Distributed State in PGo

SHIZUKO AKAMOTO, University of British Columbia, Canada
ZACK GRANNAN, University of British Columbia, Canada
SHAYAN HOSSEINI, University of British Columbia, Canada

Distributed systems are known to be hard to build correctly. As a result, scientists and practitioner started to use formal methods to verify design and implementation of such systems. PGo is a compiler that compiles specifications of distributed systems into executable implementations. PGo allows users to model check their specifications against the desired properties. Although several complex distributed systems has been built by using PGo, but all of them have used message passing as their environment. Shared memory as the other major environment of building distributed systems has not been studied.

We present a new approach to build distributed systems using shared memory paradigm with PGo. In response to fundamental trade offs in building distributed systems, we encorporate an improved version two-phase commit protocol and CRDT objects to provide solutions with different consistency guarantees. Our evalution demonstrates promising results. We built, model checked and evaluated performance of several applications such as a distributed set and a distributed key value store.

1 INTRODUCTION

Distributed systems are hard to reason about due to complex and asynchronous interactions 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 [Amazon 2011; Costa 2019].[1]

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,potentially introducing bugs into the system [Chandra et al. 2007]. The process of making real-world implementations from simplified specifications is a manual error-prone process.[1]

PGo [PGo Developers 2021] is a distributed system compiler, designed to help convert from formal, model-checkable specifications written in the Modular PlusCal [Do 2019] algorithm language into usable implementations written in the Go [Go Developers 2009] programming language. A user writes a 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.[1]

A specification in MPCal consists of multiple processes that communicate via updates on *resources*. Resources represent the system's environment. A resource can be as low-level network channel, as in message passing systems, or can be as high-level as a distributed shared variable, as in shared memory systems. Modularity-oriented extensions to PlusCal, such 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 concrete implementation 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 resource implementations for low-level abstractions of the system's environment, such as network channels. However, there is no abstraction provided for high-level distributed shared state, such as distributed shared variables. 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 requires a higher level of abstraction. In fact, many distributed algorithms and concurrent systems have been built on top of shared memory.

In this paper, we present a new set of resources that can be used for distributed shared state in PGo. As it is well-known that no distributed system can satisfy consistency, performance, and fault-tolerance properties [Gilbert and Lynch 2012; Ousterhout 1991], we provide implementations for both strong and eventually consistent shared state. MPCal modularity enables us to model check systems with respect to guarantees of each resource.

2 BACKGROUND

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.[1]

Model-checking is a method to determine whether an abstract model of a system satisfies a formal specification. A model describes possible system behavior in a mathematically precise manner. A model-checker explores all possible system states to determine if the specified model satisfies the required properties [Baier and Katoen 2008]. In case of a property violation, a model-checker 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; the state space can be extremely large, making an exhaustive search intractable. 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 [Costa 2019].[1]

Authors' addresses: Shizuko Akamoto, shizuko@cs.ubc.ca, University of British Columbia, Vancouver, Canada; Zack Grannan, zgrannan@cs.ubc.ca, University of British Columbia, Vancouver, Canada; Shayan Hosseini, sshayanh@cs.ubc.ca, University of British Columbia, Vancouver, Canada.

[1]Paragraph has been taken from Shayan's CPSC 508 report.

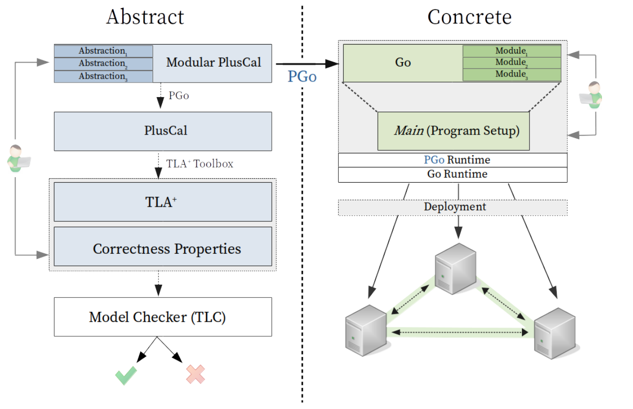

Fig. 1. PGo workflow, taken from Renato Costa's Master's thesis [Costa 2019].

TLA+ [Lamport 1994, 2002] is a high-level language for modeling programs and systems, especially concurrent and distributed ones. It has a simple, but powerful mathematical foundation [TLA Developers 2021].[1]

The PlusCal algorithm language [Lamport 2009] 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+'s declarative style [Do 2019].[1]

Modular PlusCal (MPCal for short) is a language built on top of PlusCal that enables modular specifications, allowing users to separate abstract and implementation-dependent details. The PGo compiler [PGo Developers 2021] then uses implementation-dependent parts of the specification to generate Go code. PGo converts the entire MPCal specification into PlusCal, which next can be compiled to TLA+ and then model-checked to be verified with the required properties [Do 2019].

Figure 1 depicts PGo's workflow. 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.[1]

To enable separation between the algorithm and the system's environment, MPCal introduces the two constructs, which we describe in the following sections.

2.1 Archetype

Archetypes describe the behavior of the processes being specified. Figure 2 shows an example archetype.

An archetype has a list of arguments that defines the resources it has access to. These resources represent the system's environment and their behavior can be defined by mapping macros (Sec. 2.2). For example, `connection` is a resource in archetype `Coordinator`.

