Universitetet
i Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study program / specialization: Computer Science Reliable and Secure Systems	Spring Semester 2023 Open
Author: Daniel Gundersen	
Supervisor: Nejm Saadallah	
Thesis title: Go-MC - An implementation level model checker for Go	
Credits (ECTS): 30	
Keywords: Distributed System, Model Checking, Golang,	Number of Pages: 50 + appendix: 109 Stavanger, June 13, 2023

Go-MC

An implementation level model checker for Go

Daniel Gundersen

June 13, 2023

Abstract

Implementation level model checkers haven proven a good tool for identifying bugs in implementations of distributed algorithms. In recent years many new model checkers have been developed. These often include new state space reduction techniques which increase their effectiveness, but they are often locked to use a specific state space reduction technique and to support specific abstractions. This makes it hard to compare different state space reduction techniques and to change between different abstractions.

We propose Go-MC, a modular implementation level model checker for the Go programming language. Go-MC consists of four modules: the Scheduler, the State Manager, the Checker and the Failure Manager. Each module can easily be swapped for different implementations, which makes it easy to change between different abstractions and scheduling techniques.

Go-MC also uses Event Managers to control the execution of the algorithm. Event Managers are flexible and custom implementations can be made to utilize specific frameworks or to mock components of distributed systems. This allows us to take a modular approach when simulating distributed systems, which will reduce the number of events in a simulation and thus reduce the size of the state space. It also makes it possible to efficiently capture events and support different frameworks.

Acknowledments

I would like thank my supervisor Nejm Saadallah for guidance throughout the process of writing this thesis. Your feedback and advice has been invaluable.

I would also like to thank Hein Meling, Leander Nikolaus Jehl and Hanish Gogada for feedback and suggestions when implementing and using Go-MC.

Thank you to my family who have supported me throughout the process of writing this thesis.

Contents

1 Introduction

Distributed systems are a class of systems that are known for being hard to reason about and difficult to troubleshoot [30, 20, 18, 28, 22]. This is, at least in part, because distributed systems consists of autonomous nodes communicating over a network. Consequently, distributed system are inherently concurrent and they contain all the complexities associated with concurrent programs. Individual nodes can also fail, further increasing the complexity in the system. Since an ever increasing number of services relies on distributed algorithms it is essential that we develop effective methods for verifying both the design and implementation of the algorithms to ensure that these services can operate with the high uptime that are required of them.

Traditional verification techniques, such as testing, are ill-equipped to deal with the large amount of randomness inherent in distributed systems. The randomness causes tests to provide widely different output when run multiple times with the same input. This is because they only test one possible execution of the algorithm, while the randomness means that there can be many different executions corresponding to the same input. Any tool that wants to guarantee the correctness of a distributed system must verify all possible executions of the algorithm.

There has been a large amount of research into verification methods for distributed systems. An early set of methods are traditional model checkers. They use models of the system, such as Petri Nets[24], TLA+[29] or Promela[3] specifications, to generate all possible states of the system. They then systematically check all generated states of the system for property violations. These model checkers are useful for finding logical faults in the design of the algorithm, but since they only run a model of the system, instead of the final implementation, they are unable to find implementation errors.

A natural development of such model checkers is to apply them to the final implementation of the algorithm. Such implementation level model checkers include MaceMC [11], MODIST [28] and SAMC [18]. Implementation level model checkers are more efficient at detecting implementation level bugs than traditional model checkers, since they test the actual implementations instead of a model of the system. They can therefore detects bugs that occur when translating the algorithm from specification into the actual implementation. Implementation level model checker repeatedly run trough the implementation of the algorithm while recording the state of the processes and controlling the ordering of important events such as message flow, process crashes and timeouts. This allows it to explore the state space and discover subtle bugs caused by rare race conditions.

One of the major problems faced by both traditional and implementation level model checkers is that the number of possible executions grows quickly as the number of events (e.g. messages, timeouts, node crashes, etc.) increases. This is called the state space explosion problem. The state space is the space of all possible states that can occur in a distributed system and the size of the state space generally increases with the number of possible executions. As the

size of the state space increases both the computation time required to simulate the execution and the memory required to maintain the state space increases.

Several techniques for reducing the size of the state space have been developed [23, 30, 2, 19, 18, 10, 28, 11, 20]. However, these techniques are often implemented as a part of their own model checkers, which are not designed to make it possible to insert other state space reduction techniques. This has two disadvantages. Firstly, it creates a situation where, to implement a new state space reduction technique, you either need to implement a new model checker or you have to spend time to adapt an existing model checker. Secondly, since the different state space reduction techniques are run on different model checkers it is hard to compare them. At best one can compare the model checkers as a whole, but this fails to provide a good way of comparing the state space reduction techniques as the different designs of the model checkers would affect the results.

We propose Go-MC (Go Model Checker), an implementation level model checker for the Go programming language. Go-MC makes it possible to approach verification of distributed systems in much the same way as we approach verification of other software: by writing deterministic tests that shows that the desired properties holds under the given input. When a property is violated it provides counterexamples and the ability to replay the run, which simplifies the process of identifying and fixing the bug.

Go-MC uses a modular design consisting of four modules: the Scheduler, the State Manager, The Checker and the Failure Manager. Each module has a defined interface and can easily be swapped by other implementations of the module. This ensures that Go-MC can easily be modified to support the assumptions and abstractions required by the algorithm, instead of adapting the algorithm to fit the model checker. It also makes it easy to implement and compare different versions of state space reduction techniques.

Go-MC uses Event Managers as the interface to the nodes. The Event Managers are placed at a high level of the algorithm. This allows them to intercept events close to the actual implementation of the algorithm, which reduces noise. This ensures that the Event Managers can utilize the features of the frameworks to capture events efficiently. Event Managers are flexible and can also be used to mock components of distributed algorithms. This makes it possible to utilize a modular design, where each module is verified independently, when verifying distributed systems. This reduces the number of events in the simulation, which helps reduce the impact of the state space explosion problem.

Go-MC also contains a tool for recording the execution of events in a live run of the algorithm. The tool provides the functionality to pause the execution of events, trigger node crashes or send requests to nodes. This makes it possible to visualize and control the execution of the algorithm under different scenarios. The tool uses Event Managers in a similar way as the simulation tool, and can therefore easily be applied to the same algorithms that are verified.

2 Background

Distributed systems are systems consisting of multiple nodes that communicate by passing messages between them. The nodes in the system cooperates to achieve some common goal, such as agreeing on a common value or operating a key-value storage. They are typically passing messages over some network, but can also use some other method.

A node is some kind of computing element, it could be a computer, a process or thread on a computer or some other kind of device capable of participating in the distributed system. We make no assumption about the speed of the nodes. It can vary over time and all nodes in a distributed system do not have to run at the same speed.

Each node in a distributed system is controlled by a local algorithm. The local algorithm define the behavior of the node, such as how it should respond to messages and what messages it should send to other nodes. The distributed algorithm is the aggregation of the local algorithms of the nodes in the system. Thus, the behavior of the distributed algorithm is defined by the behavior of its nodes [27, 5].

Distributed algorithms can be grouped into modules that defines the behavior an algorithm should present to the user. While all algorithms implementing the module should present the same external behavior to the user, the methods used to achieves this behavior and the conditions under which they are achieved can vary. Consider, for example, the *Reliable Broadcast* module. The module has a set goal, which is ensuring that all correct processes agrees on the delivered messages, but we can implement multiple different algorithms that achieves this goal. One algorithm can solve the problem by detecting when the node sending a message has crashed and resending the message to all other nodes. Another algorithm can solve the problem by always relaying messages to all other nodes [5].

This enables the use of a modular approach when designing distributed systems. We can divide the system into several modules, each encapsulating a specific external behavior, such as reliable broadcasts or consensus(i.e. agreeing on a common value). New modules can be designed that rely on or extends the behavior of other modules. The modules can interact with each other by passing events between them.

This modular approach has several advantages. Firstly, it makes it easy to replace one implementation of the module with another. This can be useful when, for example, we want to improve the performance of the algorithm or when we need the algorithm to support a different set of assumptions. Secondly, it allows us to verify each module individually. If we have verified an implementation of a module under a set of assumptions once, we know that it is correct and we do not need to verify it every time we use it in a distributed system. Lastly, since modules often encapsulate common behavior it is often possible to reuse modules across different distributed systems, thus saving time when implementing new algorithms.

The execution of a distributed algorithm is modelled as a sequence of events,

we will call this sequence a *run*, that are executed by the processes in the distributed system. An event is one of the three following actions[5]:

1. Sending a message, m.
2. Receiving a message, m.
3. Perform some action that changes the local state of the node.

Sending the same message to multiple nodes, i.e. broadcasting a message, can be considered a single event, as it has no practical impact on the ordering of events [15]. We can also consider the sending of a message to be a part of the previous event, since the sending of a message does not randomly happen, but is triggered by the execution of the previous event.

As an example, let us consider a basic ping algorithm(Figure 1) where Node 1 sends a *ping* message to two other nodes, Node 2 and 3, and the two nodes respond with a *ping response* message each. This algorithm consists of four events, which we give the names e_1, e_2, e_3 and e_4: the *ping* arrives at Node 2 (e_1), the *ping* arrives at Node 3(e_2), the *ping response* from Node 2 arrives at Node 1(e_3) and the *ping response* from Node 3 arrives at Node 1 (e_4). The sequences $e_1e_2e_3e_4$ and $e_1e_3e_2e_4$ are two different runs of this algorithm. Not all sequence correspond to a valid execution of the algorithm. An example of this is the sequence $e_4e_1e_2e_3$, which is invalid since the *ping response* from Node 3 to Node 1(e_4) can not be sent before Node 3 receives the *ping* message from Node 1 (e_2). Only sequences that corresponds to an actual execution of the algorithm is a possible run.

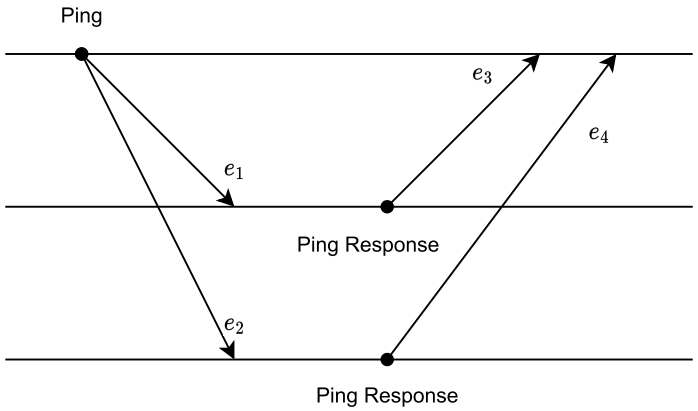


Figure 1: One possible execution of the basic Ping Algorithm. Events are marked with their name. When messages are sent they are marked with the type of message.

The nodes in a distributed system are autonomous, that is, they are independent and do not share any resources such as memory or persistent storage. They only communicate by sending messages to each other. This means that

the nodes does not share access to a common global clock, and each node therefore has its own notion of time. Each node is equipped with a local clock to keep track of its local time. Each of the local clocks runs at approximately the same speed, but due to the heterogeneity of the nodes they will deviate from each other. It is possible to device algorithms that can synchronize the local clocks, ensuring that there is a known upper bound to the amount of deviation, but such algorithms are only possible under certain timing conditions that can not be guaranteed to hold in all environments. In practice this means that while each node might have a local clock which it can use to get timestamps, comparing two timestamps from different nodes is generally meaningless. This complicates several parts of designing distributed algorithms, such as establishing an ordering of events [27, 16, 5].

In the absence of global clocks, the *happened before* relationship is often used to establish a partial ordering of the events. The ordering is partial because it is unable say definitively what events happened first, but it captures which events have potentially caused other events and which events might have happened at the same time. Event e_1 happened before (or potentially caused) e_2, we write $e_1 \rightarrow e_2$, if one of the following conditions hold [15]:

- e_1 and e_2 are events of the same node and e_1 was executed before e_2 on the node
- e_1 is the sending of a message, m, from a node and e_2 is the receipt of the same message, m, on a node
- given a third event, e_3. If $e_1 \rightarrow e_3$ and $e_3 \rightarrow e_2$

Given two events, e_1 and e_2, if neither $e_1 \rightarrow e_2$ nor $e_2 \rightarrow e_1$, then the two events are concurrent.

The *happened before* relationship is very important and an important state space reduction techniques called Dynamic Partial Order Reduction uses it to identify runs which are redundant because they correspond to the same *happened before* ordering of events [8]. Many state-of-the-art state space reduction techniques utilizes Dynamic Partial Order Reduction.

As an example we will investigate the *happened before* relationships in the execution of the Ping Algorithm presented in Figure 1. We will represent events representing the sending of a message as a separate events. We will use a $\prime$ to distinguish between this send event and the event that triggered the sending of a message, e.g. the sending of the *ping response* message that is a result of e_1 will be denoted as e_1'. In this example run Node 1 receives e_3 before e_4, although in a different run it is possible that e_4 arrives before e_3, which would result in a different *happened before* relationship. We can establish that:

- $e_1 \rightarrow e_1'$, $e_2 \rightarrow e_2'$ and $e_3 \rightarrow e_4$ since the events are executed on the same node.
- $e_1' \rightarrow e_3$ and $e_2' \rightarrow e_4$, since they are the sending and receipt of a message.
- $e_1 \rightarrow e_3$, since $e_1 \rightarrow e_1' \rightarrow e_3$

- $e_2 \rightarrow e_4$, since $e_2 \rightarrow e_2' \rightarrow e_4$.
- $e_1 \rightarrow e_4$, since $e_1 \rightarrow e_3 \rightarrow e_4$.
- e_1 and e_2 are concurrent, since neither $e_1 \rightarrow e_2$ nor $e_2 \rightarrow e_1$.

This can be summarized as: $e_1 \rightarrow e_3 \rightarrow e_4$ and $e_2 \rightarrow e_4$.

When designing distributed algorithms we specify a set of properties that define the desired behavior of the distributed system. The properties defines goals that the individual nodes in the system should be working toward and behaviour that they should avoid. The properties must hold for all possible executions of the algorithm for it to be correct. These properties are divided into two categories: *Safety* and *Liveness* properties.

Informally, safety properties define what an algorithm should not do. More formally a safety property is a property that can be violated at some time, t, and never be satisfied again after. This can for example be that a broadcast abstraction should not deliver messages that was not sent.

Liveness properties describe what must happen during the execution of the algorithm. They are properties where it suffices that for any time, t, in an execution there is some later time, $t' > t$ where the property is satisfied. We say that the property must eventually be satisfied, meaning that it must be satisfied at some point, but not necessarily at the current time. An example of a liveness property is a property stating that a broadcast algorithm should eventually deliver a message that is sent [1, 13].

2.1 Common Abstractions

It is desirable that distributed algorithms should be applicable to many devices, no matter what hardware, or operating system is used on the device and no matter the details of the communication method used to transfer the messages. We therefore define abstractions which encapsulate common properties of these systems. These abstractions make it easier to reason about the system and makes it possible to design algorithms that match the abstractions instead of making a new algorithm for each specific device.

One abstraction that is used is the notion of links (also called channels). Links are connections between two nodes used to pass messages between them. In the most basic abstraction of a link any message can be dropped by the network, but the probability of a dropping a message is less than one, ensuring that eventually some message will be delivered. This type of link abstraction is often called a *fair-loss* link. In general, the link abstractions imposes no limitations to the processing or transmission time of a message. In practice this means that a message can be delayed for an arbitrary amount of time and that the order in which messages arrive on a node can be different from the order in which they were sent (Figure 2).

The *fair-loss* link abstraction is often used to create several more advanced link abstractions. An example is the perfect link abstraction, which guarantees that all messages that are sent are eventually delivered, which can be created

Figure 2: Two space-time diagrams showing an example of how messages can arrive in different orderings depending on the delay. Left: $m_1 m_2$. Right: $m_2 m_1$.

by continuously re-sending a message. Various broadcast abstractions, where messages are sent to a group of nodes instead of only one node, can also be created [5, 26].

Using the abstraction of point-to-point links does not mean that every node must be physically connected to all other nodes. It only means that every node must be able to send messages to all other nodes in the system. The messages can be routed trough other nodes or even networks before they arrive at the target node. The abstraction captures this and since it makes no assumptions on delay it is applicable to a wide range of practical implementation and network topologies.

Distributed systems are defined by the presence of partial failures where some of the nodes may fail, while the rest of the nodes continue to execute. To ensure that the system remains as fault tolerant as possible it is desirable that the remaining nodes in the system is able to correctly execute the distributed algorithm even if some nodes fails. Furthermore, a node may fail in different ways, each of them them having a different impact on the execution of the algorithm. For example if a node crashes and therefore stops processing messages it will have a different impact than if a network error occurs that temporarily disconnects a node from the rest of the nodes. Different abstractions are therefore used to capture the properties of different types of failures. The failure abstractions makes it easier to reason about different types of failures and encapsulates the properties that must be considered when designing an algorithm for the abstraction. Three important failure abstractions are introduced below.

The first abstraction is the *crash-stop* fault, where a node executes the algorithm as specified until a set point where it crashes and stops performing any action. The node is said to be faulty if it crashes at some point during the execution, and correct if it never crashes [5, 25].

The second important failure abstraction is the *crash-recovery* fault. In the *crash-recovery* abstraction a correct node might crash if at some point later it recovers again and continue to execute the algorithm. A correct node can crash and recover multiple times, as long as it only crashes a finite number of times. An incorrect node is a node that either crashes and never recovers, or crashes and recovers infinitely often. When a node recovers it may have lost all state stored in memory and some kind of stable storage will then be required to ensure that the node can continue correctly [5, 4].

The final failure abstraction is the *Byzantine*, or *Arbitrary*, fault, where

faulty nodes may deviate from the specified algorithm and can take any arbitrary action. This can happen for several reasons for example if a malicious party manages to get control of one of the machines running the node or by some bug that causes the node to behave in some way that deviates from the algorithm. Under a *byzantine* fault arbitrary messages can be read, edited and inserted into any communication link [17].

A final important abstraction is the timing assumptions. The timing assumptions encapsulate information about time bounds on communication and processing delay. We presents three common timing assumptions: Synchronous, asynchronous, and partially synchronous systems.

Synchronous systems represents systems where there are a known upper bound to the communication and processing delay. In synchronous systems it is possible to design algorithms that synchronizes clocks and accurately detects failures. The major limitation of the synchronous systems is that they are only applicable to a limited amount of real world systems.

Many distributed systems runs over complex networks where it is hard to establish a upper limit to communication delay, which means that the synchronous model can not be used. The asynchronous systems model represent systems where no timing assumptions are made. The model can be applied to all scenarios, but imposes limitations on the kind of algorithms that can be implemented. It is, for example, impossible to detect crashes by using timing mechanisms.

The partially synchronous system model represent a combination of the asynchronous and synchronous systems: there exists an upper bound to the communication and processing delay, but it is unknown or it only holds eventually [7]. This system model is particularly useful, because it is can be applied to a large range of real world systems while also making it possible to implement certain algorithms that can not be implemented in the asynchronous system model, for example fault-tolerant consensus algorithms[6].

An important class of algorithms that can be used to encapsulate timing assumptions are failure detectors. Failure detectors are used to detect failures in distributed systems. The strongest failure detector abstraction is the Perfect Failure Detector. A Perfect Failure Detector will eventually detect a crashed node and will never be wrong when it does. The Perfect failure Detectors relies on the synchronous timing assumption. A weaker failure detector is the Eventually Perfect Failure Detector. It will eventually detect a crashed node, but it is not necessarily correct. However, after some unknown time crashes are detected correctly, but before this time detection of node crashes does not have to be accurate. We say that it is eventually correct [5]. Other groups of failure detectors that rely on other assumption also also exists [6], some of which does not rely on timing assumptions at all [21].

Failure detectors can be used to implement leader electors, which are algorithms that are used to elect some node which will control the algorithm. This is especially useful in consensus algorithms where all nodes have to agree on a common value. Both a perfect leader elector and an eventual leader elector can be implemented by using the respective failure detectors. In addition a byzan-

tine leader elector, which can elect a leader in the presence of byzantine nodes, can be implemented.

2.2 Randomness in Distributed Systems

We have made several assumptions in the model we have presented for distributed systems: the nodes can execute events at different speeds and the speed can vary over time, messages can be delayed or dropped, and nodes can crash during the execution of events. These assumptions ensure that the model we use can be applied to many types of systems, with little regard for what hardware is used or the network topology. They also represent sources of randomness: the factors are out of control of the algorithm and we can not rely on any specific behaviour from them. We could – of course – make stronger assumptions that reduces the randomness, but then we would have to ensure that the real world system that the distributed system is deployed in is accurately described by the model we use, which would make it harder to deploy. Furthermore, the different abstractions we have presented does this, but even when we use the stronger abstractions we can not remove all randomness.

This inherent randomness is one reason that it is hard to implement and verify distributed systems. Because of the randomness there are rarely a single run that can represent the execution of the algorithm. Instead, there are many runs, each representing a scenario where two messages arrives in a different order, a node slows down or crashes, or a message is dropped by the network. The number of runs quickly grows, even for relatively simple algorithms, and reasoning about all possible runs that can occur is challenging: Runs that appear similar can have subtle differences that must be considered.

The randomness also makes testing of implementations using traditional methods difficult. Simply running through the algorithm is not an effective way of identifying bugs, since there is no guarantee that this execution would result in a run where the bug occur. Even if the bug occur in one execution there are no guarantee that the bug can be reproduced at a later execution, making it hard to troubleshoot and verify that the bug has actually been fixed. Comprehensive unit testing would have to predict all possible states that can occur during the execution and then execute all possible events on those nodes. This is equally hard as predicting all runs that can occur, and defining the tests would be time consuming.

Implementation level model checkers solves these problems by performing a linearization of the concurrent execution of the algorithm. That is, instead of executing multiple events on different nodes simultaneously, model checkers executes all events sequentially, even if the events are executed on different nodes. By linearizing the execution of the events we reduce the randomness in the execution: it no longer matter how long it takes to execute an event, because no other event will be executed at the same time, nor how long a message is delayed, because no other event will be executed before the message has arrived. The model checker will still have to resolve other sources of randomness, such as node crashes or the network dropping messages, but these sources can be

modelled as events that are added to the linearization. Thus, they no longer occur at random times, but rather at pre-planned and deterministic times. The linearization of the algorithm is therefore deterministic in the sense that each time a given linearization is run it will always produce the same sequence of states. Each run corresponds to a different linearization of the algorithm and the model checkers can create all possible runs of the algorithm by performing all possible linearizations. The runs can then be checked to ensure that the properties of the algorithm holds for all discovered states. Any bugs that are found can be reproduced, ensuring that it is easy to troubleshoot.

