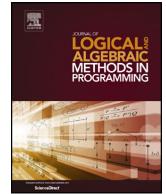
Journal of Logical and Algebraic Methods in Programming 104 (2019) 254–273

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp



Automated test case generation for the Paxos single-decree protocol using a Coloured Petri Net model

Rui Wang [a,*], Lars Michael Kristensen [a], Hein Meling [b], Volker Stolz [a]

[a] Department of Computing, Mathematics, and Physics, Western Norway University of Applied Sciences, Norway
[b] Department of Electrical Engineering and Computer Science, University of Stavanger, Norway

ARTICLE INFO

Article history:
Received 27 March 2018
Received in revised form 10 December 2018
Accepted 13 February 2019
Available online 14 February 2019

Keywords:
Coloured Petri Nets
Distributed systems
Model-based testing

ABSTRACT

Implementing test suites for distributed software systems is a complex and time-consuming task due to the number of test cases that need to be considered in order to obtain high coverage. We show how a formal Coloured Petri Net model can be used to automatically generate a suite of test cases for the Paxos distributed consensus protocol. The test cases cover both normal operation of the protocol as well as failure injection. To evaluate our model-based testing approach, we have implemented the Paxos protocol in the Go programming language using the quorum abstractions provided by the Gorums framework. Our experimental evaluation shows that we obtain high code coverage for our Paxos implementation using the automatically generated test cases.

© 2019 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

1. Introduction

Systematic testing is an important activity in software development. This is especially important for fault-tolerant distributed systems, because they are notoriously difficult to implement correctly [1]. The reason for this difficulty is that such systems must cope with both concurrency and failures, e.g. due to crashes and network partitions. Distributed systems therefore employ protocols with complex logic to tolerate individual component failures without causing service disruption for users. Testing approaches and programming abstractions that can be used to systematically test and simplify the implementation of protocols for distributed systems are therefore important.

Model-based testing (MBT) [2] has emerged as a powerful approach for testing software, and as part of our ongoing research effort, we are investigating the application of MBT on protocols for state machine replication (SMR). SMR is a core technique for developing fault-tolerant distributed systems that can tolerate a bounded number of server failures. In MBT, we construct a model of the system under test (SUT) and its environment, in order to generate test cases. The goal of MBT is validation and error-detection by finding observable differences between the behavior of the implementation and the intended behavior of the SUT, as defined by the model. A test case consists of inputs to the SUT and the expected output, which determines whether the execution of the test was successful or failed. Finally, it involves implementing a test adaptor that can be used to embed the SUT, enabling the test cases to be executed against the SUT, and their output compared against the expected output.

Coloured Petri Nets (CPNs) [3] are a formal modeling language that can model distributed systems combining Petri Nets and the Standard ML programming language. Petri Nets provide the foundation for modeling concurrency, synchronization,

* Corresponding author.
E-mail addresses: rwa@hvl.no (R. Wang), lmkr@hvl.no (L.M. Kristensen), hein.meling@uis.no (H. Meling), vsto@hvl.no (V. Stolz).

https://doi.org/10.1016/j.jlamp.2019.02.004
2352-2208/© 2019 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

communication, and resource sharing, while Standard ML provides the primitives for compact data modeling and sequential computations. Construction, simulation, validation, and verification of CPN models are supported by CPN Tools [4]. CPNs and CPN Tools have been widely used for modeling, validation, and verification of distributed systems protocols [5], but their application in software testing has only been explored to a limited extent [6–8]. Recently, CPNs have been explored in the context of automated code generation to obtain an implementation of a modeled system [9]. Even though the automated code generation is applied to obtain such an implementation of the modeled system, it is seldom that the correctness of the model-to-text transformations and their implementation can be formally proved. Thus, it is also an important task in the engineering of distributed systems to comprehensively test the implementation. Therefore, we have developed the MBT/CPN library [10] that extends CPN Tools with the support for model-based test case generation. The reason we chose CPNs as the foundation for our testing approach is that CPNs have a strong track record for modeling distributed systems and are able to create parametric models and perform model validation. Moreover, CPNs also have mature tools to support both simulation and state space exploration, which is important for implementing our approach and for our practical experiments.

The contribution of this paper is the application of CPNs and the MBT/CPN library [10] for model-based testing of an implementation of Paxos [11]. Paxos is a fault-tolerant consensus protocol that makes it possible to construct a replicated service, or SMR, using a group of server replicas. Paxos is an important foundational building block, and a whole family of Paxos-based protocols have been developed [12–15], focusing on different attributes such as latency and throughput. Moreover, Paxos is also the basis for many production systems such as Google's Chubby [16] and Spanner [17], and Amazon Web Services [18]. However, Paxos is also known for being difficult to understand and implement correctly [19]. The main aim of our work has been to develop a practically-oriented approach that narrows the gap between the provably correct in theory, and a correct implementation in practice. We use finite-state model checking to automatically validate the correctness of small configurations of the CPN model used for test case generation. This increases confidence in the test cases that are then subsequently extracted from running a set of simulations of the CPN model. The use of simulation to extract test cases (which are then executed against the SUT) ensures scalability of our approach. It also means that our approach (in general) only tests the SUT against a subset of the behaviors specified by the CPN model. As such our approach should be seen as aimed at validating an implementation and detecting implementation errors.

A secondary contribution is an implementation of the single-decree Paxos protocol that is especially amenable to testing. Single-decree Paxos allows a collection of servers to operate as a coherent group and to agree on a common value, while tolerating the failure of some of its members. The implementation, written in Go, takes advantage of quorum abstractions provided by the Gorums middleware library [20]. These abstractions include the ability to perform invocations on a set of server replicas, and collect, analyze, and combine a quorum of replies into a single representative reply to be used in the next protocol step. These abstractions also help to shield the programmer from having to explicitly deal with low-level communication and error handling.

The paper is organized as follows. §2 introduces the Paxos consensus protocol and gives an overview of the constructed CPN model, while §3 provides detailed models of the various Paxos agents. §4 gives an overview of Gorums and its abstractions, and outlines our Gorums-based implementation of Paxos. §5 presents our testing approach, and our test adapter developed to execute the test cases generated from the CPN model. §6 discusses test case generation and our experimental results obtained using CPN Tools and the MBT/CPN testing library. §7 discusses related work. Finally, §8 concludes the paper and discusses future work. The reader is assumed to be familiar with the basic syntactical and semantical concepts of Petri Nets (places, transitions, tokens, and transition enabling and occurrence). The CPN model and Paxos implementation presented herein are only partial. The full details of the CPN model are available via [21].

2. The Paxos consensus protocol and CPN model overview

The objective of a distributed *consensus* algorithm such as Paxos is to have a single value chosen among those proposed, while the *safety* (S) and *liveness* (L) properties [22,23] below are upheld with a *correct replica* being a replica that does not fail:

- S1 Only a proposed value may be chosen.
- S2 Only a single value is chosen.
- S3 Only a chosen value may be learned by a correct replica.
- L1 Some proposed value is eventually chosen.
- L2 Once a value is chosen, correct replicas eventually learn it.

Note that the definition permits multiple values to be proposed for consensus. An algorithm satisfying the above safety properties is considered safe in that all replicas that learn the chosen value remain consistent with each other. However, we note that distributed consensus is impossible in an asynchronous system model [24]. Therefore, to satisfy liveness, periods of synchrony are required.

The single-decree Paxos consensus protocol can be used by a distributed application, in which the Paxos replicas need to agree on a single common value among potentially many input values. We assume that the input values are sent to the Paxos replicas from one or more clients, and then the decided output value is returned to these clients.

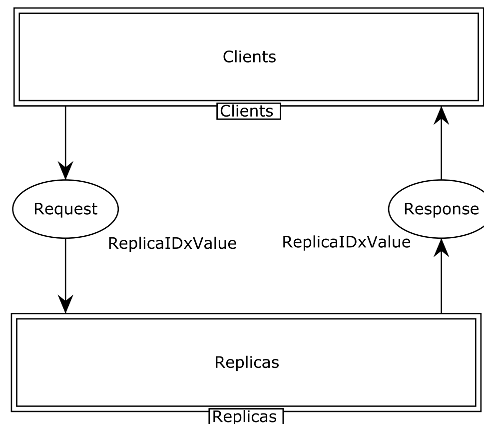


Fig. 1. Top-level CPN module for Paxos.

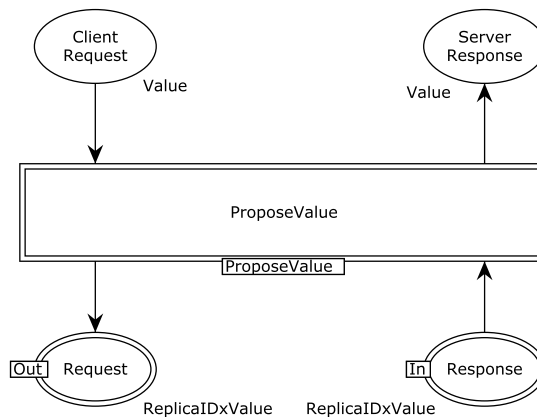


Fig. 2. The Clients CPN module.

The constructed CPN model of the single-decree Paxos protocol is comprised of 23 hierarchically organized modules. Fig. 1 shows the top-level module of the CPN model consisting of two substitution transitions (drawn as rectangles with a double border) connected by the two places Request and Response. The name of the submodule associated with each substitution transition is written in the name-tag positioned next to the substitution transition. The substitution transition Clients and its associated submodule Clients are modeling the behavior of the clients that may propose values to be chosen. The substitution transition Replicas and its associated submodule are modeling the behavior of the distributed replicas executing the Paxos protocol in order to reach consensus on a value proposed by the clients. The client sends a request to the Paxos replicas by putting a token on the place Request and then waits for the decided response value to be returned as a token on place Response.

Fig. 2 shows the submodule of the Client substitution transition. The port places Request and Response are associated with the identically named socket places in Fig. 1. This means that any tokens added to or removed from these places by transitions in the ProposeValue module will be reflected in the top-level module. The submodule of the ProposeValue substitution transition models the behavior of sending a client request value for consensus to the Paxos replicas.

Paxos [11,22] is often described in terms of three separate agent roles: *proposers* that can propose values, *acceptors* that accept a value among those proposed, and *learners* that learn the chosen value. A Paxos replica may take on multiple roles: in a typical configuration (and also in the CPN model), all replicas play all roles. Paxos is safe for any number of crash failures, and can make progress with up to f crash failures, given $n = 2f + 1$ acceptors.

Fig. 3 shows the Replicas module which is the submodule of the substitution transition Replicas in Fig. 1. The module has a substitution transition for each Paxos agent connected by socket places to model the communication between the different agents. The detailed behaviors of the agents are then modeled in the submodule of the substitution transitions. The Replicas module has been constructed such that we can configure any number of replicas, each encapsulating the three Paxos agents, without modifying the net-structure. This allows us to easily generate test cases for differently sized Paxos configurations.

