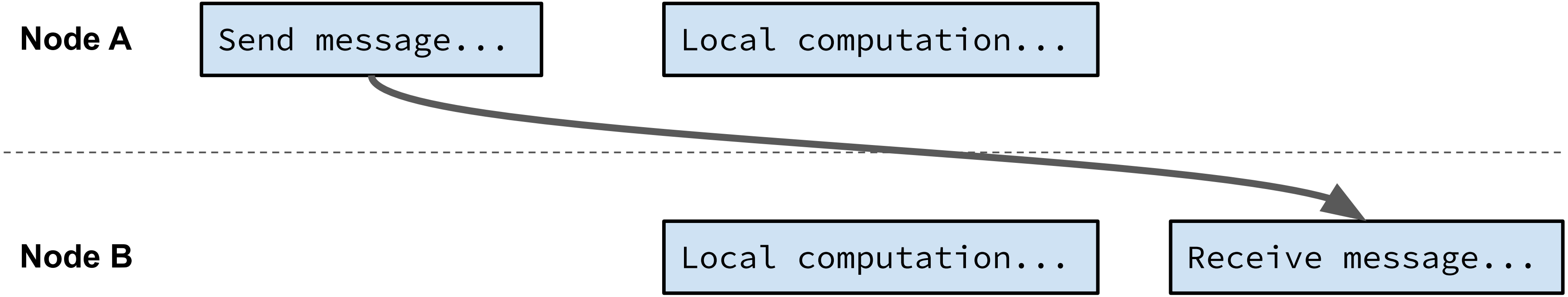


```
readMessage:
  msg := network[self];
  goto sendPage;
```

```
read .pc -> “readMessage”
read network[self] -> value
write msg <- value
write .pc <- “sendPage”
```

Tracing Causality with Vector Clocks

- ➔ Some critical sections are **causally related**, others are not
- ➔ Implementation must record causality via **vector clocks**



What is a Vector Clock?

- ➔ Track whether one event **happens-before** another by marking each event with per-node logical clocks
- ➔ Defines a **partial order between events**
  - ◆ Locally, each event necessarily happens-before the next
  - ◆ Across nodes, events *might* happen-before one another
  - ◆ Some remote events do not have a relative order: they are concurrent, and could have happened in any order

