Application of Model-based Testing on a Quorum-based Distributed Storage

Rui Wang[1], Lars Michael Kristensen[1],
Hein Meling[2], Volker Stolz[1]

[1] Department of Computing, Mathematics, and Physics
Western Norway University of Applied Sciences
Email: {rwa@hvl.no,lmkr@hvl.no,vsto@hvl.no}
[2] Department of Electrical Engineering and Computer Science
University of Stavanger, Email: {hein.meling@uis.no}

Abstract. Data replication is a central mechanism for the engineering of fault-tolerant distributed systems, and is used in the realization of most cloud computing services. This paper explores the use of Coloured Petri Nets (CPNs) for model-based testing of quorum-based distributed systems. We have used model-based testing to validate a distributed storage implemented using the Go language and the Gorums framework. We show how a CPN model of a single-writer, multi-reader distributed storage system can be used to obtain both unit test cases for the quorum logic functions, and system level test cases consisting of read and write quorum calls to the storage. The CPN model is also used to obtain the test oracles against which the result of running a test case can be compared. Our experimental results show that we obtain 100 % code coverage for the quorum functions, around 84 % coverage on the quorum calls, and around 40 % coverage on the Gorums library.

1 Introduction

Distributed systems serve millions of users in many important applications and domains. However, such complex systems are known to be difficult to implement correctly because they must cope with concurrency, failures, and a host of other challenges [6]. Thus, when designing and implementing distributed systems, it is important to ensure correctness and fault-tolerance. Distributed systems can rely on a quorum system to achieve fault-tolerance, yet it remains challenging to implement fault-tolerance correctly. Therefore, use of testing techniques can help to detect bugs and to improve the correctness of such systems.

One promising testing approach is *model-based testing* (MBT) [15]. MBT is a paradigm based on the idea of using models of a system under test (SUT) and its environment to generate test cases for the system. The goal of MBT is validation and error-detection aimed at finding observable differences between the behavior of the implementation and the intended behavior of the SUT. A test case consists of test input and expected output and can be executed on the SUT.

Typically, MBT involves: (a) build models of the SUT from informal requirements; (b) define test selection criteria for guiding the generation of test cases and the corresponding test oracle representing the ground-truth; (c) generate and run test cases, and finally; (d) compare the output from test case execution with the expected result from the test oracle. The component that performs (c) and (d) is known as a *test adaptor* and it uses a *test oracle* to determine whether a test has passed or failed.

Xu suggested using high-level Petri nets for MBT and implemented their approach [18]. The benefits of using high-level Petri nets over finite state machines and UML was: a) the ability to include data in the models and hence directly derive concrete test cases; and b) a compact modeling of parallelism making it simpler to obtain test cases for systems with concurrency.

The contribution of this paper is to investigate the use of Coloured Petri Nets (CPNs) [5] for model-based testing applied to quorum-based distributed systems [16]. Quorum systems are fundamental to building fault-tolerant distributed systems, and recently the Gorums library [9] has been developed to ease the implementation of quorum-based distributed systems. Our long-term research goal is to validate the Go implementation of the Gorums framework using MBT. As a first step towards this goal, we consider a Gorums-based implementation of a single-writer, multi-reader distributed storage. The storage system is implemented with a read and a write *quorum call*, that clients can use to access the distributed storage. The distributed storage may return multiple replies to a quorum call. To simplify client access to the storage, Gorums uses a user-defined *quorum function* to coalesce the different replies into a single reply that can then be returned to the client. For this particular storage system, we use a simple majority quorum.

CPNs have been widely used for modeling and verifying models of distributed systems spanning domains such as workflow systems, communication protocols, and distributed algorithms. Recently, work has also been done on automated code generation allowing an implementation of the modeled systems to be obtained [7]. Comprehensive testing of an implementation is, however, an equally important task in the engineering of distributed systems, independently of how the implementation has been obtained. This also applies in the case of automated code generation, as it is seldom the case that the correctness of the model-to-text transformations and their implementation can be formally proved.