The Paxos protocol operates in *rounds*, which refer to a set of semantically related messages that may or may not conclude the consensus protocol. We say that the protocol solves consensus in some round. Due to asynchrony and failures,

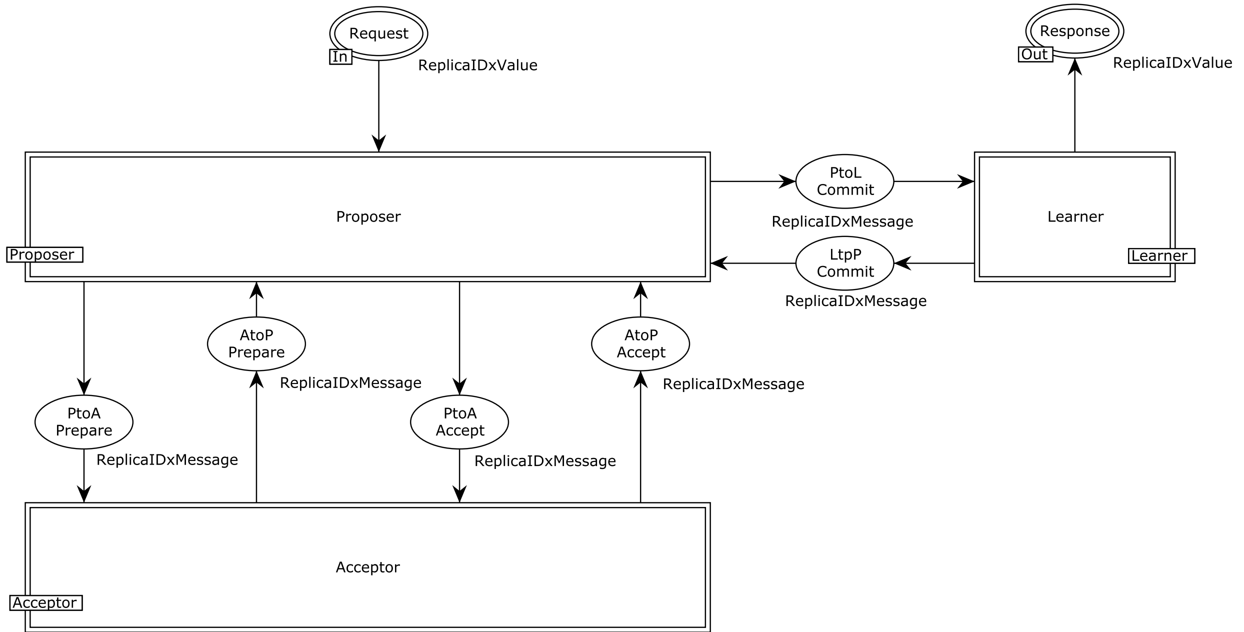


Fig. 3. The single-decree Replicas module.

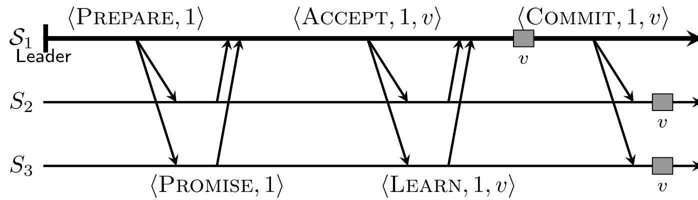


Fig. 4. The single-decree Paxos consensus protocol with three phases.

a consensus protocol may need to run several rounds to solve consensus. When describing the protocol, we use the variables *rnd*, *crnd*, and *vrnd* to denote round number, current round, and voted in round. Every round is associated with a single proposer, which is the *leader* for that round. Other proposers can start new (higher) rounds concurrently by sending a ⟨PREPARE⟩ message to acceptors, to collect ⟨PROMISE⟩s from the acceptors to follow a new proposer. This is essential for Paxos to make progress in case the current leader goes mute. Every round runs in three phases:

1. A proposer sends a ⟨PREPARE⟩ message to the acceptors and collects at least $f + 1$ ⟨PROMISE⟩ messages;
2. the proposer then sends ⟨ACCEPT⟩ messages for some value v to the acceptors, who respond by sending ⟨LEARN⟩ messages back to the proposer acknowledging the value v;
3. the proposer sends the decided value in ⟨COMMIT⟩ messages to learners.

The common case execution of the three phases is shown in Fig. 4. The first number in each message is the $rnd = 1$, and v is the value that the proposer wants the acceptors to choose. The gray boxes labeled v represent the execution of a state machine command derived from the decided value v. While not shown in the figure, each replica has instances of each of the Paxos agents. The communication between the different Paxos agents has been modeled based on the quorum abstractions provided by the Gorums framework [20], which we discuss in §4. Specifically, the communication takes the form of quorum calls, one for each of the Paxos phases: Prepare, Accept, and Commit.

The value v to choose is the value with the highest round among those provided in the ⟨PROMISE⟩ messages, or if no votes are provided in the ⟨PROMISE⟩ messages, any value can be chosen by the proposer; this would typically be a value that the proposer received from a client. In Paxos, acceptors are said to have *chosen* a value v, if a majority of acceptors have voted for v in the same round. Once a value has been chosen by acceptors in a round, no other value can be chosen in any other round. However, if there is no majority of acceptors that have voted for v, then the acceptors may vote for different values in other rounds. Since rounds execute concurrently, there is no guarantee of progress even if there are no failures or message loss. Therefore, Paxos typically relies on an eventual leader detection protocol, often implemented from

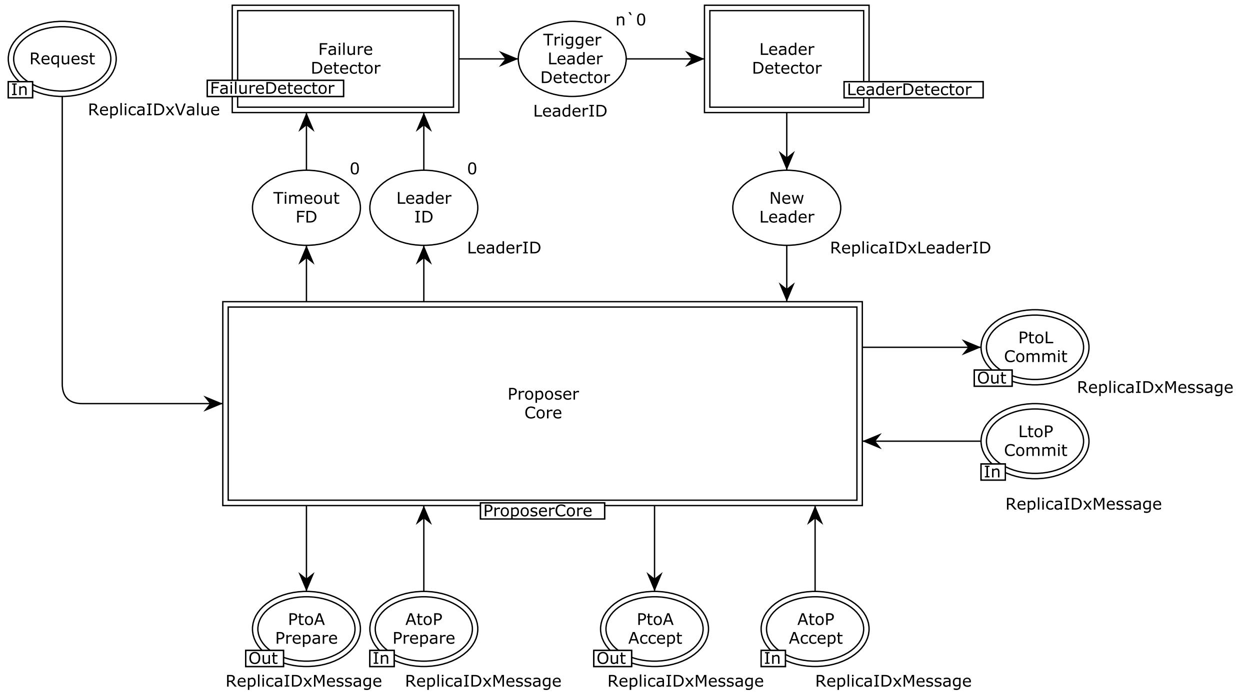


Fig. 5. The Proposer module.

the Ω failure detector [25]. While Ω may be inaccurate for some time, eventually it makes correct proposers agree on which proposer is the leader. Using Ω, and under the assumption that $n \geq 2f + 1$ acceptors, Paxos satisfies Properties L1 and L2.

3. Modeling the Paxos agents

This section presents the behavioral modeling of the Paxos agents. One of the Proposers is designated as a leader and proposes the client request value for consensus. The Acceptors choose the consensus value among those proposed, and the Learner of each replica learns the decided value. Once a value has been learned by a Learner, a response may be sent by this Learner to the client. This response is presented as a token on the port place Response.

3.1. Proposers

The submodule of the Proposer substitution transition is shown in Fig. 5. It contains three substitution transitions: LeaderDetector, FailureDetector, and ProposerCore. The Proposer of each replica receives the client request (presented as a token on the port place Request) for consensus, sent from the submodule of the Clients substitution transition.

In Paxos, one of the Proposers is responsible for driving the consensus process, namely the *leader*. However, due to the asynchronous nature of the environment in which we are operating, we may have that many Proposers believe they are the leader, thus the Paxos protocol can only guarantee progress if one of them is eventually chosen. Therefore, the objective of the first phase of Paxos is to obtain permissions from the Acceptors that a particular Proposer can serve as the leader. However, to be able to detect if a new proposer should initiate the first phase, we use the LeaderDetector substitution transition which has a submodule to pick a leader among the Proposers. This submodule is informed about failures from the failure detector. The FailureDetector substitution transition has a submodule that can detect the failure of any of the Proposers. Then, another leader can be selected by the submodule of the LeaderDetector substitution transition and it can take over the leadership by starting the first phase of Paxos with a higher round number than previous leaders.

Paxos uses round numbers to rank replicas, and each replica has a unique set of round numbers. More specifically, each round is assigned to a single proposer. The choice of the proposer for round i is determined by a deterministic mapping $p : B \to P$, where B is the set of round numbers and P is the set of proposers. In this paper, we assume that B is the set of natural numbers, and that proposers have assigned identities $0, 1, \ldots, |P| - 1$, where $|P| = n$. Then, we can choose mapping p such that $p(i) = i \pmod{|P|}$.

A client request presented as a token value on the port place Request will be sent to the ProposerCore, waiting to be handled by the leader as can be seen from Fig. 5.