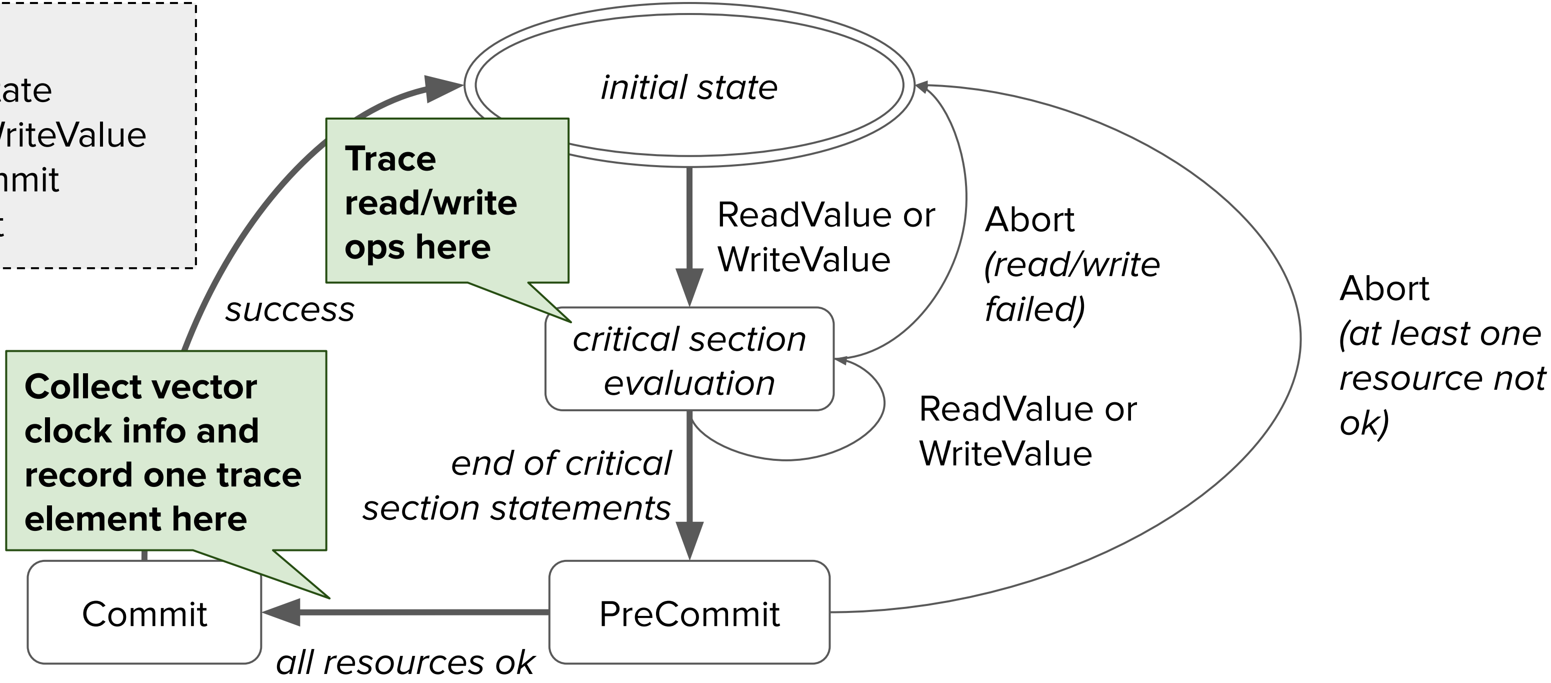
Implementation Tracing Challenges

- Critical sections can spontaneously **abort and roll back**:
    - Network timeout
    - Attempt to read unavailable information
    - Custom condition (e.g await x = 5)
- **Multiple heterogeneous environment implementations** (resources) coexist
- Need to **achieve consensus** between environment components whether the critical section can finish

Two-Phase Commit-like Critical Section Operation

Happy path

1. Initial state
2. Read/WriteValue
3. PreCommit
4. Commit

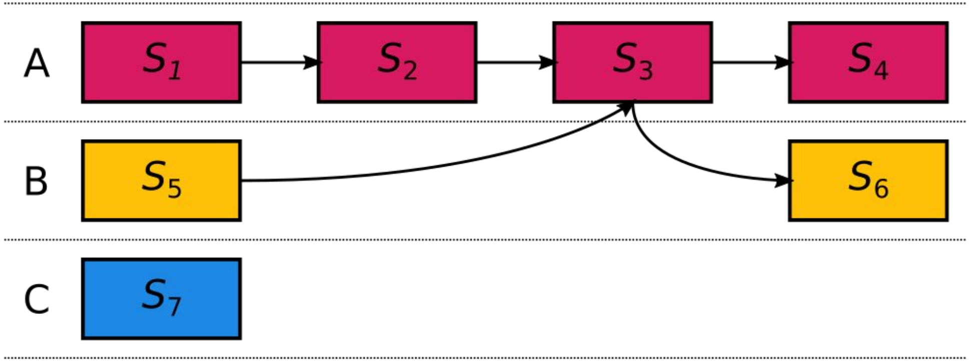

initial state
ReadValue or WriteValue
critical section evaluation
ReadValue or WriteValue
Abort (read/write failed)
end of critical section statements
PreCommit
Abort (at least one resource not ok)
all resources ok
Commit
success