Implementation level model checkers can be divided into two components: the controller and the hooks. The controller is a central unit that manages the overall execution of the simulation. It controls the hooks, informing them when to execute events. The controller can be divided into smaller modules, such as a scheduler and a property checker. The hooks connect to a node. They control and report actions that the nodes perform back to the controller. They are the interface between the node and the simulation and are used to collect state and control the execution of the algorithm. To make it easy to move between testing and production it should be easy to insert and remove the hooks from an algorithm.

3 Related Works

In recent years several implementation level model checkers for different programming languages have been developed.

DSLab is an implementation level model checker implemented in Java and designed for use while teaching distributed systems. It assumes an asynchronous network model where nodes are represented as a single-threaded event loop and can crash, but not recover. The distributed algorithm is defined as a set of message and timer handlers, called event handlers, that are run upon receiving a message or a timer firing. Each event handler should be deterministic, in the sense that the output state should only depend on the input state and the event. Limited randomness is however supported trough random timer duration which is controlled by DSLab. It combines two techniques to search the state space: search for progress and guided search. In search for progress the model checker will search for states in which progress has been made before searching for invariant violating states. In guided search the user provides information that guides the model checker towards areas of the state space that are likely to contains errors, for example when a node has crashed. Finally, DSLab also provides a graphical interactive debugger that can be used to assist troubleshooting bugs. It allows for manually controlling the order of events.[20]. Go-MC takes a similar approach as DSLab and models nodes as a single-threaded event loop with each event being deterministic. Go-MC also provides support for the guided search approach used by DSLap through one of the implemented schedulers.

MODIST is a model checker that assumes an asynchronous and unreliable network, and will simulate faults by reordering and dropping messages. Nodes are modelled as processes running multiple threads and communicating with each other by sending messages over socket connections. MODIST consists of a frontend that is used to control event execution and a backend that is responsible for scheduling decisions. The frontend is interposed between the application and the operating system and intercepts system calls performed by the application. The process is then paused and the system calls reported to the backend. The frontend waits until it receives a command from the backend before returning the system call. Errors can also be introduced by forcing the system call to return errors. This allows the backend to control the execution of the distributed algorithm. System calls are discovered by intercepting calls to the WinAPI. The frontend of MODIST is therefore Windows specific, although the authors claims that it would be easy to port MODIST to other operating systems[28].

MaceMC is a model checker that focuses on identifying liveness violations. It tests programs implemented in Mace, a C++ language extension designed to simplify the task of implementing distributed algorithms. In Mace a user defines event handlers which specify how the system should react to events, such as incoming messages. MaceMC controls message ordering by controlling the order in which the event handlers are executed. Event handlers should be deterministic to allow MaceMC to search the state space. MaceMC ensures this by modifying the Mace random number generator to return deterministic values

during simulation[12, 11].

Morpheus is a model checker for testing algorithms implemented in Erlang. Erlang is a programming language designed for use in distributed systems. Programs in Erlang is run in Erlang virtual machines that communicates by exchanging messages. Morpheus controls the ordering of events by intercepting the communication primitives and replacing them with specific implementations that report the call and wait for instructions before resuming. Morpheus simulates clocks by maintaining a queue of timeouts for each node, according to the order of the deadlines. When all pending messages have been delivered the next timer is fired. Morpheus focuses on concurrency testing and does not inject failures, such as dropping messages. Morpheus utilizes an randomized algorithm called Partial Order Sampling to provide strong probabilistic guarantees of sampling any partial order of a program. Partial Order Sampling assigns independently random priorities to operations and schedule operations according to their priority [30].

Demeter is an implementation level model checker that uses an approach called dynamic interface reduction. The approach divides the system into a set of components with well defined interfaces. The interface of the components are discovered dynamically when messages are exchanged by the components. Dynamic interface reduction uses the discovered interface behaviour to run state exploration of the individual components separately. This allows it to explore local decisions, such as thread scheduling, individually instead of performing global scheduling of all possible interleavings of all nodes [10]. Go-MC expects events to be deterministic and does not consider local thread scheduling when exploring the state space. It also mainly considers events that represents communication between nodes.

Dara is an implementation level model checkers for distributed systems implemented in Go. It uses a modified version of the Go runtime, called *dgo*, to control several types of nondeterminism, including thread interleaving, network randomness, the *time* library and the *random* library. Dara uses static analysis of the code to report the coverage of the simulation to a the model checker which makes scheduling decisions aiming to increase the coverage. State variables are reported to Dara using two API calls provided by the custom runtime. These must be called manually by the user in the implementation of the algorithm [2]. Compared with Go-MC, Dara takes a lower level approach where it controls more detailed events, such as thread scheduling.

SAMC is a implementation level model checker that uses a white box approach to increase the effectiveness of the scheduling. In the approach the user provides semantic information to the algorithm by defining policies. The policies are then used to reduce the reordering of messages, crashes and recoveries [18]. Another implementation level model checker that uses a white box approach is FlyMC. FlyMC uses symmetry inherent in the algorithm, such as message symmetry, to reduce the number of messages that are reordered. It also reorders multiple independent message-pairs at a time by performing *parallel flips*. This allows it to quickly explore deep runs [19]. The scheduling module of Go-MC is designed to make it possible to use white box approaches when designing

schedulers.

4 Go-MC

This section will provide an overview of Go-MC[1]. It provide a conceptual model of the system that can be used to understand how Go-MC works and how to use it. It presents the model in a top down approach, building the various components that are used. Section 5 provide a technical overview of the system, detailing the implementation and the design of the modules.

Go-MC uses a modular design that consists of 4 modules (Figure 3). Each of the modules is responsible for a separate task and Go-MC can be configured to use implementations of the modules to represent different abstractions or to use different techniques. This makes it easy to design modules that represent different abstractions and to insert own implementation of modules into the simulation. We will introduce each module and the problems they solve in the following sections.

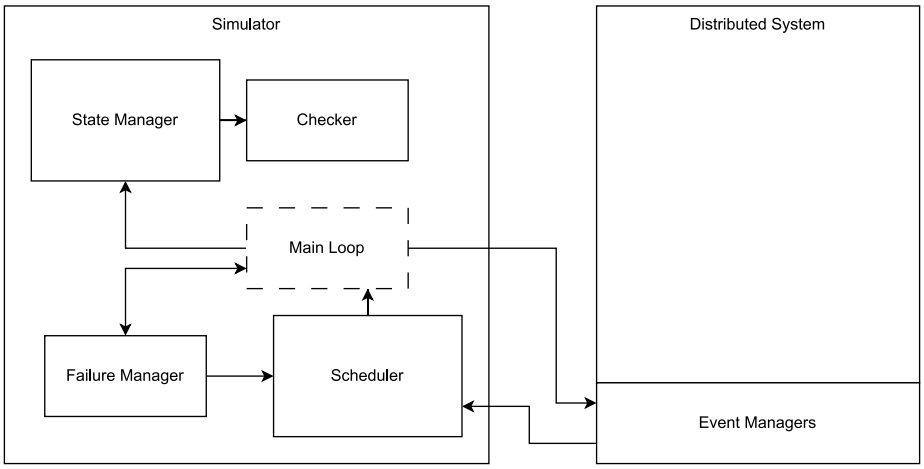


Figure 3: System Architecture of the simulator of Go-MC. The simulator consists of four modules and the main loop. The arrows show how the modules communicate with each other. The simulator uses Event Managers to interface with the distributed system under testing.

Throughout this section we will use the Hierarchical Consensus algorithms as an example. We have used Go-MC to test our implementation of hierarchical consensus (Appendix A.1). The specifications of the algorithm is presented by Cachin, Guerraoui, and Rodrigues [5](Algorithm 5.2). It is a consensus algorithm where the correct node with the lowest id is the leader. The leader imposes its value on all other nodes in the system. The algorithm consist of one round for each node. A node waits until its round before deciding on the value received by the leader. The algorithm proceeds to the next round when the node

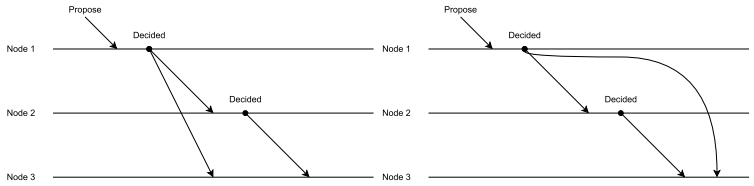
[1]The source code of Go-MC can be found at the github repository, `https://github.com/erthbison/GoMC`, or as an attachment to this document.

has either decided or crashed. The algorithm uses a Perfect Failure Detector to detect when a node has crashed. Figure 4 shows the two possible execution of the Hierarchical Consensus algorithm with three nodes and no crashes.

```
1  package main
2
3  type Value[T any] struct
4
5  func NewHierarchicalConsensus[T any](id int, nodes []int, send
   ↪  func(int, string, ...any))
6
7  type HierarchicalConsensus[T any] struct
8  func (hc *HierarchicalConsensus[T]) Crash(id int, _ bool)
9  func (hc *HierarchicalConsensus[T]) Propose(val Value[T])
10 func (hc *HierarchicalConsensus[T]) Decided(from int, val
   ↪  Value[T])
11 func (hc *HierarchicalConsensus[T]) decide()
```

Listing 4.1: Signature of the Hierarchical Consensus implementation. For a full overview of the implementation of the Hierarchical Consensus algorithm see Appendix A.1.

We configure the algorithm for testing with three nodes, where node 1 receives a Propose request and we want all correct nodes to agree on the same value (Listing A.2). We first configure the simulation by selecting a Scheduler and what state should be collected. Then we run the simulation by specifying how to start the nodes, what requests will be used to start the simulation, which nodes should fail and what properties should hold for the algorithm.



(a) (b)

Figure 4: Space-Time diagrams showing the two possible executions of the Hierarchical Consensus Algorithm with three nodes and no crashes.

4.1 Main Loop - Linearizing the Execution

To ensure that we get consistent results when running the tests we need to execute every possible run of the algorithm and verify that the properties holds

for each of the states that are generated. To do this we need a mechanism that control the random variable to manually decide the order in which the events are executed.

First, we formalize the notion of events in our simulation as actions that changes the state of a node. Events target a specific node and can not change the state of any other nodes. An event is executed when the action is taken and the state of the node is changed. The execution of an event must be atomic, i.e. it is not possible to pause an event and begin executing a different event, and deterministic, i.e. executing the same event on two equivalent input states should result in the same output state. Delivering a message and executing the corresponding message handler is an example of an event.

This notion of events matches well with the traditional model of an algorithm as nodes each performing a sequential execution of events, but an actual implementation might want to execute events concurrently on the same node to increase efficiency. The Go concurrency scheduler could then pause the execution of one event on one goroutine to execute another event on another goroutine, thus breaking the atomicity of events. To ensure that this does not happen during the simulation of the algorithm we impose rule 1 on the execution of events. The rule is enforced by the simulator by waiting for an event to signal that its execution is done before starting the execution of a new event on the same node. This ensures that the Go concurrency scheduler can not break the atomicity of events since there are no other events to switch to. This rule makes it hard to find local concurrency bugs during concurrent execution of events on a node, but Go-MC focuses on identifying distributed concurrency bugs and we therefore consider identifying local concurrency bugs as out of scope.

Rule 1. Each node execute events in a sequential manner.

Rule 1 ensures the atomicity of events on a node, but since events that are executed on different nodes can still be executed concurrently it is not a linearization of the events. To ensure that the execution is deterministic we want to perform a linearization of the events. We therefore introduce Rule 2, which strengthen rule 1. Rule 2 is also enforced by the simulator in a similar manner as rule 1, but it only allows one event to run at the same time, instead of allowing multiple events to be started as long as they targets different nodes.

Rule 2. All events are executed in a sequential manner.

Conceptually, we create a global clock and assign one event to each time slot until there are no more available events. The simulator then executes the event of the current time slot and advances the clock to the next time slot when the event has been completely executed. This continues until the end of the clock. The sequence of events formed by the clock correspond to one possible run of the algorithm.

The runs generated by Go-MC must be finite in length, otherwise the simulation will never complete. In practice some algorithms are designed to run forever, like failure detectors, or can not guarantee progress under the conditions

that are tested. To ensure that the simulation progresses even when infinite runs are simulated Go-MC enforces a maximum length on runs. If a run is longer than the maximum length the simulation of the run is stopped.

As an example of how Go-MC linearizes the execution let us consider the example algorithm with three nodes (Figure 5). Say that node 1 receives a *Propose* request and begins executing the event at time slot 1. During the execution of the event node 1 sends *Decided* messages to node 2 and node 3. During the normal execution of the algorithm the messages would arrive at their respective nodes and the nodes would begin executing the events concurrently, possibly before the *Propose* request on node 1 has been completely executed. During the simulation only one event can be executed at a time, so the simulator first waits until node 1 has completed executing the *Propose* request. It then selects one of the two pending events to execute next. In this example it selects the the *Decided* messages to Node 2 in time slot 2. During the handling of the *Decided* message Node 2 sends a *Decided* message to node 3. There are now two pending events in the simulation: The *Decided* message from Node 1 to Node 3 and the *Decided* message from Node 2 to Node 3. After the previous event is completed, the simulation selects one of the pending events, in this case the *Decided* message From Node 1 to Node 3, in time slot 3. It selects and executes the final event in time slot 4 after the event in time slot 3 has been completed.

4.2 Event Managers - Discovering and Executing Events

Conceptually events are created during normal execution of an algorithm when a node performs some action that triggers a node to perform an action in response. The response is the event that is created and it is executed when the algorithm performs the action. Go-MC needs to be able to discover these events as they are created. This is done using Event Managers.

Event Managers are hooks that are inserted into the implementation to replace primitives that would cause an event to be created. Consider the `send` function of the example implementation. When the algorithm is executed it will provide the functionality of a perfect link by sending messages to nodes. During simulation the regular `send` function can be replaced with a modified `send` function that is connected to the simulator. It will still provide the functionality of a perfect link, but it will also report any created events to the simulator and wait until the event is executed before actually delivering the message to the target node.

Event Managers can be used to cover many cases where we want to test the ordering of events. Consider, for example, a case where we want to check if a message arrives within a certain time. In this case we can create an event manager for timeouts which will add a timeout event to the simulator when it is called. The simulator can then control when the timer will fire, allowing it to create both a run where the timer fires before the message arrives and a run where the timer fires after the message arrives, allowing us to verify that the algorithm works for both scenarios. The event manager can easily be designed to have the same signature as the `time.Sleep` function, and inserting it into

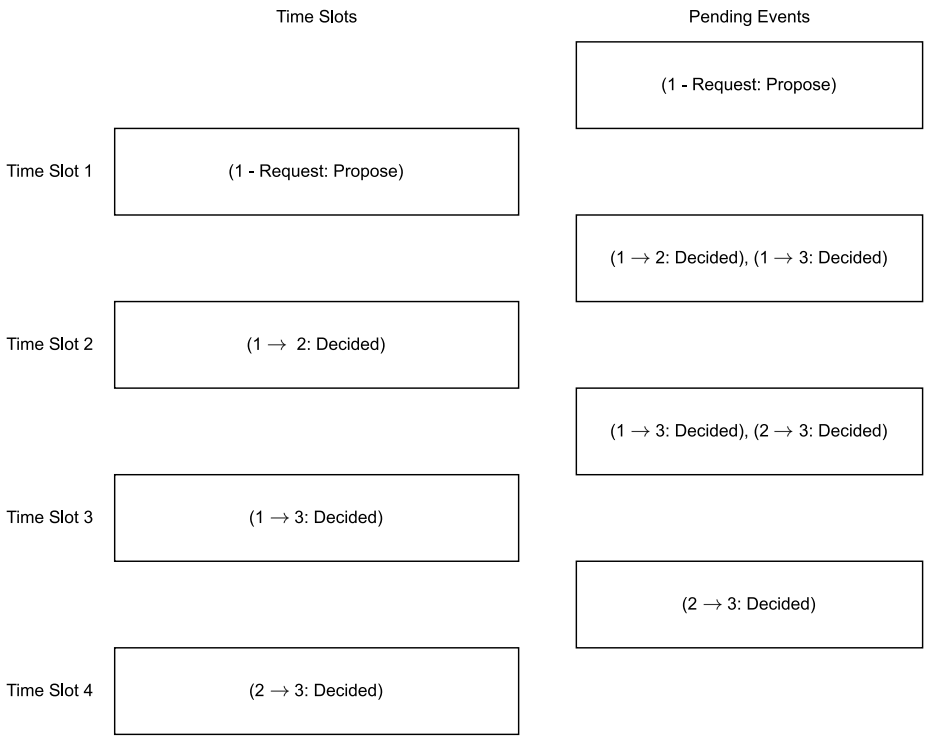


Figure 5: Example of global clock. The time slots are shown in the left column. The right column shows the pending events after the event assigned to the previous time slot has been executed, but before an event is assigned to the next time slot. The figure shows how one of the pending events is assigned to each time slot where it is executed, while the others wait until they are assigned to a time slot.

the implementation will therefore only require that the function is stored as a variable that can be replaced.

One of the advantages of the Event Managers is that they can be inserted at a high level of the implementation which reduces noise and prevents the creation of more events than necessary. Consider for example a network library that passes messages between nodes by performing calls to an underlying network connection. The network library might choose to split the message into several calls, merge multiple messages to the same call or transmit additional data that is unrelated to any specific message. If the events are captured in a low level, such as at the underlying network connection or at the OS layer, all these separate calls would be interpreted as individual events. This could create scenarios where certain events have no impact on the algorithm, where one event carries multiple messages or where you need multiple events to fire before a message can be delivered. This could break the atomicity or the determinism of events depending on how the network library choose to handle the message. Adding more events than necessary is also undesirable since it would increase the number of possible runs, and therefore increase simulation time, while not increasing the accuracy of the simulation. If we instead insert the Event Managers at a high level we can ignore the details of sending the message over a network and focus on capturing and executing the event.

Another advantage of using Event Managers is that they are adaptable and versatile. A new event manager can be created to handle new instances where we need to test the ordering of events or support new frameworks. This allows us to take advantage of the features of each specific framework, which will reduce the number of event created compared with a more general approach, and makes sure that all the features that are needed by the implementation are properly modeled.

This also makes it possible to use Event Managers to mock other services, similar to how we would mock services when testing non-distributed software. As an example, consider a scenario where we are testing an algorithm that uses a consensus instance. Instead of simulating the algorithm with the actual implementation of the consensus module we can use Event Managers to mock it. The Event Manager would add an event that represent the consensus module deciding on a value, and when the consensus module would have decided on a value the event fires. This has two important advantages. Firstly, it allows us to take a modular approach to testing distributed systems, which simplifies testing and makes it easier to identify faults. Secondly, it reduce the size of the state space: instead of scheduling all the events that would be necessary for the module, we only need to schedule the event representing the end of the operation.

Event Managers can also be used to represent different abstractions. The Event Manager that provides the `send` function used in the example implementation is based on the perfect link abstraction, but new event managers could be created to represent other abstractions. For example we could create an event manager that represent the *fair-loss* link abstraction by creating some mechanism for dropping messages or we could create an event manager that

ensures that messages from the same node are delivered in the same order as they where sent. This allows us to better represent the assumptions we make about the system and ensures that we don't need to implement the relevant functionality directly into the algorithm.

4.3 Scheduler - Assigning Events to Time Slots

In Go-MC a scheduler is responsible for deciding the order of events in each run. Different scheduler can be implemented based on different algorithms and the specific goals of the scheduler varies between different implementations, but it should attempt to order the events in such a way that the most errors can be found while also reducing the number of equivalent runs that are executed, to keep the simulation time as low as possible.

The scheduler receives events as they are discovered by the Event Managers and maintains a list of all pending events. A pending event is an event that has been discovered by an Event Manager in this run, but have not yet been executed. For each time slot the Scheduler select one of the pending events and assigns it to that time slot. After the event is executed the Scheduler proceeds to the next time slot, where it receives new events that was created by the execution of the previous event and selects an event from all the pending events. The selected event is assigned to the new time slot. Since the events are received during the execution of the algorithm the schedule is also created live, while executing the algorithm.

Conceptually, the Scheduler builds a tree of events, where each node represents an event and the depth of the node represent the time slot of the event in this run. The children of a node are the pending events after the event has been executed. The root of the tree is an empty *init* event and its children are the pending events at the first time slot. The same event will occur multiple times in the tree, since it will be part of multiple runs. One path from the root of the tree to a leaf node represent one run of the distributed system, and the complete tree represent all runs created by the Scheduler.

An example of an event tree can be seen in figure 6. After receiving the *Propose* request Node 1 broadcasts a *Decided* message to nodes 2 and 3. There are therefore two pending events that can be executed in time slot two: $1 \rightarrow 2 : Decided$ or $1 \rightarrow 3 : Decided$. When $1 \rightarrow 2 : Decided$ is executed a new event is added to the pending events: $2 \rightarrow 3 : Decided$. This can either occur in time slot 2 or in time slot 3, depending on which of the two pending events are scheduled to be executed in time slot 2.

There are many different scheduling algorithms and the work of developing new, more efficient, algorithms is still ongoing. Go-MC therefore uses a modular design where different scheduling algorithms can easily be designed and used while running simulations. We provide a set of basic scheduling algorithms as a proof of concept, but leave the development of more advanced schedulers for future work.

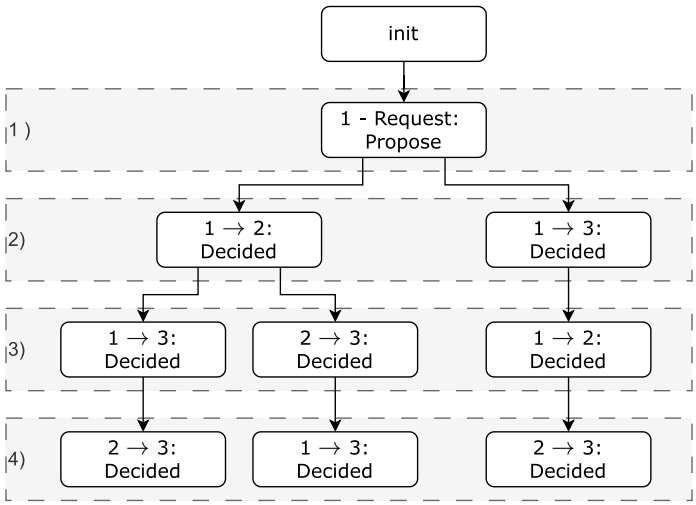


Figure 6: Event tree that is created when sending *Propose* request to example algorithm with three nodes. The specification $a \rightarrow b : t$ specifies that a message of type t was sent from node a to node b. The event in Time Slot 1 is a request event and the number specifies the target node.

