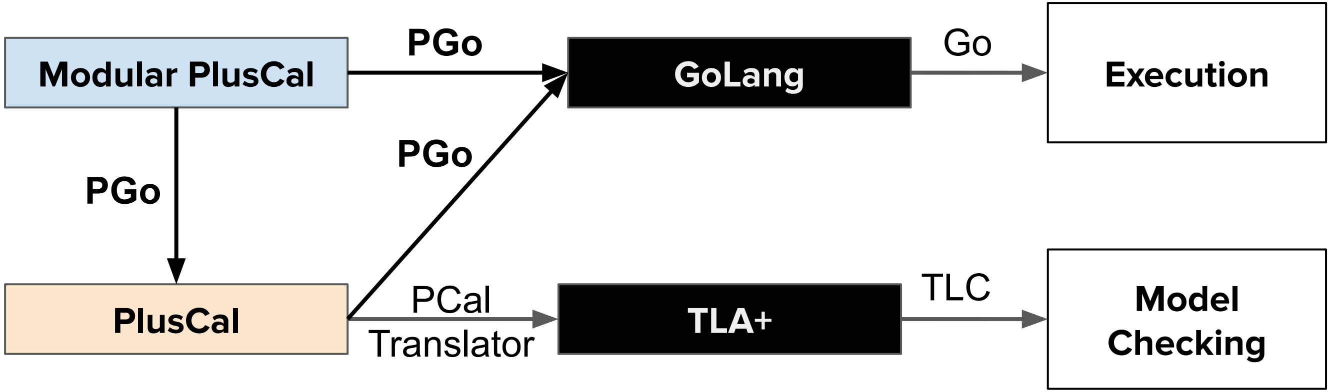
Compiling Distributed System Models into Implementations with PGo

Finn Hackett, Ivan Beschastnikh
Renato Costa, Matthew Do



Motivation

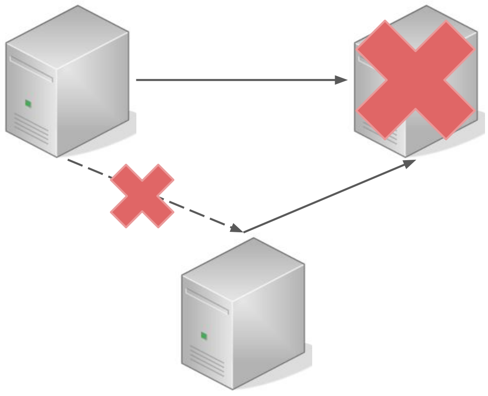
- Distributed systems are widely deployed
- Despite this fact, writing correct distributed systems is **hard**
  - Asynchronous network
  - Crashes
  - Network delays, partial failures...
- Systems deployed in production **often have bugs**

Cloud abstraction

Google's data center, Council Bluffs, IA
https://www.google.com/about/datacenters/gallery

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**



Bugs in Distributed Systems

October 30, 2018 —— Engineering, Featured, Product

October 21 post-incident analysis

Jason Warner

Degraded Performance

Google Compute Engine Incident #17003

New VMs are experiencing connectivity issues

Incident began at **2017-01-30 10:54** and ended at **2017-01-30 12:50** (all times are **US/Pacific**).

Feb 10, 2017 - GitLab

Postmortem of database outage of January 31

Postmortem on the database outage of January 31 2017 with the lessons we learned.

Service Outage

Data loss

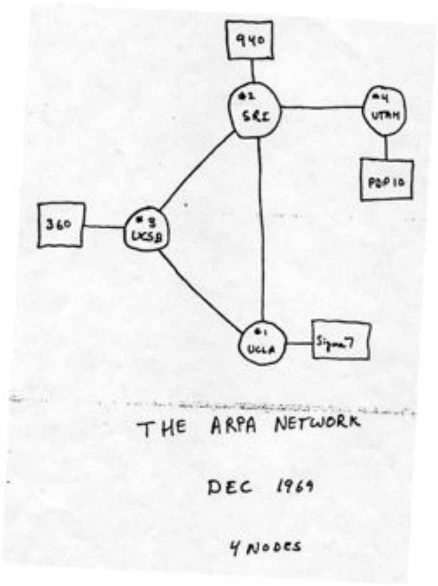
[1] Mark Cavage. 2013. There's Just No Getting around It: You're Building a Distributed System. Queue 11, 4, Pages 30 (April 2013)
[2] Fletcher Babb. Amazon's AWS DynamoDB Experiences Outage, Affecting Netflix, Reddit, Medium, and More. en-US. Sept. 2015
[3] Shannon Vavra. Amazon outage cost S&P 500 companies $150M. axios.com, Mar 3, 2017

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, making **protocol changes harder**
- Production implementations resort to **ad-hoc error handling** [PODC'07, OSDI'14, SoCC'16, SOSP'19]

One key problem for distributed systems

Design



gap

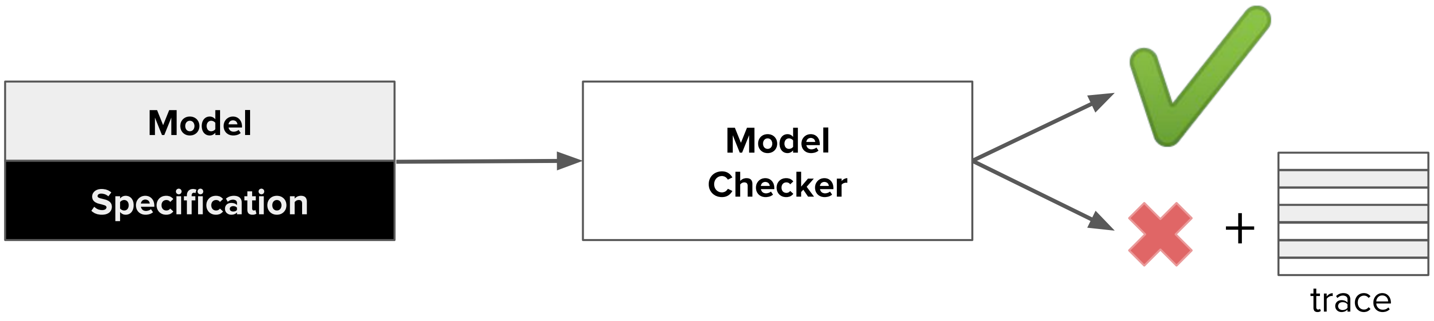
Implementation

Related Work

- ➔ Using **proof assistants** to prove system properties
  - ◆ Verdi [PLDI'15], IronFleet [SOSP'15]
  - ◆ Require a lot of developer **effort** and **expertise**
- ➔ Model checking **implementations**
  - ◆ FlyMC [EuroSys'19], CMC [OSDI'02], MaceMC [NSDI'07], MODIST [NSDI'09]
  - ◆ **State-space explosion**: many states irrelevant to high-level properties
- ➔ Systematic **testing**, **tracing**, and **debugging**
  - ◆ P# [FAST'16], D^3S [NSDI'08], Friday [NSDI'07], Dapper [TR'10]
  - ◆ **Incomplete**; requires runtime detection or extensive test harness

