Distributed State in PGo

CPSC 538B, Fall 2021

Shizuko Akamoto, Zack Grannan, Shayan Hosseini

Introduction[1]

Distributed systems are hard to reason about due to complex and asynchronous commutations among their parts. It is hard to consider every possible execution scenario when building a distributed system. Therefore, an uncovered execution scenario may result in severe bugs. For example, Amazon's Elastic Compute Cloud (EC2) hit a rare race condition that caused serious downtime for the service and took down major sites on the Internet [1, 6].

The concurrent and asynchronous nature of distributed systems makes it hard to reason about them. Designers have tried to use simplified specifications to model distributed systems and then reason about them. However, this approach introduces a gap between the simplified model and the real-world implementation. As a result of this gap, engineers have to fill in the missing pieces and potentially produce bugs in the system [4]. The process of making real-world implementations from simplified specifications is a manual error-prone process.

PGo [25] is a distributed system compiler, designed to help convert from formal, model-checkable specifications written in the Modular PlusCal [8] algorithm language into usable implementations written in the Go [23] programming language. A user writes her specification in MPCal and can model-check its correctness. After being sure about the correctness of the spec, the user can use PGo to compile the spec into implementation in the Go programming language.

A specification in MPCal consists of multiple processes that communicate via updates on the system's environment. The system's environment can be a low-level network channel or can be as high-level as a distributed shared variable. Modularity-oriented extensions to PlusCal, as the Modular PlusCal language, allow the user to separate the system's environment behavior from the algorithm being described. Then for the compiled implementation, PGo provides a collection of "resources" to replace modelled abstractions of a system's environment with a real interface to that environment, allowing the high-level specification logic to be used in practice.

Currently, PGo provides resources for low-level abstractions of the system's environment, such as network channels. However, there is no abstraction provided to be used as a high-level distributed shared state, for instance, a distributed shared variable. Having access to low-level resources such as network channels is sufficient for systems that use message passing for their communication, however, a system that relies on shared memory needs a higher level of abstraction. In fact, many distributed algorithms and concurrent systems have been built on top of shared memory. We present a new set of resources that can be used as the distributed

[1] First three paragraphs have been taken from Shayan's CPSC 508 report.

shared state. The trade-offs amongst consistency, performance, and fault-tolerance pose important challenges here. As we have seen [9, 20] before, it's not possible to have all these at the same time. Thus, we will provide different solutions for different requirements.

Background[2]

This work builds on top of multiple languages, tools, and concepts previously designed and built to solve specific problems. We begin by briefly outlining the most important ideas upon which this work is based. *

Model-checking is a method to determine whether an abstract model satisfies a formal specification. Models describe the possible system behavior in a mathematically precise manner. The model-checker explores all possible system states in a brute-force manner to find out that the specified model satisfies the required properties [2]. In case of a property violation, it can provide a counterexample that helps the system designer to debug the model specification. The main issue associated with this approach is the state space explosion problem. This issue is even more serious when modeling a distributed system, due to the high degree of concurrency and the need for exploring every possible execution interleaving [6]. *

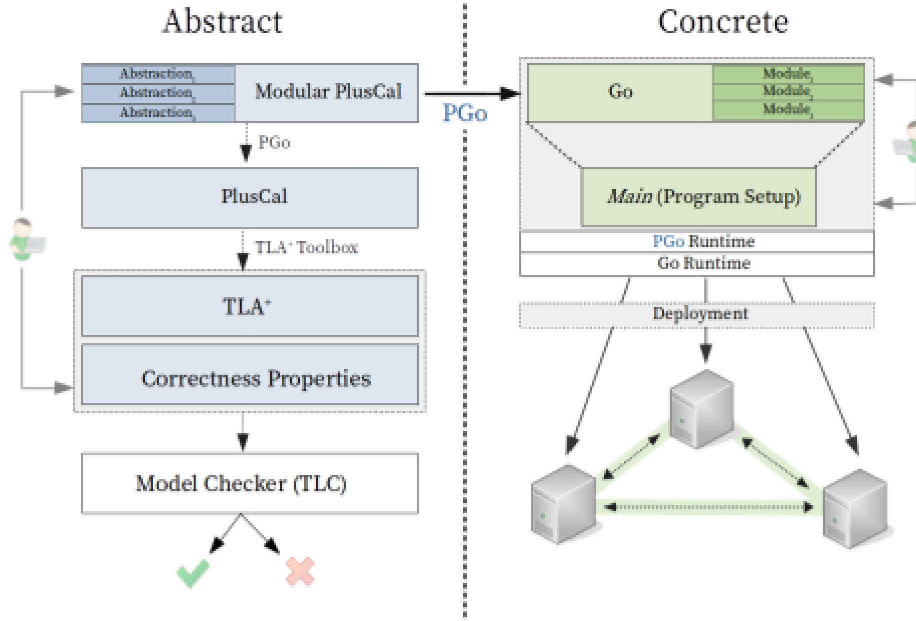
TLA [15, 17] is a high-level language for modeling programs and systems, especially concurrent and distributed ones. It has a simple, but powerful mathematical foundation [24]. *