4.4 State Manager - Collecting State From Nodes

To be able to verify that the properties of the algorithm holds we need to be able to collect the state of the system after each event has been executed. One solution is to record all state by creating copies of each node at each time slot. This is very costly and, since many of the variables stored in the node handles practical matters such as sending messages, it is an unnecessarily comprehensive approach. When verifying an algorithm we are interested in key variables showing that the algorithm progresses and that it takes the expected actions. We therefore only record a set of key variables that are specified by the user.

In addition to storing some selected variables we also store the status of each node and a record of the event that cause the transition into the current state. The status indicates whether a node has crashed or not and is therefore needed to verify the properties of a system. Storing events allows us to store each run as an alternating sequence of states and the event that caused transition from one state to the next, ensuring that we have both the information to verify and reproduce runs.

The event that caused the transition into this state and the local state and status of each node form the global state of the algorithm at a time slot. The global state of each time slot of each run is stored by the State Manager. By default, a State Manager that organizes the states into a tree is used, but Go-MC can use other implementations of State Managers for other representations of the state space.

Let us again consider the example implementation. One of the properties

that should hold for the algorithm is the *Termination* property, which states that every correct process eventually decides some value. To be able to test this property we need to know the decided values for a Node. We therefore store a slice of all decided values of a node as the local state of the node. Later, we will use this variable to verify that the property holds for the algorithm.

4.5 Checker - Verifying the Implementation

When checking the implementation we want to be able to provide the state space and a set of properties and we expect to get a boolean value indicating whether all the properties hold or not. If some property did not hold, the system should also return an example of a run that where the property did not hold, so that it is easier to identify and correct the error in the implementation.

Go-MC uses Checkers to verify that properties hold. The Checker uses the states space discovered during simulation of the algorithm and verifies that the specified properties hold for all states. The provided Checker specifies properties using Go functions. This ensures that the process of testing distributed systems is as similar as possible to the process of testing software. We also considered using a type of temporal logic, such as linear temporal logic or computation tree logic, to specify the properties, however temporal logic is not necessarily a subject that is familiar to all developers. We therefore believe that by specifying the properties using code we make it easier for all users to define and understand the properties.

We will now return to our task of verifying the *Termination* property of our example implementation. The local state stores a slice of all the values that a node has decided on. To define the property we create a Go function checking that the all nodes has decided on at least one value. Since the property is a liveness property we only check the states at the end of a run. To only check correct nodes we first check the status of a node, which is stored as a part of the global state, and only proceeds to check the rest of the property if the node is correct.

4.6 Failure Manager - Crashing nodes and detecting failures

An important part of testing distributed systems is ensuring that they work in the presence of failures. Go-MC therefore provide the option of triggering node crashes during the simulation of the algorithm. This is handled by the Failure Manager. The Failure Manager has three primary responsibilities: perform node crashes, notify nodes of changes in the status of other nodes and maintain global information about the status of nodes.

The Failure Manager performs node crashes by creating crash events for each of the faulty nodes at the start of the simulation. The Scheduler will then schedule the crash events among the other events, ensuring that all possible timings of crashes are tested. The user must configure how a node crash should

manifest on the node depending on the implementation used. For example, a node crash might close the network connection used to receive messages.

Distributed algorithms normally use the failure detector abstractions to identify crashed nodes. This usually involves sending additional messages which would increase the state space and increase the simulation time. Failure detectors are usually infinite, in the sense that they continuously send new messages, which would result in infinite runs causing the simulation to run for much longer than required. The Failure Manager therefore provides the functionality of a failure detector, where nodes can subscribe to the Failure Manager and be updated when the status of nodes change.

The Failure Manager maintains the global status of all nodes. This means that whenever the Failure Manager performs a change in the status of a node it also updates the global status of nodes that are reported to the State Manager. The global status of a node reflect whether it has actually crashed or not and can be different to the local view presented to the individual nodes. If the Failure Manager provides the Eventually Perfect Failure Detector abstraction the local view can for example show that a node crashes and later recovers – or, more correctly, that a node is suspected and then later restored – while in fact it might not have crashed at all. The global status would then show that the node was correct at all times during the run.

The default Failure Manager provided by Go-MC provides the *crash-stop* failure abstraction in a *synchronous* system. Different implementations of the Failure Manager can be created to represent different timing and failure abstractions. For example, the *crash-recovery* failure abstraction can be supported by implementing a mechanism for recovering crashed nodes. An asynchronous system could be represented by nodes not subscribing to status updates.

The example algorithm uses a Perfect Failure Detector to detect if a node has crashed and to advance to the next phase if it has. To use this provided failure detector abstraction, each node subscribe to the Failure Manager by providing a callback function that is called when a node is detected to have crashed. We also configure the simulation to trigger node 2 to crash during the simulation.

4.7 Runner - Capturing a Live Run

In addition to providing a simulator that simulates the execution of the algorithm, Go-MC also provides a Runner which runs the algorithm while recording the messages that are sent between the nodes and the local executions performed on a node. The Runner also provide functionality to pause and resume the execution of an algorithm on a specified node, to crash a specified node or to send requests to a specified node. The goal of the Runner is to generate a visualization that can be used to better understand and troubleshoot the algorithm.

The Runner does not enforce Rule 2. Instead it enforces the weaker Rule 1, where each nodes execute its events sequentially, but multiple nodes execute events simultaneously. Rule 1 allows us to produce the sequence of events that was executed on each node, which is useful when producing visualizations of the

algorithm such as a space-time diagram. It also enforces the common modeling of nodes as executing events in a sequential manner. The rule is enforced in a similar way as in the simulator, where the simulator waits for an event to signal that is has been completely executed before starting to execute the next event, but the runner maintains one such loop for each node, instead of one global loop for all nodes. The loop is maintained by by a Node Controller, which maintains the incoming event for a node and assigns them to the time slots in a first in first out order.

The Runner records when a node sends or receive a message or when a node performs some local execution. After a node executes some event, i.e. receive a message or perform local execution, the Runner record the new state of the node. The Runner does not store the state of the nodes, instead it sends the collected records to subscribed users. The sequence of records represent a partial ordering of the events that was executed during the run and the order of events that was executed concurrently on different nodes is arbitrary.

5 Implementation

This section will provide a technical overview of Go-MC. It presents a overview of the types used to implement Go-MC and its modules. Sections 5.1 provides details on how to configure the Go-MC simulator and runner to test an algorithm. The remaining sections details the implementations of each module.

Conceptually, the system works as described in section 4, but the modules are designed to be able to simulate multiple runs in parallel. This allows us to take advantage of stronger processors and multi-threaded computing to reduce the overall simulation time. To be able to do this many of the modules are split into two parts: a global part, which manages global state across several runs, and a run-specific part, which manages the information of a single run at a time and communicates with the global part of the module when necessary.

Go-MC uses two generic types to make it easy to use the system with different algorithms. The first generic type is represented by the name `T` and it represents the node that runs the algorithm. All nodes in the simulation must be represented by the same type. For the purpose of the simulation all nodes are considered to have a unique integer id. The second generic type is represented by the name `S` and it is a type storing the local state that is collected by the simulation. The use of generics make it easy to create functions to collect and compare state, and to perform actions on a node without having to change the implementation of the system.

5.1 Configuring Go-MC

Go-MC provides the `PrepareSimulation` function (Listing 5.1, `gomc/configSimulator.go`) to simplify the configuration of the simulation. The function provides default values for several parameters that are required by the `Simulator`. A full overview of the available options, their default values and the functions used to configure them is provided in Appendix B.1.

```
1 func PrepareSimulation[T, S any](smOpts StateManagerOption[T,
  ↪  S], opts ...SimulatorOption) Simulation[T, S]
```

Listing 5.1: Signature of the `PrepareSimulation` function. The function initializes the simulation with the provided configuration. The returned `Simulation` type can be used to start several simulation with different parameters. The `smOpts` is mandatory and configures the State Manager that should be used. All other parameters are optional.

The instance of the `Simulation` type returned by the function contains the configured simulator. The simulator can be used to run a simulation by calling the `Run` method (Listing 5.2, `gomc/configSimulator.go`) and providing the required parameters. One `Simulation` instance can be used to perform multiple simulations, but multiple simulations can not be performed in parallel.

```
1 func (sr Simulation[T, S]) Run(InitNodes InitNodeOption[T],
  ↪  requestOpts RequestOption, checker CheckerOption[S], opts
  ↪  ...RunOptions) checking.CheckerResponse
```

Listing 5.2: Signature of the `Simulation.Run` method. The method starts a simulation with the provided nodes and requests. `InitNodeOption` specifies how the nodes are initialized. `RequestOption` configures a list of the requests that will be sent to the nodes and which are used to start the simulation. The `CheckerOption` configures the checker used to test the algorithm. The `RunOptions` are optional parameters that can be configured. The simulation will use default values if they are not configured.

The `Run` method has three mandatory parameters (Appendix B.1.3) and the `RunOptions` which are optional parameters (Appendix B.1.4). These parameters configure the scenario used to run the simulation, such as how many nodes are a part of the system, what requests are sent to the system and how to collect the state from the nodes. The simulation will use default values for the `RunOptions` if no configuration is provided.

The `InitNodeOption` configures a function used to create the nodes that are used in each run. The function should create the nodes and register any Event Managers that are used during the simulation. It returns a map, containing (node id, node) key-value pairs for each of the nodes in the simulation. The `SimulationParameters`(`gomc/event/simulationParameter.go`) variable contains run specific parameters provided by the simulator. These parameters are used to initialize the Event Managers.

```
1 func(sp SimulationParameters) map[int]*T
```

Listing 5.3: Signature of the function used to create the nodes. The returned map contains (node id, node) key-value pairs. The function should be used to create the nodes and initialize the Event Manager used during the initialization of a run.

`RequestOption` configures the simulation to use the provided set of `Requests` when running the simulation. A `Request` represent some incoming request to the system and is used to start the simulation. Requests are created by the `gomc.NewRequest` function (Listing 5.4, `gomc/requests.go`). The function is called with the id of the node receiving the request, a string which is the name of the method of the node that will be called to handle the request and any parameters that should be provided to the method.

The `Simulation.Run` method returns a `CheckerResponse` type. The `CheckerResponse` contains the results from the simulation. It contains information about which properties held as well as counterexamples if some of the properties

```
func NewRequest(id int, method string, params ...any) Request
```

Listing 5.4: Signature of the `NewRequest` function. The function creates a `Request` type. The `id` parameter specifies the node that receives the request. The `method` parameter specifies the method that will be called to handle the request and the `params` parameter specifies the parameters that will be provided to the method.

where broken. More details about the `CheckerResponse` can be found in Section 5.8.

Go-MC also provides the `PrepareRunner` function to simplify the configuration of the `Runner` (Listing 5.5, `gomc/configRunner.go`). The function takes two mandatory parameters and three optional parameters (Appendix B.2).

```
func PrepareRunner[T, S any](initNodes InitNodeOption[T],
↪  getState GetStateOption[T, S], opts ...RunOptions)
↪  *Runner[T, S]
```

Listing 5.5: Signature of the `PrepareRunner` function. The function starts the `Runner` with the provided configuration. The running of the algorithm can be started by calling the provided functions on the returned `Runner` type. More details about the available functionality can be found in section 5.9.

The `PrepareRunner` function returns a configured and running instance of the `Runner` type. It is ready for use and commands can be sent to the nodes by calling the respective methods on the provided `Runner`.

5.2 Simulator

The Simulator is the controller of the simulation. It maintain the main loop used to perform the simulation of each run and controls the execution of the various modules. It consists of a global type, the `Simulator` (`gomc/simulator/simulator.go`), and a run specific type, the `runSimulator` (`gomc/simulator/runSimulator.go`).

The `Simulator` manages the global information about the runs that are simulated. It coordinates the multiple `runSimulators` as they each simulate a run, by signaling to the `runSimulators` when to start a new run and managing the errors that occur during simulation of a run.

The `Simulator` start the simulation by initializing the `runSimulators` with their own instances of a `runScheduler`, a `runStateManager` and `runFailureManager`. The number of `runSimulators`, and therefore the number of concurrent runs that can be executed, is decided by the `numConcurrent` variable.

After the `runSimulator`s have been initialized the main loop of the `Simulator` is started. The main loop receives the status of completed runs from the `runSimulator` and signals to the `runSimulator` to start new runs. It keeps track of the number of runs that have been simulated and stops the simulation if the maximum number of runs have been reached. The maximum number of runs can be configured by the `maxRuns` variable.

The main loop handles errors reported by the `runSimulator`s according to how the `ignoreErrors` flag is set. If the `ignoreErrors` flag is `true`, the error will be recorded, but the simulation will continue to the end. After the simulation have been completed a summary of all the errors that occurred will be provided. If the `ignoreErrors` is `false` the simulation will stop immediately and return the error.

The main loop runs until all `runSimulator` have stopped. This can happen for three reasons: The maximum number of runs have been reached and the main loop is no longer starting new runs, there are no more runs to simulate, or an error occurred during the simulation of a run and the `ignoreErrors` is set to `false`.

The `runSimulator` performs the simulation of each run. Before starting the simulation of a run the `runSimulator` waits for a signal from the `Simulator`. When the signal is received it begins simulating a run. The simulation of a run consists of three phases: initialization of the run, execution of the run and teardown of the run. After the simulation of a run has been completed any errors that occurred during the simulation is sent to the `Simulator`. If no errors occurred a `nil` is sent instead.

The initialization of the run consists of creating the nodes that constitutes the distributed system and initializing the modules for a new run. The State Manager collects the initial state of the system and the configured requests and crashes are added to the scheduler. The nodes are created using the `initNodes` function provided by the user.

The execution of a run consists of running the main loop described in section 4.1 until either all events have been executed or the maximum depth have been reached. The main loop consists of getting the next event from the Scheduler, executing the event and collecting the global state of the system. If any errors is detected during the execution of a run the execution is ended and the error is returned.

The maximum depth of the simulation is set by the `maxDepth` parameter. It ensures that the simulation has a finite size, even if we are simulation an infinite run. This is important to ensure that the simulation completes. It also assists us in reducing the size of the state space in complex algorithms. If the maximum depth is reached, we can not guarantee that the algorithm is correct, even if no states violating the properties are found, because there are valid states which we have not discovered in which the properties might be violated.

By default `runSimulator` will not recover from panics that occur during the execution of an event on a node. This can be changed by setting the `ignorePanic` flag to `true`, which will cause the panic to be reported as an error and the simulation of other runs can potentially continue, depending on other settings.

The teardown of the run is always performed at the end of the run, even if an error occurs during the initialization or execution of a run. The teardown signals to the `runScheduler` and the `runStatemanager` that the run is over and closes all the created nodes. This prepares the `runSimulator` for the next run and sends information about the run to the global `Scheduler` and `StateManager`.

5.3 Events

In Go-MC an event represent an action that causes a transition from one state to another. In general an event correspond to a block of code in the algorithm that is executed.

Each event (Listing 5.6, `gomc/event/event.go`) is identified by an id. Two events that have the same effect, that is two events which given the same input state produces the same output, should have the same id. The id can be accessed by the `Event.Id` method. Good practice is to add an identifier of the event type in the id, to avoid unintended collision with other implementations of events.

After the event has been completely executed it must send a signal to the simulator that the event is completed and that the simulator can continue. This is done using the `nextEvt` channel. If any errors occurred during the execution the error is sent on the channel, otherwise `nil` is sent. Events are executed in a separate goroutine. This allows events, such as the `SleepEvent`, to split a message or request handler into two events.

```
1 type Event interface {
2     Id() EventId
3     Execute(node any, nextEvt chan error)
4     Target() int
5 }
```

Listing 5.6: Interface of `Event`. The `Id` returns the id of the event. Two events that, given the same input state, produces the same output state should have the same id. `Execute`, executes the event on the provided node. `Target` returns the id of the target node.

Go-MC also provides an interface for `MessageEvents` (Listing 5.7, `gomc/event/event.go`). `MessageEvents` are events that represent messages. In addition to the methods provided by the `Event` interface, `MessageEvents` also provide methods returning the id of the sender and the receiver of the message. Message events are created when the message is sent and executed when it is received on the target node. They are not used by the simulation, but can be used to identify when an event was sent, for example when visualizing the simulation.

```
type MessageEvent interface {
    Event

    To() int
    From() int
}
```

Listing 5.7: Interface of `MessageEvent`. The `To() int` method returns the id of the target of the message and `From() int` returns the id of the sender of the message.

5.4 Event Manager

Event Managers are types or functions that are inserted into the algorithm. They form the interface between the simulator and the algorithm. They do not implement a specific interface, but they often adopt the signature or interface of primitives used by the algorithm, to make it easy to insert them into the algorithm. They replicate the behaviour of these primitives and provides functionality that allows the simulator to record and control the execution of events.

The Event Managers uses the `EventAdder` interface (Listing 5.8, `gomc/eventManager/eventAdder.go`) to add events that are discovered during the execution of the algorithm. The `EventAdder` are an interface representing some type that the Event Manager can add events to. The Simulator uses Schedulers as `EventAdders` while the runner uses `RunnerController`.

```
type EventAdder interface {
    AddEvent(event.Event)
}
```

Listing 5.8: Interface of `EventAdder`. Represent some type that can receive events.

Some Event Managers also uses the `NextEvent(error, int)` function to signal to the system that the execution of an event is completed. The first parameter passed to the function is any error that occurred during the execution of the event. If no error was occurred, the value is `nil`. The second parameter is the id of the node that completed the event.

The Event Managers are configured when the nodes are initialized. The `SimulationParameters` variable that is passed to the function contains the instance of `EventAdder` and `NextEvent` function that is used to simulate the run. The Event Managers should be initialized with these variables to be able to add events to the correct run.

5.4.1 Sender

Go-MC provides Event Managers that represents several common primitives used to design distributed systems. The `Sender`(`gomc/eventManager/sender.go`) event manager represent the perfect link abstraction. It assumes that the nodes have a set of message handlers, which are methods used to handle incoming messages. The message contains a set of parameters, which are used to call the method on the node. The `Sender` Event Manager provides an abstract, but convenient, representation of a network link and can therefore represent many different solutions for sending messages between nodes.

There are two reasons that the `Sender` does not represent the fair loss links. Firstly, many commonly used technologies guarantees delivery of messages. Therefore, there is no strong need to provide functionality that simulate fair loss links. Secondly, it is not obvious how to implement such a link without dramatically increasing the size of the state space. For the simulation to be correct we must test all possible combinations of message drops, in addition to the different ordering of the messages. This dramatically increases the size of the state space, for little gain in the applicability of the simulation.

The `Sender` is initialized with the `EventAdder` that is used for this run. It provides one method, which is a factory method that returns the send function. The send function is then used by the node to send the messages. The factory method configures the send function with the id of the node, so that the messages can be created with the correct sender. One `Sender` can be used to create multiple send functions for different nodes.

The send function created by the `Sender` creates a `MessageHandlerEvent` for the message and adds it to the `EventAdder`. When the event is executed it calls the specified message handler on the target node with the provided parameters.

5.4.2 gRPC Manager

gRPC is a multi-environment framework that handles many aspects of communication between nodes. This includes guaranteed delivery of messages, calling the correct message handler, etc [9]. Go-MC provides an Event Manager that can be used to control messages sent using the gRPC framework for Go. The Event Manager can be used when the RPC are made asynchronously, using goroutines, and where we do not wait for a response. An example of this can be seen in listing 5.9. This pattern is a common way of using RPCs to send messages in a distributed systems, since it makes it possible to concurrently send messages to multiple nodes simultaneously and because the execution can continue without waiting for a response. Which makes it easy to use and more efficient than traditional RPC calls.

The Event Manager is implemented as a client side unary interceptor that can be inserted into the gRPC client when it is configured. The Event Manager waits until the event is executed before the message message is sent to the server, it then waits until an (empty) response is received before signaling that the event

```
1 go ExampleGrpcService.ExampleRpc(context.Background(), msg)
```

Listing 5.9: Example of an asynchronous RPC call. Note that the `ExampleRpc` call is made in a separate goroutine and that it does not wait for a response.

is completed. Since physical time is not accurately simulated it is important that functionalities that uses timeouts, such as contexts, are not used.

Since messages are sent in individual goroutines it is possible that the calling method returns before the RPC call has been made and the corresponding events have been added to the `EventAdder`. This would cause the event to signal its completion before the event was actually completed. This could be solved by waiting an arbitrary amount of time after the event signals that the event has been completed, but this does not actually guarantee that all events have been added and it causes unnecessary delay which would slow down simulation. Instead, the Event Manager uses a `waitForSend` method. The method waits until it receives confirmation from the unary interceptor that the specified number of events have been added before it continues, thus ensuring that all events have been added before continuing. This function must be called by the algorithm, with the number of events to wait for as the parameter. An example of how the `WaitForSend` method is used can be seen in Listing 5.10.

```
1 for _, node := range p.nodes {
2     go node.Accept(context.Background(), msg)
3 }
4 p.waitForSend(len(p.nodes))
```

Listing 5.10: Example usage of the wait for send function. The function is called after sending a number of asynchronous RPC calls. It is used to ensure that all messages are properly added before continuing.

5.4.3 Sleep Manager

Timeouts are an important part of many algorithms, but since physical time is not represented in the simulation it is hard to accurately test algorithms containing timeouts. Go-MC therefore provide the `SleepManager` (`gomc/event Manager/sleepManager.go`), which is an event manager that can be used to represent timeouts. The Event Manager provides a sleep function that imitates the signature of the `time.Sleep` function, making it easy to replace the Event Manager with the actual function after testing.

When the sleep function is called it creates a `SleepEvent` (`gomc/event /sleepEvent.go`) and informs the system that the previous event has been completed. The function does not return until the event is executed, at which point the rest of the code block that called the sleep function will be executed.

The `SleepEvent` splits a code block into two pieces. The first piece that started the timeout and is a part of the previous event, and the second piece which is the code that is executed after the timeout fires and is a part of the `SleepEvent`. When the sleep function is called it will therefore call the `NextEvent` function to signal to the simulation that the previous event has been completed. Other events can then be executed before the `SleepEvent` is executed. This represent that events can be executed while the node is waiting for the timeout to fire and allows us to interleave the firing of the timeout with the receipt of messages.