Model Checking

- Verifies a **model** with respect to a **correctness specification**
- Specification can define **safety** and **liveness** requirements
- Produces a **counterexample** when a property is violated



Model Checking a Bank Transfer

$$\text{VARIABLES } AliceSavings,\ BobSavings,\ Amount$$

Initial state: both accounts have **positive** balance

$$Init \triangleq \land AliceSavings \in Nat \\ \land BobSavings \in Nat \\ \land Amount \in Nat$$

Transfer `Amount` between accounts

$$Transfer \triangleq \land AliceSavings' = AliceSavings - Amount \\ \land BobSavings' = BobSavings + Amount$$

$$Spec \triangleq Init \land Transfer$$

Property: transfer should preserve positive balances

$$ValidBalances \triangleq AliceSavings > 0 \land BobSavings > 0$$

$$\text{THEOREM } Spec \Rightarrow ValidBalances$$

Visualizing an Error Trace

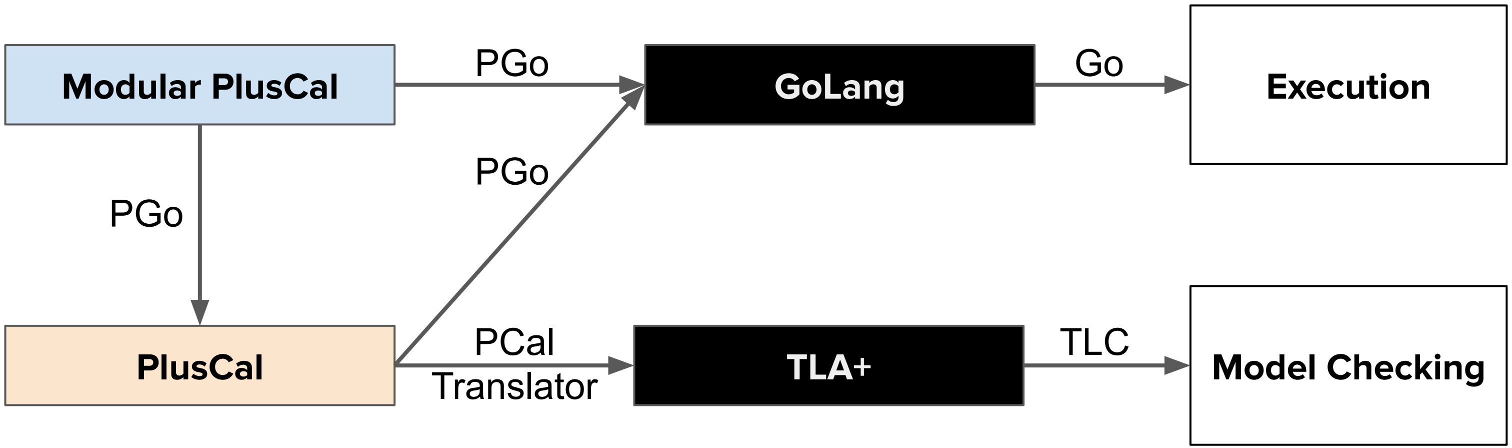
Name	Value
<Initial predicate>	State (num = 1)
AliceSavings	1
Amount	2
BobSavings	1
<Transfer line 10, col 13 to line 12, col 31 o	State (num = 2)
AliceSavings	-1
Amount	2
BobSavings	3

Error: our model does not check if Alice has **sufficient** funds!

Overview of **PGo** and **Modular PlusCal**

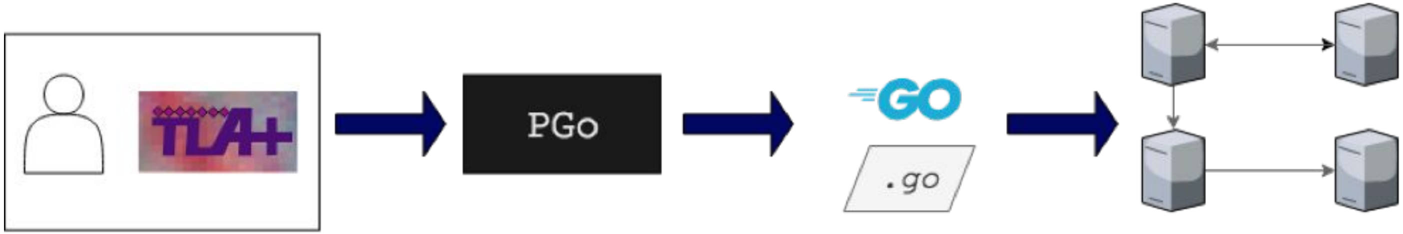
PGo compiler toolchain

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



PGo workflow

Making building distributed systems easier



Developer writes **specification**

PGo compiles it to a *matching implementation*

Source is compiled

Running Distributed System!

✓ **TLC**

✓ **PGo**

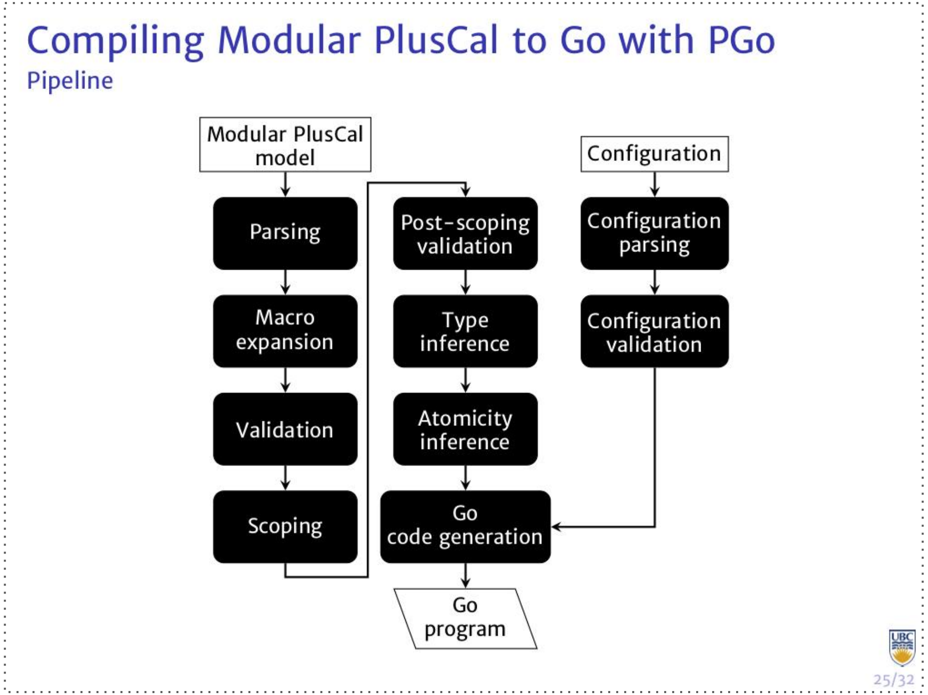
Transition from *design* (specification) to *implementation* is **automated**

PGo trade-offs

- Advantages
  - Compatible with existing PlusCal/TLA+/TLC eco-system
  - Mechanize the implementation = less dev work
  - Maintain one definitive version of the system
- Limitations
  - **No free lunch: concrete details have to be provided somehow**
    - **Environment** is abstract: developer must **edit** generated source
    - **Bugs** can be introduced in this process
  - **Software evolution**: unclear how to **reapply** the changes to model?