The PlusCal algorithm language [16] is a pseudocode-like language that translates to TLA. The main goal of PlusCal is to be simpler and more familiar to the users since it uses a procedural style rather than TLA declarative style [8]. *

Modular PlusCal (MPCal for short) is a language built on top of PlusCal that provides modularity in its specifications. MPCal allows the users to separate abstract and implementation-dependent details in the specification. The PGo compiler then uses implementation-dependent parts of the specification to generate Go code. The PGo compiler uses the whole MPCal specification to compile to PlusCal, which next can be compiled to TLA and then model-checked to be verified with the required properties [8]. *

PGo [25] is a compiler from MPCal specifications to PlusCal specifications or Go implementations, as its workflow is depicted in figure below. PGo generates almost the entire Go implementation. The user only has to invoke generated functions with appropriate available resources provided by the PGo distributed runtime. These resources are part of the PGo distributed runtime, and they provide concrete implementation of abstractions in the system's environment. *

[2] Paragraphs with asterisk have been taken from Shayan's CPSC 508 report.



PGo workflow, taken from Renato Costa's Master's thesis [costa19].

To provide the mentioned separation between the algorithm being described and the system's environment, MPCal introduces the following two constructs:

Archetypes

Archetypes describe the behavior of the processes being specified. They are declared as such (example taken from PGo wiki [25]):

```
archetype Coordinator(connection)
variables local = 10, success = FALSE;
{
    l1: statement1;
    l2: statement2;
}
```

Each archetype consists of several labels (at least one). In the above example, archetype `Coordinator` has labels `l1` and `l2`. Each label will be executed atomically during model-checking. A user can change the degree of concurrency in a spec by adding or removing labels. The compiled Go implementation has to provide the same critical section semantics for labels. Assume that a resource r is shared between archetypes $a_1, a_2, \ldots, a_n$. If archetype a_i changes r in a label, then all $a_1, \ldots, a_n$ must agree on that change before execution of a_i's next label. In case of agreement the next label in a_i will be executed, otherwise, a_i reverts the change

and executes the previous label again. PGo distributed runtime uses an approach based on two-phase commit protocol to provide atomicity for each label. For more information, please see the design document PGo critical section model.

Mapping macro

Mapping macros allow developers to isolate the system's environment behavior from archetypes. As an example, suppose that we want to model a network channel that is both lossy and re-ordering. This behavior can be expressed using a mapping macro (example taken from PGo wiki [25]):

```
mapping macro LossyReorderingNetwork {
    read {
        with (msg \in $variable) {
            $variable := $variable \ msg;
            yield msg;
        }
    }

    write {
        either { yield $variable } or { yield Append($variable, $value) };
    }
}
```

Read and write operations in a mapping macro define what happens when the mapped variable is read and written to, respectively. `$variable` denotes the name of the variable being mapped and `$value` is the value being assigned to the mapped variable (only accessible in write operation).

The PGo compiler doesn't compile mapping macros in the implementation. It only replaces them with an abstract resource interface. Then the user should provide the appropriate concrete implementation by using the implementations available in the PGo distributed runtime.

Related Work

Other approaches for generating implementations from specification include the domain-specific language P [7], and Mace [14], a language for converting specification into a C++ implementation.

Alternative approaches ensure correctness by verifying that the implementation obeys the specification. Ironfleet [12] and Verdi [11] are frameworks for Dafny [18] and Coq [22], respectively. However, both approaches require the developer to add the appropriate proof steps to ensure the correctness of the implementation. In contrast, PGo generates an implementation automatically to implement the specification.

MoDIST [26] is another approach for verifying distributed systems via model checking. However, the approach suffers due to the state space explosion in real systems. In contrast, model checking PlusCal code generated by PGo is tractable due to the higher level of abstraction in the specification.

Maude transformer [19] takes a similar approach to PGo to compile models into implementations. Similar to PGo, it uses model checking to verify the spec. However, Maude doesn't have the same flexibility as PGo to model the system's environment. To define the environment's behavior, Maude provides a Sockets abstraction, which is quite similar to network channel resources in PGo, and it's not straightforward how it can be used to provide higher level abstractions, such as the distributed state in this work. Also, we hypothesize that PGo generated implementations will have a better performance than Maude because PGo compiles implementations to Go, but Maude transforms the formal model to an executable model in Maude language.