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

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

Fig. 2. An archetype in MPCal, taken from PGo wiki [PGo Developers 2021].

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

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

Fig. 3. A mapping macro in MPCal, taken from PGo wiki [PGo Developers 2021].

labels. The compiled Go implementation provides 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, which is described in Section 2.3.

2.2 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 the mapping macro in Figure 3.

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 the write operation).

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

`ReadValue() → (Value, error)`	Read the current value of the resource.
`WriteValue(v: Value) → error`	Sets the value of the resource to `v`.
`PreCommit() → error`	Determines if it is fine to go do a `Commit`.
`Commit()`	Unconditionally commits current state.
`Abort()`	Resets to the last `Commit`.

Fig. 4. PGo's `ArchetypeResource` API

```
archetype Send(ref network[_]) {
send:
    network[dest] := "hello";
}
```

(a) Using a network channel as a resource.

```
archetype Increase(ref counter) {
inc:
    counter := counter + 1;
}
```

(b) Using a shared counter as a resource

Fig. 5. Archetypes using different resources in MPCal.

2.3 PGo Resource API

PGo uses an API based on the two-phase commit protocol (2PC) to implement critical section semantics. Resources in the PGo distributed system runtime are required to implement this interface (named `ArchetypeResource`). The `ArchetypeResource` interface is shown in Figure 4.

3 DISTRIBUTED STATE

MPCal archetypes represent concurrent running processes that use resources to communicate and synchronize. The user determines the behavior of the resources. In the MPCal their behaviors are defined by mapping macros and they are not compiled to Go implementation. Instead, their implementations are provided by the PGo distributed runtime.

Figure 5 shows two different archetypes. `Send` (Figure 5a), is simple message passing system that uses a network channel to a message. The behavior of network will be defined by a mapping macro. For example, we can use the mapping macro in Figure 3 to do this. The PGo distributed runtime provides implementations for network channels with different guarantees. `Increase` (Figure 5b), resembles a shared memory system that uses a shared counter. Although, several complex systems has been built in PGo, all of them were implemented using message passing; not much work has been done for thein a shared memory environment.

The ideal implementation of shared memory 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 the local memory if mutual exclusion semantics are provided. However, if the archetypes are deployed on different nodes, they must communicate through a network to construct the shared state. In this work we focus on distributed setting.

It has been shown [Gilbert and Lynch 2012; Ousterhout 1991] that there is a tradeoff between consistency, fault-tolerance and performance in distributed systems. For instance, implementing a system with strong consistency semantics usually entails a large performance penalty compared to an eventually consistent implementation. Consequently, we present different solutions for different requirements. First, we present a solution based on the two-phase commit protocol (2PC) for workloads that require strong consistency. Then we propose another solution based on conflict-free replicated data types (CRDT) that relaxes consistency semantics and allows having a better performance when there is no need for strong consistency. We also show that how we build MPCal models that satisfy the exact guarantees of each resource. This allows us to use model-checking to ensure that the resulting systems work correctly in the desired environment.

4 STRONG CONSISTENCY WITH TWO-PHASE COMMIT

Two-phase commit is a well-known protocol for implementing atomic state transitions. In the protocol, a *coordinator* attempts to initiate a state transition by sending a PRECOMMIT message to all other nodes. Each node replies with either an ACCEPT or REJECT message. If any node rejects the message, then the coordinator must send a ROLLBACK message to the other nodes; otherwise, if all nodes have accepted the message, the coordinator sends a COMMIT message to all other nodes to complete the state transition. 2PC can tolerate temporary (but not permanent) node failures. In particular, the coordinator cannot perform a state transition if any node is unresponsive.

2PC enables 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 semantics enabled by 2PC, combined with transaction support provided by PGo, can ensure such an invariant.

We chose 2PC rather than other well-known approaches, such as Paxos [Lamport et al. 2001], because it is simpler to implement and reason about. Although we have extended the protocol to support multiple coordinators (Sec. 4.1) and fault-tolerance (Sec. 4.2), the design is still less complex than other approaches.

Our 2PC implementation replicates the shared state amongst all nodes, which have write access to the state, and coordinates the changes using the 2PC protocol. Expressing strong consistency in an MPCal model is trivial, since every shared variable in the model satisfies strong consistency semantics by default. To build models that works in a shared memory system with strong consistency semantics, we simply provide the same variable (resource) to the various archetypes.

Our implementation of 2PC is based on PGo critical section semantics and the original paper [Gray 1978]. In our implementation, any node in the system is allowed to act as a coordinator. Additionally, our use case for PGo entails two requirements that are not defined by the 2PC protocol. First, since any node may wish to initiate a state transition, there must be some way to handle the case when two or more nodes request to update the state simultaneously. Second, it's quite possible that a node could go offline permanently; the system should continue to make progress in the presence of such failures. We now describe how we support these requirements in our implementation.

4.1 Coordinator Conflict

To understand the challenge posed by coordinator conflict, consider a set of nodes $n_1, \ldots, n_j$, where both n_1 and n_2 desire to update the state simultaneously. Both n_1 and n_2 send a PRECOMMIT message to all other nodes. For a state transition to occur, either n_1 or n_2 must receive an ACCEPT response from all other nodes. However, if the messages were sent simultaneously, then this is unlikely: for each other node it's possible to receive either message first. Therefore, the most likely scenario is that neither n_1 nor n_2 are able to commit their state transition, and must send a ROLLBACK message to all participants.