In today’s talk

- Focus on explaining ModularPlusCal (MPCal)
- Examples and demo
- Omit PGo compiler details:



How would you naively 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**

```
readMessage: // blocking read from the network
env.Lock(“network”)
network := env.Get(“network”)
if !(Len(network.Get(self)) > 0) {
  env.Unlock(“network”)
  goto readMessage
}
msg = Head(network.Get(self))
env.Set(“network”, network.Update(self, Tail(network.Get(self))))
env.Unlock(“network”)
```

Go

Not a blocking network read

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

Use macros?

```
variables network = <<>>;
...
readMessage:
    NetworkRead(msg, self);
```

PlusCal

Semantics still rely on **global variables**

The **macro body** could be replaced by a **real-world implementation**

All processes will share the same **view of** and **access to** the **environment**

Network semantics become a **one-liner**

```
readMessage:
msg := ReadNetwork(self)
```

Go

Assumes one canonical network

Invent a new kind of macro: archetype

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

MPCal

Processes are **parameterised** by an **abstraction** over the environment

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

Any number of model checker and implementation behaviors can be defined elsewhere, since the environment is abstract

```
readMessage:
msg := network.Read(self)
```

Modular PlusCal: System vs Environment

- **Goal**: isolate system **definition** from abstractions of its **execution environment**
- Semantics of new primitives:
  - Archetypes can only interact with arguments passed to them
  - Archetype arguments encapsulate their environment and are called **resources**
  - Each resource can be **mapped** to an abstraction for model checking when archetypes are **instantiated**

The Modular PlusCal Language

- **Archetypes**: define **API** to be used to interact with the concrete system
- **Mapping Macros**: allow definition of **abstractions**
- **Instances**: Configures abstract **environment** for model checking

MPCal

```
variables network = <<>>;

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

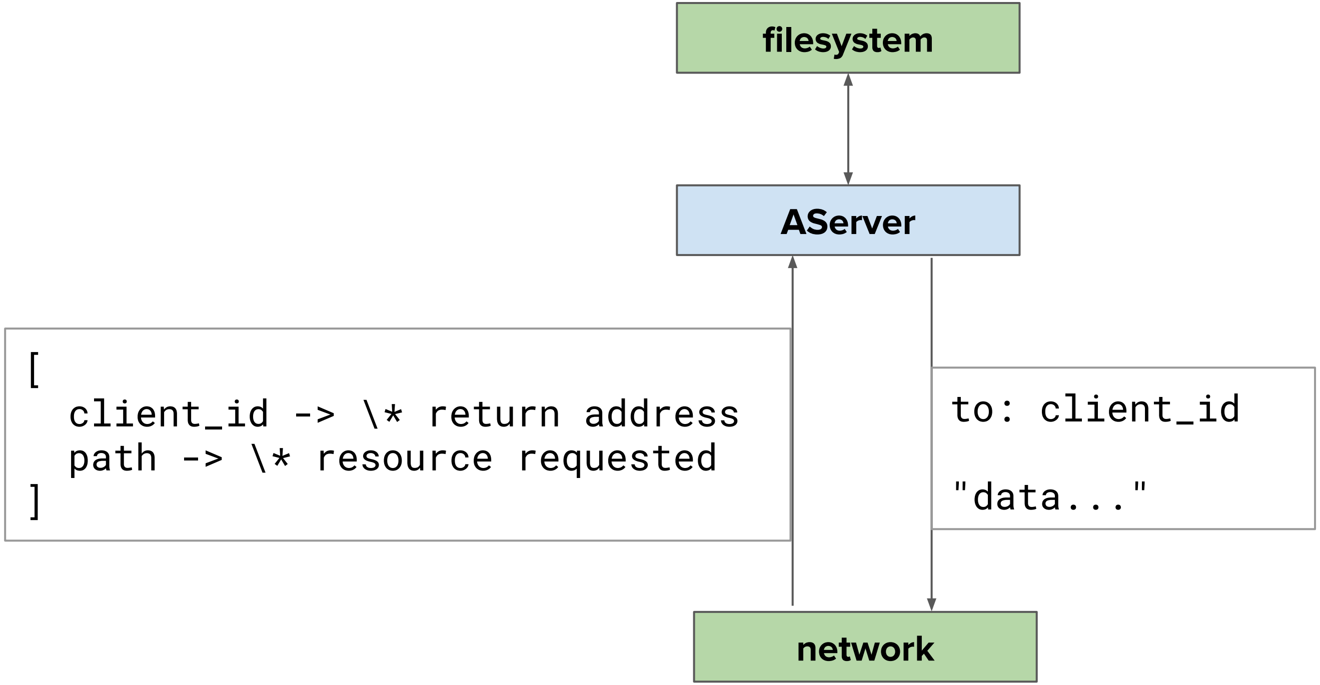
```
archetype AServer(ref network, ...)
...
readMessage:
    msg := network[self];
```

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

Web server example



Abstract Server with Buffered Network (PlusCal)

```
variables network = <<>>;

process (Server = 0)
variable msg;
{
  readMessage:
    await Len(network[self]) > 0;
    msg := Head(network[self]);
    network := [network EXCEPT ![self] = Tail(network[self])];

  sendPage:
    await Len(network[msg.client_id]) < BUFFER_SIZE;
    network := [network EXCEPT ![msg.client_id] = Append(network[msg.client_id], WEB_PAGE)];
    goto readMessage;
}
```

PlusCal

Abstract environment: network as **sequences**

Abstractly represents **reading** a message from the **network**

Model checking concern: only send messages if the buffer has space

Model website data **as a constant** called WEB_PAGE

Abstract Server with Buffered Network (MPCal)

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

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

MPCal

Archetype has access to: a **network**, a **filesystem**

Interacting with the **network** becomes straightforward

Reading from the **filesystem** becomes clear, unlike just passing around a WEB_PAGE **placeholder**

Environment Abstractions: Buffered Network

What happens when a variable is read, transform the underlying value `$variable` and `yield` the result.