The submodule of the ProposerCore substitution transition is shown in Fig. 6 and models the internal behavior of the Proposer. In this module, the InitProposer substitution transition has a submodule to initialize Proposers, obtain a new

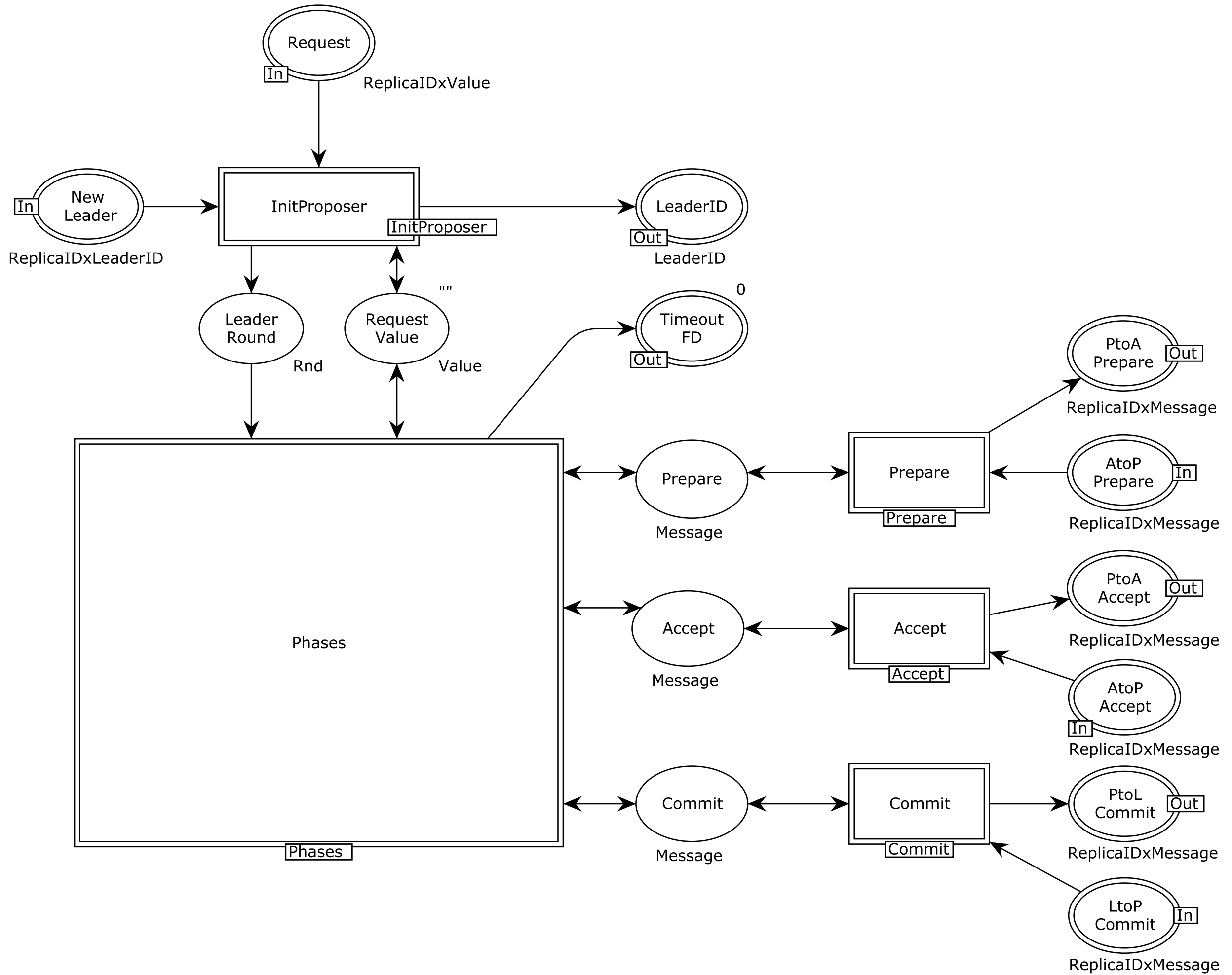


Fig. 6. The ProposerCore module.

leader, and receive a client request for consensus. Then, the value of the current round number of the leader and the value of the received client request will be presented on the port places as tokens, respectively. These tokens will be handled by the submodule of the Phases substitution transition.

A successful round of the single-decree Paxos protocol has three phases, modeled by submodules of the three substitution transitions shown in Fig. 7. The first phase involves two types of messages known as the ⟨Prepare⟩ and ⟨Promise⟩ messages. The leader candidate creates a ⟨Prepare⟩ message with its current round number and invokes a Prepare quorum call. This is modeled by the submodule of the Prepare substitution transition, shown in Fig. 6, which sends the ⟨Prepare⟩ message to Acceptors in order to propose itself to be a leader. After the Acceptors receive the ⟨Prepare⟩ message, and if they accepted it, then each Acceptor returns back a ⟨Promise⟩ message to the leader candidate by the Prepare quorum call. This is modeled by the submodule of the Acceptor substitution transition shown in Fig. 3. When the leader candidate receives enough ⟨Promise⟩ messages from Acceptors (obtain a quorum), then the first phase is finished, which means the leader candidate now can become a leader, and propose the client request to Acceptors for consensus.

In the second phase, the leader creates an ⟨Accept⟩ message with its current round number, *crnd*, and the value v obtained from the client request, and invokes the Accept quorum call, modeled by the submodule of the Accept substitution transition, shown in Fig. 6. This quorum call sends the ⟨Accept⟩ message to the Acceptors, requesting them to vote for consensus value v. Upon receiving an ⟨Accept⟩ message whose round number is greater or equal to the Acceptor's round number, the Acceptor will return a ⟨Learn⟩ message to the leader. Once the leader has received a quorum of ⟨Accept⟩ messages from Acceptors, the second phase is done. For the third phase, the leader invokes the Commit quorum call on the Learners, as modeled by the submodule of the Commit substitution transition, shown in Fig. 6. This enables the Learners to learn the chosen consensus value and can send it to the client.

Fig. 8 shows the submodule of the PhaseOne substitution transition. The leader uses its current round number to create a ⟨Prepare, 0, *crnd*⟩ message by triggering the transition SendPrepareMessage so that this message can be placed on the port

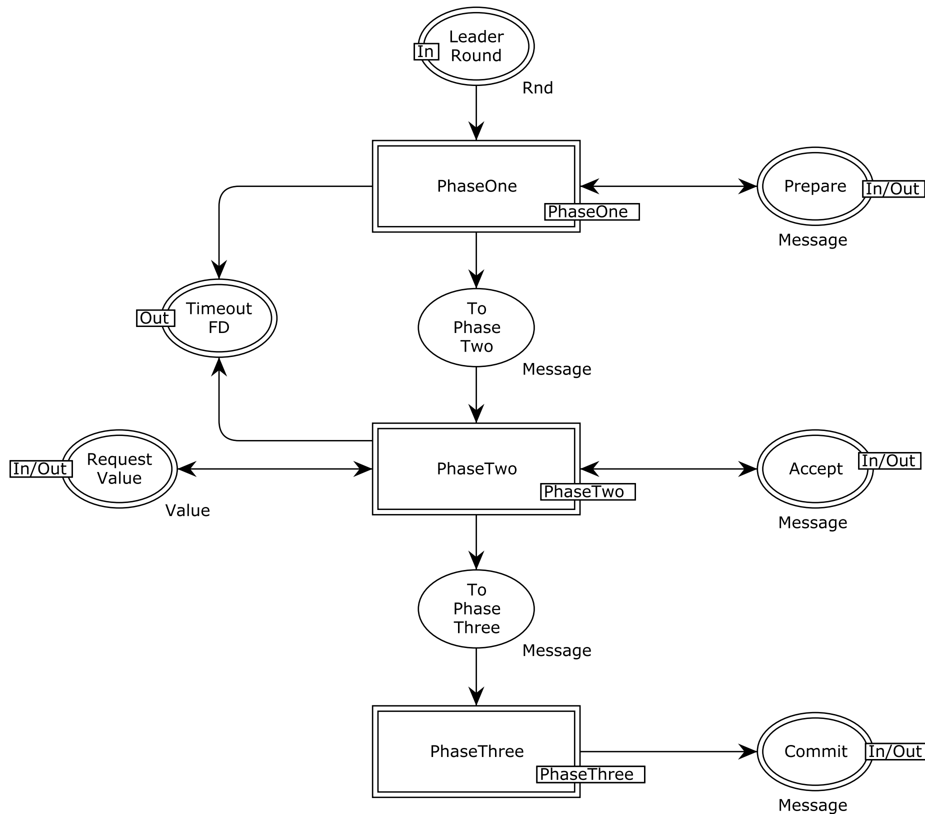


Fig. 7. The Phases module.

place Prepare as a token to invoke the Prepare quorum call. This quorum call returns a ⟨PROMISE, *cid*, *rnd*, (*vrnd*, *vvalue*)⟩ message as we already discussed, where *cid* is the call ID (initialized as 0 in ⟨PREPARE⟩ message); *rnd* is the round number confirmed by the Acceptor; *vrnd* is the most recent round in which the Acceptor voted, and *vvalue* is the consensus value it voted for. The place FDControl provides an upper bound on the number of timeouts/failures in our test cases. This place is not part of the CPN model for single-decree Paxos, but is used to control the test environment. If no timeout occurs, and the leader obtained a quorum of ⟨PROMISE⟩ messages, the second phase can start. The place FailedReplica is used to collect the identity of failed replicas, which we use in §6 for validation of the model. The second and third phases are modeled in a similar manner as the first phase, and we do not include them here.

After the ⟨PROMISE⟩ message returns, a timeout could happen to trigger the failure detector when the ProcessPromiseMessage transition occurs. This is used to capture scenarios where a failure of any replica occurs. Such failure is modeled as an event that may occur after a quorum has been obtained for the quorum call, which, in this case, is represented as a token of the ⟨PROMISE⟩ message appearing on the port place Prepare. At this stage, an occurrence of the ProcessPromiseMessage transition (Fig. 8) may result in a timeout modeled by the creation of a token on the port place TimeoutFD signaling that a failure has occurred. We may then have a finite sequence of transition occurrences for the accomplishment of the Prepare quorum call (in this case) and for finishing the remaining transitions in the submodule of the LeaderDetector substitution transition. After this, the transitions for leader detection in the submodule of the FailureDetector substitution transition will occur as they are given higher priority compared to other transitions in the model. This ensures that the execution of the failure detector cannot be forever postponed and the current leader ID (round number) for this failed round is obtained, which then causes the execution of the leader detector to elect a new leader. The fact that the failure detector will be executed in a finite number of steps from when a failure has occurred, restricts the behavior of the model and in turn implies that the model satisfies properties L1 (a proposed value is eventually chosen) and L2 (that correct replicas eventually learns the chosen value).

3.2. Acceptors

This section details the model for the acceptors. Fig. 9 shows the Acceptor module. The submodule consists of HandlePrepare and HandleAccept substitution transitions. The former handles ⟨PREPARE⟩ messages sent by the submodule of the Proposer substitution transition shown in Fig. 3 through the port place PtoAPrepare. The latter similarly handles ⟨ACCEPT⟩ messages also sent by the Proposer through port place PtoAAccept.