5.5 Scheduler

The scheduler is responsible for managing the events and controlling the order in which they are executed. It receives events as they occur during the execution of a run and continuously returns the next event in the run from the events it has received, until all events have been returned. The mechanism for selecting the next event to return is dependent on the specific implementation of the scheduler, but the goal should be to select events in such a way that the most errors can be detected using the least amount of time.

The scheduler is divided into two parts: the `GlobalScheduler` and the `RunScheduler` (`gomc/scheduler/scheduler.go`). The main role of the `GlobalScheduler` is creating and configuring the corresponding `RunSchedulers` and coordinating between the `RunSchedulers`. It maintains the global scheduling information, for example which runs have been explored and which runs have not been explored.

The `RunScheduler`d are created by the `GlobalScheduler` and they are responsible for performing the local scheduling of a run. They communicate with the `GlobalScheduler` when necessary. A `RunScheduler` might for example report a run that it discovered but did not explore to the `GlobalScheduler`. The `GlobalScheduler` can then instruct the next `RunScheduler` to explore this run. A `RunScheduler` can be used to simulate multiple runs, but only one run at a time.

Go-MC provides four basic schedulers: The Prefix scheduler, the Random scheduler, the Replay scheduler and the Guided Search scheduler. They do not implement the state of the art within scheduling techniques, but are intended as proof of concepts, showcasing the different types of schedulers that can be created and providing a basic suite of schedulers that can be used to simulate algorithms. The development of more advanced schedulers that utilize state of the art state space minimization techniques is left for future work.

The prefix scheduler (`gomc/scheduler/prefix.go`) is a stateful, deterministic scheduler that completely explores the state space. It guarantees that all possible states are visited. The scheduler operates by systematically testing all possible permutations of runs. As it simulates runs it builds a view of the state space, which it uses to ensure that all possible runs are executed. It relies on the determinism of events to be able to build a view of the events and correctly explore the entire state space. The algorithm used by the prefix scheduler is explained in Appendix C.

The Random scheduler(`gomc/scheduler/random.go`) is a randomized, stateless scheduler. It does not explore the entire state space, but randomly samples possible runs. Thus, it does not provide any guarantees to discover all errors, but provide larger variation in the runs it produces compared with a Prefix Scheduler. This makes the Random scheduler useful for simulations where there is not enough time to explore the entire state space. It operates by randomly selecting the next event from the pending events.

The replay scheduler(`gomc/scheduler/replay.go`) replays a single run. The run is provided as a sequence of event ids and can be retrieved from a `CheckerResponse` (See section 5.8 for more details on the `CheckerResponse`). The replay scheduler is intended as a tool that helps when troubleshooting algorithms and it might be desirable to replay a specific run and debug the execution of the algorithm. When using the replay scheduler it is important that the configuration of the simulation is the same as the configuration of the simulation that was used to generate the original run.

The guided search scheduler (`gomc/scheduler/guidedSearch.go`) is a variation of the replay scheduler. The scheduler initially follows a provided run until all the events in the run has been selected, then it begins searching using some other scheduler. The guided search scheduler makes it possible to guide the search towards some interesting state before the proper exploration of the state space begins.

5.6 State Manager

The State Manager is responsible for collecting, storing and organizing the global state of the system during the simulation.

The State Manager consists of a `StateManager` (`gomc/stateManager/stateManager.go`) and a `RunStateManager`. The `StateManager` maintains the state across multiple runs. It collects the state from the individual runs and organizes it into some state space representation. It consists of four methods: `GetRunStateManager`, which creates a `RunStateManager` that can be used to simulate runs; `AddRun`, which is called by the `RunStateManager` when a run have been completed and which should add the provided run to the globally discovered state space; `State`, which returns a representation of the discovered state space; and `Reset` which reset the collected state, so that the `StateManager` can be used for a new simulation. The `AddRun` method will be called by multiple `RunStateManager`, so it must be concurrency safe.

Go-MC provides an implementation of the `StateManager` interface, called the `TreeStateManager` (`gomc/stateManager/treeStateManager.go`). This state manager organizes the state space as a tree. It is initialized with two variables: `getLocalState`, which collects the local state of a node, and `stateEq`, which compares two local states and returns true if they are equal. The two functions ensures that the State Manager can easily be adapted to different algorithms. The root of the tree is the initial state, captured before any event is executed. Each path from the root to a terminal node represents one run of the algorithm.

The `RunStateManger` (`gomc/stateManager/runStateManager.go`) is responsible for collecting the global state of the system and managing it across a run. It manages the `run` variable, which is a sequence of the global states that have been recorded in this run.

The `UpdateGlobalState` method is called in each time slot, after the event has been executed, and it collect the global state of the system and adds it to the run. It is also called at the start of each run, before the first event is executed, to capture the initial state of the system. The global state is created from the local state of the nodes, the map of correct processes and the event that caused the transition into this state, that is the event that was executed in this time slot, and it is stored as a variable of type `GlobalState` (`gomc/state/globalState.go`).

The `EndRun` method is called at the end of each run and it sends the sequence of events collected as a part of the run to the `StateManager`, by calling the `StateManager.AddRun` method with the collected run. The method then resets the `run` variable, preparing for the next run.

5.7 Failure manager

The Failure Manager is used to perform node failures and manage information about the status of the nodes. It also provides the functionality of a failure detector where nodes can subscribe to notifications on changes in the status of the nodes. Different implementations of Failure Managers can be used to represent different abstractions of failure detectors.

The global part of the Failure Manager is the `FailureManger` (`gomc/failureManager/failureManager.go`). It manages the global information across runs, for example the configuration of the Failure Manager, and creates the `RunFailureManagers`. The `GetRunFailureManager` methods creates a configured `RunFailureManager` that can be used when simulating a run.

The `RunFailureManager` is the run specific part of the Failure Manager. The `FailureManager` contains a `Subscribe` method, which allows nodes to subscribe to notifications about changes in status of the nodes in the system. The `Subscribe` method is exposed to the nodes through the `SimulationParameter` type that is provided when initializing the nodes.

Go-MC provides an implementation of the `FailureManager` called the `PerfectFailureManager` (`gomc/failureManager/perfectFailureManager.go`). The `PerfectFailureManager` represents the Perfect Failure Detector abstraction. It is configured with a set of faulty nodes which will crash at some point during the simulation. The `PerfectFailureManager` will create and add a `CrashEvent`(`gomc/event/crashEvent.go`) for each of these faulty nodes and the scheduler will interleave the crashes with the other events.

The `PerfectFailureManager` is configured with a `crashFunction` which performs the crash on the provided node. This allows users to specify how a crash manifests on the specific implementation. The `crashFunction` should ensure that the node does not execute any events or respond to any received messages.

When the `CrashEvent` is executed it calls the `nodeCrash` method of the `PerfectFailureManager` with the node id of the crashing node. The method sets the node as crashed in its internal structure. It then calls the configured `crashFunction` on the target node, triggering the crash of the node. Finally it adds an `CrashDetection` (`gomc/event/crashDetection.go`) event for each of the nodes that have subscribed to the status notifications. When the `CrashDetection` event is executed it calls the provided crash callback provided by the node when it subscribed to the Failure Manager. `CrashDetection` events will be interleaved with the other events. This represents the fact that a node can not discover that another has crashed immediately, but will only eventually detect that a node has crashed.

When configuring the `PerfectFailureManager` a user must provide the node id of the nodes that will crash during the simulation. Another option would be to automatically try all combinations of crashed nodes. This is inefficient, since many nodes in a distributed system perform the same task and any of these nodes crashing will have the same impact on the overall system. It would therefore be inefficient to simulate runs where each of them crashes, since it causes the same outcome. By relying on the user to configure the nodes that will crash we can use the users knowledge of the algorithm to make smart decisions. This will ensure that the simulation is focused on the parts of the algorithm where we expect to find errors.

5.8 Checker

The `Checker`(`gomc/checking/checker.go`) is responsible for verifying that the algorithm is correct. It is invoked on the discovered state space after the simulation has been completed and verifies that a set of properties holds for all the states that where discovered during the simulation of the system. It returns a `CheckerResponse` (`gomc/checking/checker.go`) object which hold the result for the simulation.

The `CheckerResponse` stores the result of the checking. It provides two methods that are used to provide and explain the results to the user. The formatted text that is returned from the `Response` method should provide a detailed description of which property is violated and which sequence of events caused the state where the property is violated. The sequence of events that are returned from the `Export` method can be used with the `ReplayScheduler` to recreate the run that caused the property to be violated.

Go-MC provides the `PredicateChecker` (`gomc/checking/predicateChecker.go`), which is a `Checker` that uses predicates defined as Go functions to specify properties. The `PredicateChecker` performs a depth-first search through the state space and runs all predicates on each of the discovered states (`gomc/checking/state.go`). If the checker finds a state for which a predicate does not hold the search is stopped and a `PredicateCheckerResponse` is created. The `PredicateCheckerResponse` contains the index of the predicate that failed and the sequence of global states that resulted in the failed predicate. It can also export the sequence of events that caused the failed predicate, which

can be used to replay the run.

To simplify the process of specifying properties Go-MC provides two helper functions (`gomc/checking/predicate.go`). `Eventually` ensures that the provided predicate is only run on terminal states. This makes it useful when specifying properties that are eventually true. These properties can not be tested on all states, since there is still the possibility that it will be true in the future, but they can be tested on the terminal state, since this is the state after the system has completed its execution. If the depth of a run reach the configured `maxDepth` the last recorded state of the run will not be the state after the system has completed its execution and the predicate can not be used to verify properties that are eventually true, since it is still possible that the property would be true at some point after the simulation stopped. `ForAllNodes` checks that a condition holds for all nodes. It is also possible to use it to check that a condition holds for all correct nodes.

5.9 Runner

The Runner provides functionality for running the algorithm and recording the messages that are sent between the nodes using the Event Managers. The runner gives a more realistic view of the algorithm by executing events on multiple nodes concurrently and can be used to visualize and increase understanding of the algorithm.

The `Runner`(`gomc/runner/runner.go`) initializes the running of the algorithm and provides an interface that can be used to send commands to the nodes. It uses a Runner Controller to manage the running of the algorithm and the Node Controller to control individual nodes. The `Runner` allows a user to pause and resume nodes, crash nodes and send requests to nodes. A user can also subscribe to `Records` collected during the running of the algorithm.

The running of an algorithm is started with the `Start` method. The user provides a function that creates the nodes that will be used during the simulation. This is the same function that is used to start the nodes in the `Simulator`. The `getState` is a function that collects the local state from a node, similar to the function used to create the `TreeStateManager`, and the `stop` parameter is a function specifying how to stop a node, similar to the `stop` function passed to the `Simulator`. The `stop` function is used to both stop and crash the node.

`Records` (`gomc/runner/records.go`) are information collected during the running of the algorithm and reported to the user. They represent messages that are sent or received by nodes, local executions on nodes and the local states of nodes. `ExecutionRecord` and `MessageRecord` represent different types of events, while `StateRecord` is the new state after some event has been executed on the node. `MessageRecord` are created by both the sending node when it sends the message and on the receiving node when it receive the message.

The Runner Controller(`gomc/runner/runnerController.go`) performs the actions instructed by the `Runner`, manages the Node Controllers and relays the `Records` received from the nodes to the subscribed users. It also implements the `EventAdder` interface (`gomc/eventManager/eventAdder.go`) and provides

the variables used to create the `SimulationParameters` (`gomc/eventManager/simulationParameter.go`).

The Runner Controller exposes the four commands that are given to the nodes. When the Runner Controller receives the command it selects the specified node and executed the command on the node. The commands that are exposed by the Runner Controller is: `Pause`, which pauses the execution of events on a node; `Resume`, which resumes the execution of events on a node; `CrashNode`, which triggers the node to crash; and `NewRequest`, which sends a request to the node. The `CrashNode` method first selects the node and closes it. It then adds a `CrashDetection` event to all subscribed nodes.

The Runner Controller provides the methods used to create the `Simulati onParameters` type. The `AddEvent` method receives the events from the event managers. It then selects the target node of the event and adds the event to the pending events of the node by calling the `addEvent` method of the node. Similarly, when an Event Manager calls the `NextEvent` method with the error and the id of the node the Runner Controller selects the specified node and calls the `nextEvent` method on that node. The `CrashSubscribe` method registers the provided callback function under the provided node id.

The Node Controller (`gomc/runner/nodeController.go`) manages the execution of events on a single node. The events are executed in first-in first-out order, according to when they where added to the node. The node controller maintains the pending events for the node and execute the pause, resume and crash commands on the node. One Node Controller is created for each node.

Each node controller maintains one main loop for its node which ensures that events are executed sequentially. The main loop is similar to the main loop used by the `runSimulator` to simulate one run of a simulation, but it does not utilize a Scheduler. It waits for an event to be added to the Node Controller. The event is recorded and executed in a separate goroutine and the loop waits until it have been completed by waiting on the `nextEvtChan`. After the event has been executed the new state of the node is recorded. The Node Controller pauses the execution of event on a node by pausing the main loop. It simulates a crash of the node by stopping the main loop and preventing the execution of new events. The `crashFunc` is then called on the node.

6 Evaluation

We performed two tests that will help us evaluate the ability of Go-MC to verify distributed algorithms. In the first test we investigated the time required to run simulations on simple scenarios and in the second test we investigated the ability to find errors in a distributed algorithm.

6.1 Simulation Time

In the first test we investigated the time required to perform the simulation. We created six implementations of distributed algorithms and used Go-MC to simulate the execution of a simple scenario for each of the implementations (Table 1). We created two implementations of the Hierarchical Consensus algorithms (Algorithm 5.2[5]): one using the *Sender* Event Manager and one using the *GrpcEventManager* Event Manager. We also implemented the Eager Reliable Broadcast (Algorithm 3.3 [5]) and the Majority Voting Regular Register(Algorithm 4.2 [5]) using the *Sender* Event Manager. Finally we implemented two versions of the Paxos algorithm using the *GrpcEventManager*, one using the basic version, where phase 1 is performed for each value(Paxos), and one using the optimization where phase 1 is only performed once when a new leader is elected(Multipaxos)[14]. For more details about the algorithm see Appendix A.1. We benchmarked each scenario with 10 repetitions and recorded the total time for executing all repetitions.

Name	**Scheduler**	**Nodes**	**Faulty Nodes**
Hierarchical Consensus (Sender)	Prefix	$\{1, 2, 3\}$	$\{2\}$
Hierarchical Consensus (gRPC)	Prefix	$\{1, 2, 3\}$	$\{2\}$
Eager Reliable Broadcast	Prefix	$\{1, 2, 3\}$	$\{2\}$
Majority Voting Regular Register	Random	$\{1, 2, 3\}$	$\emptyset$
Paxos	Random	$\{1, 2, 3\}$	$\{2\}$
Multipaxos	Prefix	$\{1, 2, 3\}$	$\{2\}$

Table 1: Summary of configuration for test of simulation time. All simulations have been run with default paramters, unless otherwise states. When using the random scheduler the seed was set to 0. For a full overview of the configuration see D.1.

The results (Figure 7) shows that Go-MC can simulate small scenarios in a relatively short time. The results also show that there is a notable difference in the simulation time between algorithms that used the *Sender* Event Manager and the algorithms that used the *GrpcEventManager*. This could be caused by the additional overhead of using the gRPC framework.

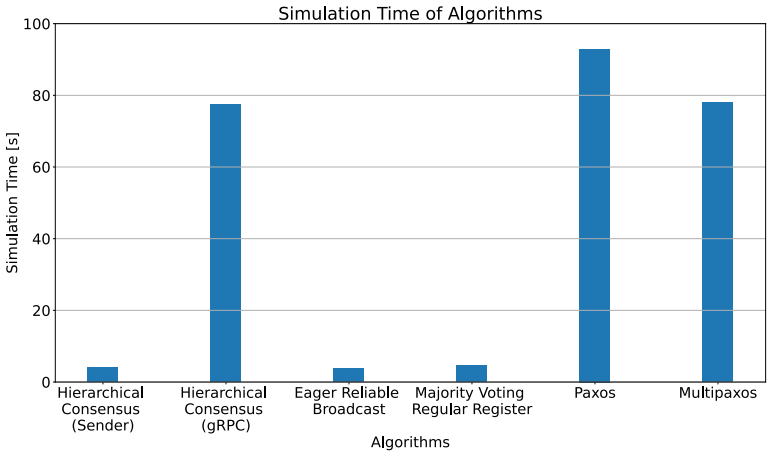


Figure 7: Simulation time of each algorithm. The simulation time is the total time to perform 10 simulations.

6.2 Finding Bugs

To evaluate how good Go-MC is at identifying and reporting bugs we have applied it to the example algorithm from 4. We then inserted a known bug into the algorithm, and ran several scenarios to see if Go-MC was able to identify the bug.

We used an implementation of the Hierarchical Consensus algorithm introduced in section 4. Recall, that the algorithm works by having each node decide its value in a specified round, and that all nodes impose the value it selects on all subsequent nodes. We introduce a bug where the algorithm does not progress multiple times to the next round if a node crashes and the algorithm enters a round where the node has already crashed or decided a value (Listing 6.1, Figure 8). This causes the algorithm to freeze and violates the *Termination* property.

We ran the simulation of the algorithm six times with different configurations (Table 2). We ran the scenarios with both a `Prefix` scheduler and a `Random` scheduler, to compare how different scheduling techniques affects the results. Since the bug only occurs when a node has crashed, we do not expect any errors to be detected when the simulation is configured with no faulty nodes. We configure the simulation with different number of nodes to investigate how the size of the distributed system affected the simulation. We measured the time used to run each simulation to compare how different configurations affected the simulation time.

The results from the test of the Hierarchical Consensus algorithm (Table 3, Figure 9) shows that Go-MC is able to find bugs in distributed systems, but that the configuration that is used affect the effectiveness. In the simulations with seven nodes we can see that the `Prefix` scheduler is unable to find the bug,

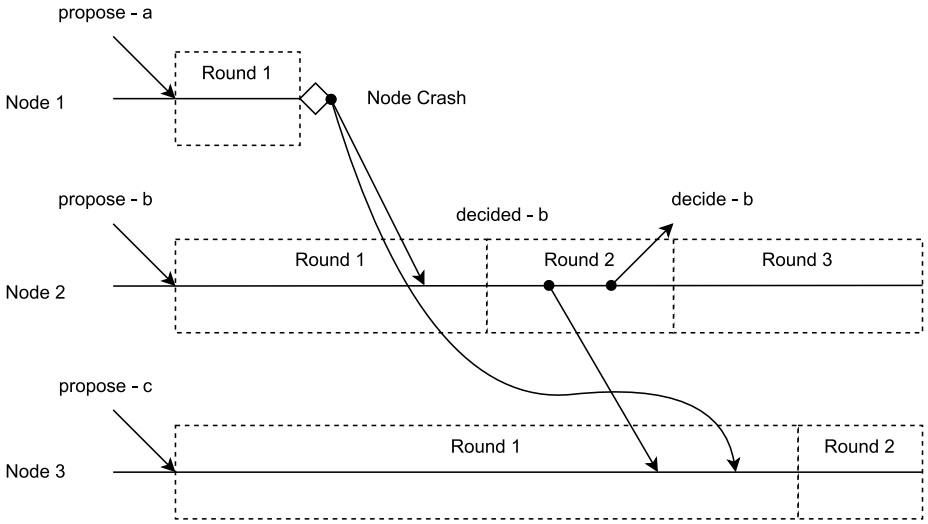


Figure 8: Space-Time diagram showing one run where the bug introduced in the Hierarchical Consensus algorithm occurs. The dotted boxes shows which round of the algorithm each Node is currently in. Node 1 crashes immediately. Node 2 detects this and decides on its value. Node 3 receives the *decided* message from Node 2 before it detects that Node 1 has crashed. It is therefore still in round 1 and it does not decide on a value. Node 3 then detects that Node 1 has crashed and enters round 2. The algorithm should have detected that Node 2 has already decided and Node 3 should therefore enter round 3, where it should decide on the same value. However due to the bug it will only change round once after a node crash and it will therefor never leave Round 2.

Simulation	**Scheduler**	**Nodes**	**Faulty Nodes**
1	Prefix	$\{1, 2, 3\}$	$\emptyset$
2	Prefix	$\{1, 2, 3\}$	$\{1\}$
3	Prefix	$\{1, 2, 3, 4, 5, 6, 7\}$	$\{2\}$
4	Random	$\{1, 2, 3\}$	$\emptyset$
5	Random	$\{1, 2, 3\}$	$\{1\}$
6	Random	$\{1, 2, 3, 4, 5, 6, 7\}$	$\{2\}$

Table 2: Testing of the Hierarchical Consensus algorithm. The algorithm was modified to include a bug. A *Propose* request was made by each node. When using a random scheduler the seed was set to 0. Unless states otherwise, default values for parameters was used. The full configuration of the tests can be found in Appendix D.2.

```
func (gc *GrpcConsensus) Crash(id int, _ bool) {
    if gc.stopped {
        return
    }
    gc.detectedRanks[int32(id)] = true
    // Violates C1: Termination
    // The algorithm does not advance the round again if it enters a round where the node has already crashed
    // The correct implementation would be:
    // for gc.delivered[gc.round] || gc.detectedRanks[gc.round] {
    if gc.delivered[gc.round] || gc.detectedRanks[gc.round] {
        gc.round++
        gc.decide()
    }
}
```

Listing 6.1: The bug introduced in the gRPC version of the Hierarchical Consensus algorithm. The algorithm does not increase the round if a node crashe causing the algorithm to advance to a round where the node for this round has already decided on a value or crashed.

while the `Random` scheduler do find it. We also see that the `Random` scheduler uses approximately the same amount of time to schedule all scenarios with the same amount of nodes (simulation 4 and simulation 5), while the `Prefix` scheduler uses significantly less time in simulation 1 compared with simulation 2.

Simulation	Expected Error	Found Error
1	No	No
2	Yes	Yes
3	Yes	No
4	No	No
5	Yes	Yes
6	Yes	Yes

Table 3: Results from Testing of the Hierarchical Consensus algorithm. The bug can only appear when a node crashes. We therefore do not expect to find an error in simulation 1 and 4.