```
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

Abstract **blocking network read** semantics

What happens when a variable is written, apply the new `$value` to the underlying `$variable` and `yield` the new underlying value.

Abstract **buffered network write** semantics

Environment Abstractions: Filesystem Read

```
mapping macro WebPages {
  read {
    yield WEB_PAGE;
  }
  write {
    assert(FALSE);
    yield $value;
  }
}
```

MPCal

Reading modeled **lossily** by returning a **constant**

Writing **not modeled**, so represented by **failure**

Putting it All Together: Instances

```
variables network = <<>>;

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

MPCal

Same **model checking** abstractions

Server is an **instance** of AServer, with all the mapping macros and parameters **expanded**

Function-mapping syntax

Mappings without the [_] also exist:

```
mapping pipe via ... ;
```

Reviewing Source Languages

PlusCal	Modular PlusCal
Abstract environment; require **manual edits** in the generated implementation that can **introduce bugs**	Abstractions are **isolated**: not included in archetypes. Behavior can be **preserved** if abstractions have **implementations** with **matching semantics**
Protocol updates are **difficult**; developer needs to **reapply** manual changes	Protocol updates can be applied **any time**; generated code is **isolated from execution environment**

PGo Workflow

1. User writes system model in MPCal

2. PGo compiles model to PlusCal

3. User defines correctness properties

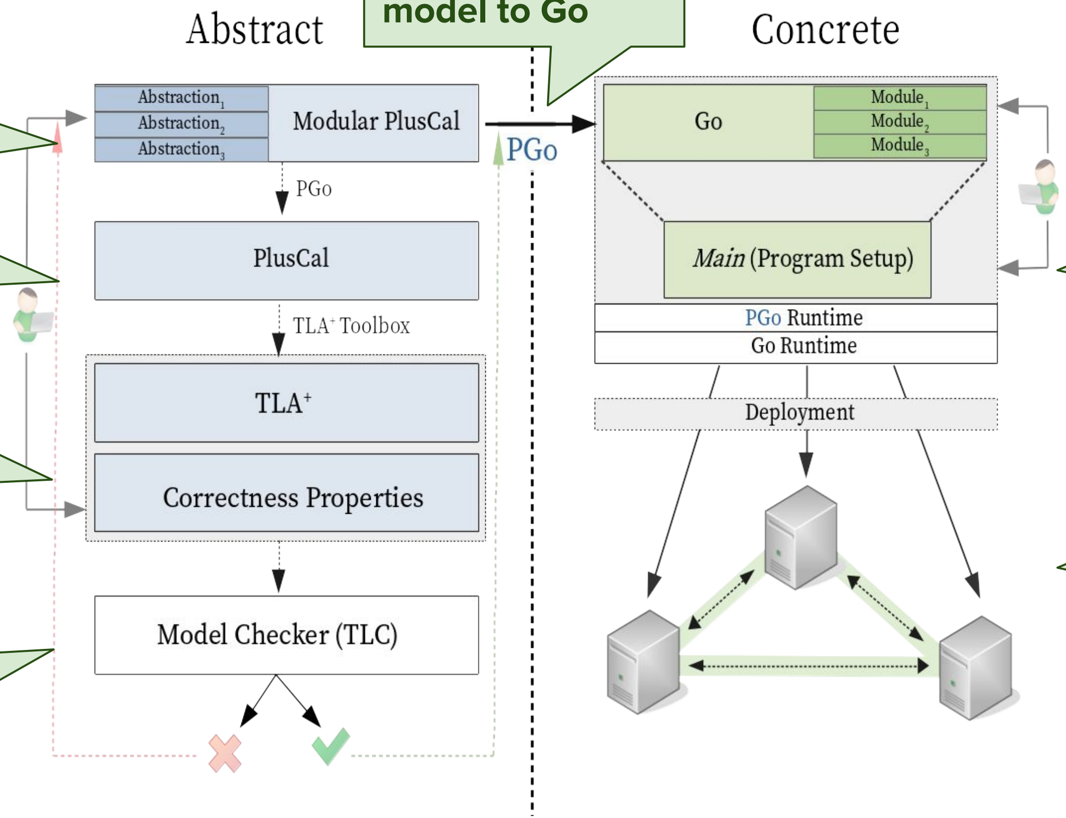
4. TLC validates or produces counterexample

5. PGo compiles model to Go

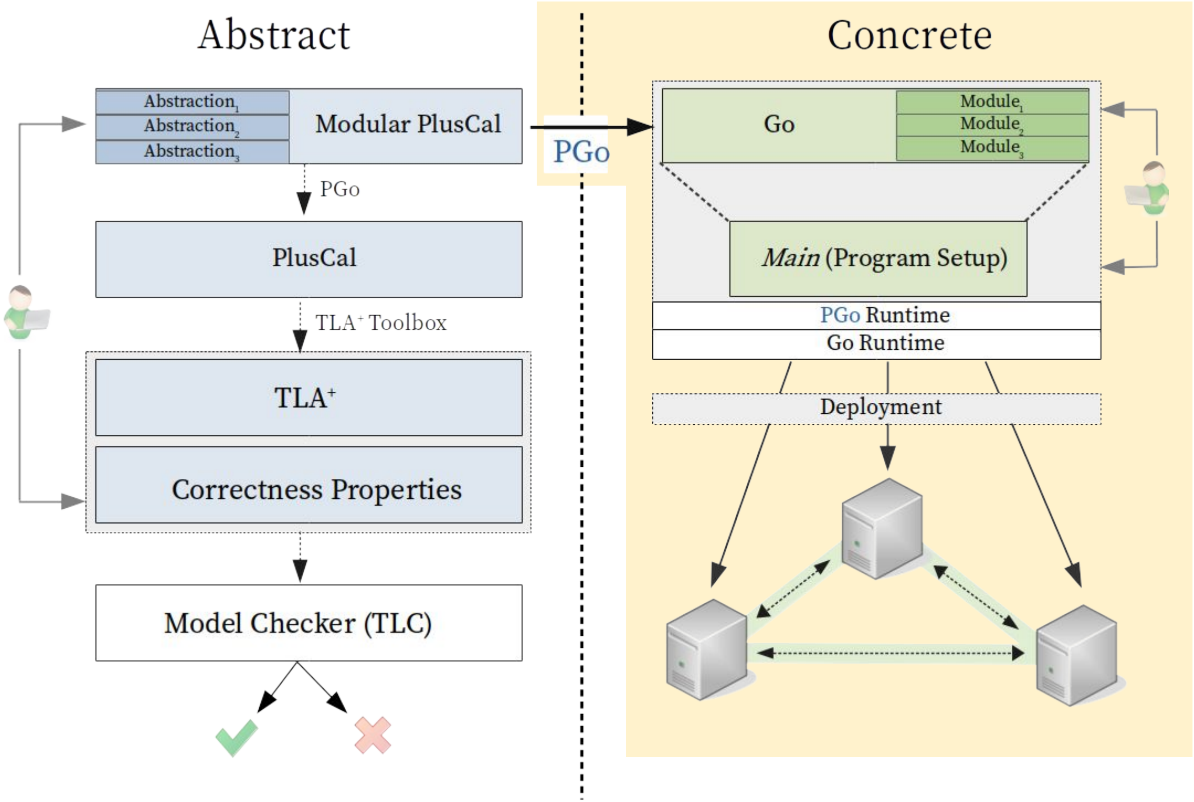
6. User choses concrete implementation for abstractions

7. User writes *main* function (bootstrap)

8. System is deployed in a distributed environment



PGo Workflow



Compiling Modular PlusCal to Go

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**
- We need to understand how TLC explores behaviors **defined** by a model

Coming Back to the Server Example

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

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

MPCal