The rest of this paper is organized as follows. §2 introduces quorum-based system and the Gorums framework, and §3 considers the Gorums-based distributed storage which constitutes our system under test. §4 presents the constructed CPN model for test case generation, and §5 shows how state-spaces and simulation can be used to obtain test cases and test oracles. In §6 we present the Go implementation of our test adapter and how it is connected to the Gorums implementation of the distributed storage in order to execute the test cases. We also report on the experimental results. §7 presents related work, and in §8 we sum up conclusions and presented directions for future work. The reader is assumed to be familiar with the basic concepts of high-level Petri Nets.

2 Quorum-based Distributed Systems and Gorums

In this section we briefly describe Gorums [9], a framework for implementing quorum-based distributed systems. We have used Gorums to implement the distributed storage system that constitutes our system under test.

Gorums is a library whose goal it is to alleviate the development effort for building advanced distributed algorithms, such as Paxos [8] and distributed storage [1]. These algorithms are commonly used to implement replicated services, and they rely on a quorum system [16] to achieve fault tolerance. That is, to access the replicated state, a process only needs to contact a quorum, e.g. a majority of the processes. In this way, a system can provide service despite the failure of individual processes. However, communicating with and handling replies from sets of processes often complicate the protocol implementations.

To reduce this complexity, Gorums provides two core abstractions: (a) a flexible and simple quorum call abstraction, which is used to communicate with a set of processes and to collect their responses, and (b) a quorum function abstraction which is used to process responses. These abstractions can help to simplify the main control flow of protocol implementations.

Gorums consists of a runtime library and code generator that extends the gRPC [3] remote procedure call library from Google. Specifically, Gorums allow 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. Fig. 1 illustrates the interplay between the main abstractions provided by Gorums.

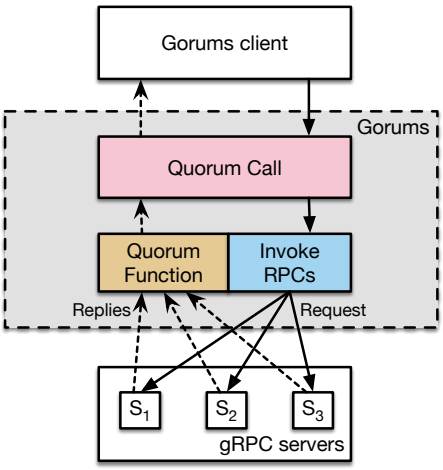

Fig. 1. Overview of Gorums abstractions.

With Gorums, developers can specify several RPC service methods using `protobuf` [4], and from this specification, Gorums' code generator will produce code to facilitate quorum calls and collection of replies. However, each RPC/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.

Our goal in this paper is to provide a framework for generating test cases to verify the correctness of different quorum function and quorum call implementations. 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 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 to verify 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.

3 System Under Test: Distributed Storage

We have implemented a simple distributed storage system, with a single writer and multiple readers. The storage system is replicated for fault-tolerance, and is implemented using Gorums. To test this storage implementation, we have designed a corresponding CPN model that we use to generate test cases (see §4). In this section, we describe the different components of the distributed storage.

As with any RPC library, Gorums also requires that the server implements the methods specified in the service interface. For our distributed storage, we have implemented two server-side methods: Read() and Write(). These can be invoked as quorum calls from storage clients, to read/write the state of the storage. In our current implementation, we allow only a single writer client, but any number of clients can read the state of the storage. A client reading from the storage may observe different replies returned by the different server replicas, since the read may be interleaved with one or more writes generated by the writer client.

To allow a reader to pick the correct reply value to return from a quorum call, the servers also maintains a timestamp that is incremented for each new Write(). That is, the reader will always return the value associated with the reply with the highest timestamp. Thus, to implement the reader client using Gorums, we can simply implement a user-defined ReadQF quorum function for the Read() quorum call as shown in Algorithm 1. As this code illustrates, a set

of replies from the different servers are coalesced into a single reply, the one with the highest timestamp, that can then be returned from the quorum call.

The user-defined quorum functions are implemented as methods on an object of type QuorumSpec, named *qs* in Algorithm 1. This object holds information about the quorum size, such as ReadQSize, and other parameters used by the quorum functions. This *qs* object must satisfy an interface generated by Gorums' code generator. In Algorithm 1, ReadQSize is used to determine if enough replies have been received to search for the reply with the highest timestamp.