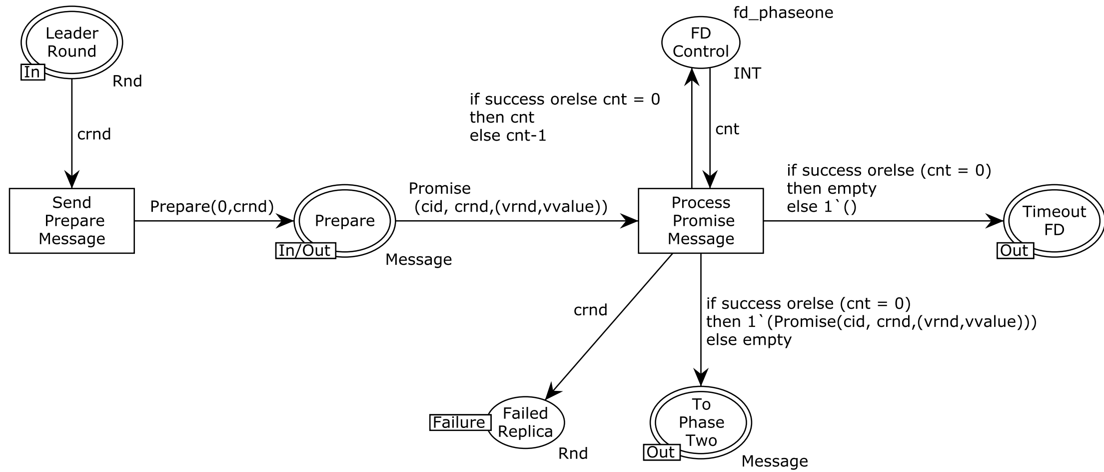


Fig. 8. The PhaseOne module.

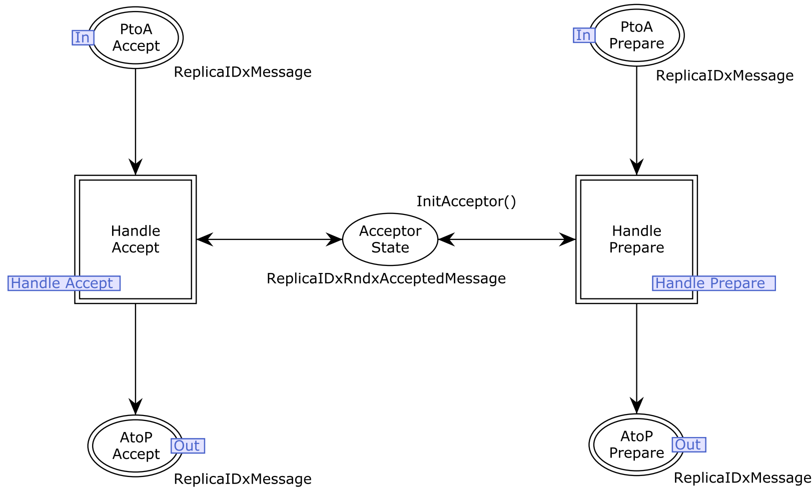


Fig. 9. The Acceptor module.

The AcceptorState place represents the state of each Acceptor. It is initialized with each replica's ID, round number $rnd = 0$, last voted round $vrnd = 0$, and voted value $vvalue = \varepsilon$ (empty string).

The submodule of the HandlePrepare substitution transition is shown in Fig. 10. The HandlePrepare transition handles ⟨Prepare⟩ messages sent by the Proposer. If the *crnd* of the ⟨Prepare⟩ message is higher than the Acceptor's *rnd*, then the token placed on the AcceptorState port place is updated accordingly, and a new token for the ⟨Promise⟩ message can be placed on the AtoPPrepare port place according to the expression of the arc connecting the HandlePrepare transition and AtoPPrepare place. We do not show the submodule of the HandleAccept substitution transition as it is similar to HandlePrepare. The main difference is that it updates the triplet $(rnd, vrnd, vvalue)$ in the AcceptorState port place, and places a ⟨Learn⟩ message on the AtoPAccept place.

3.3. Learners

Finally, we discuss the Learner substitution transition shown in Fig. 3, which has a submodule with a single HandleCommit substitution transition, as shown in Fig. 11. This submodule handles the ⟨Learn⟩ message sent by the Proposer, checking that a quorum of learn messages have been received before returning the decided consensus value to the client by placing a token on the Response port place. This behavior is modeled by the submodule of the HandleCommit substitution transition, shown in Fig. 12.

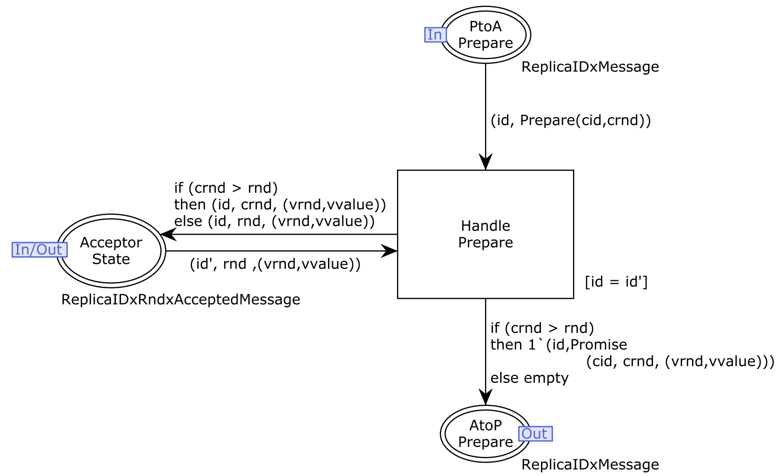


Fig. 10. The HandlePrepare module.

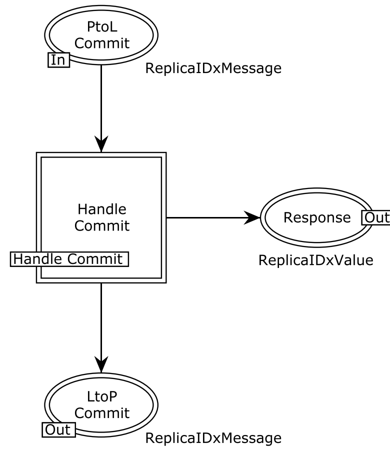


Fig. 11. The Learner module.

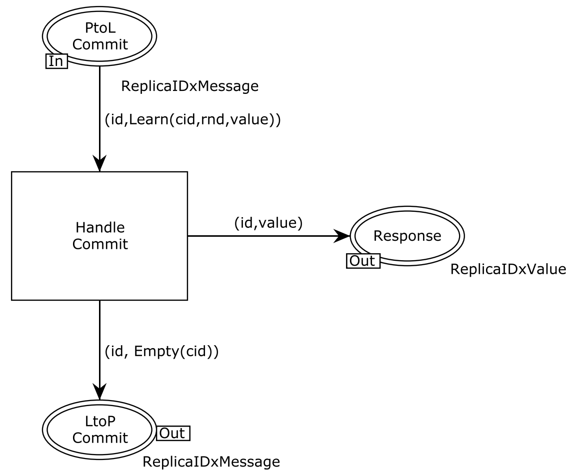


Fig. 12. The HandleCommit module.

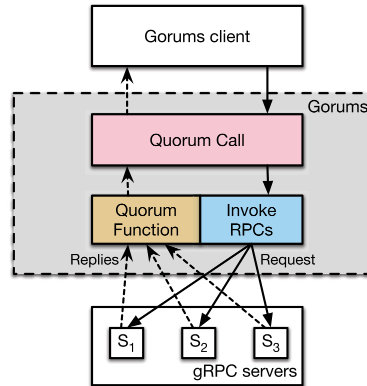


Fig. 13. Gorums abstractions.

4. Gorums and single-decree Paxos implementation

Gorums [20] is a framework for implementing quorum-based distributed systems. This section describes Gorums and how we use it to implement single-decree Paxos [11]. Our goal here is to provide a Paxos implementation that is amenable to testing based on the CPN model in §3, and in §5 we describe our testing approach.

4.1. Gorums abstractions

Gorums is a library whose goal is to alleviate the development effort for building advanced distributed algorithms, such as Paxos [11] and distributed storage [26,27]. These algorithms are commonly used to implement replicated services, and they rely on a quorum system [28] to achieve fault tolerance. In a quorum system, such as Paxos, protocol replicas need to exchange and update information about each other's state. However, to ensure consistency, a replica must contact a quorum, e.g. a majority of the replicas. In this way, a system can provide service despite the failure of individual replicas. However, communicating with and handling replies from sets of replicas often complicate the protocol implementations.

To reduce this complexity, Gorums provides three core abstractions: (a) configurations that group a set of replicas to hide the existence of individual replicas, (b) a flexible and simple quorum call abstraction, which is used to communicate with a configuration, i.e. a set of replicas, and to collect their responses, and (c) a quorum function abstraction which is used to process responses. These abstractions can help to simplify the main control flow of protocol implementations, as we illustrate later in this section.

Fig. 13 illustrates the interplay between the main abstractions provided by Gorums. Gorums consists of a runtime library and code generator that extends the gRPC [29] remote procedure call library from Google. Specifically, Gorums allows clients to invoke a quorum call, i.e. a set of RPCs, on a group of servers, and to collect their replies. The replies are processed by a quorum function to determine if a quorum has been received. Note that the quorum function is invoked every time a new reply is received at the client, to evaluate whether or not the received set of replies constitutes a quorum.

Protocol developers using Gorums can specify RPC service methods using `protobuf` [30], and from this specification, Gorums's code generator will produce code to facilitate quorum calls and collection of replies. Each quorum call method must provide a user-defined quorum function that Gorums will invoke to determine if a quorum has been received for that specific quorum call. In addition, the quorum function will also provide a single reply value, based on a coalescing of the received reply values from the different server replicas. This coalesced reply value is then returned to the client as the result of its quorum call. That is, the invoking client does not see the individual replies.

The quorum functions for a specific protocol implementation must follow a well-defined interface generated by Gorums. These only require a set of reply values as input and a return of a single reply value together with a boolean quorum decision. Hence, quorum functions can easily be tested using manually written unit tests. However, some quorum functions involve complex logic, and their input and output domains may be large, and so generating test cases from a model, provide significant benefit for verifying their correctness.

A quorum call is implemented by a set of RPCs, invoked at different servers, and so must consider different interleavings due to invocations by different clients. Hence, using model-based testing we can produce sequences of interleavings aimed at finding bugs in the server-side implementations of the RPC methods and also in the Gorums runtime system.

Fig. 14 gives the Prepare quorum call module of the Prepare substitution transition in Fig. 6. This module models the behavior of the quorum call and quorum function abstractions provided by Gorums for sending the ⟨PREPARE⟩ messages from a Proposer (leader) to Acceptors when the transition SendPrepareMessages occurs. Then, after such ⟨PREPARE⟩ messages are handled by Acceptors, the replied ⟨PROMISE⟩ messages from Acceptors can be processed when the transition ApplyPrepareQF occurs, which models the behavior of the Prepare quorum function. The logic to implement such quorum function will be discussed in §4.2.