```
archetype ALoadBalancer(ref network)
variables msg, next = 0;
{
  rcvMsg:
    msg := network[LoadBalancerId];
    assert(msg.message_type = GET_PAGE);

  sendServer:
    next := (next % NUM_SERVERS) + 1;
    mailboxes[next] := [
      message_id |-> next,
      client_id |-> msg.client_id,
      path |-> msg.path
    ];
    goto rcvMsg;
}
```

MPCal

Behaviors in a Model

```
variables network = <<>>;

process (Server = 0) == instance AServer(ref network, filesystem)
    mapping network[_] via TCPChannel
    mapping filesystem[_] via WebPages;

process (LoadBalancer = 1) == instance ALoadBalancer(ref network)
    mapping network[_] via TCPChannel;
```

MPCal

TLC explores all possible **interleavings** between two processes (instances)

Interleavings between Processes

```
archetype ALoadBalancer(ref network)
variables msg, next = 0;
{
  rcvMsg:
    msg := network[LoadBalancerId];
    assert(msg.message_type = GET_PAGE);

  sendServer:
    next := (next % NUM_SERVERS) + 1;
    mailboxes[next] := [
      message_id |-> next,
      client_id |-> msg.client_id,
      path |-> msg.path
    ];
    goto rcvMsg;
}
                                    MPCal
```

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

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

Labels define **atomic steps** in the model (or **actions**)

Possible behaviors ✓

```
rcvMsg
sendServer
readMessage
sendPage
```

```
readMessage
sendPage
rcvMsg
sendServer
```

```
rcvMsg
readMessage
sendServer
sendPage
```

Impossible behavior ✗

```
sendServer
rcvMsg
readMessage
sendPage
```

Preserving Modular PlusCal Semantics in Go

- Trivial solution: **runtime scheduler** that chooses which step to run next
  - Prohibitively **expensive**, especially in a distributed system context
- **Goal**: achieve as much **concurrency** as possible across archetypes without changing behavior:
  - Exploit the fact that archetypes can only perform **externally visible** operations by interacting with its **resources** (parameters)
  - Achieve concurrency while preserving **atomicity** when it matters
  - Devise an algorithm to **safely** execute the statements in a step

Reasoning about Concurrency (part 1)

- Steps that **do not use** any resource are **safe** to be executed **concurrently** with other steps
  - Their effects are "**invisible**"
  - **Equivalent** to some **sequential** execution explored by TLC

```
archetype AServer(ref network,
                      file_system)
variable msg;
{
  start:
    print "Waiting for message";

  readMessage:
    msg := network[self];

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

MPCal

Reasoning about Concurrency (part 2)

```
archetype ALoadBalancer(ref network)
variables msg, next = 0;
{
  rcvMsg:
    msg := network[LoadBalancerId];
    assert(msg.message_type = GET_PAGE);

  sendServer:
    next := (next % NUM_SERVERS) + 1;
    network[next] := [
      message_id |-> next,
      client_id |-> msg.client_id,
      path |-> msg.path
    ];
    goto rcvMsg;
}
                                                    MPCal
```

```
archetype AServer(ref network,
                      file_system)
variable msg;
{
  start:
    print "Waiting for message";
  readMessage:
    msg := network[self];
  sendPage:
    network[msg.client_id] := file_system[msg.path];
    goto readMessage;
}
                                                    MPCal
```

- ➔ Steps that use the the **same resource** (environment) **may not** be safe to run concurrently
  - Let **implementation** dictate **safety** of concurrent execution
  - If **exclusive access** is needed (such as in our log), **locks** can be used

Executing an Atomic Step in Go

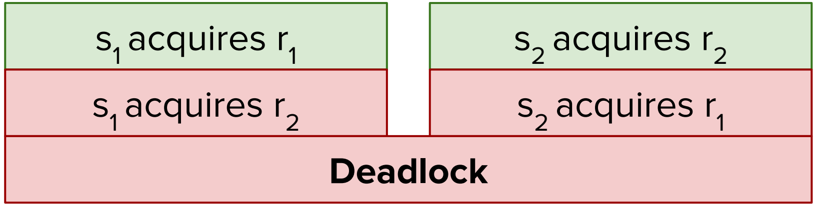
- ➔ We generate a **Go function** for each **archetype** instantiated in the model
  - ◆ Steps in an archetype may be executed **concurrently** with steps from other archetypes
- ➔ Overview of the execution model of a **single step**:
  - ◆ **Acquire** all resources used in the step
  - ◆ Execute all **statements** in order
  - ◆ **Release** all resources at the end
- ➔ Is this always safe?

Deadlocks!

- Steps s_1 and s_2 interact with resources r_1 and r_2, but in different orders
- Suppose also that they both require **exclusive access**
- **Deadlock** becomes possible

```
s1:
  if (r1 > 0) {
      r2 := 0;
  }
                    MPCal
```

```
s2:
  if (r2 > 0) {
      r1 := 0;
  }
                    MPCal
```



Updating our Execution Model

- Resources are acquired in **consistent order**
  - Either <r_1, r_2> or <r_2, r_1>, **always**
- Updated execution model:
  - Acquire all resources used in the step, **in consistent order**
  - Execute all statements in order
  - Release all resources at the end

```
s1:
  if (r1 > 0) {
      r2 := 0;
  }
```

MPCal

```
s2:
  if (r2 > 0) {
      r1 := 0;
  }
```

MPCal

Reasoning about the Execution Model

- We offer a **reduction argument** about the safety of the execution model
- Take any two labels. There are three cases to consider:
  - One of the labels **does not use any resource**: equivalent to sequential execution
  - Labels use **disjoint** sets of resources: equivalent to sequential execution (steps interact with **different parts** of the environment)
  - Labels use **overlapping** sets of resources: if resources require **exclusive access**, they should implement that behavior when being **acquired**.