Trace read/write ops here

Collect vector clock info and record one trace element here

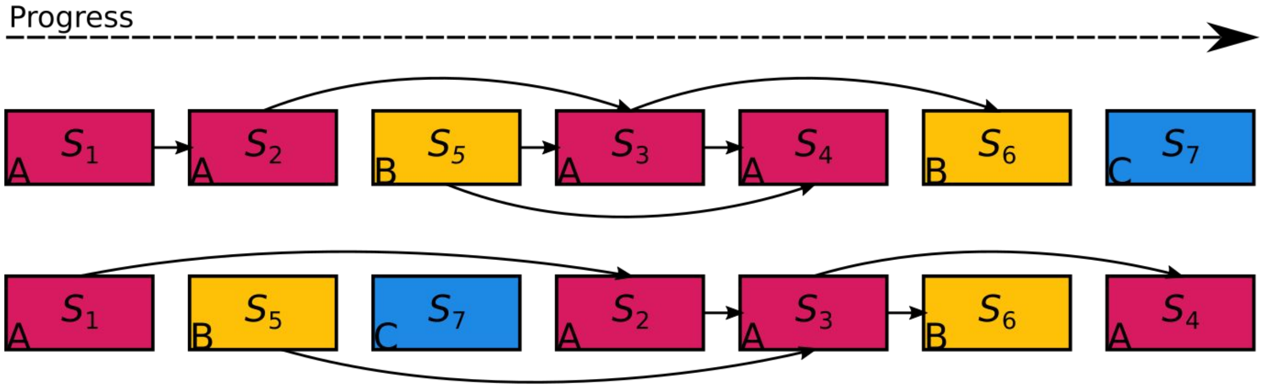
Implementation Traces Have Multiple Possible Orderings

Example 3-node system trace



Edges represent direct causality

<u>For example:</u>



Total order A

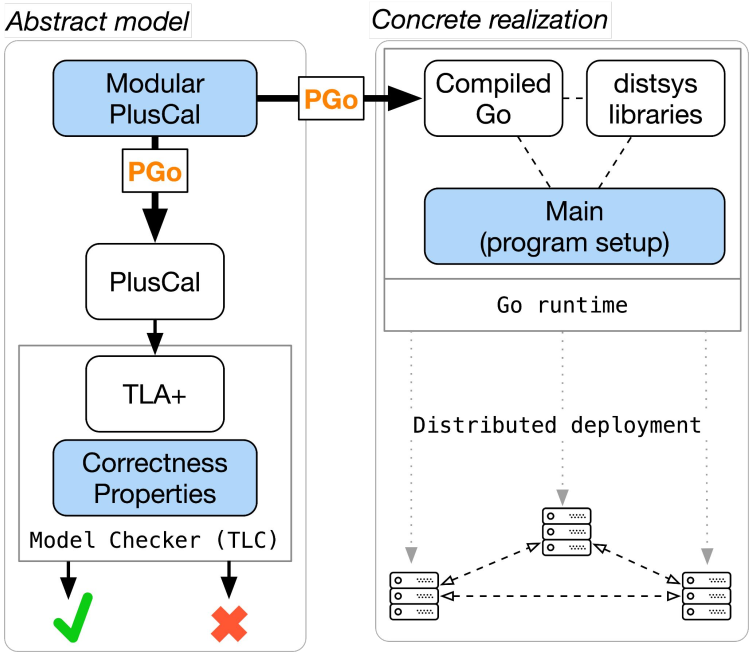
Total order B

Matching Partial Order with Total Order

- Model explorations form a **total order**, while implementation executions form a **partial order**
  <u>So, we need to totally-order the implementation tracing.</u>
- Any implementation path respecting partial order should be valid: **if one path is invalid, there is definitely a bug**
- But, if one path passes, **it does not guarantee all paths do**
  - Our current prototype checks only one trace
  - We have found bugs despite this limitation

PGo Takeaways

- ➔ MPCal cleanly separates the **system** from its **environment**
- ➔ PGo generates **correct** distributed systems
- ➔ Results **improve on state of the art** solutions that require years of manual work
- ➔ We are actively improving PGo's output and tooling to match **production quality** systems code



https://github.com/DistCompiler/pgo