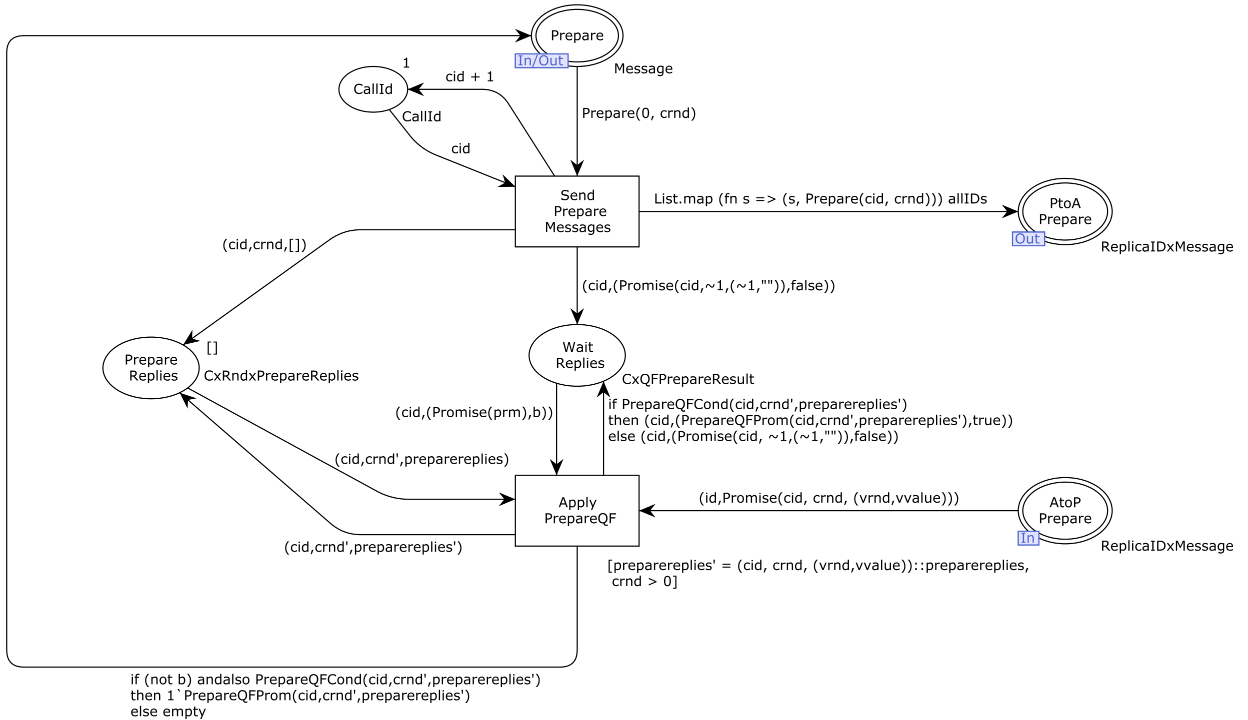


Fig. 14. The Prepare quorum call module.

```
type SinglePaxosServer interface {
    Prepare(context.Context, *PrepareMsg) (*PromiseMsg, error)
    Accept(context.Context, *AcceptMsg) (*LearnMsg, error)
    Commit(context.Context, *LearnMsg) (*Empty, error)
    ClientHandle(context.Context, *Value) (*Response, error)
    Ping(context.Context, *Heartbeat) (*Heartbeat, error)
}
```

Listing 1: The `SinglePaxosServer` interface that Paxos replicas must implement.

Our goal in this paper is to provide a framework for generating test cases to validate the correctness of the Gorums implementation itself, in addition to different quorum function and quorum call implementations for our Gorums-based Paxos implementation.

4.2. Implementing single-decree Paxos using Gorums

We have implemented the single-decree Paxos protocol as our system under test, using Gorums and the Go programming language. The system consists of $n = 2f + 1$ replicas that run the Paxos protocol, taking client requests as input, aimed at reaching consensus on a single output response. The implementation corresponds to the CPN model discussed in §2 and §3.

In our implementation, we first define a set of RPC service methods for Paxos using the interface description language (IDL) of protocol buffers [30]. This IDL is then supplied to the Gorums code generator, which generates the code necessary to invoke quorum calls for the methods defined in the IDL. Each of the Paxos replicas must implement the `SinglePaxosServer` interface shown in Listing 1, which is generated from the IDL. The methods Prepare(), Accept() and Commit() in this interface represent Paxos quorum calls that can be invoked by the different replicas in order to access and update each other's Paxos state.

In addition to the Paxos methods mentioned above, the `SinglePaxosServer` interface also contains ClientHandle() and Ping(). The former is a quorum call used by clients to communicate their proposed value to the Paxos replicas and receive the decided value. Recall that multiple clients can propose a value, possibly simultaneously, but only one of the proposed values will be decided, and returned to all clients. The Ping() is simply a regular RPC call used by the failure detector to determine if a replica has failed.

In the following, we explain the main control flow of the single-decree Paxos protocol, as shown in Listing 2; ignoring error handling and `ctx` initialization. On Line 3 of the `Proposer`, the Prepare() quorum call sends a ⟨Prepare⟩ message to the `Acceptors`, whom return ⟨Promise⟩ messages back to the `Proposer`. Once a quorum of promises has been received,

```
1  func (p *Proposer) runPaxosPhases() error {
2      preMsg := &PrepareMsg{Rnd: crnd}
3      prmMsg, err := p.config.Prepare(ctx, preMsg)
4      if prmMsg.GetVrnd() != Ignore {
5          p.cval = prmMsg.GetVval()
6      }
7      accMsg := &AcceptMsg{Rnd: crnd, Val: p.cval}
8      lrnMsg, err := p.config.Accept(ctx, accMsg)
9      ackMsg, err := p.config.Commit(ctx, lrnMsg)
10     return nil
11 }
```

Listing 2: `Proposer`'s code for Paxos phases, slightly simplified, and without error handling.

```
type QuorumSpec interface {
    PrepareQF(replies []*PromiseMsg) (*PromiseMsg, bool)
    AcceptQF(replies []*LearnMsg) (*LearnMsg, bool)
    CommitQF(replies []*Empty) (*Empty, bool)
    ClientHandleQF(replies []*Response) (*Response, bool)
}
```

Listing 3: The QuorumSpec interface must be implemented to process replies.

```
1  type PaxosQSpec struct {
2      quorum int
3  }
4
5  func (q PaxosQSpec) PrepareQF(replies []*PromiseMsg) (*PromiseMsg,bool) {
6      if len(replies) < q.quorum {
7          return nil, false
8      }
9      reply := &PromiseMsg{Rnd: replies[0].GetRnd()}
10     for _, r := range replies {
11         if r.GetVrnd() >= reply.GetVrnd() {
12             reply.Vrnd = r.GetVrnd()
13             reply.Vval = r.GetVval()
14         }
15     }
16     return reply, true
17 }
```

Listing 4: The `PrepareQF` processes ⟨Promise⟩ replies from replicas.

the Prepare() quorum call returns with a single combined ⟨Promise⟩ message. We explain later in this section, how we leverage Gorums's quorum function abstraction to determine whether or not a quorum has been received, and how to combine the replies into a single ⟨Promise⟩ message.

Next, the `Proposer` determines from the ⟨Promise⟩ if any of the `Acceptors` have voted in a previous round, *vrnd*. If so, the corresponding value from the ⟨Promise⟩ message (Line 5) that was voted for, must also be used by the `Proposer` when constructing its ⟨Accept⟩ message on Line 7. Otherwise, the `Proposer` uses the value *cval* that it received from a client.

At this stage the `Proposer` invokes the Accept() quorum call, asking the `Acceptors` to choose the value included in the ⟨Accept⟩ message. The `Acceptors` respond back with a ⟨Learn⟩ message, followed by the `Proposer` invoking the Commit() quorum call to propagate the decision to the `Learners`, which concludes the protocol.

We have implemented the `SinglePaxosServer` interface methods on an object of type `PaxosReplica`, encapsulating the state and behavior of the Paxos agents: `Proposer`, `Acceptor`, and `Learner`. The behavior of each agent corresponds to different CPN models in §3. Further, the `PaxosReplica` takes care of dispatching the method calls to their respective Paxos agents.

We now turn our attention to the handling of replies from quorum calls. For each quorum call defined in the IDL, Gorums adds a quorum function signature to an interface called `QuorumSpec`, as shown in Listing 3. This interface must be implemented by the protocol developer; Listing 4 shows the implementation of the `PrepareQF` quorum function. These methods are implemented on the `PaxosQSpec` type, which holds information about the quorum size (Line 2).

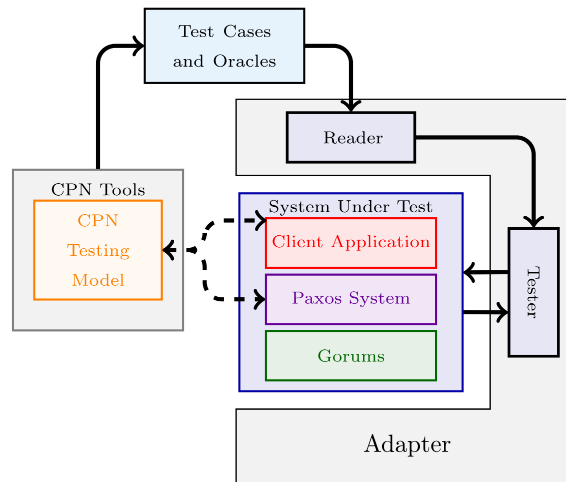


Fig. 15. Overview of the test framework.

`PrepareQF` is called by the Gorums runtime with the set of replies that have been received so far; it is called once for each reply. In the first part (Lines 6–8), we check if there are enough replies to return from the quorum call, or return `false` to signal to Gorums that we must wait for more replies.

If enough replies have been received, we continue to construct a combined ⟨PROMISE⟩ message by examining all the replies, and picking the value, *vval*, from the ⟨PROMISE⟩ message with the highest voted round, *vrnd*. If such a locked-in value is found in the replies, this means that the `Proposer` is constrained and must continue to use this value in the remainder of the protocol. Otherwise, the `Proposer` is unconstrained, and can pick its own client value.

Similar constructs are used for all the methods in the QuorumSpec interface, but we do not show them here. However, we note that one of the benefits of using Gorums's quorum functions is that they are amenable to unit testing.

5. Test case execution

To perform model-based testing of our Paxos implementation described in §4.2, we have implemented a client application, which together with the Paxos implementation and Gorums constitute the SUT. Fig. 15 gives an overview of our test framework, which consists of CPN Tools and a test adapter. Our test approach involves three steps: (a) use CPN Tools to construct a test model of our SUT; (b) perform simulation-based test case generation by using the MBT/CPN library to generate test cases with oracles represented in an XML format; (c) implement a test adapter to execute the generated test cases on the SUT, and compare the test results against generated oracles.

5.1. The test adapter

A central part of our test approach is the development of a test adapter which can execute the system and unit test cases generated from CPN Tools using our MBT/CPN library [10] (discussed in §6). The test adapter consists of a reader and a tester, both implemented in Go. The reader of the adapter can read test cases with oracles in the XML format generated from the CPN test model. The tester component has been implemented using the *testing* package from the Go standard library. Go's testing infrastructure comprises the `go test` command which allows us to simply run and execute our generated tests and obtain pass/fail information for each test case execution. Moreover, the Go testing infrastructure includes a tool which can be used to evaluate our approach by measuring the statement coverage for both unit and system tests.