Resources Mapped as Functions

```
process (Server = 0) == instance AServer(ref network,
                                         filesystem)
    mapping filesystem[_] via WebPages
                                                      MPCal
```

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

- → Resources can be **mapped as functions**
- → **Entire function applications** is seen as the resource
- → **Challenge**: statically analysing MPCal model is **no longer sufficient** to determine resources used in a step

Solution

- ➔ Resources mapped as functions are acquired **in the statement they are used**
- ➔ Drawback: they **cannot** be acquired in consistent order
- ➔ Instead, we allow actions to be **restarted** during a potential deadlock

Executing a Modular PlusCal step (action) in Go

Main loop: only exit when the step is **complete**

Execute the **statements** defined in the **model**

If it cannot be acquired (potential **deadlock**), **restart from scratch**

Algorithm 1: Action Execution

```
actionCompleted = FALSE;
while ¬ actionCompleted do
    Acquire all resources not mapped as functions in consistent order;
    ... action statements ...;
    /* A resource mapped as function is required at
       this point                                   */
    error = Acquire(resourceMappedAsFunction);
    if error ≠ NULL then
        Abort all resources acquired so far;
        if ShouldRetry(err) then
            continue;
        end
        return error
    end
    ... more action statements ...;
    /* End of action */
    Release all acquired resources;
    actionCompleted = TRUE;
end
```

Resources **not mapped as functions** acquired in **consistent order** as described

Resource **mapped as function** is used in a statement

When **all** statements are executed, make environment changes **externally visible**

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
    - ◆ If implementation matches abstraction, code generated by PGo **does not need to be manually edited**

Environment Implementations: Requirements

- What is needed from these implementations?
  - A way to "**acquire**" them before use (enforcing **exclusive access** if necessary)
  - **Interacting** with the environment (reading, writing)
  - Making environment changes **visible** at the end of the atomic step
  - **Aborting** local interaction if step needs to be **restarted**

Archetype Resources API (in Go)

```go
type ArchetypeResource interface {
    Acquire(ResourceAccess) error
    Read() (interface{}, error)
    Write(interface{}) error
    Release() error
    Abort() error
    Less(ArchetypeResource) error
}
```

Go

Called **before** the resource is read or written

Discards interactions when actions need to be **restarted**

Only call that can make **externally visible** effects

Allow resources to be **comparable** (enforcing **consistent order**)

Handling Errors

- API functions implemented by resources may return **errors**
- Errors are used for **two purposes** during execution:
  - To flag unrecoverable **environment errors**
  - To request that an action be **restarted** (e.g., potential deadlock)

Environment Errors	Restart Request
I/O error reading or writing to a file or socket; network operation **timeout**	Attempt to read a socket when **no message is available**; attempting to **lock shared data** that is already locked

DEMO

Compiling and Running `load_balancer.tla`

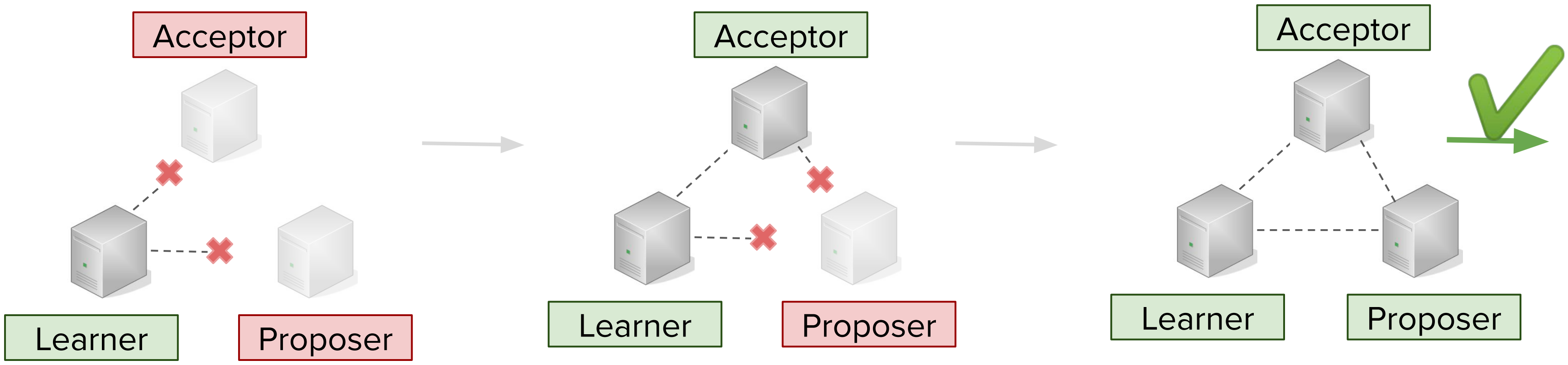
Distributed **Runtime**

Execution Runtime

- **Goal**: reduce the **burden** on developers by providing resources often used in distributed systems
  - **Scheduling** setup
  - **Network** communication
  - **Global state**
  - Others: file system, time, shared resources, etc…

Synchronized Start

- Allows processes (that may run on **different nodes** when deployed) to **coordinate** when they start execution
  - TLA$^+$ **weak fairness**
- Developer can use it to enforce a **distributed barrier**



Distributed Global State

- Provides the abstraction of **shared state** in a distributed system
- Exposed as an archetype **resource** implementation
  - Makes it easier to **migrate** PlusCal spec to Modular PlusCal
- Data is stored **across all nodes** in the system
  - Objects are **owned** by only one at a time, but can **move** over time
  - (Many exciting future work directions hide here :-)

Distributed Global State: Data Store

Name	Value	Owner
counter	`42`	P_1
queue	`nil`	P_3
history	`nil`	P_2

Node has value for data it **owns**

No state kept if **not owned**

Name	Value	Owner
counter	`nil`	P_3
queue	`nil`	P_3
history	`["e1", "e2"]`	P_2

Name	Value	Owner
counter	`nil`	P_1
queue	`["j1", "j2"]`	P_3
history	`nil`	P_1

Ownership may be **outdated**

Evaluation

- ➔ PGo is 25K LOC (compiler) and 3K (runtime)
- ➔ Able to compile concurrent and distributed systems
- ➔ Supports different dist. state strategies

Evaluation

- Is the implementation **sufficiently robust** to support the compilation of **complex specifications**?
- Do systems compiled by PGo have **behavior** that is **defined** by the specification?
- What is the **performance** of systems compiled by PGo, and how does it **compare** with similar, **handwritten** implementations?