To address this issue, a traditional approach is to develop a leader election protocol, where a node is designated to act as a central coordinator. However, this introduces significant complexity to the protocol, as essentially it requires a separate consensus mechanism to ensure that all nodes agree on the identity of the leader.

Instead, we choose a different, simpler strategy: in cases of coordinator conflict, each node waits for a duration of time and retries the request later. Crucially, randomization is used to reduce the likelihood of two nodes waiting the same amount of time between attempts, and nodes wait longer after consecutive failures. This strategy practically ensures that some node will eventually be able to successfully commit.

4.2 Fault-Tolerance

In the traditional 2PC protocol, an offline replica will prevent any state transition. This is because, as the coordinator will never receive a response to a PRECOMMIT message sent to an unresponsive replica, it will never know if it is safe to move forward with a commit.

To address this issue, we alter the semantics of the 2PC protocol, such that the coordinator can proceed to the commit phase if it has received an ACCEPT message from at least half of the other nodes. Intuitively, this preserves strong consistency because it requires that a majority of the nodes have agreed on the new value. However, making this change involves quite a few changes to ensure that the protocol remains safe.

The key adjustment to the protocol is that each phase of the 2PC protocol is allowed to complete once an affirmative response has been received from a majority of the nodes. As a result, no single node can observe the current value, as the system can update the state without informing the node. Therefore, the consistency of the system is defined in terms of a *version*, a natural number that represents the number of state transitions in the system.

4.3 Formal Protocol Definition

To incorporate the above-mentioned changes, we define the protocol formally in terms of the following definitions.

Definition 4.1. A *2PC request* is a tuple $\langle t, v, s, \text{VER} \rangle$, where $t \in \{\text{COMMIT}, \text{PRECOMMIT}, \text{ROLLBACK}\}$. v is the the new value to be committed, s is the sender of the message, and VER is the version of the new message.

Definition 4.2. A *2PC response* is a tuple $\langle t, v, \text{VER} \rangle$, where $t \in \{\text{ACCEPT}, \text{REJECT}\}$. v and VER are the value and version of the replica at the time it received the corresponding request.

Definition 4.3. A *node state* is a tuple $\langle r, v, \text{VER}, R \rangle$. r is the currently accepted precommit, or NULL, v represents the current value, VER is a natural number representing the current version, and R is a list of replica addresses.

This change in the protocol introduces two additional considerations for replicas in the system. First, they may receive a message with an unexpected version, and second, they may receive COMMIT and ROLLBACK messages without having first received a corresponding PRECOMMIT. These considerations are addressed as follows:

(1) If the replica receives any message with a version that is not greater than its own, it rejects the message and sends its current version and value to the sender, so that the sender can update to the current version.
(2) If the replica receives a PRECOMMIT message at a version higher than the PRECOMMIT it has currently accepted, then it will accept the message.
(3) If the replica receives a COMMIT or ROLLBACK message, it will accept the message (in the case of ROLLBACK, only changing its internal state if the message corresponds to an accepted PRECOMMIT).

Figure 6 presents pseudocode attempting state transitions with the modified 2PC semantics. `broadcast` is a higher-order function that takes a function $f : R \rightarrow$ `bool` as input, and runs f simultaneously on each replica address. `broadcast`(f) returns **true** if f returns **true** for at least half of the replicas, and **false** if f returns **false** for more than half of the replicas, returning as soon as it can make this determination (each call to f that has not yet returned is allowed to continue).

`sendPreCommit` sends a single message to each replica, returning **true** if the replica replies with an ACCEPT, and returning **false** if the replica rejects the message, or if a network error or timeout occurred. If a replica has a higher version than the sender, the sender will adopt that replica's value and version.

The precommit will be successful if at least half of the nodes accept the PRECOMMIT message (meaning that their version is not greater than the sender, and they have not accepted or initiated another precommit), and the sender itself has not updated to a new version since initializing the precommit (for example, in the case where another replica is at a higher version). In `sendPhase2`, the node sends either a COMMIT or ROLLBACK message accordingly. Crucially, although the state transition attempt is complete after receiving a response from half of the nodes, the message will continue to be resent to the other nodes until they have acknowledged the request, or the sender reaches a higher version. This is necessary to ensure that any replica that has accepted a PRECOMMIT will

```
attemptStateTransition($v$) = {
  $\textsc{ver} \longleftarrow s.\textsc{ver}$
  $m \longleftarrow \langle \textsc{precommit}, v, \textsc{self}, \textsc{ver} + 1 \rangle$
  if (broadcast($r \longrightarrow$ sendPreCommit($r, m$)) $\wedge$ $\textsc{ver} = s.\textsc{ver}$) {
    $s.v \longleftarrow v$
    $s.\textsc{ver} \longleftarrow \textsc{ver} + 1$
    $m' \longleftarrow \langle \textsc{commit}, v, \textsc{self}, \textsc{ver} + 1 \rangle$
  } else {
    $m' \longleftarrow \langle \textsc{rollback}, \textsc{null}, \textsc{self}, \textsc{ver} + 1 \rangle$
    broadcast($r \longrightarrow$ sendPhase2($r, m'$))
  }
}
```

Fig. 6. Pseudocode for state transition. s is the state of the node attempting the state transition, v is the value it wishes to assign.

receive either a higher-version message, or a corresponding COMMIT or ROLLBACK.