5.2. Test case execution

We distinguish between unit and system tests for our SUT. The unit tests are used to test the central protocol logic used to implement the single-decree Paxos protocol, such as quorum functions discussed in §4. The system tests are used to test the complete Paxos implementation and Gorums library with clients. This separation provides a modular approach to testing. Additionally, under system tests, we consider failure scenarios for the Paxos replicas when in different Paxos phases, cf. Fig. 8.

5.2.1. Unit tests

The test adapter implements a Go-based tester that can execute the unit tests obtained from the reader. The tester invokes the methods to be tested with the supplied input values, and upon completion compares the results against the

```xml
<Test Name="TestPrepareQF">
  <TestCase ID="1">
    <TestValues>
      <PromiseMsg>
        <Rnd>0</Rnd>
        <Vrnd>0</Vrnd>
        <Vval></Vval>
      </PromiseMsg>
      <PromiseMsg>
        <Rnd>0</Rnd>
        <Vrnd>0</Vrnd>
        <Vval></Vval>
      </PromiseMsg>
    </TestValues>
    <TestOracles>
      <Quorum>true</Quorum>
      <PromiseMsg>
        <Rnd>0</Rnd>
        <Vrnd>0</Vrnd>
        <Vval></Vval>
      </PromiseMsg>
    </TestOracles>
  </TestCase>
</Test>
```

Fig. 16. XML format for PrepareQF().

test oracle's expected output, also obtained from the reader. The unit tests can be performed without running the Paxos protocol and clients. The methods we consider for unit tests include PrepareQF() and AcceptQF(), discussed in §3 and §4.2. Fig. 16 shows an excerpt from the XML representation of a test case for PrepareQF(), which corresponds to a test case where Paxos is configured with three replicas and the quorum size is two. The test input for the PrepareQF() method in the test case is two ⟨Promise⟩ messages with values for the fields Rnd, $Vrnd$ and $Vval$. The expected output of the PrepareQF() is a ⟨Promise⟩ message together with the Quorum boolean *true*, indicating that a quorum was obtained for these input messages.

5.2.2. System tests

Execution of the system tests requires that the Paxos replicas are running and ready to handle the requests from clients so that we can test the complete system including the Gorums library.

Therefore, for system tests, the tester first starts the Paxos replicas and then iterates through the test cases obtained from the reader. For each test case, the tester starts clients in order to send client requests to the Paxos replicas. Each client has a single request value to send for consensus. As an example, the test adapter can execute two clients concurrently to send their requests to the Paxos replicas. After the Paxos replicas have decided, a response value is sent back to the clients. The tester checks whether the response for each client belongs to the expected responses (oracles) and whether the responses are the same for all clients, i.e., the consensus was reached.

In addition to testing success scenarios, we also test scenarios with different types of failures. This includes forcing the failure detector to timeout, triggering a new leader to be promoted. In this way, we can test leader changes and fault tolerance of the Paxos protocol. To make the implementation amenable for such failure scenarios during system test execution, our test adapter must be able to observe the messages exchanged between the replicas, and to interfere, for example, in a test case where we simulate a lost message or trigger a timeout.

We have considered three major harnessing approaches below for how we can effectively test a particular scenario, and we motivate our final choice. Further details on the first two approaches and their pros and cons can be found in [31].

In the first approach, one would isolate the involved (Unix) processes in individual, networked, containers or virtual machines, and if necessary interfere with the environment by, e.g., introducing network partitions. This is a heavy-weight approach, where a lot of implementation effort will have to be spent on manipulating the environment based on a test case description. Also, the test case adapter coordinating the environment needs an understanding of the messages to be exchanged between replicas, so that it can decide that a particular setup has now been reached and it should interfere.

The second approach is more light-weight in that the Paxos replicas would connect to the test adapter instead of directly to each other. The test adapter can then observe the protocol and either relay message, or introduce faults [32]. This approach can reuse marshaling logic in the test adapter, which makes analyzing the message content easier than in the virtual machine approach above.

Our approach is even more light-weight in that we do not use an external test adapter. Instead, to track the state of a replica, we compile an instrumented version of the server that contains several points for test case interaction. By using this approach, we can use test cases to describe not only the successful scenarios, but also different failure scenarios and guide the test case execution. As an example, consider a Paxos configuration with three replicas. A test case may contain events that cause the leader to fail during either the first or the second phase of the Paxos protocol. After such a failure, a new leader will eventually emerge, restarting the Paxos phases. In a configuration with five replicas, a test case can, e.g., be

```xml
<Test Name="systemtest">
  <TestCase ID="1">
    <TestValues>
      <ClientPropose>M1</ClientPropose>
      <ClientPropose>M2</ClientPropose>
      <P1Failure>1</P1Failure>
    </TestValues>
    <TestOracles>
      <Leader>0</Leader>
      <Leader>1</Leader>
      <Response>M1</Response>
      <Response>M2</Response>
    </TestOracles>
  </TestCase>
</Test>
```

Fig. 17. XML format for testing a three-way replicated Paxos system.

configured to let the first leader fail in the first phase, and after the second leader emerges and the Paxos phases restart, the second leader can be made to fail in the second phase. Finally, the third leader can restart and complete the Paxos phases successfully.

To enable the test adapter to know when it is possible to inject a failure, we have instrumented the `Proposer` with an `AdapterConnector` to communicate the `Proposer`'s state, such as the current leader and which Paxos phases have completed, to the test adapter. Moreover, between each state change, the `Proposer` will wait for a decision from the test adapter to determine if the current Paxos phase should fail, e.g. triggering a leader failure. The decisions made by the test adapter regarding failures of the Paxos phases are configured for each test case in the XML file. Fig. 17 shows an example of a test case for the Paxos protocol with three replicas, where there is a failure in the first Paxos phase. The test input for this example consists of two clients sending requests concurrently to the Paxos replicas. The test oracles include the legal responses from Paxos replicas, and the expected leaders. Leader 0 is the first leader, and after it fails, leader 1 becomes the new leader. The test adapter checks whether the correct leaders are chosen, and whether the response returned to each client belongs to the set of legal responses. Furthermore, it also tests whether the responses obtained by all clients are equal, so that we can determine if they have reached consensus.

6. Model validation and test case generation

For the test case generation we rely on the MBT/CPN library [10], which we have developed as an extension to CPN Tools. The MBT/CPN library is based on extracting test cases from execution sequences of the CPN model by partially observing occurring events. MBT/CPN supports both state space and simulation-based test case generation. State space-based test case generation works for finite-state models and is based on computing all reachable state and state changes of the CPN model. Simulation-based test case generation is based on running a set of simulations and extracting test cases from the corresponding set of executions.

The CPN test model for the Paxos protocol has an infinite state space and also for restricted and representative configurations with a finite state space, state-based test case generation is infeasible due to the state explosion problem. We therefore only use state space for validating the CPN model for small configurations (see §6.1) in order to gain confidence in the correctness of the test generation CPN model. For the test case generation itself, we rely on simulation-based test case generation due to the high complexity of the Paxos protocol.

6.1. Model validation

A distinct advantage of relying on formal models such as CPNs for test case generation is that restricted configurations of the test case generation model with a finite state space can be verified using model checking prior to test case generation. This can be used to increase confidence in the correctness of the test case generation model and the generated test cases. To obtain configurations of the Paxos CPN test generation model with a finite state space, we have bounded the behavior of the Paxos agent roles such that only a finite number of messages can be generated in the system.

Specifically, we consider configurations of our CPN model with two clients, where each client can send one client request message (modeled as a string) into the Paxos system. These two request messages can be sent in any order, and the Paxos system then makes a decision on which client request message should be chosen and handled. The model terminates when both clients have received the decision response from the Paxos system. For the Paxos agent roles, we have limited the number of messages when executing the Paxos phases by configuring an upper bound of one on the number of timeouts/failures. This is done by means of place FDControl discussed in §3.1 and shown in Fig. 8. The most complex scenario currently covered is where the first Paxos phase fails once, then the Paxos system restarts the first phase; but this time the second phase fails once and the Paxos system restarts again, and then Paxos completes successfully for both phases. This scenario involves the `Proposer` (leader) sending the ⟨Prepare⟩ message three times, the ⟨Accept⟩ message two times, and the ⟨Commit⟩ message once.

In other words, we explicitly model failure scenarios where messages timeout or get lost in particular phases of the protocol, and combinations thereof. After the associated restarts of the Paxos protocol in the presence of these failures, the model lets the run complete successfully without further errors.

We have used the model checker and ASK-CTL library available in CPN Tools to verify that the CPN model (in the restricted configurations) satisfies the correctness properties S1–S3 and L1–L2 as formulated in §2. The ASK-CTL library makes it possible to specify temporal properties in a state and event-oriented variant of the computation tree logic (CTL). Below we show how the behavioral properties can be specified in CTL relative to the developed test case generation CPN model. We use $M(p)$ to denote the marking (multi-set of tokens) on a place p in the marking (state) M. For a token (value) t, we use $t \in M(p)$ to denote that t is a token on place p in the marking M.

S1 Only a proposed value may be chosen: To check this property we consider the place ServerResponse in Fig. 2. Proposed values are represented as tokens on place ClientRequest in the initial marking (state), and the chosen consensus value will appear as a token on place ServerResponse. The property can therefore be formulated in CTL as:

$$\mathbf{AG}\,(t \in M(\text{ServerResponse}) \Rightarrow t \in M_0(\text{ClientRequest}))$$

S2 Only a single value is chosen: As any chosen value will appear as a token on place ServerResponse we can verify this property by checking that there is at most one token on this place in any reachable state. In CTL this can be formulated as:

$$\mathbf{AG}\,(|M(\text{ServerResponse})| \leq 1)$$

S3 Only a chosen value may be learned by a correct replica: We consider the tokens on AcceptorState (Fig. 9) of the form $(r, rnd, vrnd, v)$ where the first component specifies the replica and the last component specifies the chosen value. The value learned by each replica will appear as tokens on place Response (Fig. 3), where the first component specifies the replica and the second component specifies the learned value. To account only for correct replicas, we consider the fusion place FailedReplica (Fig. 8) and restrict the property to replicas not present on this place. The property can therefore be checked using the following CTL formula where R denotes the set of replicas:

$$\mathbf{AG}(\forall r \in R \setminus M(\text{FailedReplica}) :$$
$$(r, v) \in M(\text{Response}) \Rightarrow \exists (r', rnd, vrnd, v) \in M(\text{AcceptorState}))$$

L1 Some proposed value is eventually chosen: For this property we can check that eventually a token will be put on place ServerResponse. In CTL this can be formulated as:

$$\mathbf{AG}\;\mathbf{AF}\,(M(\text{ServerResponse}) \neq \emptyset)$$

L2 Once a value is chosen, correct replicas eventually learn it: We consider the place AcceptorState holding any chosen value, and check that this value is eventually learned by non-failing replicas by considering the place Response in Fig. 3. In CTL this property can be formulated as:

$$\mathbf{AG}\,(\exists (r, rnd, vrnd, v) \in M(\text{AcceptorState}) \Rightarrow$$
$$\mathbf{AF}(\forall r \in R \setminus M(\text{FailedReplica}) : (r, v) \in M(\text{Response}))$$

We have executed the above queries against the test case generation CPN model configured with two replicas which yields a relatively small state space with less than 2000 states. In the process of checking these properties we found a number of minor modeling errors that we were then able to correct. In particular, we use the support in CPN Tools to obtain error traces (in case a property was violated) which helped in identifying the source of the problem. Even if the Paxos model is too complex to conduct model checking for larger configurations (due to the state space size), being able to verify the model for smaller configurations increases the confidence in the correctness of our test case generation for larger configurations of the Paxos protocol.