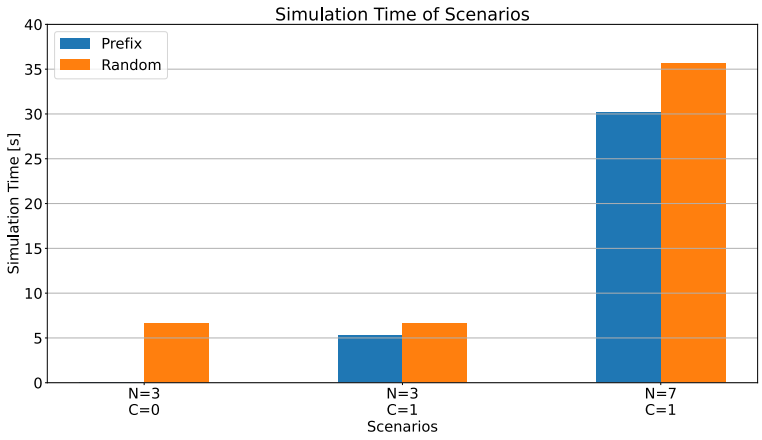


Figure 9: Simulation time of each scenario. N specifies the number of nodes in the scenario, while C specifies the number of faulty nodes.

7 Discussion

In this section we will summarize some experiences and observations from working with Go-MC. We will first discuss the results of the tests. Then, we will discuss some of our experiences from working with Go-MC and some thoughts about the design of Go-MC. Finally we will propose improvements that could be applied to Go-MC.

7.1 Tests

The first test shows that Go-MC is able to perform simulations of simple scenarios in a relatively short time, but that the simulation times varies depending on which Event Managers are used. This is somewhat expected, since the different Event Managers hooks into different frameworks, which will have different levels of overhead. It is however a significant difference, and it might indicate that it is infeasible to perform simulation with more complex frameworks that introduces large amount of overhead.

There are two interesting observations from the second test. Firstly the Prefix scheduler was unable to identify the fault in the complex scenario with seven nodes and one faulty node. We believe this is because the prefix scheduler only performs minor changes between runs. Since the size of the state space is much larger than the number of simulated runs it did not explore the particular runs where the bug occurred and was therefore unable to find the bug. The Random Scheduler schedules runs that are much more varied, and it is therefore able to find the bug. One solution to this would naturally be to increase the maximum number of runs to simulate, but doing this will also increase the time used and the memory required to maintain the entire discovered state space.

Another observation is that the simulation time of the Simulations 4 and 5 is approximately equal, despite the simulation time of Simulations 1 and 2 being very different. Simulation 1 and 2 uses the Prefix Scheduler, which keep track of the discovered state space and stops when it has explored all possible runs. Simulation 4 and 5 uses the Random Scheduler which is stateless and will continue to schedule runs until the maximum number of runs have been scheduled, even if it has already explored all possible runs. In simple scenarios, such as simulation 1 and 4, where there are only three nodes and no crashes, the state space is smaller than the maximum number of runs and the Prefix Scheduler can therefore stop significantly earlier than the random scheduler.

In conclusion, Go-MC is capable of finding bugs in distributed systems, but that the choice of scheduling algorithm has a significant impact on the results. The Prefix scheduler can guarantee that all runs have been simulated and will stop once it has simulated all runs. It is therefore good when the distributed system is simple and the state space is relatively small. The Random scheduler can sample more evenly from the state space and it is therefore useful when the state space is large.

7.2 Experiences

Go-MC proved useful when implementing the distributed algorithms, particularly in simple scenarios. We used Go-MC extensively when implementing the example algorithms and it provided good feedback that was useful during the implementation process. It was capable of providing results quickly when the maximum number of runs was low, e.g. 1000, and was still able to identify obvious errors in the algorithms. More complex scenarios could be run later to increase the chance of finding edge cases and complex bugs.

In our experience, implementing the algorithms for testing in Go-MC is relatively straightforward, but does impose some limitations. Firstly, when implementing algorithms it is important to keep in mind that events must be atomic. Thus, it is not possible to pause the handling of some event to wait for some condition to apply before performing some action. Instead the handling of the event must end and the action must be performed in the event where the condition is satisfied. This is a minor limitation to how algorithms can be implemented, but it also enforces the event driven model that is commonly used to describe algorithms. This makes it easier to implement and understand the algorithms, since the implementations are similar to the specification of the algorithm.

Another limitation that is imposed when using Go-MC is that the events must be deterministic. This has not been a problem for the algorithms we have implemented and tested, but it would limit the use of Go-MC on algorithms that rely on randomness. Go-MC does not currently provide any methods of simulating randomness, but it might be desirable to implement such mechanisms in the future to increase the applicability of Go-MC to randomized algorithms.

The choice to intercept events at a high level by using Event Manager has proven to have both advantages and disadvantages. The design makes it relatively easy to define events and it is flexible, which allows Go-MC to take advantage of the different frameworks that are used. It is also possible to define new Events for a specific use case, instead of relying on the existing events. On the other hand, the requirement for new Event Managers makes it harder to utilize new frameworks, since the user would first need to implement a new Event Manager for the framework. However, this is only a problem the first time a new framework is utilized. After that, the same Event Managers can be used, which reduces the impact of the problem.

7.3 Future Work

The tests shows that the state explosion problem is still a large factor in the efficiency of Go-MC. For Go-MC to be viable when testing larger number of nodes and more advanced algorithms it is necessary to develop it's ability to handle this problem. We believe that the approach to solving the problems should be twofold: It should work to reduce the memory usage of each run of the simulation and it should focus on reducing the size of state space by applying state of the art state space reduction techniques.

The effort to reduce the memory usage of each run can take multiple approaches. It can improve the efficiency in which state is stored, for example by only storing changes to the state instead of storing the state of each time slot, or by implementing a new State Manager that stores states more efficiently than a tree representation. Another approach could be to perform the checking of the states live, instead of performing the checking after the state space has been discovered. Using this solution we could check each run individually and if all properties hold we could discard the run. Thus we would not have to store the entire state space, which would significantly reduce the memory use.

The development of Go-MC so far has not focused on implementing advanced schedulers and the schedulers that are provided does therefore not utilize state space reduction techniques. The development of new schedulers can therefore significantly increase the efficiency of the Go-MC by reducing the size of the state space and by providing approaches that increase the likelihood of finding errors. New schedulers can utilize different approaches and can easily be inserted into the simulation. In particular, a scheduler that implements some form of state space reduction technique would significantly reduce the number of redundant runs that are simulated, and therefore increase the efficiency of Go-MC

Finally, when a node crashes all current and future events targeting that node are still scheduled as normal, but they are expected to have no impact on the target node. This causes the simulation to interleave events that should have no impact on the simulation, which increases the number of redundant runs that are simulated. To prevent this we could ignore all events that target a crashed node. However, some frameworks might provide errors when messages are sent to a crashed node, and removing the event might therefore have unintended consequences. It is therefore necessary to investigate the consequences of such a change further.

8 Conclusion

We have presented Go-MC, an implementation level model checker that uses a modular design to enable us to test distributed systems in a deterministic manner, similar to how sequential software is tested. We have presented two primary design choices used in Go-MC: The modular design and the use of Event Managers. The modular approach used by Go-MC makes it easy to use different implementations of modules when running simulations. This makes it easy to implement and compare state space reduction techniques for Go-MC. It also makes it easy to represent different abstractions, allowing the user to select the abstractions that fit best for the algorithm that they are testing. The use of Event Managers means that events are collected in an efficient manner. Event Managers are flexible and new implementations can be created for different frameworks. Event Mangers can also be used to mock the behavior of modules, enabling an modular approach to testing. Thus, reducing the simulation time and allowing modules to be tested separately.

References

[1] Bowen Alpern and Fred B. Schneider. "Defining liveness". en. In: *Information Processing Letters* 21.4 (Oct. 1985), pp. 181–185. ISSN: 00200190. DOI: 10.1016/0020-0190(85)90056-0. URL: https://linkinghub.elsevier.com/retrieve/pii/0020019085900560 (visited on 05/09/2023).

[2] Vaastav Anand. "Dara The Explorer: Coverage Based Exploration for Model Checking of Distributed Systems in Go". en. PhD thesis. Aug. 2020. URL: https://vaastavanand.com/assets/pdf/msc_thesis.pdf.

[3] M. Ben-Ari. *Principles of the Spin model checker*. en. OCLC: ocn170042062. London: Springer, 2008. ISBN: 978-1-84628-770-1.

[4] R. Boichat and R. Guerraoui. "Reliable broadcast in the crash-recovery model". In: *Proceedings 19th IEEE Symposium on Reliable Distributed Systems SRDS-2000*. Oct. 2000, pp. 32–41. DOI: 10.1109/RELDI.2000.885390.

[5] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to reliable and secure distributed programming*. Second edition. Berlin ; New York: Springer, 2011. ISBN: 978-3-642-15259-7.

[6] Tushar Deepak Chandra and Sam Toueg. "Unreliable failure detectors for reliable distributed systems". In: *Journal of the ACM* 43.2 (Mar. 1996), pp. 225–267. ISSN: 0004-5411. DOI: 10.1145/226643.226647. URL: https://dl.acm.org/doi/10.1145/226643.226647 (visited on 05/09/2023).

[7] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. "Consensus in the presence of partial synchrony". en. In: *Journal of the ACM* 35.2 (Apr. 1988), pp. 288–323. ISSN: 0004-5411, 1557-735X. DOI: 10.1145/42282.42283. URL: https://dl.acm.org/doi/10.1145/42282.42283 (visited on 05/09/2023).

[8] Cormac Flanagan and Patrice Godefroid. "Dynamic partial-order reduction for model checking software". In: *ACM SIGPLAN Notices* 40.1 (Jan. 2005), pp. 110–121. ISSN: 0362-1340. DOI: 10.1145/1047659.1040315. URL: https://doi.org/10.1145/1047659.1040315 (visited on 01/25/2023).

[9] *gRPC*. en. URL: https://grpc.io/ (visited on 05/12/2023).

[10] Huayang Guo et al. "Practical software model checking via dynamic interface reduction". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. New York, NY, USA: Association for Computing Machinery, 2011, pp. 265–278. ISBN: 978-1-4503-0977-6. DOI: 10.1145/2043556.2043582. URL: https://doi.org/10.1145/2043556.2043582 (visited on 01/11/2023).

[11] Charles Killian et al. "Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code". en. In: (Jan. 2007). URL: https://www.usenix.org/legacy/event/nsdi07/tech/killian/killian.pdf (visited on 01/11/2023).

[12] Charles Edwin Killian. "Systems and language support for building correct, high performance distributed systems". en. PhD thesis. UC San Diego, 2008. URL: https://escholarship.org/uc/item/4gj3z4tw (visited on 01/11/2023).

[13] L. Lamport. "Proving the Correctness of Multiprocess Programs". In: *IEEE Transactions on Software Engineering* SE-3.2 (Mar. 1977). Conference Name: IEEE Transactions on Software Engineering, pp. 125–143. ISSN: 1939-3520. DOI: 10.1109/TSE.1977.229904.

[14] Leslie Lamport. "Paxos Made Simple". en-US. In: *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (Dec. 2001), pp. 51–58. URL: https://www.microsoft.com/en-us/research/publication/paxos-made-simple/ (visited on 05/09/2023).

[15] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". en. In: 21.7 (1978).

[16] Leslie Lamport and P. M. Melliar-Smith. "Synchronizing clocks in the presence of faults". en. In: *Journal of the ACM* 32.1 (Jan. 1985), pp. 52–78. ISSN: 0004-5411, 1557-735X. DOI: 10.1145/2455.2457. URL: https://dl.acm.org/doi/10.1145/2455.2457 (visited on 05/10/2023).

[17] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine Generals Problem". en-US. In: *ACM Transactions on Programming Languages and Systems* (July 1982), pp. 382–401. URL: https://www.microsoft.com/en-us/research/publication/byzantine-generals-problem/ (visited on 05/09/2023).

[18] Tanakorn Leesatapornwongsa et al. "SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems". en. In: (2014). URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/leesatapornwongsa.

[19] Jeffrey F. Lukman et al. "FlyMC: Highly Scalable Testing of Complex Interleavings in Distributed Systems". In: *Proceedings of the Fourteenth EuroSys Conference 2019*. EuroSys '19. New York, NY, USA: Association for Computing Machinery, Mar. 2019, pp. 1–16. ISBN: 978-1-4503-6281-8. DOI: 10.1145/3302424.3303986. URL: https://doi.org/10.1145/3302424.3303986 (visited on 01/11/2023).

[20] Ellis Michael et al. "Teaching Rigorous Distributed Systems With Efficient Model Checking". In: *Proceedings of the Fourteenth EuroSys Conference 2019*. EuroSys '19. New York, NY, USA: Association for Computing Machinery, Mar. 2019, pp. 1–15. ISBN: 978-1-4503-6281-8. DOI: 10.1145/3302424.3303947. URL: https://doi.org/10.1145/3302424.3303947 (visited on 01/12/2023).

[21] A. Mostefaoui, E. Mourgaya, and M. Raynal. "Asynchronous implementation of failure detectors". In: *2003 International Conference on Dependable Systems and Networks, 2003. Proceedings.* June 2003, pp. 351–360. DOI: 10.1109/DSN.2003.1209946.

[22] Madanlal Musuvathi et al. “Finding and reproducing Heisenbugs in concurrent programs”. In: *Proceedings of the 8th USENIX conference on Operating systems design and implementation*. OSDI'08. USA: USENIX Association, 2008, pp. 267–280. (Visited on 04/11/2023).

[23] Burcu Kulahcioglu Ozkan et al. “Randomized testing of distributed systems with probabilistic guarantees”. In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018), 160:1–160:28. DOI: 10.1145/3276530. URL: https://doi.org/10.1145/3276530 (visited on 01/17/2023).

[24] James Lyle Peterson. *Petri Net Theory and the Modeling of Systems*. USA: Prentice Hall PTR, Jan. 1981. ISBN: 978-0-13-661983-3.

[25] Richard D. Schlichting and Fred B. Schneider. “Fail-stop processors: an approach to designing fault-tolerant computing systems”. en. In: *ACM Transactions on Computer Systems* 1.3 (Aug. 1983), pp. 222–238. ISSN: 0734-2071, 1557-7333. DOI: 10.1145/357369.357371. URL: https://dl.acm.org/doi/10.1145/357369.357371 (visited on 05/09/2023).

[26] Fred B. Schneider, David Gries, and Richard D. Schlichting. “Fault-tolerant broadcasts”. en. In: *Science of Computer Programming* 4.1 (Apr. 1984), pp. 1–15. ISSN: 0167-6423. DOI: 10.1016/0167-6423(84)90009-1. URL: https://www.sciencedirect.com/science/article/pii/0167642384900091 (visited on 05/10/2023).

[27] Maarten van Steen and Andrew S. Tanenbaum. “A brief introduction to distributed systems”. en. In: *Computing* 98.10 (Oct. 2016), pp. 967–1009. ISSN: 0010-485X, 1436-5057. DOI: 10.1007/s00607-016-0508-7. URL: http://link.springer.com/10.1007/s00607-016-0508-7 (visited on 03/31/2023).

[28] Junfeng Yang et al. “MODIST: Transparent Model Checking of Unmodified Distributed Systems”. en. In: (2009). URL: https://www.cs.columbia.edu/~junfeng/papers/modist-nsdi09.pdf.

[29] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. “Model Checking TLA+ Specifications”. en. In: *Correct Hardware Design and Verification Methods*. Ed. by Laurence Pierre and Thomas Kropf. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1999, pp. 54–66. ISBN: 978-3-540-48153-9. DOI: 10.1007/3-540-48153-2_6.

[30] Xinhao Yuan and Junfeng Yang. “Effective Concurrency Testing for Distributed Systems”. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, Mar. 2020, pp. 1141–1156. ISBN: 978-1-4503-7102-5. DOI: 10.1145/3373376.3378484. URL: https://doi.org/10.1145/3373376.3378484 (visited on 01/11/2023).

A Distributed Algorithms

A.1 Hierarchical Consensus

5.1 Regular Consensus 209

Algorithm 5.2: Hierarchical Consensus

Implements:
Consensus, **instance** c.

Uses:
BestEffortBroadcast, **instance** *beb*;
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event $\langle$ c, *Init* $\rangle$ **do**
detectedranks := $\emptyset$;
round := 1;
proposal := $\perp$; *proposer* := 0;
delivered := $[\text{FALSE}]^N$;
broadcast := FALSE;

upon event $\langle$ $\mathcal{P}$, *Crash* | p $\rangle$ **do**
detectedranks := *detectedranks* $\cup$ $\{rank(p)\}$;

upon event $\langle$ c, *Propose* | v $\rangle$ **such that** *proposal* $= \perp$ **do**
proposal := v;

upon *round* $=$ *rank*(*self*) $\wedge$ *proposal* $\neq \perp \wedge$ *broadcast* $=$ FALSE **do**
broadcast := TRUE;
trigger $\langle$ *beb*, *Broadcast* | [DECIDED, *proposal*] $\rangle$;
trigger $\langle$ c, *Decide* | *proposal* $\rangle$;

upon *round* $\in$ *detectedranks* $\vee$ *delivered*[*round*] $=$ TRUE **do**
round := *round* + 1;

upon event $\langle$ *beb*, *Deliver* | p,[DECIDED, v] $\rangle$ **do**
r := *rank*(p);
if $r <$ *rank*(*self*) $\wedge$ $r >$ *proposer* **then**
proposal := v;
proposer := r;
delivered[r] := TRUE;

Figure 10: The Hierarchical Consensus algorithm as presented by Cachin, Guerraoui, and Rodrigues [5]. (Algorithm 5.2)

A.1.1 Sender

```
1  package main
2
3  type Value[T any] struct {
4      Val T
```

```
}

type HierarchicalConsensus[T any] struct {
    detectedRanks map[int]bool
    round         int
    proposal      Value[T]
    proposer      int
    proposed      bool

    DecidedSignal chan Value[T]

    // Test values used to verify algorithm
    DecidedVal  []Value[T]
    ProposedVal Value[T]

    delivered map[int]bool
    broadcast bool

    crashed bool

    id    int
    nodes []int
    send  func(int, string, ...any)
}

func NewHierarchicalConsensus[T any](id int, nodes []int, send
↪  func(int, string, ...any)) *HierarchicalConsensus[T] {
    return &HierarchicalConsensus[T]{
        detectedRanks: make(map[int]bool),
        round:         1,
        proposal:      Value[T]{},
        proposer:      0,
        proposed:      false,
        delivered:     make(map[int]bool),
        broadcast:     false,

        DecidedSignal: make(chan Value[T], 1),
        DecidedVal:    make([]Value[T], 0),

        id:    id,
        nodes: nodes,
        send:  send,
    }
}

func (hc *HierarchicalConsensus[T]) Crash(id int, _ bool) {
```

```
    if hc.crashed {
        return
    }
    hc.detectedRanks[id] = true
    for hc.delivered[hc.round] || hc.detectedRanks[hc.round] {
        hc.round++
        hc.decide()
    }
}

func (hc *HierarchicalConsensus[T]) Propose(val Value[T]) {
    if hc.crashed {
        return
    }
    hc.ProposedVal = val
    if !hc.proposed {
        hc.proposed = true
        hc.proposal = val
    }
    hc.decide()
}

func (hc *HierarchicalConsensus[T]) Decided(from int, val
↪   Value[T]) {
    if hc.crashed {
        return
    }
    if from < hc.id && from > hc.proposer {
        hc.proposed = true
        hc.proposal = val
        hc.proposer = from
        hc.decide()
    }
    hc.delivered[from] = true
    for hc.delivered[hc.round] || hc.detectedRanks[hc.round] {
        hc.round++
        hc.decide()
    }
}

func (hc *HierarchicalConsensus[T]) decide() {
    if hc.id != hc.round {
        return
    }

    if hc.broadcast {
```

```
95          return
96      }
97      if !hc.proposed {
98          return
99      }
100
101     hc.broadcast = true
102     for _, target := range hc.nodes {
103         if target > hc.id {
104             hc.send(int(target), "Decided", hc.id, hc.proposal)
105         }
106     }
107     // Decide on value
108     hc.DecidedSignal <- hc.proposal
109     hc.DecidedVal = append(hc.DecidedVal, hc.proposal)
110 }
```

Listing A.1: Implementation of the Hierarchical Consensus Algorithm using the Sender Event Manager. The `Crash` method is called when a node crashes. The `Propose` method is called when a node proposes a value. The `Decided` method is a message handler that handles a decided message.

```
1   package main
2
3   import (
4       "bytes"
5       "encoding/json"
6       "os"
7       "testing"
8
9       "golang.org/x/exp/slices"
10
11      "gomc"
12      "gomc/checking"
13      "gomc/eventManager"
14  )
15
16  type state struct {
17      proposed Value[int]
18      decided  []Value[int]
19  }
20
21  var predicates = []checking.Predicate[state]{
22      checking.Eventually(
23          // C1: Termination
```

```
24          func(s checking.State[state]) bool {
25              return checking.ForAllNodes(func(s state) bool {
26                  return len(s.decided) > 0
27              }, s, true)
28          },
29      ),
30      func(s checking.State[state]) bool {
31          // C2: Validity
32          proposed := make(map[Value[int]]bool)
33          for _, node := range s.LocalStates {
34              proposed[node.proposed] = true
35          }
36          return checking.ForAllNodes(func(s state) bool {
37              if len(s.decided) < 1 {
38                  // The process has not decided a value yet
39                  return true
40              }
41              return proposed[s.decided[0]]
42          }, s, false)
43      },
44      func(s checking.State[state]) bool {
45          // C3: Integrity
46          return checking.ForAllNodes(func(s state) bool { return
    ↪ len(s.decided) < 2 }, s, false)
47      },
48      func(s checking.State[state]) bool {
49          // C4: Agreement
50          decided := make(map[Value[int]]bool)
51          checking.ForAllNodes(func(s state) bool {
52              for _, val := range s.decided {
53                  decided[val] = true
54              }
55              return true
56          }, s, true)
57          return len(decided) <= 1
58      },
59  }
60
61  func TestConsensus(t *testing.T) {
62      sim := gomc.PrepareSimulation(
63          gomc.WithTreeStateManager(
64              func(node *HierarchicalConsensus[int]) state {
65                  return state{
66                      proposed: node.ProposedVal,
67                      decided:  slices.Clone(node.DecidedVal),
68                  }
```

```
69              },
70              func(a, b state) bool {
71                  if a.proposed != b.proposed {
72                      return false
73                  }
74                  return slices.Equal(a.decided, b.decided)
75              },
76          ),
77          gomc.PrefixScheduler(),
78      )
79
80      nodeIds := []int{1, 2, 3}
81      resp := sim.Run(
82          gomc.InitSingleNode(nodeIds,
83              func(id int, sp eventManager.SimulationParameters)
    ↪ *HierarchicalConsensus[int] {
84                  send := eventManager.NewSender(sp)
85                  node := NewHierarchicalConsensus[int](
86                      id,
87                      nodeIds,
88                      send.SendFunc(id),
89                  )
90                  sp.CrashSubscribe(id, node.Crash)
91                  return node
92              },
93          ),
94          gomc.WithRequests(
95              gomc.NewRequest(1, "Propose", Value[int]{1}),
96              gomc.NewRequest(2, "Propose", Value[int]{2}),
97              gomc.NewRequest(3, "Propose", Value[int]{3}),
98          ),
99          gomc.WithPredicateChecker(predicates...),
100         gomc.WithPerfectFailureManager(
101             func(t *HierarchicalConsensus[int]) { t.crashed =
    ↪ true },
102             1,
103         ),
104         gomc.Export(os.Stdout),
105     )
106     if ok, out := resp.Response(); !ok {
107         t.Errorf("Expected no errors while checking. Got: %v",
    ↪ out)
108
109         var buffer bytes.Buffer
110         json.NewEncoder(&buffer).Encode(resp.Export())
111         os.WriteFile("FailedRun.txt", buffer.Bytes(), 0755)
```

```
112     }
113 }
```

Listing A.2: Configuration of the simulation of the Hierarchical Consensus using the Sender Event Manager. The simulation is configured with three nodes that each propose a value. It uses a prefix scheduler to schedule the events. Node 1 is configured to crash during the simulation. Lines 20-58 define the properties that must hold for the algorithm.