Formally, we define our consistency guarantee as follows:

PROPERTY 4.1 (VERSIONED STRONG CONSISTENCY). *For any two nodes* n_1, n_2 *with states* $\langle r_1, v_1, \textit{VER}_1, R_1 \rangle, \langle r_2, v_2, \textit{VER}_2, R_2 \rangle$ *respectively,* $\textit{VER}_1 = \textit{VER}_2 \implies v_1 = v_2$

Although we have not formally proven that our implementation ensures this property, it has not been violated in practice.

5 EVENTUAL CONSISTENCY WITH CRDT

Strong consistency semantics provide guarantees about the order of operations on shared states seen by all replica nodes, but at the cost of high latency and low availability. At some times, the weaker consistency semantics 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. Strong consistency in real-time interactive systems comes with high cost, and so conflict-free replicated data types (CRDTs) come into play. CRDTs [Shapiro et al. 2011b] are a collection of data structures and operation properties such that if our data structure follows these properties, then data could be replicated across multiple nodes without divergence: even under unreliable networks. 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 phase across participating nodes. Thus, we have chosen CRDTs for implementing PGo's eventual consistent resources.

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 [Shapiro et al. 2011b] that CvRDTs can be used to emulate any CmRDT and vice-versa. For PGo, we will focus on providing CvRDTs due to its simplicity, and less requirements on the underlying network. CvRDTs lift the complexity of conflict resolution mechanisms into the states themselves, thereby allowing us to separate state management logic from other workings of the replica node, including inter-node communication and failure handlings. In the rest of this paper, we use the term CRDT and CvRDT interchangeably.

There are two parts to this solution: the MPCal specification of eventual consistency semantics and their corresponding Go runtime implementations. MPCal's shared variable reflects strong-consistency semantics, so we must provide new mapping macros modeling eventual consistency. Next, PGo's `distsys` library must provide the eventually consistent states with concrete CRDT implementations. For PGo, we have implemented a counter and a set resource, following the CvRDTs described in the previous works [Shapiro et al. 2011b].

5.1 CRDT Semantics

At the core of any CvRDT is the merge function, where a commutative, associative, and idempotent operation finds the least upper bound over the two merged states. We describe the semantics of each CRDT in terms of the supported operations and their merge function.

5.1.1 Counter. Our counter resource is grow-only; it is a monotonically increasing counter with applications to counting page views, number of likes, and clocks. Each participating node maintains a vector of partial counts, and supports the following operations depicted in Table 1.

query *get* ()	Return the sum of all partial counts from the vector.
update *increment* ()	Increment the partial count at the index given by the replica's id.
merge (S, T)	Iterate through vectors S and T, resolving conflicts with max function.

Table 1. GCounter support operation semantics.

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

5.1.2 Set. Shapiro describes a number of CvRDT set design variations such as the `U-Set`, `2P-Set`, and `LWW-Set` in the CRDT paper [Shapiro et al. 2011a]. However, these variations either only partially support the common set operations, or the outcome of state convergence relies on external details of clock synchronization and timestamp allocation. Here we want that the set resource supports addition and removal of elements any number of times, and the outcome of a series of set operations conforms to the actual causal order of the operation. Based on these, we chose to implement a state-based version of the Observed-Remove Set (`AWOR-Set`) variation [Shapiro et al. 2011a]. `AWOR-Set` ensures the set is consistent with the causal history of performed adds and removes, and resolves concurrent addition and removal of an element by two nodes with an "add-wins" rule.

To provide these semantics, we model a set as two maps keyed by the element, with vector timestamps as their values. Vector clocks are used for tracking causal order of the set operations, and its size corresponds to the number of participating replica nodes. The "Add" map records the last add time of each set element, while the "Remove" map records the last remove times. Table 2 lists the operations supported by `AWOR-Set`.

query *get* ()	Return the key set of Add map, excluding those where Add timestamp is less than Remove timestamp.
update *add* (e)	Add the element's entry into Add map, incrementing the node's clock in the vector clock.
update *remove* (e)	Add the element's entry into Remove map, incrementing the vector clock at the node's index.
merge (S, T)	1. Squash the corresponding Add and Remove maps, resolving timestamp conflicts using vector clock's merge [Baldoni and Raynal 2002]. 2. Remove from squashed Add map, any entry with timestamp less than that of corresponding Remove map entry. 3. Remove from squashed Remove map, any entry with timestamps less than or concurrent to that of corresponding Add map entry.

Table 2. `AWOR-Set` supported operation semantics

The merge operation of `AWOR-Set` is a multistep process whereby the first step computes the new state consistent to the causal history of the two merged states. Step 2 and 3, although not requried for safety, is a useful **instant garbage collection** mechanism. Instead of preserving the entries of removed or re-added entries in the Add and Remove maps, we can clean up the maps without sacrificing safety by always performing the described vector timestamp comparisons. We can observe the "add-wins" behaviour of `AWOR-Set` in its *get* () semantics. If an element has an entry in both the maps, and the timestamps were concurrent, it is in the current set.

5.2 Network Communication

The advantage of CvRDT is in that we can completely decouple inter-replica communication from the state management mechanism described previously. Our network communication is straightforward, consisting of two asynchronous actions: `Broadcast` and `Receive`. `Receive` is invoked when a replica node gets delivered remote states other peer nodes. Each node invokes `Broadcast` periodically, and it broadcasts the node's local state to the peer nodes. With this approach, propagating local state updates to other nodes happens asynchronously to the state update operations, but alternatives exist in which state update and propagation are synchronous. Often in such approach, the broadcasting node waits for ACKs from receiving nodes to ensure state is received by at least one peer node. This performs better for sporatic state updates, as asynchronous broadcast has a strict lower-bound on time that is dependent on the broadcast interval. However, the absence of waits makes asynchronous broadcast favourable for frequent state updates, and fault tolerance also becomes simple as discussed next.