6.2. Test case specification

Test case generation from the CPN model requires identification of *observable events* originating from occurrences of transitions. A test case is comprised of observable events, where the input events represent stimuli to the system and the output events represent the expected outputs used as test oracles to determine the pass/failure of a test case. The formal foundation used to check whether the execution of the SUT conforms to the specification as provided by the test case is hence based on trace equivalence.

The generation of test cases with MBT/CPN requires an implementation of a *test case specification* defined by the Standard ML signature (interface) shown in Listing 5.

```
signature TCSPEC = sig
  val detection   : Bind.Elem -> bool;
  val observation : Bind.Elem -> TCEvent list;
  val format      : TCEvent    -> string
end;
```

Listing 5: Signature for test case specification.

```
fun detection (Bind.DecidedValue'Request _) = true
  | detection (Bind.DecidedValue'Apply_RequestQF _) = true
  | detection (Bind.PhaseOne'Process_PromiseMsg _) = true
  | detection (Bind.PhaseTwo'Process_LearnMsg _) = true
  | detection _ = false;

exception obsExn;
fun observation (Bind.DecidedValue'Request (_,b)) = [InEvent (SYS_Propose (#value b)]
  | observation (Bind.DecidedValue'Apply_RequestQF (_,b) = [OutEvent (SYS_Decide (#value b))]
  | observation (Bind.PhaseOne'Process_PromiseMsg (_,b) = [InEvent (SYS_P1Failure (#crnd b))]
  | observation (Bind.PhaseTwo'Process_LearnMsg (_,b) = [InEvent (SYS_P2Failure (#rnd b)))]
  | observation _ = raise obsExn;
```

Listing 6: Implementation of test case specification for system level tests.

The type `Bind.Elem` is an existing data type in CPN Tools representing binding elements, i.e., a transition and an assignment of values to the variables of the transition. The type `TCEvent` is the type defined for observable events. The *detection function* is a predicate on binding elements that evaluates to true for binding elements representing observable events. The purpose of the *observation function* is to map an observable binding element into an observable input or output event belonging to the `TCEvent` type. The observation function may return a list of observable events in case one might want to split a binding element into several observable events in the test case. Finally, the *formatting function* maps observable events into a string representation which is used in order to export the test cases into files.

For the Paxos protocol we generated both system test and unit tests. The system level test is concerned with the proposed values, chosen value, selected leaders, and failure of replicas. The unit test are concerned with testing the quorum functions, which forms the core of the Gorums-based implementation. Listing 6 shows a slightly simplified implementation of the detection and observation function for system level tests. We omit the formatting function as the XML format for test cases is already described in §5.2.

The first two binding elements for which the detection function returns true correspond to events representing the proposal and choice of a value. The two next binding elements correspond to events representing replica failures. The observation function then generates the observable events, which can be either an `InEvent` representing stimuli to the system or an `OutEvent` representing expected outputs. The implementation of the test case specification for unit tests covers the prepare, accept, and commit quorum functions and the implementation is similar to the system test case specification.

6.3. Experimental results on statement coverage

We have used statement coverage to evaluate the quality of our test case generation. Several other metrics exist to assess test coverage, but currently only statement coverage is supported by the Go tool chain. Table 1 summarizes the experimental results obtained using simulation-based test case generation for the Paxos protocol. We have considered Paxos configurations with 3 and 5 replicas and generated 1, 2, 5 and 10 simulation runs of the CPN model. As we did not see any increase in the number of test cases by going from 5 to 10 simulations, we did not increase the number of simulation runs further. The table shows the coverage obtained for the different subsystems of our Paxos implementation. Note that the Unit tests are for the quorum functions and hence not applicable for the other subsystems. The two numbers written below System tests and Unit tests gives the total number of test cases generated for 3 and 5 replica configurations, respectively. The test case generation for each configuration considered took less than 10 seconds, and the execution of each test case took less than one minute.

The results show that, for the configuration with both 3 and 5 replicas, the statement coverage of unit tests for Prepare and Accept quorum functions are up to 90% and 85.7%, respectively. For the system tests, the statement coverage for Prepare, Accept and Commit quorum calls reaches 83.9%, respectively; the results of statement coverage for Prepare and Accept quorum functions are up to 100%; for the Paxos implementation (Paxos core in the table), the Proposer module's statement coverage reaches 97.4%; the statement coverage of the Acceptor module is up to 100%; the statement coverages of the Failure Detector and Leader Detector modules reach 75.0% and 91.4%, respectively; the statement coverage of the Paxos replica module (discussed in §4.2) reaches 91.4%; for the Gorums library as a whole, the highest statement coverage reaches 51.8%. The results and test cases considered above validate that the implementation of the single-decree Paxos system and the Gorums framework work in both correct scenarios and scenarios involving failures of replicas. The reason for the lower coverage results of the Gorums library is that Gorums contains code generated by Gorums's code generator, and among them, various auxiliary functions and error handling code that are not used by our current implementation. The total number of lines of code for the SUT is approximately 3890 lines, which include generated code by Gorums's code

Table 1
Experimental results for test case generation and execution.

Subsystem	Component	System tests	Unit tests
		Test cases for 3/5 replicas	
		15/38	74/424
		Coverage	
Gorums library		51.8%	–
Paxos core	Proposer	97.4%	–
	Acceptor	100.0%	–
	Failure Detector	75.0%	–
	Leader Detector	91.4%	–
	Replica	91.4%	–
Quorum calls	Prepare	83.9%	–
	Accept	83.9%	–
	Commit	83.9%	–
Quorum functions	Prepare	100.0%	90.0%
	Accept	100.0%	85.7%

generator (around 3150 lines), the code for Paxos replica (around 110 lines), the client code (around 80 lines), the Proposer code (around 170 lines), the Acceptor code (around 40 lines), the code for failure detector (around 170 lines), the code for leader detector (around 100 lines), and the code for quorum functions (around 70 lines).

As part of analyzing the test results and executing generated test cases, we have discovered bugs in the implementation of the Paxos protocol, which are not captured by using manually written table-driven tests in Go. We have found bugs related to: the leader detector elects a wrong leader; only the leader's failure detector is executed; the elected new leader obtains a wrong round number; clients cannot receive responses from the Paxos replicas; the Paxos system can only handle one request from one client; and after the current leader fails, the failed leader executes the Paxos phases again. This shows how our MBT approach is able to detect non-trivial programming errors in complex distributed systems protocols.

7. Related work

Chubby [33] was one of the first implementations of Paxos that were deployed in a production environment, and thus were extensively tested. The authors highlight that at the time (2007), it was unrealistic to prove correct a real system of that size. Thus to achieve robustness, they adopted meticulous software engineering practices, and tested their system thoroughly. One of their testing strategies was to test their implementation when subjected to a random sequence of network outages, message delays, timeouts, process crashes and recoveries, schedule interleavings, and so on. Using our CPN model and our generated tests, we aim to test many of the same attributes in a more systematic manner.

Modbat is an MBT tool implemented in Scala and hence compatible with Java bytecode-based applications [34]. Models are specified as annotated, non-deterministic extended finite state machines. Modbat explores the transition system and executes the calls specified on the transitions. It has been used successfully in a similar setting as ours on the ZooKeeper distributed coordination service. It explores different possible interleavings and non-deterministic outcomes due to scheduling decisions or network communication in the real system which are judged by an oracle essentially implementing a model checking component. Unlike our CPN models, the specifications are not for consumption by other tools such as model checkers, nor is there an interactive component that allows exploring a particular execution of the model. As in our approach, it requires some manual effort connecting the engine to the SUT.

A testing approach for true concurrency using I/O Petri nets has been discussed by Ponce de León et al. [35]. The authors define a concurrent conformance relation for input–output labeled transitions systems, IOLTS. They present a test case selection algorithm using criteria such as all paths of length n, or traversing each basic behavior a certain number of times. Since test case selection is also a challenge in our setting, it remains an open question how their unfoldings would work in our CPN setting.

MBT has been used with success (as measured through productivity gain) in Microsoft's Protocol Documentation Quality Assurance Process. Grieskamp et al. [31] used Spec Explorer on protocols, where a so-called model program describes the test case, including how to check an observation against a possibly non-deterministic outcome. The main difference to our work is that their model programs are rule-based, and as such only get a visual representation as a graph through state space exploration. Our CPN models give developers a better overview as they directly link client- and server interactions. Spec Explorer uses the coordination language Cord for slicing models into tractable subsets of test cases that may impact coverage and completeness, but not correctness. We have not yet tackled the issue of test case selection, relying on user interaction through simulation when state space exploration becomes infeasible.

A CPN-based test generation approach is proposed by Liu et al. [6]. This approach requires defining a conformance testing-oriented CPN (CT-CPN) model and a PN-ioco relation which specifies how an implementation conforms to its specifications. Furthermore, this approach uses simulation-based test case generation algorithm for the CT-CPN model. In our

approach, on the other hand, test cases can be generated directly by using a simulation-based approaches for an existing implementation of the SUT. In addition, Wu, Schnieder, and Krause [7] use a model-based test generation technique based on CPNs to verify a module of a satellite-based train control system. They use CPN Tools to generate the reachability graph of the test model and then use state space analysis with CPN Tools to extract the expected output of each test case from the path of the graph. However, their approach does not support simulation-based test case generation, which is of essential for scalability. Zheng et al. [8] provide a technique for test cases and sequences generation. In their method, two algorithms are used to generate test cases and sequences from a CPN model of the SUT. The CPN model is first used as input to their APCO algorithm to generate an initial set of test cases. These test cases can then be converted to test sequences by using their algorithm. After that, the set of original test cases and test sequences can be exported as XML formatted files. They have applied their technique to a radio module in a centralized railway control system. In contrast to our approach, Zheng et al. do not consider testing any failure scenarios of the system, do not handle any concurrent execution of the system, and their approach has not been used to validate any distributed systems.