A partial set of specs that we wrote

- Load Balancer model:
  - Defines interaction of a **load balancer**, multiple **servers** and multiple **clients**. Implementations interact with the **file system**
- Replicated Key-Value Store:
  - **Serializable** key-value consistency semantics
  - Replicated state machines using **Lamport logical locks** to determine **ordering** and **stability***
  - An assignment at UBC in Winter 2019
- **Raft** and **Paxos** models; no eval for these yet

* as described in *Implementing Fault-Tolerant Services Using the State Machine Approach: a Tutorial*

MPCal and Go LOC

Specification	Archetypes	Mapping macros	MPCal LOC
Load balancer	3	2	79
Replicated KV	5	6	291

Implementation	PGo-gen Go LOC	Manual Go LOC	Total Go LOC
Load balancer	494	85	579
Replicated KV	3,395	234	3,629

MPCal and Go LOC

Specification	Archetypes	Mapping macros	MPCal LOC
Load balancer	3	2	79
Replicated KV	5	6	291

PGo

Implementation	PGo-gen Go LOC	Manual Go LOC	Total Go LOC
Load balancer	494	85	579
Replicated KV	3,395	234	3,629

Semantic Equivalence

- **Proof** that resulting system is **semantically equivalent** to original model is future work (certified compilation)
- Tested both systems
  - Load balancer: different **numbers of clients/servers**; files of different **sizes**; verified result was **received by client** as expected
  - Replicated Key-Value Store: Different **numbers of clients/replicas**; keys and values as **random bytes** of configurable length; clients issue request **sequentially** or **concurrently**; at the end: all replicas are **consistent**.
  - All tested student solutions **had bugs** when the same test suite was used!

Performance Comparison

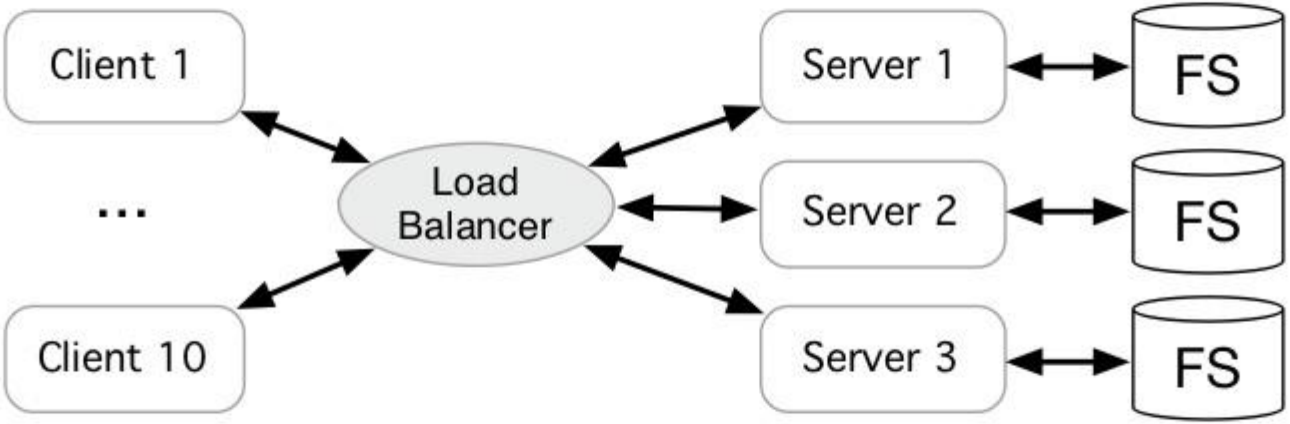
- Comparison with **handwritten** versions of the load balancer and replicated key-value store

Implementation	PGo version (gen)	Manual version
Load balancer	579 (494)	156
Replicated KV	3,629 (3,395)	406

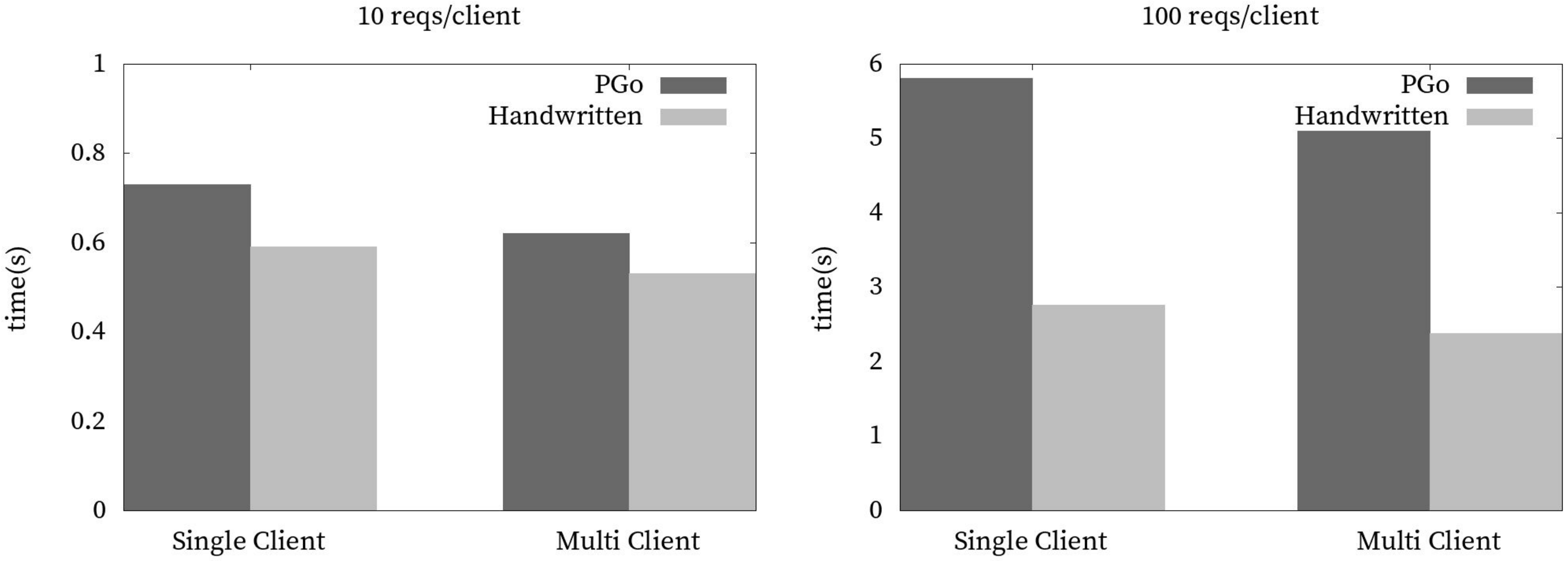
5-8x LOC increase

- Experimental setup: all processes running on the **same node**, focus on **runtime** overhead