Approach

The proposed solution for the shared state depends on the deployment setting of the system and its requirements. The archetypes that run in the same OS process as different threads can have access to shared memory if mutual exclusion semantics are provided. However, if the archetypes are deployed in different nodes, they should communicate through a network to construct the shared state, which is a harder problem. It has been shown [9, 20] that there is a tradeoff between consistency, fault-tolerance and performance in distributed systems. For instance, strong consistency usually comes with a huge performance penalty. Consequently, we present different solutions for different requirements.

Ownership Algorithm

The ownership algorithm assigns a single owner to each variable in the global state. The ownership of a variable can be transferred to another node. This algorithm is based on the Emerald [13] system and has been previously implemented in PGo, but it has been removed during the recent rewrite. The ownership algorithm provides strong consistency, but it's not fault-tolerant (consider owner failure).

In our draft, we mentioned that we want to implement the ownership algorithm as a solution. There were two main reasons behind this decision: (1) we thought it would be easy to reuse the ownership algorithm implementation in the old PGo codebase; (2) we can use this as a baseline for our benchmarks. However, we realized that neither of these are quite true, and we decided not to work on the ownership algorithm anymore.

First, we analyzed the old implementation, which is available on GitHub. Almost all of the Go code there is related to the ownership algorithm and it involves complex reference ownership and borrowing. Due to huge API changes in the new version, we think that porting the old

implementation would not be easy and would require extensive changes to the old implementation.

Second, the ownership algorithm is optimized for the workloads that mostly one process reads and writes to the shared variable. However, the replicated two-phase commit approach (described in the next section) is optimized for the workloads that different processes heavily read the shared state. Basically, these two approaches are designed for different uses and not quite comparable on the same benchmark.

Two-Phase Commit (2PC)

Two-phase commit is a protocol that allows the user to implement distributed transactions. It provides strong consistency semantics that can be implemented in MPCal by using a trivial mapping macro. 2PC can tolerate node failures, but does not guarantee liveness.

2PC is useful for distributed computations that require strong consistency. For example, financial applications that process concurrent transactions must ensure that users cannot spend money they do not have, and that corresponding debit and credit transactions are implemented atomically. The strong consistency of 2PC, combined with transaction support provided by PGo, can ensure such an invariant because applications always observe the most up-to-date state.

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

    write { yield $value; }
}
```

Implementation of a mapping macro that can be realized by the 2PC archetype.

We replicate the shared state in all nodes, which have write access to the state, and coordinate the changes using the 2PC protocol. Implementing the two-phase commit protocol only requires implementation in the PGo distributed runtime. Since PGo critical section semantics have been built based on 2PC, implementing the shared state based on 2PC should be easier than implementing other solutions with the same guarantees.

The implementation of two-phase commit will be based on PGo critical section semantics and [10]. Our implementation of 2PC will allow any node in the system to act as a coordinator. To ensure that the critical sections on each node are serializable; the 2PC state synchronization between nodes must require critical sections to abort in some cases. For example, if node A completes a critical section that updates a 2PC resource R, then all other nodes currently in a critical section having already read R must abort their transactions to ensure that stale reads are never observed. This functionality will be enabled in the resource R itself: if it accepts state updates (or even pre-commits to them), then it will signal an abort whenever it is called from a local critical section.

In principle multiple coordinators could issue queries simultaneously; in which case that no commit will succeed (as different nodes pre-commit to the queries of different coordinators). We will investigate various approaches to solve this problem. One simple solution is to implement retry logic, where each coordinator waits a random amount of time before retrying. Alternatively, we could extend the 2PC protocol slightly, such that each node is assigned a unique numeric ID. When a node rejects a pre-commit from node B due to having previously pre-commited to node A; it includes the ID of node A in its message. When node B receives all the responses, it will receive a set S of nodes with conflicting pre-commits. Then, node B will wait until the set of nodes $\{ s \in S \mid s < B \}$ have successfully committed, prior to retrying the message.

Additionally, we will ensure that the 2PC implementation supports intermittent unavailability or crashes of replica nodes. In particular, a replica could crash at the following points:

1. **Before or during receipt of the pre-commit:** In this case, there is no issue, as the coordinator will continue re-trying the pre-commit until the node is online. Upon restart, if the node has already saved the pre-commit data to stable storage, it will simply acknowledge the request. Otherwise, it will save the data to stable storage and acknowledge.
2. **After accepting or rejecting the pre-commit, but before receiving a commit or rollback:** The replica will have recorded the pre-commit information into stable storage after acknowledging the pre-commit. When the replica restarts, it can restore its state from stable storage, and accept a subsequent commit or rollback
3. **After receiving a commit or rollback, but before sending an acknowledgement:** The coordinator will continue to retry the message. When the replica restarts, if the replica has already recorded the commit or rollback locally, it can simply acknowledge the subsequent message without performing any other action. Otherwise, it restores the pre-commit state from stable storage, then handles the commit or rollback message accordingly.

As is typical in 2PC implementations, liveness cannot be maintained if the coordinator goes offline during a pre-commit or commit stage. To handle cases of replica failure, the coordinator will assume that a replica has gone offline if it does not respond within a fixed time period. Offline replicas will be marked as dead, and ignored for subsequent requests. We may implement recovery logic so that if a node marked as dead comes back online, it can obtain the current state from a live node.

Go runtime API

Our 2PC implementation must implement the existing `ArchetypeResource` interface, which provides a common API for any resource used by MPCal archetypes. Internally, each node will maintain a variable "old" to keep track of the state before entering the critical section, and "current" to keep track of the current state of the variable.