5.3 Fault Tolerance

A node in PGo can experience failures and crashes, so the CRDT-backed resources must implement failure handling as well. In turns out that providing fault tolerance is also straightforward with our approach. We adopt the retry-if-failed mechanism for inter-replica communication failures. Specifically, a failed broadcast from a node to another is simply attempted again on next broadcast event (with potentially different state from the first attempted).

5.4 Integration with the PGo API

As a resource used by the PGo runtime, all our CRDTs must implement the PGo `ArchetypeResource` API. That is, we must perform the appropriate CRDT operation upon `ReadValue` and `WriteValue` requests, and be ready to rollback the CRDT states if an `Archetype` aborts. Which means, contrary to conventional CRDTs, our CRDT resources guarantee additional requirements. They must:

(1) Maintain previously commited states for potential rollback operation.
(2) Not broadcasting uncommited states to other peer nodes.
(3) Not modify a node's local state during a critical section (except the local update being performed).

We detail in Sec. 6.2 the handlings of these addtional requirements.

5.5 Model Checking

To build applications in PGo that use CRDT resources, we should be able to reflect CRDT behavior in MPCal specs and model check the their properties. First we show that how we model the CRDTs behavior in MPCal and then we discuss the verified properties.

5.5.1 Modeling CRDT Behavior in MPCal. We express query and update methods of CRDTs using mapping macros in MPCal. For each CRDT, we define a mapping macro that its read section implements the query method and its write section implements the update method. This mapping macro for GCounter CRDT is depicted in Figure 7.

According to the CRDT paper [Shapiro et al. 2011b], every node infinitely often sends its state to every other node. Each node after receiving another node's state, merges the received state into its local state. So there is no restriction on the execution order of merge operations. To simulate this behavior, we created a new process in the specification that runs concurrently with node archetypes. This process, in each of its execution step, merges states of two nodes that do not have equal states (merging states of two nodes with the same state has no effect). The model checker will explore every possible execution orders of running this process along with that of the node archetypes. This allows us to make sure that a system works correctly in every possible way that merge operations can happen. Note that for this, we do not use an **`archetype`** and we use a **`process`** because we wish to model-check this behavior without

```
mapping macro GCounter {
    read {
        yield Sum($variable, DOMAIN $variable);
    }

    write {
        assert $value > 0;
        yield [$variable EXCEPT ![self] =
              $variable[self] + $value];
    }
}
```

Fig. 7. Specification of query and update methods of GCounter CRDT in MPCal.

```
macro Merge(crdt, i1, i2) {
    with (res = [j \in DOMAIN crdt[i1] |-> Max(crdt[i1][j],
    ↪   crdt[i2][j])]) {
        crdt[i1] := res;
        crdt[i2] := res;
    };
}

fair process (Merger = 0) {
l1:
    while (TRUE) {
        with (i1 \in NodeSet; i2 \in {x \in NodeSet:
        ↪   crdt[x] /= crdt[i1]}) {
            Merge(crdt, i1, i2);
        };
    };
}
```

Fig. 8. Specifying CRDTs merging in MPCal.

compiling it into Go. The actual broadcasting in the Go implementation is included in the CRDT resources.

5.5.2 Model Checking Properties. Our goal is to make sure our CRDT specs are strongly eventually consistent. An object is strongly eventually consistent if it satisfies the following properties:

Property 5.1 (Eventual Delivery). *An update delivered at some node is eventually delivered to all nodes.*

Property 5.2 (Strong Convergence). *Correct replicas that have delivered the same updates have equivalent state.*

Property 5.3 (Termination). *All method executions terminate.*

To specify Property 5.1 and Property 5.2 we use casual history of state-based CRDTs [Shapiro et al. 2011b]. We define casual history of CRDT object at node i as c_i and update it in each step. Note that we use casual history only during the model checking. To specify Property 5.3 we use a stronger property that states all nodes eventually terminate if no new operation being made.

Moreover, we check the following property that describes CRDT behavior from point of view of the nodes that are using it.

Property 5.4 (Value Convergence). *Infinitely often value of CRDT objects in all nodes converge to the same value.*

We have specified and model checked all of the mentioned properties in our MPCal specs for CRDT systems.

6 IMPLEMENTATION

We implmeneted the resources in Go programming language and specs are implemented MPCal language. All of our implementation are available in a branch in the PGo GitHub repository [2].

6.1 2PC Resource

`ReadValue` and `WriteValue` operate on the replica's local state, and do not involve any consensus steps. `PreCommit` performs the PRECOMMIT broadcast as in Figure 6, also performing the rollback operation if the precommit fails. `Commit` and `Abort` perform the commit and rollback broadcasts respectively.