A.1.2 gRPC

```
1  package main
2
3  import (
4      "context"
5      pb "gomc/examples/grpcConsensus/proto"
6      "net"
7
8      "google.golang.org/grpc"
9  )
10
11 type state struct {
12     proposed string
13     decided  []string
14 }
15
16 type Client struct {
17     pb.ConsensusClient
18     id   int32
19     conn *grpc.ClientConn
20 }
21
22 type GrpcConsensus struct {
23     pb.UnimplementedConsensusServer
24
25     detectedRanks map[int32]bool
26     proposal      *pb.Value
27     proposer      int32
28
29     round     int32
30     delivered map[int32]bool
31     broadcast bool
32
33     id    int32
34     nodes []*Client
```

```
35
36      stopped bool
37
38      DecidedVal  []string
39      ProposedVal string
40
41      waitForSend func(num int)
42
43      srv *grpc.Server
44  }
45
46  func NewGrpcConsensus(id int32, lis net.Listener, waitForSend
    ↪  func(int), srvOpts ...grpc.ServerOption) *GrpcConsensus {
47      srv := grpc.NewServer(srvOpts...)
48      gc := &GrpcConsensus{
49          detectedRanks: make(map[int32]bool),
50          proposer:      0,
51          round:         1,
52          delivered:     make(map[int32]bool),
53          broadcast:     false,
54          id:            id,
55          nodes:         make([]*Client, 0),
56
57          DecidedVal:  make([]string, 0),
58          ProposedVal: "",
59
60          waitForSend: waitForSend,
61
62          srv: srv,
63      }
64      pb.RegisterConsensusServer(srv, gc)
65      go func() {
66          srv.Serve(lis)
67      }()
68      return gc
69  }
70
71  func (gc *GrpcConsensus) DialServers(addrMap map[int32]string,
    ↪  dialOpts ...grpc.DialOption) {
72      for id, addr := range addrMap {
73          conn, err := grpc.Dial(addr, dialOpts...)
74          if err != nil {
75              panic(err)
76          }
77          gc.nodes = append(gc.nodes, &Client{
78              id:              id,
```

```
79              ConsensusClient: pb.NewConsensusClient(conn),
80              conn:            conn,
81          })
82      }
83  }
84
85  func (gc *GrpcConsensus) Crash(id int, _ bool) {
86      if gc.stopped {
87          return
88      }
89      gc.detectedRanks[int32(id)] = true
90      // Violates C1: Termination
91      // The algorithm does not advance the round again if it
    ↪   enters a round where the node has already crashed
92      // The correct implementation would be:
93      // for gc.delivered[gc.round] || gc.detectedRanks[gc.round]
    ↪   {
94      if gc.delivered[gc.round] || gc.detectedRanks[gc.round] {
95          gc.round++
96          gc.decide()
97      }
98  }
99
100 func (hc *GrpcConsensus) Propose(val string) {
101     hc.ProposedVal = val
102     protoVal := &pb.Value{Val: val}
103     if hc.proposal == nil {
104         hc.proposal = protoVal
105     }
106     hc.decide()
107 }
108
109 func (gc *GrpcConsensus) Decided(ctx context.Context, in
    ↪   *pb.DecideRequest) (*pb.DecideResponse, error) {
110     if in.GetFrom() < gc.id && in.GetFrom() > gc.proposer {
111         gc.proposal = in.GetVal()
112         gc.proposer = in.GetFrom()
113         gc.decide()
114     }
115     gc.delivered[in.GetFrom()] = true
116     for gc.delivered[gc.round] || gc.detectedRanks[gc.round] {
117         gc.round++
118         gc.decide()
119     }
120     return &pb.DecideResponse{}, nil
121 }
```

```
122
123 func (gc *GrpcConsensus) decide() {
124     if gc.id != gc.round {
125         return
126     }
127
128     if gc.broadcast {
129         return
130     }
131     if gc.proposal == nil {
132         return
133     }
134
135     gc.broadcast = true
136     gc.DecidedVal = append(gc.DecidedVal, gc.proposal.GetVal())
137     msg := &pb.DecideRequest{
138         Val:  gc.proposal,
139         From: gc.id,
140     }
141     num := 0
142     for _, node := range gc.nodes {
143         if node.id > gc.id {
144             num++
145             go node.Decided(context.Background(), msg)
146         }
147     }
148     gc.waitForSend(num) // Wait until all messages has been
    ↪  sent, but do not wait until an answer is received
149 }
150
151 func (gc *GrpcConsensus) Stop() {
152     gc.srv.Stop()
153     for _, node := range gc.nodes {
154         node.conn.Close()
155     }
156     gc.stopped = true
157 }
```

Listing A.3: Implementation of the Hierarchical Consensus Algorithm using gRPC. The `Crash` method is called when a node crashes. The `Propose` method is called when a node proposes a value. The `Decided` method is a message handler that handles a decided message.

A.2 Eager Reliable Broadcast

80 3 Reliable Broadcast

Algorithm 3.3: Eager Reliable Broadcast

Implements:
ReliableBroadcast, **instance** *rb*.

Uses:
BestEffortBroadcast, **instance** *beb*.

upon event ⟨ *rb*, *Init* ⟩ **do**
delivered := ∅;

upon event ⟨ *rb*, *Broadcast* | m ⟩ **do**
trigger ⟨ *beb*, *Broadcast* | [DATA, *self*, m] ⟩;

upon event ⟨ *beb*, *Deliver* | p, [DATA, s, m] ⟩ **do**
if $m \notin$ *delivered* **then**
delivered := *delivered* ∪ {m};
trigger ⟨ *rb*, *Deliver* | s, m ⟩;
trigger ⟨ *beb*, *Broadcast* | [DATA, s, m] ⟩;

Figure 11: The Eager Reliable Broadcast algorithm as presented by Cachin, Guerraoui, and Rodrigues [5]. (Algorithm 3.3)

```
1  package main
2
3  import (
4      "fmt"
5  )
6
7  type message struct {
8      From    int
9      Index   int
10     Payload string
11 }
12
13 func (m message) String() string {
14     return fmt.Sprintf("{F:%v, i:%v}", m.From, m.Index)
15 }
16
17 type Rrb struct {
18     id    int
19     nodes []int
20
```

```
21      delivered map[message]bool
22      sent      map[message]bool
23      send      func(to int, msgType string, msg ...any)
24
25      crashed bool
26
27      deliveredSlice []message
28  }
29
30  func NewRrb(id int, nodes []int, send func(int, string,
    ↪  ...any)) *Rrb {
31      return &Rrb{
32          id:    id,
33          nodes: nodes,
34
35          delivered: make(map[message]bool),
36          sent:      make(map[message]bool),
37          send:      send,
38      }
39  }
40
41  func (rrb *Rrb) Broadcast(msg string) {
42      if rrb.crashed {
43          return
44      }
45      message := message{
46          From:    rrb.id,
47          Index:   len(rrb.sent),
48          Payload: msg,
49      }
50
51      for _, target := range rrb.nodes {
52          rrb.send(target, "Deliver", message)
53      }
54      rrb.sent[message] = true
55  }
56
57  func (rrb *Rrb) Deliver(message message) {
58      if rrb.crashed {
59          return
60      }
61
62      // violation of RB2:No Duplication
63      rrb.deliveredSlice = append(rrb.deliveredSlice, message)
64      if !rrb.delivered[message] {
65          rrb.delivered[message] = true
```

```
66          for _, target := range rrb.nodes {
67              rrb.send(target, "Deliver", message)
68          }
69      }
70  }
```

Listing A.4: Implementation of the Eager Reliable Broadcast Algorithm. The `Broadcast` method is called when a node should broadcast a message. The `Deliver` method is a message handler that handles a message.

A.3 Majority Voting Regular Register

4.2 $(1, N)$ Regular Register 147

Algorithm 4.2: Majority Voting Regular Register

Implements:
$(1, N)$-RegularRegister, **instance** *onrr*.
Uses:
BestEffortBroadcast, **instance** *beb*;
PerfectPointToPointLinks, **instance** *pl*.

upon event ⟨ *onrr*, *Init* ⟩ **do**
$(ts, val) := (0, \perp)$;
$wts := 0$;
acks := 0;
rid := 0;
readlist := $[\perp]^N$;

upon event ⟨ *onrr*, *Write* | v ⟩ **do**
$wts := wts + 1$;
acks := 0;
trigger ⟨ *beb*, *Broadcast* | [WRITE, wts, v] ⟩;

upon event ⟨ *beb*, *Deliver* | p, [WRITE, ts', v'] ⟩ **do**
if $ts' > ts$ **then**
$(ts, val) := (ts', v')$;
trigger ⟨ *pl*, *Send* | p, [ACK, ts'] ⟩;

upon event ⟨ *pl*, *Deliver* | q, [ACK, ts'] ⟩ **such that** $ts' = wts$ **do**
acks := *acks* + 1;
if *acks* $> N/2$ **then**
acks := 0;
trigger ⟨ *onrr*, *WriteReturn* ⟩;

upon event ⟨ *onrr*, *Read* ⟩ **do**
rid := *rid* + 1;
readlist := $[\perp]^N$;
trigger ⟨ *beb*, *Broadcast* | [READ, *rid*] ⟩;

upon event ⟨ *beb*, *Deliver* | p, [READ, r] ⟩ **do**
trigger ⟨ *pl*, *Send* | p, [VALUE, r, ts, val] ⟩;

upon event ⟨ *pl*, *Deliver* | q, [VALUE, r, ts', v'] ⟩ **such that** $r =$ *rid* **do**
readlist[q] := (ts', v');
if $\#$(*readlist*) $> N/2$ **then**
v := highestval(*readlist*);
readlist := $[\perp]^N$;
trigger ⟨ *onrr*, *ReadReturn* | v ⟩;

Figure 12: The Majority Voting Regular Register algorithm as presented by Cachin, Guerraoui, and Rodrigues [5]. (Algorithm 4.2)

```
1 package main
2
```

```
3  type Value struct {
4      Ts  int
5      Val int
6  }
7
8  type BroadcastWriteMsg struct {
9      Val Value
10 }
11
12 type AckMsg struct {
13     Ts int
14 }
15
16 type BroadcastReadMsg struct {
17     Rid int
18 }
19
20 type ReadValueMsg struct {
21     Rid int
22     Val Value
23 }
24
25 type onrr struct {
26     val      Value         // Current value stored in the
   ↪  register
27     wts      int           // Write timestamp
28     acks     int           // Number of acks received for the
   ↪  current value
29     rid      int           // A read request identifier
30     readList map[int]Value // A slice of all values
31
32     WriteIndicator chan bool
33     ReadIndicator  chan int
34
35     // Used for testing.
36     ongoingRead   bool
37     ongoingWrite  bool
38     possibleReads []int
39
40     // Id of the current node
41     id int
42     // Used to keep track of all nodes
43     nodes []int
44     // Used to send messages to other types.
45     send func(to int, msgType string, params ...any)
46 }
```

```go

func NewOnrr(id int, send func(to int, msgType string, params ...any), nodes []int) *onrr {
    return &onrr{
        val:      Value{Ts: 0, Val: 0},
        wts:      0,
        acks:     0,
        rid:      0,
        readList: make(map[int]Value),

        // Indicator channels
        WriteIndicator: make(chan bool, 1),
        ReadIndicator:  make(chan int, 1),

        ongoingRead:   false,
        ongoingWrite:  false,
        possibleReads: []int{0},

        id:    id,
        nodes: nodes,
        send:  send,
    }
}

func (onrr *onrr) Write(val int) {
    onrr.wts++
    onrr.acks = 0

    onrr.possibleReads = []int{onrr.val.Val, val}

    value := Value{
        Ts:  onrr.wts,
        Val: val,
    }

    onrr.ongoingWrite = true

    msg := BroadcastWriteMsg{Val: value}
    for _, target := range onrr.nodes {
        onrr.send(target, "BroadcastWrite", onrr.id, msg)
    }
}

func (onrr *onrr) BroadcastWrite(from int, msg BroadcastWriteMsg) {
    bwMsg := msg
```

```
91      if bwMsg.Val.Ts > onrr.val.Ts {
92          onrr.val = bwMsg.Val
93      }
94      ackMsg := AckMsg{
95          Ts: bwMsg.Val.Ts,
96      }
97      onrr.send(from, "AckWrite", ackMsg)
98  }
99
100 func (onrr *onrr) AckWrite(msg AckMsg) {
101     ackMsg := msg
102     if ackMsg.Ts != onrr.wts {
103         return
104     }
105     onrr.acks++
106     if onrr.acks > len(onrr.nodes)/2 {
107         if onrr.ongoingWrite {
108             onrr.possibleReads = onrr.possibleReads[1:]
109         }
110         onrr.ongoingWrite = false
111         onrr.acks = 0
112         onrr.WriteIndicator <- true
113     }
114 }
115
116 func (onrr *onrr) Read() {
117     onrr.ongoingRead = true
118
119     onrr.rid++
120     onrr.readList = make(map[int]Value)
121     msg := BroadcastReadMsg{
122         Rid: onrr.rid,
123     }
124     for _, target := range onrr.nodes {
125         onrr.send(target, "BroadcastRead", onrr.id, msg)
126     }
127 }
128
129 func (onrr *onrr) BroadcastRead(from int, msg BroadcastReadMsg)
    ↪ {
130     readMsg := msg
131     valMsg := ReadValueMsg{
132         Rid: readMsg.Rid,
133         Val: onrr.val,
134     }
135     onrr.send(from, "ReadValue", onrr.id, valMsg)
```

```
}

func (onrr *onrr) ReadValue(from int, msg ReadValueMsg) {
    valMsg := msg
    if valMsg.Rid != onrr.rid {
        return
    }
    onrr.readList[from] = valMsg.Val
    if len(onrr.readList) > len(onrr.nodes)/2 {
        val := getvalue(onrr.readList)
        onrr.readList = make(map[int]Value)

        onrr.ongoingRead = false
        onrr.ReadIndicator <- val.Val
    }
}

func getvalue(valueMap map[int]Value) Value {
    var highest Value
    for _, val := range valueMap {
        if val.Ts > highest.Ts {
            highest = val
        }
    }
    return highest
}
```

Listing A.5: Implementation of the Hierarchical Consensus Algorithm. The `W⌋rite` method is called when a node write a value and the `Read` method is called when a node reads the value . The `BroadcastWriteMsg`, `AckWrite`, `Broadcas⌋tRead` and `ReadValue` methods are message handlers handling their respective messages in the algorithm.

A.4 Paxos

```
package paxos

import (
    "gomc/examples/paxos/proto"
    "net"

    "google.golang.org/grpc"
)

type paxosClient struct {
```

```
    conn *grpc.ClientConn

    proto.ProposerClient
    proto.AcceptorClient
    proto.LearnerClient
}

func newPaxosClient(conn *grpc.ClientConn) *paxosClient {
    return &paxosClient{
        ProposerClient: proto.NewProposerClient(conn),
        AcceptorClient: proto.NewAcceptorClient(conn),
        LearnerClient:  proto.NewLearnerClient(conn),
    }
}

type paxos struct {
    *Proposer
    *Acceptor
    *Learner

    Proposal string
    Decided  string

    stopped bool

    Id      int64
    correct map[int64]bool
    leader  int64
}

func newPaxos(id int64, nodes map[int64]string, waitForSend
↪   func(int)) *paxos {
    nodeId := &proto.NodeId{Val: id}
    var leader int64
    correct := make(map[int64]bool)
    for nodeId := range nodes {
        if nodeId > leader {
            leader = nodeId
        }
        correct[nodeId] = true
    }
    l := NewLearner(nodeId, len(nodes))
    p := &paxos{
        Proposer: NewProposer(nodeId, waitForSend),
        Acceptor: NewAcceptor(nodeId, waitForSend),
        Learner:  l,
```

```
56
57          Id:      id,
58          leader:  leader,
59          correct: correct,
60      }
61
62      l.Subscribe(p.decided)
63      return p
64  }
65
66  func (p *paxos) NodeCrash(id int, _ bool) {
67      if p.stopped {
68          return
69      }
70      p.correct[int64(id)] = false
71      if int64(id) == p.leader {
72          p.newLeader()
73      }
74  }
75
76  func (p *paxos) newLeader() {
77      p.Proposer.IncrementCrnd()
78      p.leader = p.nextLeader()
79      if p.leader == p.Id && p.Proposal != "" {
80          p.performPrepare(p.Proposal)
81      }
82  }
83
84  func (p *paxos) nextLeader() int64 {
85      var leader int64
86      for id, ok := range p.correct {
87          if ok {
88              if id > leader {
89                  leader = id
90              }
91          }
92      }
93      return leader
94  }
95
96  func (p *paxos) Propose(val string) {
97      if p.stopped {
98          return
99      }
100     p.Proposal = val
101     if p.leader == p.Id {
```

```
102             p.performPrepare(p.Proposal)
103         }
104     }
105
106     func (p *paxos) decided(val string) {
107         p.Decided = val
108     }
109
110     type Server struct {
111         srv         *grpc.Server
112         connections []*grpc.ClientConn
113         addrMap     map[int64]string
114
115         *paxos
116     }
117
118     func NewServer(id int64, addrMap map[int64]string, waitForSend
    ↪   func(int), srvOpts ...grpc.ServerOption) (*Server, error) {
119         srv := grpc.NewServer(srvOpts...)
120         paxos := newPaxos(id, addrMap, waitForSend)
121         proto.RegisterProposerServer(srv, paxos)
122         proto.RegisterAcceptorServer(srv, paxos)
123         proto.RegisterLearnerServer(srv, paxos)
124         return &Server{
125             srv:     srv,
126             paxos:   paxos,
127             addrMap: addrMap,
128         }, nil
129     }
130
131     func (p *Server) StartServer(lis net.Listener) error {
132         return p.srv.Serve(lis)
133     }
134
135     func (p *Server) Stop() {
136         p.srv.Stop()
137         for _, c := range p.connections {
138             c.Close()
139         }
140         p.stopped = true
141     }
142
143     func (p *Server) DialNodes(dialOpts ...grpc.DialOption) error {
144         nodes := make(map[int64]*paxosClient)
145         for id, addr := range p.addrMap {
146             conn, err := grpc.Dial(addr, dialOpts...)
```

```
147             if err != nil {
148                 return err
149             }
150             nodes[id] = newPaxosClient(conn)
151             p.connections = append(p.connections, conn)
152         }
153         p.paxos.Proposer.nodes = nodes
154         p.paxos.Acceptor.nodes = nodes
155         return nil
156     }
```

Listing A.6: Implementation of the Paxos Algorithm. The type implements the basic paxos algorithm as presented by Lamport [14]. It performs phase 1 first when proposing a value. It is designed to only decide on one value.

```
1   package paxos
2
3   import (
4       "context"
5       "gomc/examples/paxos/proto"
6
7       "github.com/golang/protobuf/ptypes/empty"
8       "google.golang.org/protobuf/types/known/emptypb"
9   )
10
11  type Proposer struct {
12      proto.UnimplementedProposerServer
13
14      id *proto.NodeId
15
16      v *proto.Value
17
18      // Current round
19      crnd *proto.Round
20      // Constrained consensus value
21      cval *proto.Value
22
23      numPromise int
24      largestVal *proto.Value
25
26      phaseOne chan bool
27
28      nodes       map[int64]*paxosClient
29      waitForSend func(num int)
30  }
```

```
31
32  func NewProposer(id *proto.NodeId, waitForSend func(num int))
    ↪  *Proposer {
33      return &Proposer{
34          id: id,
35
36          crnd: &proto.Round{Val: id.GetVal()},
37
38          nodes:       make(map[int64]*paxosClient),
39          waitForSend: waitForSend,
40      }
41  }
42
43  func (p *Proposer) performPrepare(propsedVal string) {
44      // Use a zero value for round. This will always be smaller
    ↪  than any value returned by the acceptor.
45      // This ensures that the value is only chosen if no value
    ↪  is returned by an acceptor
46      p.largestVal = &proto.Value{
47          Val: propsedVal,
48      }
49
50      msg := &proto.PrepareRequest{
51          Crnd: p.crnd,
52          From: p.id,
53      }
54      for _, n := range p.nodes {
55          go n.Prepare(context.Background(), msg)
56      }
57      p.waitForSend(len(p.nodes))
58  }
59
60  func (p *Proposer) Promise(_ context.Context, in
    ↪  *proto.PromiseRequest) (*empty.Empty, error) {
61      if in.GetRnd().GetVal() != p.crnd.GetVal() {
62          return &emptypb.Empty{}, nil
63      }
64
65      p.numPromise++
66      if in.GetVal().GetRnd().GetVal() >
    ↪  p.largestVal.GetRnd().GetVal() {
67          p.largestVal = in.GetVal()
68      }
69
70      if p.numPromise <= len(p.nodes)/2 {
71          return &emptypb.Empty{}, nil
```

```
72      }
73
74      msg := &proto.AcceptRequest{
75          Val: &proto.Value{
76              Rnd: p.crnd,
77              Val: p.largestVal.GetVal(),
78          },
79          From: p.id,
80      }
81
82      for _, node := range p.nodes {
83          go node.Accept(context.Background(), msg)
84      }
85      p.waitForSend(len(p.nodes))
86
87      p.numPromise = 0
88      p.largestVal = nil
89
90      return &emptypb.Empty{}, nil
91  }
92
93  func (p *Proposer) IncrementCrnd() {
94      newRnd := p.crnd.GetVal() + int64(len(p.nodes))
95      p.crnd = &proto.Round{Val: newRnd}
96  }
```