```
ReadValue()         // Returns the value of the "current" variable. Abort the
                    critical section if a 2PC commit message has been received, or
```

```
                     if a 2PC pre-commit has been received without a corresponding
                     rollback.

WriteValue(value)    // Writes the value of the "current" variable. Abort the
                     critical section if a 2PC commit message has been received, or
                     if a 2PC pre-commit has been received without a corresponding
                     rollback.

PreCommit()          // Abort the critical section if a 2PC commit message has been
                     received, or if a 2PC pre-commit has been received without a
                     corresponding rollback. Otherwise, perform the 2PC precommit
                     phase, returning a failure if any node rejects the precommit or
                     if there is a timeout. Optionally, retry logic could be
                     implemented here, to handle cases when multiple coordinators are
                     attempting to commit transactions simultaneously.

Commit()             // Perform the 2PC commit, and put the value of the "current"
                     variable in the "old" variable.

Abort()              // Puts the value of "old" variable into the "current" variable
```

In addition, each node will also define callbacks for handling 2PC pre-commit, commit, and rollback messages.

```
ReceivePreCommit(value)     If this node has accepted a precommit message from
                            another node, or if PreCommit() has already been called
                            on this resource, then reject the message. Otherwise,
                            save the precommit data to durable storage and accept
                            the message. If the resource is in a local critical
                            section; calls to PreCommit(), ReadValue(), and
                            WriteValue() will abort the critical section unless
                            this node receives a rollback message.

ReceiveCommit()             Update the 2PC variable with the data from the saved
                            precommit data, if any. Send an acknowledgement. If the
                            resource is in a local critical section; calls to
                            PreCommit(), ReadValue(), and WriteValue() will abort
                            the critical section.

ReceiveRollback()           Clear the saved precommit data. Send an
                            acknowledgement.
```

We may also implement common optimizations, such as the presumed commit and presumed abort variations of the 2PC protocol.

We chose 2PC rather than consensus-based approaches, such as Paxos, because of two main reasons. First, it is much simpler to implement and reason about 2PC. Second, due to semantics of PGo critical sections, 2PC would have better performance. We don't formally prove this, however, intuitively show that if we use Paxos, then we need two rounds of consensus for commit or abort, which is less performant than one 2PC round. To show, we should know more

about the details of PGo critical section semantics. PGo tries to reach an agreement for all the resources that have been changed in a critical section. If all of them agree on commit, then it commits all the changes in the critical section, otherwise it aborts. Suppose that we have two changed resources named a and b in a transaction and a uses Paxos. Consider the case that a agrees to commit (by running Paxos) but b rejects. Now we have to abort the whole transaction that means aborting changes of both a and b. Therefore, a should run Paxos for the second time to abort the changes.

(Strong) Eventual Consistency with CRDTs

Strong consistency semantics provides guarantees about constant up-to-date state information on every replicated node, but at the cost of high latency and low availability. At some times, the weaker consistency semantics that eventual consistency provides may just be sufficient for the use of the application.

For instance, DNS is a well-known service employing eventual consistency that resolves its high availability requirements by sacrificing strong consistency. Domain name modifications take some time to propagate across all the nameservers, and during the propagation period, stale information may be returned. But this is generally considered acceptable to the client.

Another use case of eventual consistency is in real-time collaborative editing systems. Traditional editing systems using strong consistency semantics are not designed to scale, as the overhead from the involved protocols make real-time updates and collaboration experience poor. So strong consistency in real-time interactive systems comes with high cost, and so conflict-free replicated data types (CRDTs) come into play. CRDTs are a collection of data structure and operation properties such that if our data structure follows these properties, then data could be replicated across multiple nodes without divergence: even under problems such as network failures and out-of-order deliveries. Each CRDT has inherent, deterministic conflict resolution rules which is an advantage over other quorum-based or merge-based eventual consistency protocols that require special conflict-resolution mechanisms across participating nodes. Thus, we have chosen CRDTs for implementing PGo resources with eventual consistency semantics.

CRDT comes in two flavours: state-based convergent data types (CvRDTs) and operation-based commutative data types (CmRDTs). The equivalency of these two approaches has been proved in that CvRDTs can be used to emulate any CmRDT and vice-versa. In this project, we will focus on providing CvRDTs because they are generally simpler to reason about, and require less channel guarantees than its counterpart.

There are two parts to this solution: the MPCal specification of eventual consistency semantics and their corresponding Go runtime implementations. MPCal's trivial mapping macro reflects strong-consistency semantics, so we must provide a new mapping macro with eventual consistency. The underlying Go runtime implementation of eventually consistent resources will

be built with different CRDTs, for example, grow-only counters and add-wins-observed-remove sets.

Go runtime API

A CRDT must implement the existing `ArchetypeResource` interface, which provides a common API for any resource used by MPCal archetypes. That is, a CRDT will implement:

```
ReadValue()        // Return CRDT's current value.

WriteValue(value)  // Update CRDT with value, and buffer old state for potential
                   rollback.

PreCommit()        // Part of PGo's critical section semantics to signify it is
                   ready to commit. CRDT updates involve only its local state, so
                   this should always be yes.

Commit()           // Part of PGo's critical section semantics to definitely commit
                   the update. CRDT can discard the buffered old state at this
                   point.

Abort()            // Part of PGo's critical section semantics telling CRDT to
                   rollback due to sibling PreCommit failures and other spurious
                   aborts. Restore CRDT to the last committed state.
```

In addition to local state updates, state changes must be propagated to other nodes for eventual consistency. This is performed in the background by each node periodically broadcasting its last committed state.

The core of CvRDT is merging of local and received state, whereby a commutative, associative, and idempotent merge function finds the least upper bound over join-semilattice [21] (A comprehensive study of Convergent and Commutative Replicated Data Types). When a node receives a state from another node, it applies the `merge` function on the two states. This reconciles a node's local current state with the received state.

```
broadcast()

merge(other)
```

Because any CvRDT would have common implementations of `PreCommit`, `Commit`, and `Abort`, where old and new states are managed for potential rollback during critical sections, as well as `broadcast()`, which disseminates state to other peer nodes, we would like to provide an abstraction over these. Ultimately, we will be introducing a single additional CvRDT `ArchetypeResource` that accepts different `ReadValue`, `WriteValue`, and `merge` semantics implementing the different CvRDTs.

A node can experience intermittent failures and crashes, and our CvRDT state implementations must consider failure handling as well. In particular, our CvRDTs must persist to stable storage the last state update that has successfully been committed, otherwise potentially committed changes can be lost before broadcast. Another failure condition is when a node performing broadcast observes another node failing. In this case, the broadcasting node will simply retry at the next broadcast. This is safe and will not violate the eventual consistency semantics of CvRDT.

Finally, the CvRDT `ArchetypeResource` will also implement resource initialization and termination functions, where any connection and resource is set up/teared down as needed.

Here, we describe some CvRDTs we could provide as eventually consistent distributed states in PGo. Their designs reflect the corresponding CvRDTs discussed in [21]. This list is not exhaustive however, and we plan to support more advanced data types, such as an Add-Wins map, which utilizes the basic CRDTs discussed more in detail below.

Grow-only counter (GCounter)

GCounter is a monotonically increasing counter with applications for counting page views, number of likes. Each participating node maintains a vector of partial counts, indexed by its id. Operations on a node with a GCounter proceed as the following:

```
ReadValue    Returns the sum of all partial counts from the vector.

WriteValue   Increment the partial counts at index given by the node’s id.

merge        Iterate through the two vectors, resolving conflicts by
             applying max function.
```

The `max` operation satisfies all of commutativity, associativity, and idempotency required for a merge function.

Grow-only Set (GSet)

GSet is another basic CRDT that only supports addition of new elements. Each participating node keeps a local set.

```
ReadValue    Returns the current local set

WriteValue   Add element to the local set

merge        Apply the union function to local and received sets
```

Add-Wins Observed Remove Set (AWORSet)

AWORSet allows an element to be added and removed to/from it any number of times. Add-Wins dictates that when conflicting addition and removal of an element are performed concurrently by two nodes, the addition will be prioritized over the removal. To provide these semantics, each node keeps two maps keyed by the element with a timestamp as its value. The Add map records when the last add time for each element, while the Remove map records the last remove times. We must be able to partially compare these timestamps, so vector clocks can be used.