Critical sections in PGo are required to be *serializable*: the outcome of concurrent critical sections should be equivalent to any sequential ordering of the sections. Our protocol's use of versioning provides a mechanism for ensuring serializability. Suppose the a 2PC resource is at version VER during its first usage in a critical section. At the end of the critical section, the node will only attempt to initiate a state transition if the resource version has not changed, i.e., it has not received a COMMIT message with a higher version. If the version has not changed, then the node will attempt to initiate a state transition to current state value (which may have been modified via `WriteValue`, or unchanged from the beginning of the critical section), at version VER + 1. This critical section will only succeed if the state transition succeeds. As no commits could have occurred between VER and VER + 1, this ensures that the critical section has not observed any effects from concurrent transactions, thus ensuring serializability.

6.2 CRDT Resources

We provide a generic CRDT `ArchetypeResource` that can be instantiated with one of many different CRDTs. The semantics for each CRDT (their access, merge functions) as described in Sec. 5.1 reside in these concrete CRDT implementations, while the generic CRDT abstraction handles replica node initialization, network communication and failure handling (as in Sec. 5.2 and Sec. 5.3), which are common to all CRDTs. By this, introducing new CvRDTs requires no change in the networking code.

CRDT resources service `ReadValue` and `WriteValue` based purely on the node's local state. Network communicaion is orthogonal to state accesses and relies on Go's RPCs. `Broadcast` occurs as a separate goroutine spawned at resource initialization. On every user-specified `broadcastInterval`, the `runBroadcast` goroutine calls the `ReceiveValue` RPC method on its peer nodes with its local CRDT state.

Moreover, we satisfy requirements 1. and 2. from Sec. 5.4 by storing previously commited state during a critical section, and explicitly performing checks in broadcast action to ensure the state has been committed. Otherwise, the node aborts the particular broadcast round. To provide requirement 3, we implement a merge queue

[2] https://github.com/UBC-NSS/pgo/tree/cpsc538b

for remote states received from other nodes while a node is within a critical section. Only after a node exists critical section, (in `Commit` or `Abort`) can the remote states affect a local state.

It is important to keep the time taken to process incoming merges short, as a long- running `ReceiveValue` call or `Commit` , `Abort` methods impact scalability and state convergence time. We implemented two simple optimizations to keep merge time minimal. First, during a critical section, we pre-merge the queued states instead of retaining all received states. This is safe as any CRDT merge operation is commutative. Second, a node may receive states that it has previously merged. We detect such reduntant merges by performing a pre-flight merge on local state, without actually modifying the state. A node can bypass acquiring exclusive write lock to the local state for redundant merges.

7 EVALUATION

Our evaluation focuses on the following four research questions:

RQ1: How efficient is the Go implementation of the systems built using the shared state?
RQ2: How much model-checking overhead is added?
RQ3: Does the implementation satisfy its consistency guarantees in practice?
RQ4: Does the implementation satisfy its fault-tolerance semantics in practice?

7.1 Test Applications

We developed a suite of test applications to evaluate these research questions.

Counter. In this application, each node simultaneously attempts to increment a shared counter, and then waits for the counter to reach its maximum value before exiting. We implemented two versions of these app, one that uses 2PC resource and we call it *Counter*-2PC. The other version uses GCounter CRDT resource for the counter and we call it *Counter*-CRDT.

Lock. Each node attempts to acquire a lock simultaneously. After a node acquires the lock, it releases it. The application ends once every node has acquired the lock a single time. We implemented this application only using 2PC resource.

Key-Value Store. Each node inserts and reads arbitrary values from a key-value store. We implemented this application only using 2PC resource.

ShopCart. Shop cart is a set that supports add and remove operations. We implemented this application using AWORSet CRDT resource.

7.2 Efficiency of the Go Implementation

7.2.1 2PC resource evalution. To evaluate our implementation of 2PC resource, we compare *Key-Value Store* with two other strongly consistent key value stores. The first system is a key value store built in PGo and uses Raft consensus algorithm [Ongaro and Ousterhout 2014], called PGo-Raft. Also, we present benchmark results for etcd [etcd 2021] as a baseline. etcd is key value store written in Go and it is being used by both academia and industry.

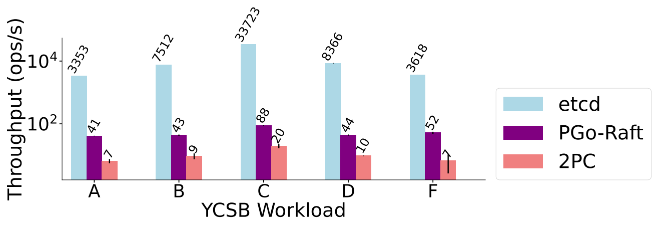

Fig. 9. Throughput of *Key-Value Store* as compared to various key value stores.

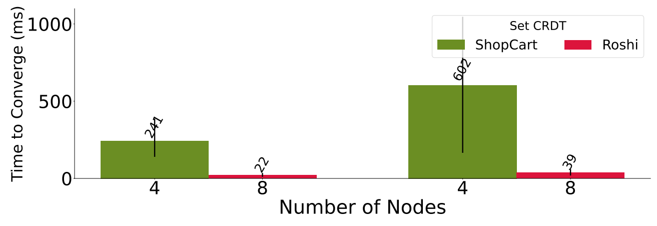

Fig. 10. Convergence times for implementations of a set CRDT.

We ran our experiments on five machines on Azure cloud. We used Ubuntu 20.04 for the operating system and used VMs of type `Standard B4ms`, which have 4 CPU cores and 16 GB RAM.