Listing A.7: Implementation of the Proposer in the Paxos Algorithm.

```
1   package paxos
2
3   import (
4       "context"
5       "gomc/examples/paxos/proto"
6
7       "github.com/golang/protobuf/ptypes/empty"
8       "google.golang.org/protobuf/types/known/emptypb"
9   )
10
11  type Acceptor struct {
12      proto.UnimplementedAcceptorServer
13
14      id *proto.NodeId
15
16      // Current round
17      rnd *proto.Round
```

```
18
19      // The last accepted value and the round in which it was accepted
20      vval *proto.Value
21
22      nodes       map[int64]*paxosClient
23      waitForSend func(num int)
24  }
25
26  func NewAcceptor(id *proto.NodeId, waitForSend func(num int)) *Acceptor {
27      return &Acceptor{
28          id: id,
29
30          nodes:       make(map[int64]*paxosClient),
31          waitForSend: waitForSend,
32      }
33  }
34
35  func (a *Acceptor) Prepare(_ context.Context, in *proto.PrepareRequest) (*empty.Empty, error) {
36      if in.GetCrnd().GetVal() > a.rnd.GetVal() {
37          a.rnd = in.GetCrnd()
38      }
39
40      go a.nodes[in.GetFrom().GetVal()].Promise(
41          context.Background(),
42          &proto.PromiseRequest{
43              Rnd:  a.rnd,
44              Val:  a.vval,
45              From: a.id,
46          },
47      )
48      a.waitForSend(1)
49      return &emptypb.Empty{}, nil
50  }
51
52  func (a *Acceptor) Accept(_ context.Context, in *proto.AcceptRequest) (*empty.Empty, error) {
53      if in.GetVal().GetRnd().GetVal() < a.rnd.GetVal() {
54          return &emptypb.Empty{}, nil
55      }
56      a.vval = in.GetVal()
57
58      msg := &proto.LearnRequest{
59          Val:  in.GetVal(),
```

```
60          From: a.id,
61      }
62      for _, node := range a.nodes {
63          go node.Learn(context.Background(), msg)
64      }
65      a.waitForSend(len(a.nodes))
66      return &emptypb.Empty{}, nil
67  }
```

Listing A.8: Implementation of the Acceptor in the Paxos Algorithm.

```
1   package paxos
2
3   import (
4       "context"
5       "gomc/examples/paxos/proto"
6
7       "github.com/golang/protobuf/ptypes/empty"
8       "google.golang.org/protobuf/types/known/emptypb"
9   )
10
11  func ValEquals(a, b *proto.Value) bool {
12      if a.GetRnd().GetVal() != b.GetRnd().GetVal() {
13          return false
14      }
15      if a.GetVal() != b.GetVal() {
16          return false
17      }
18      return true
19  }
20
21  type Learner struct {
22      proto.UnimplementedLearnerServer
23
24      id *proto.NodeId
25
26      recvLrn map[int64]*proto.Value
27
28      numNodes int
29
30      // Consensus Value
31      learnSubscribe []func(string)
32  }
33
34  func NewLearner(id *proto.NodeId, numNodes int) *Learner {
```

```
35      return &Learner{
36          id: id,
37
38          numNodes: numNodes,
39          recvLrn:  make(map[int64]*proto.Value),
40
41          learnSubscribe: make([]func(string), 0),
42      }
43  }
44
45  func (l *Learner) Learn(_ context.Context, in
    ↪  *proto.LearnRequest) (*empty.Empty, error) {
46      l.addValue(in.GetFrom().GetVal(), in.GetVal())
47      numLrn := 0
48      freqVal := &proto.Value{
49          Rnd: &proto.Round{Val: -1},
50      }
51      for _, val := range l.recvLrn {
52          if val.GetRnd().GetVal() > freqVal.GetRnd().GetVal() {
53              freqVal = val
54              numLrn = 0
55          }
56          if ValEquals(freqVal, val) {
57              numLrn++
58          }
59      }
60      if numLrn > l.numNodes/2 {
61          l.emmitLearn(freqVal)
62      }
63      return &emptypb.Empty{}, nil
64  }
65
66  // Add the value to the received value map, if it is the
    ↪  highest round number received from that node
67  func (l *Learner) addValue(from int64, val *proto.Value) {
68      oldVal, ok := l.recvLrn[from]
69      if !ok {
70          l.recvLrn[from] = val
71          return
72      }
73      if val.GetRnd().GetVal() > oldVal.GetRnd().GetVal() {
74          l.recvLrn[from] = val
75      }
76  }
77
78  func (l *Learner) Subscribe(f func(val string)) {
```

```
79      l.learnSubscribe = append(l.learnSubscribe, f)
80  }
81
82  func (l *Learner) emmitLearn(val *proto.Value) {
83      for _, f := range l.learnSubscribe {
84          f(val.GetVal())
85      }
86  }
```

Listing A.9: Implementation of the Learner in the Paxos Algorithm.

A.5 Multipaxos

```
1   package multipaxos
2
3   import (
4       "gomc/examples/multipaxos/proto"
5       "net"
6
7       "google.golang.org/grpc"
8   )
9
10  type server struct {
11      srv   *grpc.Server
12      conns []*grpc.ClientConn
13  }
14
15  func (s *server) StartServer(mp *MultiPaxos, lis net.Listener,
    ↪  srvOpts ...grpc.ServerOption) error {
16      s.srv = grpc.NewServer(srvOpts...)
17      proto.RegisterProposerServer(s.srv, mp)
18      proto.RegisterAcceptorServer(s.srv, mp)
19      proto.RegisterLearnerServer(s.srv, mp)
20      return s.srv.Serve(lis)
21  }
22
23  func (s *server) Stop() {
24      s.srv.Stop()
25      for _, c := range s.conns {
26          c.Close()
27      }
28  }
29
30  func (s *server) DialNodes(addrMap map[int64]string, dialOpts
    ↪  ...grpc.DialOption) (map[int64]*multipaxosClient, error) {
```

```
    nodes := make(map[int64]*multipaxosClient)
    for id, addr := range addrMap {
        conn, err := grpc.Dial(addr, dialOpts...)
        if err != nil {
            return nil, err
        }
        nodes[id] = NewMultipaxosClient(conn)
    }
    return nodes, nil
}
```

Listing A.10: Implementation of the Multipaxos Algorithm. The type implements the optimized variant of the paxos algorithm presented by Lamport [14]. It decides on one value for each time slot. Phase 1 is run once every time a new leader is elected.

```
package multipaxos

import (
    "context"
    "gomc/examples/multipaxos/proto"
    "sync"
    "time"

    "github.com/golang/protobuf/ptypes/empty"
)

type proposer struct {
    proto.UnimplementedProposerServer

    sync.Mutex

    nodeId int64
    qourum int

    Adu      int64
    nextSlot int64

    crnd int64
    cval *proto.Value

    completedPhase1 bool

    leader *LeaderElector

```

```
30      promises map[int64]*proto.PromiseRequest
31
32      pendingProposals []string
33
34      nodes       map[int64]*multipaxosClient
35      waitForSend func(num int)
36      wait        func(time.Duration)
37  }
38
39  func newProposer(id int64, waitForSend func(num int), l
    ↪  *LeaderElector) *proposer {
40      p := &proposer{
41          nodeId: id,
42
43          crnd: id,
44
45          leader: l,
46
47          pendingProposals: make([]string, 0),
48
49          promises:    make(map[int64]*proto.PromiseRequest),
50          waitForSend: waitForSend,
51      }
52      l.LeaderSubscribe(p.newLeader)
53      return p
54  }
55
56  func (p *proposer) performPhaseOne() {
57      p.nextSlot = p.Adu
58      msg := &proto.PrepareRequest{
59          Crnd: p.crnd,
60          Slot: p.Adu,
61          From: p.nodeId,
62      }
63
64      for _, n := range p.nodes {
65          go n.Prepare(context.Background(), msg)
66      }
67      p.waitForSend(len(p.nodes))
68  }
69
70  func (p *proposer) Propose(_ context.Context, prop
    ↪  *proto.ProposeRequest) (*empty.Empty, error) {
71      p.Lock()
72      defer p.Unlock()
73
```

```
74      if !p.leader.IsLeader() {
75          return &empty.Empty{}, nil
76      }
77
78      if !p.completedPhase1 {
79          // Store this value for when phase 1 is completed
80          p.pendingProposals = append(p.pendingProposals,
    ↪   prop.GetVal())
81          return &empty.Empty{}, nil
82      }
83
84      p.nextSlot++
85      // create accept message
86      acc := &proto.AcceptRequest{
87          Val: &proto.Value{
88              Val: prop.GetVal(),
89              Rnd: p.crnd,
90          },
91          Slot: p.nextSlot,
92          From: p.nodeId,
93      }
94
95      // Send accept message
96      for _, n := range p.nodes {
97          go n.Accept(context.Background(), acc)
98      }
99      p.waitForSend(len(p.nodes))
100
101     return &empty.Empty{}, nil
102 }
103
104 func (p *proposer) Promise(_ context.Context, prm
    ↪   *proto.PromiseRequest) (*empty.Empty, error) {
105     p.Lock()
106     defer p.Unlock()
107
108     // Wrong round: ignore
109     if prm.GetRnd() != p.crnd {
110         return &empty.Empty{}, nil
111     }
112
113     // We have already received a promise from this node: ignore
114     if _, ok := p.promises[prm.GetFrom()]; ok {
115         return &empty.Empty{}, nil
116     }
117
```

```
118     // We have already completed phase1, so we just ignore this
    ↪   message
119     if p.completedPhase1 {
120         return &empty.Empty{}, nil
121     }
122
123     p.promises[prm.GetFrom()] = prm
124
125     // Still have not reached qourum size
126     if len(p.promises) < p.qourum {
127         return &empty.Empty{}, nil
128     }
129
130     accepts := p.getValues()
131
132     // Accept messages are added to the front of the queue
133     for _, val := range accepts {
134         p.nextSlot++
135         // create accept message
136         acc := &proto.AcceptRequest{
137             Val: &proto.Value{
138                 Val: val,
139                 Rnd: p.crnd,
140             },
141             Slot: p.nextSlot,
142             From: p.nodeId,
143         }
144
145         // Send accept message
146         for _, n := range p.nodes {
147             go n.Accept(context.Background(), acc)
148         }
149         p.waitForSend(len(p.nodes))
150     }
151
152     for _, val := range p.pendingProposals {
153         p.nextSlot++
154         // create accept message
155         acc := &proto.AcceptRequest{
156             Val: &proto.Value{
157                 Val: val,
158                 Rnd: p.crnd,
159             },
160             Slot: p.nextSlot,
161             From: p.nodeId,
162         }
```

```
163
164         // Send accept message
165         for _, n := range p.nodes {
166             go n.Accept(context.Background(), acc)
167         }
168         p.waitForSend(len(p.nodes))
169     }
170
171     p.pendingProposals = make([]string, 0)
172
173     p.completedPhase1 = true
174     p.promises = make(map[int64]*proto.PromiseRequest)
175
176     return &empty.Empty{}, nil
177 }
178
179 func (p *proposer) getValues() []string {
180     slots := make(map[int64]*proto.PromiseSlot)
181     highestSlot := p.Adu
182     for _, prm := range p.promises {
183         for _, slot := range prm.Slots {
184             if slot.GetSlot() > highestSlot {
185                 highestSlot = slot.GetSlot()
186             }
187
188             oldSlot, ok := slots[slot.GetSlot()]
189             if !ok {
190                 slots[slot.GetSlot()] = oldSlot
191                 continue
192             }
193
194             if slot.GetVal().GetRnd() >
    ↪  oldSlot.GetVal().GetRnd() {
195                 slots[slot.GetSlot()] = slot
196             }
197         }
198     }
199
200     if len(slots) == 0 {
201         return []string{}
202     }
203
204     length := highestSlot - p.Adu
205     vals := make([]string, length)
206     for i := 0; i < int(length); i++ {
207         slot := slots[p.Adu+int64(i)]
```

```
208             vals[i] = slot.GetVal().GetVal()
209         }
210         return vals
211     }
212
213     func (p *proposer) ProposeVal(val string) {
214         // Send value to leader
215         go p.nodes[p.leader.Leader()].Propose(
216             context.Background(),
217             &proto.ProposeRequest{Val: val},
218         )
219         p.waitForSend(1)
220     }
221
222     func (p *proposer) newLeader(newLeader int64) {
223         p.Lock()
224         defer p.Unlock()
225
226         p.completedPhase1 = false
227         p.IncrementCrnd()
228
229         if p.leader.IsLeader() {
230             p.performPhaseOne()
231         }
232     }
233
234     func (p *proposer) IncrementCrnd() {
235         p.crnd += int64(len(p.nodes))
236     }
237
238     func (p *proposer) IncrementAdu() {
239         p.Adu++
240     }
```

Listing A.11: Implementation of the Proposer in the Multipaxos Algorithm.

```
1   package multipaxos
2
3   import (
4       "context"
5       "gomc/examples/multipaxos/proto"
6       "sync"
7
8       "github.com/golang/protobuf/ptypes/empty"
9   )
```

```
10
11  type acceptor struct {
12      proto.UnimplementedAcceptorServer
13
14      sync.Mutex
15
16      nodeId int64
17
18      rnd int64
19
20      slots map[int64]*proto.PromiseSlot
21
22      nodes       map[int64]*multipaxosClient
23      waitForSend func(num int)
24  }
25
26  func newAcceptor(id int64, waitForSend func(num int)) *acceptor
    ↪   {
27      return &acceptor{
28          nodeId: id,
29
30          slots:       make(map[int64]*proto.PromiseSlot),
31          waitForSend: waitForSend,
32      }
33  }
34
35  func (a *acceptor) Prepare(_ context.Context, prp
    ↪   *proto.PrepareRequest) (*empty.Empty, error) {
36      a.Lock()
37      defer a.Unlock()
38
39      if prp.GetCrnd() <= a.rnd {
40          return &empty.Empty{}, nil
41      }
42
43      a.rnd = prp.GetCrnd()
44
45      slots := make([]*proto.PromiseSlot, 0)
46      for _, slot := range a.slots {
47          if slot.Slot >= prp.Slot {
48              slots = append(slots, slot)
49          }
50      }
51
52      go a.nodes[prp.GetFrom()].Promise(
53          context.Background(),
```

```
54          &proto.PromiseRequest{
55              Rnd:   a.rnd,
56              Slots: slots,
57              From:  a.nodeId,
58          },
59      )
60      a.waitForSend(1)
61
62      return &empty.Empty{}, nil
63  }
64
65  func (a *acceptor) Accept(_ context.Context, acc
    ↪  *proto.AcceptRequest) (*empty.Empty, error) {
66      a.Lock()
67      defer a.Unlock()
68
69      if acc.GetVal().GetRnd() < a.rnd {
70          return &empty.Empty{}, nil
71      }
72
73      a.rnd = acc.GetVal().GetRnd()
74      a.slots[acc.GetSlot()] = &proto.PromiseSlot{
75          Slot: acc.GetSlot(),
76          Val:  acc.GetVal(),
77      }
78
79      lrn := &proto.LearnRequest{
80          Val:  acc.GetVal(),
81          Slot: acc.GetSlot(),
82          From: a.nodeId,
83      }
84
85      for _, n := range a.nodes {
86          go n.Learn(context.Background(), lrn)
87      }
88      a.waitForSend(len(a.nodes))
89
90      return &empty.Empty{}, nil
91  }
```

Listing A.12: Implementation of the Acceptor in the Multipaxos Algorithm.

```
1  package multipaxos
2
3  import (
```

```
4       "context"
5       "gomc/examples/multipaxos/proto"
6       "sync"
7
8       "github.com/golang/protobuf/ptypes/empty"
9   )
10
11  type learntSlots struct {
12      learnt bool
13      rnd    int64
14      votes  map[int64]*proto.Value
15  }
16
17  type learner struct {
18      proto.UnimplementedLearnerServer
19
20      sync.Mutex
21
22      nodeId int64
23
24      learns map[int64]*learntSlots
25
26      learnSubscribe []func(string, int64)
27
28      qourum int
29  }
30
31  func newLearner(id int64) *learner {
32      return &learner{
33          nodeId: id,
34
35          learns: make(map[int64]*learntSlots),
36      }
37  }
38
39  func (l *learner) Learn(_ context.Context, lrn
    ↪  *proto.LearnRequest) (*empty.Empty, error) {
40      l.Lock()
41      defer l.Unlock()
42
43      slotId := lrn.GetSlot()
44      slot, ok := l.learns[slotId]
45      if !ok {
46          slot = &learntSlots{}
47          l.learns[slotId] = slot
48      }
```

```
49
50      if slot.learnt {
51          return &empty.Empty{}, nil
52      }
53
54      // If the round of the stored slot is higher than the round
    ↪   of the received slot.
55      if lrn.GetVal().GetRnd() < slot.rnd {
56          return &empty.Empty{}, nil
57      }
58
59      // The new learn is larger then the current. Set the
    ↪   current to the new and reset votes
60      if lrn.GetVal().GetRnd() > slot.rnd {
61          slot.rnd = lrn.GetVal().GetRnd()
62          slot.votes = make(map[int64]*proto.Value)
63      }
64
65      // The learn is for the current round. Add it to the slot
66      if lrn.GetVal().GetRnd() == slot.rnd {
67          // We have already received a learn from this node for
    ↪   this slot and this round.
68          // Ignore this one
69          if _, ok := slot.votes[lrn.GetFrom()]; ok {
70              return &empty.Empty{}, nil
71          }
72
73          slot.votes[lrn.GetFrom()] = lrn.GetVal()
74          if len(slot.votes) >= l.qourum {
75              slot.learnt = true
76              l.emmitLearn(lrn.GetVal(), slotId)
77          }
78      }
79      return &empty.Empty{}, nil
80  }
81
82  func (l *learner) emmitLearn(val *proto.Value, slotId int64) {
83      for _, callback := range l.learnSubscribe {
84          callback(val.GetVal(), slotId)
85      }
86  }
87
88  func (l *learner) LearnSubscribe(callback func(string, int64)) {
89      l.learnSubscribe = append(l.learnSubscribe, callback)
90  }
```

Listing A.13: Implementation of the Learner in the Multipaxos Algorithm.

```go
package multipaxos

type LeaderElector struct {
    id      int64
    correct map[int64]bool
    leader  int64

    leaderSub []func(int64)

    stopped bool
}

func NewLeaderElector(id int64, addr map[int64]string)
↪   *LeaderElector {
    var leader int64
    correct := make(map[int64]bool)
    for id := range addr {
        correct[id] = true
        if id > leader {
            leader = id
        }
    }
    return &LeaderElector{
        id:      id,
        correct: correct,
        leader:  leader,
    }
}

func (l *LeaderElector) NodeCrash(id int, _ bool) {
    if l.stopped {
        return
    }

    l.correct[int64(id)] = false

    if int64(id) == l.leader {
        leader := l.nextLeader()
        l.leader = leader
        for _, f := range l.leaderSub {
            f(leader)
        }
```

```
42      }
43  }
44
45  func (l *LeaderElector) nextLeader() int64 {
46      var leader int64
47      for id, ok := range l.correct {
48          if ok {
49              if id > leader {
50                  leader = id
51              }
52          }
53      }
54      return leader
55  }
56
57  // Subscribe to leader change calls
58  func (l *LeaderElector) LeaderSubscribe(f func(id int64)) {
59      l.leaderSub = append(l.leaderSub, f)
60  }
61
62  func (l *LeaderElector) IsLeader() bool {
63      return l.leader == l.id
64  }
65
66  func (l *LeaderElector) Leader() int64 {
67      return l.leader
68  }
```

Listing A.14: Implementation of the Leader Elector in the Multipaxos Algorithm.

B Configuration Details

B.1 Configuring the Simulation

B.1.1 Parameters for PrepareSimulation

- `StateManagerOption`: Export the discovered state space after simulation. The default value is to not export the state.
  - `WithTreeStateManager`: Configure a `TreeStateManager` to be used during the simulation. The option must be configured with a function that collects the local state of a node and a function that checks the equality of two states.
  - `WithStateManager`: Configure a custom State Manager to be used during the simulation.

B.1.2 Simulator Option

- `schedulerOption`: Select the scheduler to be used during the simulation. The default value is a `PrefixScheduler`. The functions that can be used to provide this option are:
  - `PrefixScheduler`: A systematic scheduler that searches the entire state space.
  - `RandomWalkScheduler`: A randomized scheduler that randomly selects the next event from the available events. Does not guarantee to search all states, but samples more varied runs compared to a prefix scheduler. Is configured with a seed.
  - `ReplayScheduler`: A scheduler that replays a specific run. Does not explore the state space, but can be used to retry a specific run. Returns an error if it is unable to replay the run.
  - `WithScheduler`: An option that can be used to provide a different implementation of the scheduler.
- `ignoreErrorOption`: If true will ignore errors while the simulation runs. A summary of all collected errors will be provided at the end. If false the simulation will stop when an error is encountered. Default value is `false`.
  - `IgnoreError`: Configure the simulation to ignore errors during the simulation.
- `ignorePanicOption`: If true will ignore panics that occurs when executing an event on a node. If false the simulation will recover from the panic and return it as an error. The simulation may continue or return depending on the value of the `ignoreErrorOption`. Default value is `false`.
  - `IgnorePanic`: Configure the simulation to ignore panics during the simulation.

- `maxRunsOption`: Maximum number of runs to be simulated. The maximum number of runs to be simulated is an important factor in the total simulation time, but only simulating a limited amount of runs will impact the guarantees of the simulation. Default value is `1000`.
  - `MaxRuns`: Set the maximum number of runs to the provided value.
- `maxDepthOption`: Maximum depth of the runs that are simulated. Default value is `1000`.
  - `MaxDepth`: Set the maximum depth of runs to the provided value.
- `numConcurrentOption`: Maximum number of runs that are concurrently simulated. Default value is the value of the `GOMAXPROCS` environmental variable.
  - `NumConcurrent`: Set the maximum number of concurrent runs to the provided value.