```
ReadValue    Return the key set of Add map, excluding those where Add
             timestamp < Remove timestamp.

WriteValue   Add -> Add the element’s entry into Add map, incrementing the
             node’s clock in the vector clock.
             Remove -> Add the element’s entry into Remove map, incrementing
             the node’s clock in the vector clock.

merge       1. Merge the corresponding Add and Remove maps, resolving
               timestamp conflicts using vector clock’s merge semantics as
               in [3]. This results in the Add-Wins property.
            2. Remove from merged Add map, any entry with timestamp <
               timestamp of corresponding Remove map entry. This converges
               the removed elements.
            3. Remove from Remove map, any entry with timestamps < or ||
               timestamp of corresponding Add map entry. This converges the
               added elements.
```

Set addition and removals also conform to commutativity, associativity, and idempotency.

CRDT behavior in MPCal

If users use CRDT in a spec, they should be able to verify that their system works correctly with CRDT guarantees, which is weaker than strong consistency semantics. As mentioned above, CRDTs provide strong consistency semantics that formally means satisfying the following properties:

- **Eventual delivery.** All replicas eventually deliver the same set of updates.
- **Termination.** All method executions terminate.
- **Strong convergence.** Correct replicas that have delivered the same updates have equivalent state.

Therefore, expressing CRDTs in MPCal, should provide the exact above properties. To present our approach, let's consider the state-based grow-only counter. If all the nodes deliver the same set of updates, then by using the merge function for grow-only counters the strong convergence property will be satisfied. Termination also can be easily satisfied by appropriate modeling. So the main challenge is expressing eventual delivery property in MPCal. To demonstrate our solution, consider a simple example. Each node has access to a local counter that is actually an

array that grow-only counter semantics are applying to it. There is a new archetype in the system (`AGlobalGCntr`) that, in each step, picks two nodes that don't have the same state and merges their state. It's easy to see that running this archetype concurrently with the rest of the system, will produce every possible way of eventual delivery. The model checking performance can be improved if one uses `CHOOSE` statement rather than `with` statement, although this won't explore all the possible executions but still is a good approximation. Note that the following code snippet doesn't use the MPCal exactly and has some simplifications in the syntax.

```
mapping macro LocalGCntr {
    read {
        yield Sum($variable);
    }

    write {
        assert $value > 0;
        yield [$variable EXCEPT ![self] = $variable[self] + $value]; \* when a
process increases the counter, it increases the element with its id in the array
    }
}

archetype AGlobalGCntr(ref localcntrs)
{
l1:
    while (TRUE) {
        with (i1, i2 \in NODE_SET: localcntrs[i1] # localcntrs[i2]) {
            Merge(localcntrs[i1], localcntrs[i2]);
        };
    }
}
```

Let's consider the following simple application. There are N nodes having access to a shared grow-only counter. First, a node increments the counter, and then waits for all the updates to propagate, i.e. waiting until the value of the counter equals to N, before deciding to increment again. The following is the specification of the nodes in this example. Note that updating and retrieving the `cntr` value follows the `read` and `write` semantics as defined in `LocalGCntr` mapping macro.