We evalute these KV store with using YCSB benchmark [Cooper et al. 2010]. We consider the following YCSB workloads: (A) 50/50 read/update Zipfian, (B) 95/5 read/update Zipfian, (C) read-only Zipfian, (D) 95/5 read/update Zipfian (newest records are at the head of the distribution), (F) 50/50 read/read-modify-write (casually related read/write) Zipfian. Due to some limitations in our 2PC resource implementation, we have only used values that are smaller than 10 bytes.

Figure 9 shows the maximum throughput of each system for the mentioned workloads. We can see that our *Key-Value Store* performs worse than two other systems. This is mainly due to generality of 2PC resource implementation. Currently, each *Key-Value Store* node sends the whole key value store data to all other nodes for every operation and this has a huge performance cost. Also, this limits us to only using small values in the benchmark. In the future, we should build a version of this resources that is optimized for the map data structure.

7.2.2 CRDT resource evalution. We evaluated the *ShopCart* application and compare it against an open source CRDT set from SoundCloud called Roshi [Bourgon 2014]. We did this experiment on the same setup on Azure cloud but with the different number of machines.

To compare these systems, we measured how long it took for all nodes' states to converge to the same value (*convergence time*). In our experiment, every node executes multiple rounds. In round r, node n adds the pair $\langle r, n \rangle$ to the set and then waits until its set has all pairs of form $\langle r, t \rangle$, for every node t. For each round, we measured the time from when a node updates its local set until the above condition is satisfied. We repeated this process for a total

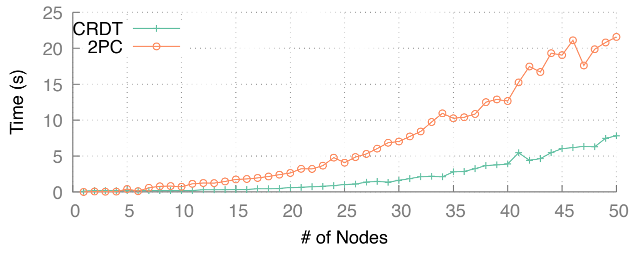

Fig. 11. Time for the *Counter* test application to complete, averaged over three runs.

Application	**Properties Checked**	**# States**	**Time (s)**
Counter-2PC	Termination	31	1
Counter-CRDT	Strong eventual consistency and value convergence	3×10^3	4
Lock	Mutual Exclusion and Liveness	48	1
Key-Value Store	Strong consistency	625	22
ShopCart	Strong eventual consistency and value convergence	14×10^3	143

Table 3. The applications we built in MPCal and their model checking results.

of 100 rounds. Note that in both systems the updates are applied locally, and then each node broadcasts its state every 50ms.

The four bars in Figure 10 show that Roshi has up to 20× better performance than *ShopCart*. This is because Roshi uses timestamps while *ShopCart* keeps a vector clock for each set element. As a result, *ShopCart* performance degrades as the set grows. However, timestamps require clock synchronization among nodes. *ShopCart* uses the stronger causality semantics, which also allowed us to more easily model check its spec.

7.2.3 2PC and CRDT Performance Comparison. We compared the performance of the *Counter* test application using both 2PC and CRDT as the underlying storage implementation. Our test recorded the average time for the application to computing using configurations with between 1 and 50 instances of the shared state, where each instance ran as a separate goroutine, communicating via TCP on a loopback interface. We performed the test for each configured three times, recording the average time for the application to terminate. Figure 11 shows the results of our experiment. As expected, the 2PC implementation required a longer time to reach convergence than the CRDT version, due to the overhead required to ensure strong consistency. We did these experiments on an M1 Macbook Air machine (8 core CPU, 16 GB RAM).

7.3 Model-Checking Performance

Table 3 for each test application lists the propeties we specified and checked, the number of states and the checking time. Our model checking configuration had four nodes for each application. We did these experiments on the same machine as we used in Section 7.2.3.

7.4 Consistency Guarantees

We used the *Counter* and *Lock* applications to verify the consistency of our 2PC implementation. In *Counter*, we ensured that at the end of the application, all nodes have agreed upon the correct value. In *Lock*, we inserted a log statement whenever a node acquires or releases the lock. We analyzed the log file to ensure that only one node had acquired the lock at a time.

The consistency of CRDT-based states relies on the commutativity of the implemented merge operation. For our CRDTs, we performed a commutativity check by generating a permutation of a series of state operations, then folding the merge operation over them. We verified that outcome states from all permutations converge.

7.5 Fault-Tolerance Semantics

We used the *Key-Value Store* application to test the fault-tolerance semantics of our 2PC implementation. In the test, we start with three nodes reading and writing from the store. Two seconds into the test, we terminated one of the nodes, and verified that the other two nodes can continue to make progress. Note that to ensure liveness, we ensured that the node is not acting as a coordinator before terminating it, otherwise the other nodes could have been stuck waiting on a COMMIT or ROLLBACK message.

For evaluating CRDT fault-tolerance semantics, we used the *ShopCart* application. The test starts with 3 nodes adding items to the cart, then we terminated one of the nodes and verified the application with two nodes is live by issuing additional cart update requests.

8 RELATED WORK

Other approaches for generating implementations from specification include the domain-specific language P [Desai et al. 2013], and Mace [Killian et al. 2007], a language for converting specification into a C++ implementation.