B.1.3 Parameters for Simulation.Run

- `InitNodeOption`: Provide the function used to create the nodes that are used in the simulation. The function should initialize the Event Managers used during the run. It returns a map with (node id, node) key-value pairs.
  - `InitNodeFunc`: Provide a function that initializes all nodes and returns the node map.
  - `InitSingleNode`: The Provide a slice of the node ids that will be created and a function that creates a single node with a provided node id. The option uses the provided function to create each of the specified nodes and add it to the map.
- `RequestOption`: Specify the requests sent to the system. The requests are used to start the simulation.
  - `WithRequests`: Configure a list of requests that are sent to the system during the simulation.
- `CheckerOption`: Set the Checker used to test the algorithm.
  - `WithPredicateChecker`: Use a `PredicateChecker` to check the algorithm. The function takes a set of properties defined as Go functions of the type `Predicate` as parameters.
  - `WithChecker`: use a custom checker to check the algorithm.

B.1.4 Run Options

- `failureManagerOption`: Select and configure the Failure Manager to be used during simulation. Default value is `PerfectFailureManager` with no crashes configured.
  - `WithPerfectFailureManager`: Configure the simulation to use a `PerfectFailureManager`. The function is configured with a `crashFunc` specifying how the crash manifests on the node and a slice of node ids of the nodes that will crash during the simulation.
  - `WithFailureManager`: An option that can be used to provide a different implementation of the Failure Manager.
- `stopOption`: Function used to close a node after a run has been completed. The function should completely clean up a node after the simulation. Should be configured to close network connections, stop goroutines etc. The default value is an empty function.
  - `WithStopFunction`: Set the function used to stop the nodes.
- `exportOption`: Export the discovered state space after simulation. The default value is to not export the state.
  - `Export`: Configure a list of `io.Writer`s that that the state will be exported to.

B.2 Configuring The Runner

B.2.1 Parameters for PrepareRunner

- `InitNodeOption`: Provide the function used to create the nodes that are used in the simulation. The function should initialize the Event Managers used during the run. It returns a map with (node id, node) key-value pairs.
  - `InitNodeFunc`: Provide a function that initializes all nodes and returns the node map.
  - `InitSingleNode`: The Provide a slice of the node ids that will be created and a function that creates a single node with a provided node id. The option uses the provided function to create each of the specified nodes and add it to the map.
- `GetStateOption`: Configure a function that is used to to collect the state of a local node. Similar to the function used to configure the `TreeStateManager`.
  - `WithStateFunction`: Configure a function used to collect the state of a node.

B.2.2 Runner Options

- `stopOption`: Function used to close a node after a run has been completed. The function should completely clean up a node after the simulation. Should be configured to close network connections, stop goroutines etc. The default value is an empty function.
  - `WithStopFunction`: Set the function used to stop the nodes.
- `eventChanBufferOption`: Specify the size of the channel used to store pending events on each node. Default value is `100`.
  - `EventChanBufferSize`: Set the the event channel buffer size
- `recordChanBufferOption`: Specify the size of the channel used to send records. Default value is `100`.
  - `RecordChanSize`: Set the the record channel buffer size.

C Prefix Scheduler

The prefix scheduler works by maintaining a set of unexplored prefixes. A prefix is some sequence of the first n events of a run, where n can be any number between 0 and the total length of the run. In each run the scheduler selects one of the unexplored prefixes and schedules the first n events of the run so that it matches the prefix. Then, for all the pending events it creates new sequences with the pending event appended to the current run. These are discovered prefixes, that the scheduler knows are possible to schedule, but that it has not explored yet. It selects one of the prefixes to continue to explore in this run, and adds the remaining to the set of unexplored prefixes. If there are no pending events then the run is over. If there are no more unexplored prefixes then the entire state space has been visited. The first run uses an empty prefix and the scheduler immediately starts adding new unexplored prefixes.

Consider an example where there are two pending events, e_1 and e_2. We name the set of unexplored prefixes as U. The simulation start and the first run is therefore empty. The scheduler creates two prefixes, $P_1 = e_1$ and $P_2 = e_2$, it selects one of them to be explored in this run, say P_1, and adds the other to U. When e_1 is executed the event e_3 is added to the pending events. The pending events are now e_2 and e_3. e_2 is a pending event since it has not been executed in this run. The scheduler now creates two new prefixes: $P_3 = e_1e_2$ and $P_4 = e_1e_3$. It selects P_3 to be explored in this run, selecting the event e_2, and adds P_4 to U. Now there is only one pending event, e_3, which is added to a new prefix, $P_5 = e_1e_2e_3$, which is then executed. There are no more pending events and the run is therefore completed. For the next run the scheduler selects one of the prefixes from U, say P_4. It follows the prefix, selecting e_1 for the first event and e_3 for the second. There is only one pending event, e_2, which is used to create a new prefix $P_6 = e_1e_3e_2$, which is the second completed run. For the third run it selects P_2 from U and using the same process it explores the prefix and schedules the final run $P_7 = e_2e_1e_3$.

D Test Configuration

D.1 Simulation Time

```
package main

import (
    "gomc"
    "gomc/eventManager"
    "testing"

    "golang.org/x/exp/slices"
)

func BenchmarkConsensus(b *testing.B) {
    sim := gomc.PrepareSimulation(
        gomc.WithTreeStateManager(
            func(node *HierarchicalConsensus[int]) state {
                return state{
                    proposed: node.ProposedVal,
                    decided:  slices.Clone(node.DecidedVal),
                }
            },
            func(a, b state) bool {
                if a.proposed != b.proposed {
                    return false
                }
                return slices.Equal(a.decided, b.decided)
            },
        ),
        gomc.PrefixScheduler(),
    )

    nodeIds := []int{1, 2, 3}
    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        sim.Run(
            gomc.InitSingleNode(nodeIds,
                func(id int, sp
↪   eventManager.SimulationParameters)
↪   *HierarchicalConsensus[int] {
                    send := eventManager.NewSender(sp)
                    node := NewHierarchicalConsensus[int](
                        id,
                        nodeIds,
                        send.SendFunc(id),
```

```
                    )
                    sp.CrashSubscribe(id, node.Crash)
                    return node
                },
            ),
            gomc.WithRequests(
                gomc.NewRequest(1, "Propose", Value[int]{1}),
                gomc.NewRequest(2, "Propose", Value[int]{2}),
                gomc.NewRequest(3, "Propose", Value[int]{3}),
            ),
            gomc.WithPredicateChecker(predicates...),
            gomc.WithPerfectFailureManager(
                func(t *HierarchicalConsensus[int]) { t.crashed
↪   = true },
                2,
            ),
        )
    }
}
```

Listing D.1: Configuration of the simulation of the Hierarchical Consensus algorithm using the Sender Event Manager.

```
package main

import (
    "gomc"
    "testing"
)

func BenchmarkConsensus(b *testing.B) {
    sim := gomc.PrepareSimulation(
        gomc.WithTreeStateManager(getState, cmpState),
        gomc.PrefixScheduler(),
    )

    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        sim.Run(
            gomc.InitNodeFunc(
                createNodes(addrMap),
            ),
            gomc.WithRequests(
                gomc.NewRequest(1, "Propose", "1"),
                gomc.NewRequest(2, "Propose", "2"),
```

```
                gomc.NewRequest(3, "Propose", "3"),
            ),
            gomc.WithPredicateChecker(predicates...),
            gomc.WithPerfectFailureManager(
                func(t *GrpcConsensus) { t.Stop() }, 2,
            ),
            gomc.WithStopFunctionSimulator(func(t
↪   *GrpcConsensus) { t.Stop() }),
        )
    }
}
```

Listing D.2: Configuration of the simulation of the Hierarchical Consensus algorithm using gRPC.

```
package main

import (
    "gomc"
    "gomc/eventManager"
    "testing"

    "golang.org/x/exp/maps"
    "golang.org/x/exp/slices"
)

func BenchmarkRrb(b *testing.B) {
    nodeIds := []int{0, 1, 2}
    crashedNodes := []int{1}

    sim := gomc.PrepareSimulation(
        gomc.WithTreeStateManager(
            func(node *Rrb) State {
                return State{
                    delivered:      maps.Clone(node.delivered),
                    sent:           maps.Clone(node.sent),
                    deliveredSlice:
↪   slices.Clone(node.deliveredSlice),
                }
            },
            func(s1, s2 State) bool {
                if !maps.Equal(s1.delivered, s2.delivered) {
                    return false
                }
```

```
                if !slices.Equal(s1.deliveredSlice,
↪   s2.deliveredSlice) {
                    return false
                }
                return maps.Equal(s1.sent, s2.sent)
            },
        ),
        gomc.PrefixScheduler(),
    )

    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        sim.Run(
            gomc.InitNodeFunc(
                func(sp eventManager.SimulationParameters)
↪   map[int]*Rrb {
                    send := eventManager.NewSender(sp)
                    nodes := map[int]*Rrb{}
                    for _, id := range nodeIds {
                        nodes[id] = NewRrb(
                            id,
                            nodeIds,
                            send.SendFunc(id),
                        )
                    }
                    return nodes
                },
            ),
            gomc.WithRequests(
                gomc.NewRequest(0, "Broadcast", "Test Message"),
            ),
            gomc.WithPredicateChecker(predicates...),
            gomc.WithPerfectFailureManager(
                func(t *Rrb) { t.crashed = true },
                crashedNodes...,
            ),
        )
    }
}
```

Listing D.3: Configuration of the simulation of the Eager Reliable Broadcast algorithm.

```
package main

```

```
import (
    "gomc"
    "gomc/eventManager"
    "testing"

    "golang.org/x/exp/slices"
)

func BenchmarkOnrr(b *testing.B) {

    nodeIds := []int{1, 2, 3}

    sim := gomc.PrepareSimulation(
        gomc.WithTreeStateManager(
            func(node *onrr) State {
                reads := []int{}
                reads = append(reads, node.possibleReads...)

                // If there has been a read indication store it. Otherwise ignore it
                read := 0
                currentRead := false
                select {
                case read = <-node.ReadIndicator:
                    currentRead = true
                default:
                }

                return State{
                    ongoingRead:   node.ongoingRead,
                    ongoingWrite:  node.ongoingWrite,
                    possibleReads: reads,
                    read:          read,
                    currentRead:   currentRead,
                }
            },
            func(a, b State) bool {
                if a.ongoingRead != b.ongoingRead {
                    return false
                }
                if a.ongoingWrite != b.ongoingWrite {
                    return false
                }
                if a.currentRead != b.currentRead {
                    return false
                }
```

```
48                  if a.read != b.read {
49                      return false
50                  }
51                  return slices.Equal(a.possibleReads,
    ↪  b.possibleReads)
52              },
53          ),
54          gomc.RandomWalkScheduler(1),
55          gomc.MaxRuns(10000),
56      )
57
58      b.ResetTimer()
59      for i := 0; i < b.N; i++ {
60          sim.Run(
61              gomc.InitNodeFunc(
62                  func(sp eventManager.SimulationParameters)
    ↪  map[int]*onrr {
63                      send := eventManager.NewSender(sp)
64
65                      nodes := make(map[int]*onrr)
66                      for _, id := range nodeIds {
67                          nodes[id] = NewOnrr(id,
    ↪  send.SendFunc(id), nodeIds)
68                      }
69                      go func() {
70                          for {
71                              <-nodes[1].WriteIndicator
72                          }
73                      }()
74                      return nodes
75                  },
76              ),
77              gomc.WithRequests(
78                  gomc.NewRequest(1, "Write", 2),
79                  gomc.NewRequest(2, "Read"),
80                  gomc.NewRequest(3, "Read"),
81              ),
82              gomc.WithPredicateChecker(predicates...),
83          )
84      }
85  }
```

Listing D.4: Configuration of the simulation of the Majority Voting Regular Register algorithm.

```go
package paxos

import (
    "context"
    "gomc/eventManager"
    "net"
    "testing"

    "gomc"

    "google.golang.org/grpc"
    "google.golang.org/grpc/credentials/insecure"
    "google.golang.org/grpc/test/bufconn"
)

func BenchmarkPaxos(b *testing.B) {
    addresses := map[int64]string{
        1: ":1",
        2: ":2",
        3: ":3",
    }

    sim := gomc.PrepareSimulation(
        gomc.WithTreeStateManager(
            func(t *Server) State {
                return State{
                    proposed: t.Proposal,
                    decided:  t.Decided,
                }
            },
            func(s1, s2 State) bool {
                return s1 == s2
            },
        ),
        gomc.RandomWalkScheduler(1),
    )

    addrToIdMap := map[string]int{}
    for id, addr := range addresses {
        addrToIdMap[addr] = int(id)
    }

    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        sim.Run(
```

```
46              gomc.InitNodeFunc(func(sp
    ↪ eventManager.SimulationParameters) map[int]*Server {
47                  lisMap := map[string]*bufconn.Listener{}
48                  for _, addr := range addresses {
49                      lisMap[addr] = bufconn.Listen(bufSize)
50                  }
51                  gem :=
    ↪ eventManager.NewGrpcEventManager(addrToIdMap, sp)
52
53                  nodes := make(map[int]*Server)
54                  for id, addr := range addresses {
55                      srv, err := NewServer(id, addresses,
    ↪ gem.WaitForSend(int(id)))
56                      if err != nil {
57                          b.Errorf("Error while starting
    ↪ simulation: %v", err)
58                      }
59                      go srv.StartServer(lisMap[addr])
60                      sp.CrashSubscribe(int(id), srv.NodeCrash)
61                      nodes[int(id)] = srv
62                  }
63
64                  for id, node := range nodes {
65                      node.DialNodes(
66                          grpc.WithUnaryInterceptor(gem.UnaryClie
    ↪ ntControllerInterceptor(id)),
67                          grpc.WithContextDialer(
68                              func(ctx context.Context, s string)
    ↪ (net.Conn, error) {
69                                  return
    ↪ lisMap[s].DialContext(ctx)
70                              },
71                          ),
72                          grpc.WithBlock(),
73                          grpc.WithTransportCredentials(insecure.
    ↪ NewCredentials()),
74                      )
75                  }
76                  return nodes
77              }),
78              gomc.WithRequests(
79                  gomc.NewRequest(1, "Propose", "1"),
80                  gomc.NewRequest(2, "Propose", "2"),
81                  gomc.NewRequest(3, "Propose", "3"),
82              ),
83              gomc.WithPredicateChecker(predicates...),
```

```
            gomc.WithPerfectFailureManager(func(t *Server) {
↪   t.Stop() }, 1),
            gomc.WithStopFunctionSimulator(func(t *Server) {
↪   t.Stop() }),
        )
    }
}
```

Listing D.5: Configuration of the simulation of the Paxos algorithm.

```
package multipaxos

import (
    "gomc"
    "testing"
)

var nodes = map[int64]string{
    1: ":1",
    2: ":2",
    3: ":3",
}

func BenchmarkMultipaxos(b *testing.B) {
    sim := gomc.PrepareSimulation(
        gomc.WithTreeStateManager(getState, cmpState),
        gomc.PrefixScheduler(),
    )

    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        sim.Run(
            gomc.InitNodeFunc(
                InitNodes(nodes),
            ),
            gomc.WithRequests(
                gomc.NewRequest(1, "ProposeVal", "1"),
                gomc.NewRequest(2, "ProposeVal", "2"),
                gomc.NewRequest(3, "ProposeVal", "3"),
            ),
            gomc.WithPredicateChecker(predicates...),
            gomc.WithPerfectFailureManager(
                func(t *MultiPaxos) { t.Stop() }, 2,
            ),
```

```
            gomc.WithStopFunctionSimulator(func(t *MultiPaxos)
↪   { t.Stop() }),
        )
    }
}
```

Listing D.6: Configuration of the simulation of the Multipaoxs algorithm.

D.2 Finding Bugs

```
package main

import (
    "context"
    "net"
    "testing"
    "time"

    "gomc"
    "gomc/checking"
    "gomc/eventManager"
    "gomc/request"

    "golang.org/x/exp/slices"
    "google.golang.org/grpc"
    "google.golang.org/grpc/credentials/insecure"
    "google.golang.org/grpc/test/bufconn"
)

var predicates = []checking.Predicate[state]{
    checking.Eventually(
        // C1: Termination
        func(s checking.State[state]) bool {
            return checking.ForAllNodes(func(s state) bool {
                return len(s.decided) > 0
            }, s, true)
        },
    ),
    func(s checking.State[state]) bool {
        // C2: Validity
        proposed := make(map[string]bool)
        for _, node := range s.LocalStates {
            proposed[node.proposed] = true
        }
        return checking.ForAllNodes(func(s state) bool {
```

```
36              if len(s.decided) < 1 {
37                  // The process has not decided a value yet
38                  return true
39              }
40              return proposed[s.decided[0]]
41          }, s, false)
42      },
43      func(s checking.State[state]) bool {
44          // C3: Integrity
45          return checking.ForAllNodes(func(s state) bool { return
    ↪ len(s.decided) < 2 }, s, false)
46      },
47      func(s checking.State[state]) bool {
48          // C4: Agreement
49          decided := make(map[string]bool)
50          checking.ForAllNodes(func(s state) bool {
51              for _, val := range s.decided {
52                  decided[val] = true
53              }
54              return true
55          }, s, true)
56          return len(decided) <= 1
57      },
58  }
59
60  func createNodes(addrMap map[int32]string) func(sp
    ↪ eventManager.SimulationParameters) map[int]*GrpcConsensus {
61      var addrToIdMap = map[string]int{}
62      for id, addr := range addrMap {
63          addrToIdMap[addr] = int(id)
64      }
65      return func(sp eventManager.SimulationParameters)
    ↪ map[int]*GrpcConsensus {
66          gem := eventManager.NewGrpcEventManager(addrToIdMap, sp)
67          lisMap := map[string]*bufconn.Listener{}
68          for _, addr := range addrMap {
69              lisMap[addr] = bufconn.Listen(bufSize)
70          }
71
72          nodes := map[int]*GrpcConsensus{}
73          for id, addr := range addrMap {
74              gc := NewGrpcConsensus(id, lisMap[addr],
    ↪ gem.WaitForSend(int(id)))
75              sp.CrashSubscribe(int(id), gc.Crash)
76              nodes[int(id)] = gc
77          }
```

```
78
79          for id, node := range nodes {
80              node.DialServers(
81                  addrMap,
82                  grpc.WithContextDialer(
83                      func(ctx context.Context, s string)
    ↪  (net.Conn, error) {
84                          return lisMap[s].DialContext(ctx)
85                      },
86                  ),
87                  grpc.WithBlock(),
88                  grpc.WithTransportCredentials(insecure.NewCrede
    ↪  ntials()),
89                  grpc.WithUnaryInterceptor(gem.UnaryClientContro
    ↪  llerInterceptor(int(id))),
90              )
91          }
92          return nodes
93      }
94  }
95
96  func getState(node *GrpcConsensus) state {
97      return state{
98          proposed: node.ProposedVal,
99          decided:  slices.Clone(node.DecidedVal),
100     }
101 }
102
103 func cmpState(a, b state) bool {
104     if a.proposed != b.proposed {
105         return false
106     }
107     return slices.Equal(a.decided, b.decided)
108 }
109
110 var simulations = []struct {
111     nodes        map[int32]string
112     crashedNodes []int
113 }{
114     {
115         map[int32]string{
116             1: ":1",
117             2: ":2",
118             3: ":3",
119         },
120         []int{},
```

```
121     },
122     {
123         map[int32]string{
124             1: ":1",
125             2: ":2",
126             3: ":3",
127         },
128         []int{1},
129     },
130     {
131         map[int32]string{
132             1: ":1",
133             2: ":2",
134             3: ":3",
135             4: ":4",
136             5: ":7",
137             6: ":6",
138             7: "127:0:0:1",
139         },
140         []int{2},
141     },
142 }
143
144 func TestGrpcConsensusPrefix(t *testing.T) {
145     sim := gomc.PrepareSimulation(
146         gomc.WithTreeStateManager(getState, cmpState),
147         gomc.PrefixScheduler(),
148     )
149
150     for i, test := range simulations {
151         requests := []request.Request{}
152         for id, addr := range test.nodes {
153             requests = append(requests,
    ↪  gomc.NewRequest(int(id), "Propose", addr))
154         }
155         start := time.Now()
156         resp := sim.Run(
157             gomc.InitNodeFunc(createNodes(test.nodes)),
158             gomc.WithRequests(requests...),
159             gomc.WithPredicateChecker(predicates...),
160             gomc.WithPerfectFailureManager(func(t
    ↪  *GrpcConsensus) { t.Stop() }, test.crashedNodes...),
161             gomc.WithStopFunctionSimulator(func(t
    ↪  *GrpcConsensus) { t.Stop() }),
162         )
163         duration := time.Since(start)
```

```
164         _, desc := resp.Response()
165         t.Logf("Test %v - Duration %v: %v", i, duration, desc)
166     }
167 }
168
169 func TestGrpcConsensusRandom(t *testing.T) {
170     sim := gomc.PrepareSimulation(
171         gomc.WithTreeStateManager(getState, cmpState),
172         gomc.RandomWalkScheduler(0),
173     )
174
175     for i, test := range simulations {
176         requests := []request.Request{}
177         for id, addr := range test.nodes {
178             requests = append(requests,
    ↪  gomc.NewRequest(int(id), "Propose", addr))
179         }
180         start := time.Now()
181         resp := sim.Run(
182             gomc.InitNodeFunc(createNodes(test.nodes)),
183             gomc.WithRequests(requests...),
184             gomc.WithPredicateChecker(predicates...),
185             gomc.WithPerfectFailureManager(func(t
    ↪  *GrpcConsensus) { t.Stop() }, test.crashedNodes...),
186             gomc.WithStopFunctionSimulator(func(t
    ↪  *GrpcConsensus) { t.Stop() }),
187         )
188         duration := time.Since(start)
189         _, desc := resp.Response()
190         t.Logf("Test %v - Duration %v: %v", i, duration, desc)
191     }
192 }
```

Listing D.7: Configuration of the simulation of the Hierarchical Consensus algorithm using gRPC. `createNodes` is a factory function that initializes the nodes used for the simulation. The `simulatios` variable defines the scenarios used for the tests. The `TestGrpcConsensusPrefix` test runs the simulation with a prefix scheduler while the `TestGrpcConsensusRandom` test runs the simulation with a random scheduler.