```
archetype ANode(ref cntr[_]) {
l1:
    cntr[self] := 1; \* increasing counter by 1
l2:
    await cntr[self] = N;
}
```

Evaluation

We will answer the following research questions as part of our evaluation.

***RQ1**: How efficient is the compiled Go implementation of the systems built using the shared state?*
We will compare the distributed key-value store against systems implemented using etcd, a distributed key-value store and an existing distributed key-value store in PGo that doesn't use the shared state. We will measure latency and throughput for comparison using the YCSB benchmark [5]. For the distributed shopping cart, we will compare the CRDT version with the 2PC version by measuring latency and throughput.

***RQ2:** How much model-checking overhead is added?*
For the 2PC resource, there is no model-checking overhead because PlusCal and TLA+ use strong consistency semantics by default. For CRDTs, we will compare the model-checking performance of shared counter and distributed shopping cart spec using CRDT and without CRDT.

***RQ3:** Does the implementation satisfy its consistency guarantees in practice?*
We will use assertions during execution of compiled Go implementation and end-to-end tests to ensure that the implementation for each test is consistent with the specification. For example, in the distributed lock case, each node can output the timestamps for when it has obtained the lock, and we can inspect the timestamps to ensure that no nodes obtained the lock simultaneously.
Similarly, a CRDT-based application will include assertions at each merge that the state is monotonically increasing, and finally, the states are all equal across the nodes, and to the greatest element of the join-semilattice.

***RQ4:** Does the implementation satisfy its fault-tolerance semantics in practice?*
We will evaluate the fault-tolerance semantics by injecting faults, and ensuring that the end-to-end tests pass.

The architecture of PGo enables applications to be defined orthogonally to the implementation of distributed state; this ensures that differences in performance between the implementations are due to differences in the distributed state implementations rather than application logic.

We will evaluate the performance on both a local and network environments. In the local setting, each node in the system will be implemented as a Goroutine[3] running concurrently in the same process. The network setting will be implemented via different computers on the same network. During evaluation, we will emulate faults by either including them in the test application directly (for example, introducing latency, exiting the process, ignoring messages etc), or at the operating-system / network level.

[3] Goroutines are concurrency primitives that are managed by the Go runtime, analogous to threads.

Test Applications

We will test using the following applications, and develop specifications for each application to ensure that each application meets the spec.

Distributed Counter
Nodes will iterate a shared variable in a round-robin manner. To benchmark, we will run the distributed application until the counter reaches a certain value. A counter can be implemented with both 2PC and CRDT resources.

Distributed Lock Service
This will ensure that at most one node can enter the critical section at a time. To benchmark the performance of the service, we will run an experiment where nodes take turns obtaining a lock to increment a shared variable. The experiment will end after the variable reaches a fixed value.

Distributed Key-Value Store
PGo already has implemented a distributed key-value store that implements distributed state on the application level. Re-implementing this key-value store using shared memory will provide an intuition about the efficiency of the shared memory implementation.

Distributed Shopping Cart
Nodes can concurrently add and remove items to a cart modelled as a set. This can be implemented using both CRDT and 2PC-based states. Then performance from these implementations can be benchmarked and compared against, which will tell us about the cost of consistency.

Milestones

Milestone 1: October 29

- Working implementation of 2PC, without failure handling or retry functionality.
- Simple application to demonstrate functional 2PC ⇨ working shared counter systems.

- Working implementation of CRDT GCounter with associated tests in PGo runtime.
- Complete MPCal specification of CRDT GCounter ⇨ with model check result

Milestone 2: November 15

- Implement failure handling for 2PC, with associated tests:
    - Retry in case of coordinator conflict
    - Replicas can recover from failures after pre-commit

We will demonstrate the failure handling in the shared counter system, by injecting faults.

- Working implementation of CRDT AWORSet and associated tests
- Complete MPCal specification of AWORSet ➪ with model check result
- Simple application to demonstrate GCounter ➪ eventually consistent shared counter system.

Milestone 3: November 29

- Complete all test applications for 2PC
- Initial benchmark results for 2PC test applications

- Application to demonstrate AWORSet ➪ eventually consistent distributed shopping cart
- Benchmark results from CRDT-based applications

Timeline

	2PC	**CRDT**
Week 1 (Oct 4 - Oct 8)		• Begin development of G-counter CRDT in Go
Week 2 (Oct 11 - Oct 15)	• Submit Final Project Proposal	
Week 3 (Oct 18 - Oct 22)	• Begin development of 2PC Go Implementation	• Complete GCounter CRDT and tests