Alternative approaches ensure correctness by verifying that the distributed state implementation obeys the specification. Ironfleet [Hawblitzel et al. 2017] and Verdi [Wilcox et al. 2015] are frameworks for Dafny [Leino 2010] and Coq [Bertot and Castéran 2013], 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 [Yang et al. 2009] is another approach for verifying systems sharing states 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 [Liu et al. 2020] takes a similar approach to PGo to compile models into implementations. Similar to PGo, it uses model checking to verify different nodes correctly utilize the distributed state. Another verifier [Gomes et al. 2017] uses the Isabelle/HOL interactive proof assistant to model composible network axioms and reusable lemmas for proving commutativity and strong eventual convergence in CRDT implementations. However, netiher work have the same flexibility as PGo to model the system's environment. For example, to define the environment's behavior Maude provides a Sockets abstraction, which is quite similar to network channel resources in PGo, but is not straightforward how it can be used to provide higher level abstractions, such as the distributed state in this work. Strong eventual consistency verifier [Gomes et al. 2017] retricts its support to operation-based CRDTs, while we take a state-based approach in our work. Finally, 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.

9 CONCLUSION

We presented a new way of building distributed systems in PGo using shared memory abstraction. Our first solution provides strong consistency semantincs and we designed an improved version of the two-phase commit protocol to build a resource with the same guarantees in the PGo distributed runtime. Then, we showed how to model CRDTs in MPCal and build systems using them. We were able to model check correctness of CRDTs and systems that are using them.

As a direction for future work, we consider to verify the handwritten code for 2PC and CRDT resources. PGo recently proposed a new feature that allows to use an archetype as a resource in the other systems. We think this can be a promising approach to reduce trusted computing base in our systems.

10 ACKNOWLEDGMENTS

We thank Finn Hackett, who participated in our weekly meetings and helped us as the main PGo developer. Moreover, he quickly fixed all the compiler issues we reported during this project. We also thank Ivan for the helpful milestone meetings and the superb course. We acknowledge that we took some of our evaluation results from the submitted PGo OSDI 2022 paper.

REFERENCES

Amazon. 2011. Summary of the Amazon EC2 and Amazon RDS service disruption in the US East Region. https://web.archive.org/web/20190217084311/https://aws.amazon.com/message/65648/#expand

Christel Baier and Joost-Pieter Katoen. 2008. *Principles of model checking.* MIT press.

Roberto Baldoni and Michel Raynal. 2002. Fundamentals of Distributed Computing: A Practical Tour of Vector Clock Systems. *IEEE Distributed Syst. Online* 3 (2002).

Yves Bertot and Pierre Castéran. 2013. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions.* Springer Science & Business Media.

Peter Bourgon. 2014. Roshi: A CRDT system for timestamped events. https://developers.soundcloud.com/blog/roshi-a-crdt-system-for-timestamped-events.

Tushar D Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing.* 398–407.

Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. *Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10* (2010), 143–154. https://doi.org/10.1145/1807128.1807152

Renato Mascarenhas Costa. 2019. *Compiling distributed system specifications into implementations.* Ph.D. Dissertation. University of British Columbia.

Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. 2013. P: safe asynchronous event-driven programming. *ACM SIGPLAN Notices* 48, 6 (2013), 321–332.

Minh Nhat Do. 2019. *Corresponding formal specifications with distributed systems.* Ph.D. Dissertation. University of British Columbia.

etcd. 2021. etcd. https://etcd.io/.

Seth Gilbert and Nancy Lynch. 2012. Perspectives on the CAP Theorem. *Computer* 45, 2 (2012), 30–36.

Go Developers. 2009. The Go Programming Language. https://go.dev/

Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. 2017. Verifying Strong Eventual Consistency in Distributed Systems. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 109 (oct 2017), 28 pages. https://doi.org/10.1145/3133933

James N Gray. 1978. Notes on data base operating systems. In *Operating Systems.* Springer, 393–481.

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. 2017. IronFleet: proving safety and liveness of practical distributed systems. *Commun. ACM* 60, 7 (2017), 83–92.

Charles Edwin Killian, James W Anderson, Ryan Braud, Ranjit Jhala, and Amin M Vahdat. 2007. Mace: language support for building distributed systems. *ACM SIGPLAN Notices* 42, 6 (2007), 179–188.

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

Leslie Lamport. 2002. *Specifying systems.* Vol. 388. Addison-Wesley Boston.

Leslie Lamport. 2009. The PlusCal algorithm language. In *International Colloquium on Theoretical Aspects of Computing.* Springer, 36–60.

Leslie Lamport et al. 2001. Paxos made simple. *ACM Sigact News* 32, 4 (2001), 18–25.

K Rustan M Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning.* Springer, 348–370.

Si Liu, Atul Sandur, José Meseguer, Peter Csaba Ölveczky, and Qi Wang. 2020. Generating correct-by-construction distributed implementations from formal Maude designs. In *NASA Formal Methods Symposium.* Springer, 22–40.

Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14).* 305–319.

John K Ousterhout. 1991. The role of distributed state. , 199–217 pages.

PGo Developers. 2021. PGo. https://github.com/UBC-NSS/pgo original-date: 2018-01-17T05:20:13Z.

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011a. *A comprehensive study of Convergent and Commutative Replicated Data Types.* Research Report 7506. INRIA. http://hal.inria.fr/inria-00555588/

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011b. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems.* Springer, 386–400.

TLA Developers. 2021. The TLA+ Home Page. https://lamport.azurewebsites.net/tla/tla.html

James R Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D Ernst, and Thomas Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 357–368.

Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. 2009. MODIST: Transparent model checking of unmodified distributed systems. (2009).