Load Balancer setup

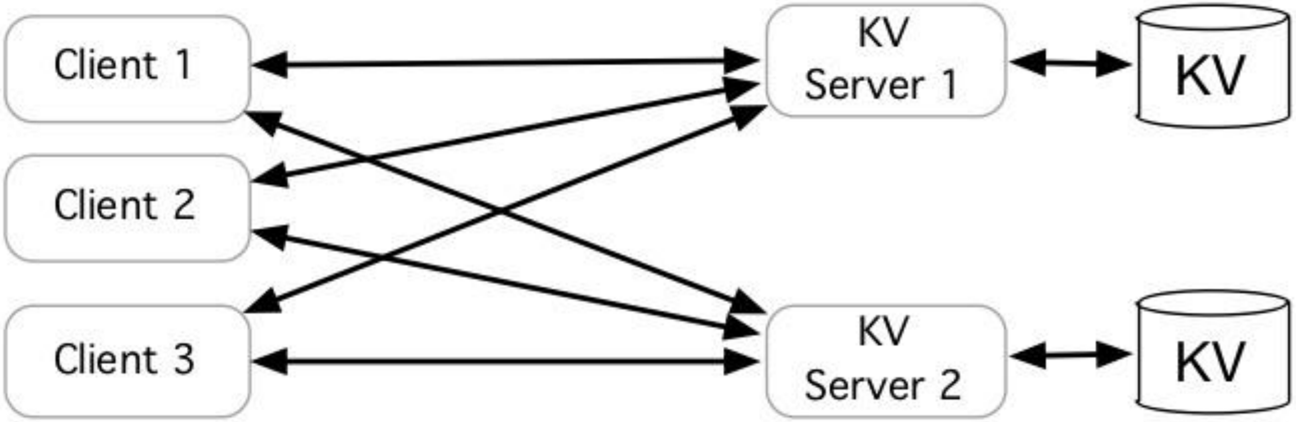


Performance results: Load Balancer

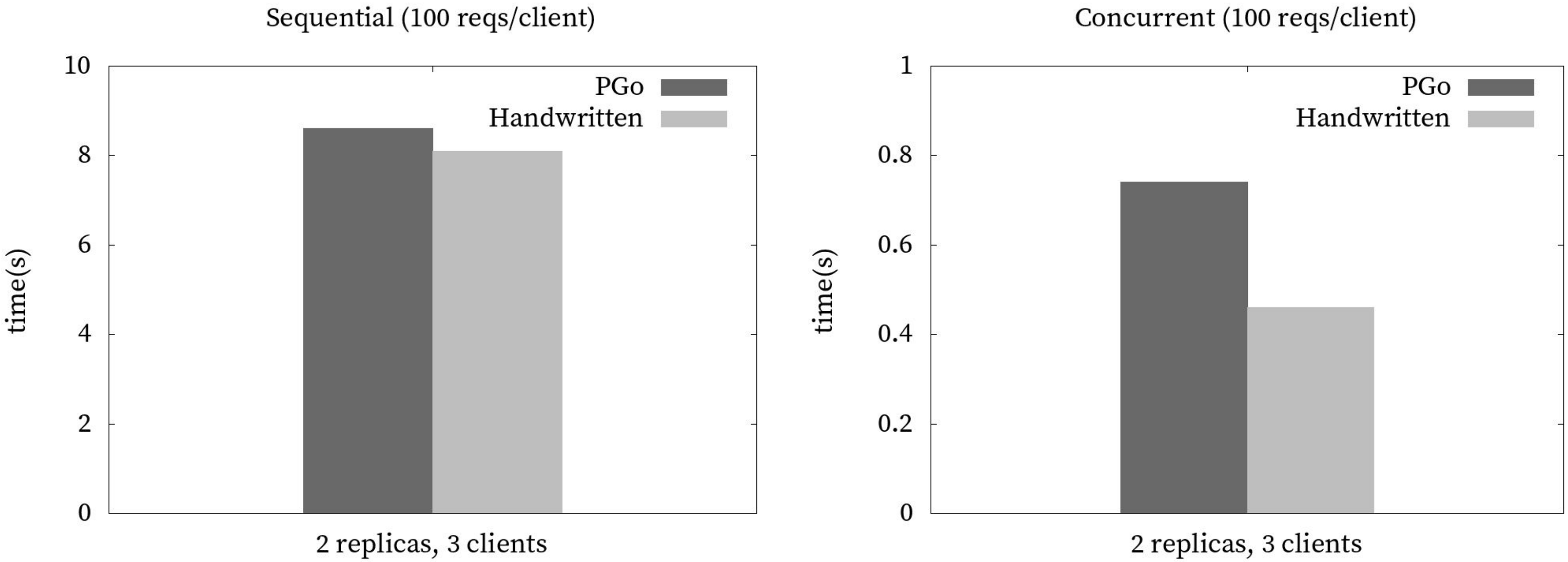


Load balancer with one or multiple clients performing 10 (left) or 100 (right) requests per client.

Replicated Key-Value Store setup



Performance results: Replicated Key-Value Store



Time it takes for three clients to perform 100 operations, first sequentially (left) and then concurrently (right).

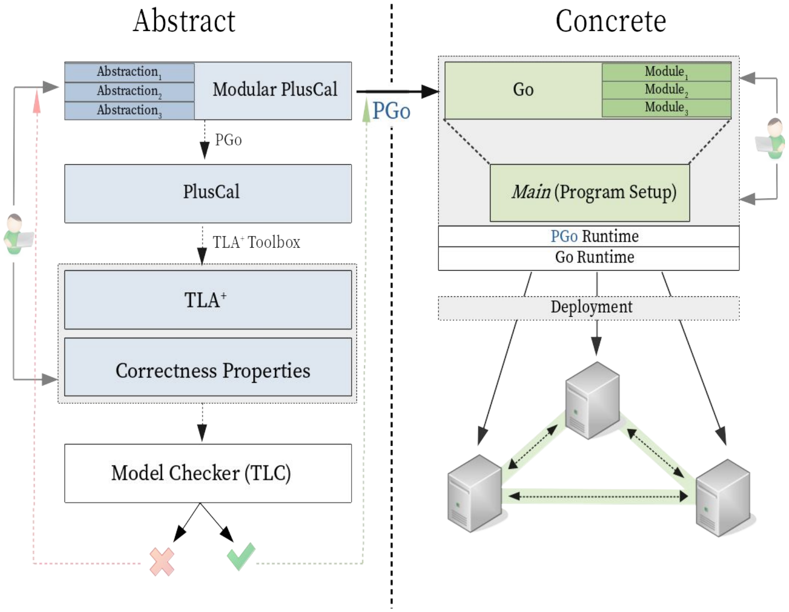
Discussion

Discussion: Limitations and Future Work

- Compilation is **not verified**
  - Trusted: **TLC** model checker, **PGo** compiler and runtime, **Java** compiler and runtime, **Go** compiler and runtime, **operating system**.
- **Fault tolerance** needs further work
  - Limited ways to deal with failures; lack of **language support**
- **Performance** can be improved
  - **Restarting** actions can be expensive
- **Fairness** is not guaranteed
  - Go favors performance over fairness; **mismatch** with original model

PGo take-aways

- ➔ Described how PGo leverages **separation** between **system** and **abstractions** to generate **correct** distributed systems
- ➔ More work is necessary to make it a viable option for the development of **production-quality** distributed systems



https://github.com/ubc-nss/pgo