Week 4 (Oct 25 - Oct 29)	• Complete Prototype 2PC Go Implementation • Implement shared counter spec • Get a working shared counter system using 2PC	• Get an eventually consistent shared counter using GCounter • Implement eventually consistent shared counter spec in MPCal
Complete 1st Project Milestone		
Week 5 (Nov 1 - Nov 5)	• Implement retry for 2PC for coordinator conflict case	• Begin development of AWORSet in Go • Complete shared counter application using GCounter
Week 6 (Nov 8 - Nov 12)	• Implement replica handler failure logic • Demonstration of failure handling and retry logic on shared counter system	• Complete AWORSet in Go • Implement eventually consistent shared set spec in MPCal • Complete shopping cart application using AWORSet

<table>
<tr><th colspan="3">Complete 2nd Project Milestone</th></tr>
<tr><td>Week 7 (Nov 15 - Nov 19)</td><td>● Start development of distributed lock service</td><td>● (Stretch) Begin AWMap in Go</td></tr>
<tr><td rowspan="2">Week 8 (Nov 22 - Nov 26)</td><td>● Finish lock service implementation</td><td>● (Stretch) Complete AWMap in Go</td></tr>
<tr><td colspan="2">● Begin project presentation
● Gather initial benchmark results on test applications</td></tr>
<tr><th colspan="3">Complete 3rd Project Milestone</th></tr>
<tr><td>Week 9 (Nov 29 - Dec 3)</td><td colspan="2">● Complete Project Presentation
● Complete evaluation</td></tr>
<tr><td>Week 10 (Dec 6 - Dec 10)</td><td colspan="2">● Work on Project Report</td></tr>
<tr><td>Week 11 (Dec 13 - Dec 17)</td><td colspan="2">● Work on Project Report</td></tr>
<tr><td>Week 12 (Dec 20 - Dec 21)</td><td colspan="2">● Complete Project Report</td></tr>
</table>

References

[1] Amazon 2008. Amazon S3 Availability Event. (2008).
[2] Baier, C. and Katoen, J.-P. 2008. *Principles of Model Checking (The MIT Press)*.
[3] Baldoni, R. and Klusch, M. 2002. Fundamentals of Distributed Computing: A Practical Tour of Vector Clock Systems. *IEEE Distributed Systems Online*. 3, 2 (2002).
[4] Chandra, T.D. et al. 2007. Paxos made live: an engineering perspective. *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing - PODC ’07*. (2007), 398–407. DOI:https://doi.org/10.1145/1281100.1281103.
[5] Cooper, B.F. et al. 2010. Benchmarking cloud serving systems with YCSB. *Proceedings of the 1st ACM symposium on Cloud computing - SoCC ’10*. (2010), 143–154. DOI:https://doi.org/10.1145/1807128.1807152.
[6] Costa, R.M. 2019. *Compiling Distributed System Specifications into Implementations*.
[7] Desai, A. et al. 2013. P: Safe Asynchronous Event-Driven Programming. *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2013), 321–332.
[8] Do, M.N. 2019. *Corresponding formal specifications with distributed systems*.

[9] Gilbert, S. and Lynch, N. 2012. Perspectives on the CAP Theorem. *Computer*. 45, 2 (2012), 30–36. DOI:https://doi.org/10.1109/mc.2011.389.
[10] Gray, J. n.d. Notes on Data Base Operating Systems. *Operating Systems, An Advanced Course* (Berlin, Heidelberg, n.d.), 393–481.
[11] Grove, D. et al. 2015. Verdi: a framework for implementing and formally verifying distributed systems. *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. (2015), 357–368. DOI:https://doi.org/10.1145/2737924.2737958.
[12] Hawblitzel, C. et al. 2017. IronFleet: Proving Safety and Liveness of Practical Distributed Systems. *Commun. ACM*. 60, 7 (Jun. 2017), 83–92. DOI:https://doi.org/10.1145/3068608.
[13] Jul, E. et al. 1988. Fine-grained Mobility in the Emerald System. *ACM Trans. Comput. Syst.* 6, 1 (1988), 109–133. DOI:https://doi.org/10.1145/35037.42182.
[14] Killian, C.E. et al. 2007. Mace: language support for building distributed systems. *ACM SIGPLAN Notices*. 42, 6 (2007), 179–188. DOI:https://doi.org/10.1145/1273442.1250755.
[15] Lamport, L. 2002. *Specifying Systems*. Addison-Wesley Longman Publishing Co., Inc.
[16] Lamport, L. 2009. 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.* (2009).
[17] Lamport, L. 1994. The Temporal Logic of Actions. *ACM Trans. Program. Lang. Syst.* 16, 3 (1994), 872–923. DOI:https://doi.org/10.1145/177492.177726.
[18] Leino, K.R.M. 2010. Logic for Programming, Artificial Intelligence, and Reasoning, 16th International Conference, LPAR-16, Dakar, Senegal, April 25–May 1, 2010, Revised Selected Papers. Lecture Notes in Computer Science. (2010), 348–370. DOI:https://doi.org/10.1007/978-3-642-17511-4_20.
[19] Liu, S. et al. 2020. NASA Formal Methods, 12th International Symposium, NFM 2020, Moffett Field, CA, USA, May 11–15, 2020, Proceedings. Lecture Notes in Computer Science. (2020), 22–40. DOI:https://doi.org/10.1007/978-3-030-55754-6_2.
[20] Ousterhout, J.K. 1991. The Role of Distributed State. (1991).
[21] Shapiro, M. et al. 2011. Stabilization, Safety, and Security of Distributed Systems, 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings. Lecture Notes in Computer Science. (2011), 386–400. DOI:https://doi.org/10.1007/978-3-642-24550-3_29.
[22] Team, T.C.D. 2021. The Coq Proof Assistant. (2021). DOI:https://doi.org/10.5281/zenodo.4501022.
[23] The Go Programming Language: n.d. *https://golang.org/*. Accessed: 2021-10-05.
[24] The TLA+ Home Page: n.d. *https://lamport.azurewebsites.net/tla/tla.html*. Accessed: 2021-04-27.
[25] UBC-NSS/pgo: PGo is a source to source compiler from Modular PlusCal specs into Go programs.: n.d. *https://github.com/UBC-NSS/pgo*. Accessed: 2021-10-05.
[26] Yang, J. et al. n.d. MODIST: Transparent Model Checking of Unmodified Distributed Systems. *NSDI* (n.d.).