Formal verification of protocols for distributed systems tackles protocols on a more abstract level, and is interested in finding flaws and inconsistencies primarily in the specification. Such approaches are not necessarily interested in a correct implementation, and only rarely can executable code directly or automatically be derived from the specification. Formal verification of such complex systems often suffers from undecidability issues that require careful management of any automation (see [36]), or substantial effort to encode the system in a decidable fragment (see Padon et al. [37] for their encoding of Paxos and Multi-Paxos in EPR, the effectively-propositional fragment of first-order logic). We see our approach of testing a concrete implementation as orthogonal to approaches that aim to validate the correctness of a protocol in general: frequently, the final, often manual, step of actually programming a proven-as-correct algorithm introduces mistakes, and also generated code may suffer from problems or assumptions about the underlying infrastructure (see e.g. Fonseca's analysis of IronFleet among others [38]).

8. Conclusions and future work

The main contribution of our work is an MBT approach for advanced distributed systems protocols based on formal modeling. As we have illustrated on the Paxos protocol, application of our approach includes constructing a CPN testing model for the system under test, executing simulation-based test case generation algorithms, and applying a test case execution framework which combines test cases obtained from CPN Tools and a test adapter. Our experiments with this testing approach on a single-decree Paxos protocol implemented by the Gorums framework have demonstrated good code coverage and considered both unit and system tests. Furthermore, for the system tests, we have considered not only tests representing successful, non-faulty executions of the Paxos protocol, but also tests in which replicas may fail during the protocol's execution, and show that the implementation can handle these failure scenarios. We have shown that our approach detected errors and bugs in the Paxos implementation.

An attribute of our testing approach is that the constructed CPN testing model can also help us to obtain a better understanding of a complex protocol to be implemented. Furthermore, our Paxos CPN testing model can also serve as a basis for MBT of multi-decree Paxos and other fault-tolerant distributed systems implemented with the abstractions of the Gorums framework. For example, given a distributed system implemented by the Gorums framework, it is only the implementation of the quorum functions that needs to be changed when modeling the behaviors of quorum calls and quorum functions.

Another attribute is that we have used simulation-based test case generation for the Paxos system with differently sized configurations, e.g. with three or five replicas. Another contribution worth mentioning is our implementation of single-decree Paxos using Gorums. It is well-known that the Paxos protocol is difficult to understand and implement correctly. However, by leveraging the Gorums framework and its abstractions, our single-decree Paxos implementation is simpler and hence more reliable than it would be without Gorums. Additionally, we expect that it will be relatively easy to extend our implementation to multi-decree Paxos.

Our work opens up several paths for future work. For MBT of both successful scenarios of our Paxos protocol and scenarios involving failure injections of replicas, we have obtained good statement coverage results for unit and system tests. However, we need to consider more of Gorums's code paths so that we can increase the results of the coverage for the Gorums library itself. In order to do this, we need to test the Paxos protocol under additional failures scenarios and adverse conditions, such as network errors and partitions. This will require extensions to the current CPN testing model and XML format to describe and configure such failure scenarios so that we can use the generated test cases to guide the test case execution based on different failures scenarios. This also requires an extension to the test adapter such that it can execute the Paxos system under test with additional configurations in test cases to handle the failure scenarios. In addition to the single-decree Paxos, we also plan to evaluate our testing approach on additional complex protocols in order to evaluate the generality of our testing approach. In the short term, we are extending our current CPN testing model and Go implementation to a multi-decree Paxos protocol, and then perform MBT for such a complex Paxos system.

References

[1] Jepsen, Distributed systems safety analysis, http://jepsen.io.

[2] M. Utting, B. Legeard, Practical Model-Based Testing: A Tools Approach, Elsevier, 2010.
[3] K. Jensen, L.M. Kristensen, Coloured Petri Nets: a graphical language for modelling and validation of concurrent systems, Commun. ACM 58 (6) (2015) 61–70.
[4] CPN Tools, CPN Tools homepage, http://www.cpntools.org, 2017.
[5] L.M. Kristensen, K.I.F. Simonsen, Applications of Coloured Petri Nets for Functional Validation of Protocol Designs, Springer, 2013, pp. 56–115.
[6] J. Liu, X. Ye, J. Li, Colored Petri Nets model based conformance test generation, in: IEEE Symp. on Computers and Communications, ISCC, IEEE, 2011, pp. 967–970.
[7] D. Wu, E. Schnieder, J. Krause, Model-based test generation techniques verifying the on-board module of a satellite-based train control system model, in: 2013 IEEE Intl. Conf. on Intelligent Rail Transportation Proceedings, 2013, pp. 274–279.
[8] W. Zheng, C. Liang, R. Wang, W. Kong, Automated test approach based on all paths covered optimal algorithm and sequence priority selected algorithm, IEEE Trans. Intell. Transp. Syst. 15 (6) (2014) 2551–2560, https://doi.org/10.1109/TITS.2014.2320552.
[9] L.M. Kristensen, V. Veiset, Transforming CPN models into code for TinyOS: a case study of the RPL protocol, in: Proc. of ICATPN'16, in: Lecture Notes in Computer Science, vol. 9698, Springer, 2016, pp. 135–154.
[10] MBT/CPN repository, https://github.com/selabhvl/mbtcpn, Aug 2018.
[11] L. Lamport, The part-time parliament, ACM Trans. Comput. Syst. 16 (2) (1998) 133–169.
[12] L. Lamport, Fast Paxos, Distrib. Comput. 19 (2) (2006) 79–103, https://doi.org/10.1007/s00446-006-0005-x.
[13] L. Lamport, D. Malkhi, L. Zhou, Vertical Paxos and primary-backup replication, in: Proceedings of the 28th ACM Symp. on Principles of Distributed Computing, PODC '09, ACM, Calgary, AB, Canada, 2009, pp. 312–313.
[14] I. Moraru, D.G. Andersen, M. Kaminsky, There is more consensus in egalitarian parliaments, in: ACM SIGOPS 24th Symp. on Operating Systems Principles, SOSP '13, 2013.
[15] H. Meling, K. Marzullo, A. Mei, When you don't trust clients: Byzantine proposer fast Paxos, in: 32nd IEEE International Conference on Distributed Computing Systems, ICDCS, IEEE, 2012, pp. 193–202.
[16] M. Burrows, The Chubby lock service for loosely-coupled distributed systems, in: Proc. of the 7th Symp. on Operating Systems Design and Implementation, OSDI '06, USENIX Association, 2006, pp. 335–350.
[17] D.F. Bacon, N. Bales, N. Bruno, B.F. Cooper, A. Dickinson, A. Fikes, C. Fraser, A. Gubarev, M. Joshi, E. Kogan, A. Lloyd, S. Melnik, R. Rao, D. Shue, C. Taylor, M. van der Holst, D. Woodford, Spanner: becoming a SQL system, in: Proc. of the 2017 ACM Intl. Conf. on Management of Data, SIGMOD '17, ACM, Chicago, Illinois, USA, 2017, pp. 331–343.
[18] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, M. Deardeuff, How Amazon web services uses formal methods, Commun. ACM 58 (4) (2015) 66–73, https://doi.org/10.1145/2699417.
[19] H. Meling, L. Jehl, Tutorial summary: Paxos explained from scratch, in: R. Baldoni, N. Nisse, M. van Steen (Eds.), 17th International Conference on Principles of Distributed Systems, OPODIS, in: Lecture Notes in Computer Science, vol. 8304, Springer, 2013, pp. 1–10.
[20] T.E. Lea, L. Jehl, H. Meling, Towards new abstractions for implementing quorum-based systems, in: Proc. of 37th IEEE Intl. Conf. on Distributed Computing Systems, ICDCS, 2017, pp. 2380–2385.
[21] CPN testing model of the single-decree Paxos, http://dkan.isp.uni-luebeck.de/story/automated-tcs-gen-cpns, March 2018.
[22] L. Lamport, Paxos made simple, ACM SIGACT News 32 (4) (2001) 18–25.
[23] J.-P. Martin, L. Alvisi, Fast Byzantine consensus, IEEE Trans. Dependable Secure Comput. 3 (3) (2006) 202–215.
[24] M.J. Fischer, N.A. Lynch, M.S. Paterson, Impossibility of distributed consensus with one faulty process, J. ACM 32 (2) (1985) 374–382, https://doi.org/10.1145/3149.214121.
[25] T.D. Chandra, V. Hadzilacos, S. Toueg, The weakest failure detector for solving consensus, J. ACM 43 (1996) 685–722, https://doi.org/10.1145/234533.234549.
[26] H. Attiya, A. Bar-Noy, D. Dolev, Sharing memory robustly in message-passing systems, J. ACM 42 (1) (1995) 124–142.
[27] L. Jehl, R. Vitenberg, H. Meling, SmartMerge: a new approach to reconfiguration for atomic storage, in: Y. Moses (Ed.), Distributed Computing – 29th Intl. Symp., DISC 2015, in: Lecture Notes in Computer Science, vol. 9363, Springer, 2015, pp. 154–169.
[28] M. Vukolić, Quorum Systems: With Applications to Storage and Consensus, Synthesis Lectures on Distributed Computing Theory, vol. 3 (1), Morgan & Claypool Publishers, 2012.
[29] Google Inc., gRPC remote procedure calls, http://www.grpc.io.
[30] Google Inc., Protocol buffers, http://developers.google.com/protocol-buffers.
[31] W. Grieskamp, N. Kicillof, K. Stobie, V. Braberman, Model-based quality assurance of protocol documentation: tools and methodology, Softw. Test. Verif. Reliab. 21 (1) (2011) 55–71, https://doi.org/10.1002/stvr.427.
[32] H. Meling, A framework for experimental validation and performance evaluation in fault tolerant distributed system, in: Workshop on Dependable Parallel, Distributed and Network-Centric Systems, DPDNS, IEEE, 2007, pp. 1–8.
[33] T.D. Chandra, R. Griesemer, J. Redstone, Paxos made live: an engineering perspective, in: Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing, PODC '07, ACM, 2007, pp. 398–407.
[34] C. Artho, Q. Gros, G. Rousset, K. Banzai, L. Ma, T. Kitamura, M. Hagiya, Y. Tanabe, M. Yamamoto, Model-based API testing of apache ZooKeeper, in: 2017 IEEE Intl. Conf. on Software Testing, Verification and Validation, ICST, 2017, pp. 288–298.
[35] H. Ponce de León, S. Haar, D. Longuet, Model-based testing for concurrent systems: unfolding-based test selection, Int. J. Softw. Tools Technol. Transf. 18 (3) (2016) 305–318, https://doi.org/10.1007/s10009-014-0353-y.
[36] C. Hawblitzel, J. Howell, M. Kapritsos, J. Lorch, B. Parno, M.L. Roberts, S. Setty, B. Zill, IronFleet: proving practical distributed systems correct, in: Proceedings of the ACM Symposium on Operating Systems Principles, SOSP, ACM, 2015.
[37] O. Padon, G. Losa, M. Sagiv, S. Shoham, Paxos made EPR: decidable reasoning about distributed protocols, Proc. ACM Program. Lang. 1 (2017) 108:1–108:31, https://doi.org/10.1145/3140568.
[38] P. Fonseca, K. Zhang, X. Wang, A. Krishnamurthy, An empirical study on the correctness of formally verified distributed systems, in: Proc. of the Twelfth European Conf. on Computer Systems, ACM, 2017, pp. 328–343.