Algorithm 1 Read quorum function

```
1: func (qs QuorumSpec) ReadQF(replies []ReadReply)
2:     if len(replies) < qs.ReadQSize then                    ▷ read quorum size
3:         return nil, false                                  ▷ no quorum yet, await more replies
4:     highest := ⊥                                           ▷ reply with highest timestamp seen
5:     for r := range replies do
6:         if r.Timestamp ≥ highest.Timestamp then
7:             highest := r
8:     return highest, true                                   ▷ found quorum
```

4 CPN Testing Model for the Distributed Storage

In this section, we describe the CPN model of our test framework for the distributed storage. We model the entire system, parametrized by a number of clients and servers. The key features are using colored tokens for the multiple outgoing and incoming messages, and the quorum specification based on the numbers of replies received so far.

Fig. 2 shows the top-most module of the CPN model developed in order to generate test cases for the distributed storage implementation. The substitution transition Clients represents the clients (users) of the distributed storage system while Servers represent the servers. The places CtoS and StoC are used for modeling the communication (exchange of messages) between the clients and the servers. The CPN model has been constructed in a folded manner so that the number of servers is a parameter to the CPN model that can be configured without making changes to the net-structure. Below we provide more details on selected modules of the CPN model. The complete CPN model including all color sets, variable declarations, and function definitions is available from [2].

Fig. 3 shows the client submodule of the Clients substitution transition in Fig. 2. The substitution transition QuorumCalls is used to model the behavior of applications running on the clients, which makes the read and write quorum calls. In particular, the submodules of QuorumCalls serve as test driver modules used to generate system tests for the distributed storage. The content of this module depends on the specific test scenarios to be investigated for the system

under test, and we give a concrete example of a test driver module in §6. The substitution transitions Read and Write represent the quorum calls provided by the distributed storage. The invocation of quorum calls is done by placing tokens on the Read and Write places. The port places StoC and CtoS are linked to the identically named socket places in Fig. 2.

Fig. 4 shows the submodule of the Read substitution transition which provides an abstract implementation of the Read() quorum call. The main purpose of the Read module is to generate test cases for the ReadQF quorum function. A read quorum call is triggered by the presence of a token with the color `READINVOKED(r)`, where `r` identifies the call and is used to match replies from servers to the call. The execution of a read quorum call starts by sending a read request to each of the servers. This is modeled by the transition SendReadReq and the expression on the arc to place CtoS, which will add tokens representing read requests being sent to the servers. In addition, a list-token is put on place ReadReplies, which is used to collect the replies received from the servers. The call then enters a WaitingReply state and waits for replies coming back from the servers. When a read's reply comes back, represented by a token on place StoC, then transition ApplyReadQF will be enabled. This transition takes the current list of `readreplies` and appends the received `readreply` to form `readreplies'`. The quorum function is then invoked, as represented by the arc expressions to WaitingReply and Read. If enough replies have been received, then a read result is returned to the Read place containing the value with the highest timestamp.

As we will see later, we use occurrences of the ApplyReadQF transition for generating test cases for the read quorum function. In addition, we record the result computed by the CPN model as the test oracle and compare it to the result of the Gorums implementation of the read quorum function.

The submodule for the write quorum call is similar. It has a transition ApplyWriteQF, which we use as a basis for generating test cases and obtain a test oracle for the write quorum function.

Fig. 5 shows the server submodule of the Servers substitution transition in Fig. 2. The replicated state of each server is modeled by the place State. The two substitution transitions are used for modeling the handling of write requests and read requests on the server side. Fig. 6 shows the submodule of the substitution HandleWriteRequest modeling the processing of a write request from a client. The incoming write request will be present as a token on place CtoS and contains

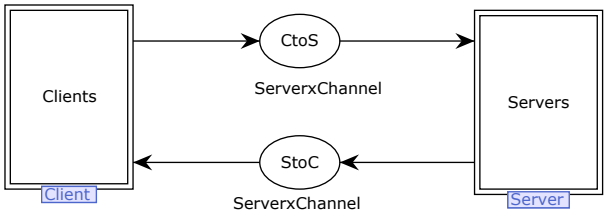

Fig. 2. Top-level module of the CPN model.

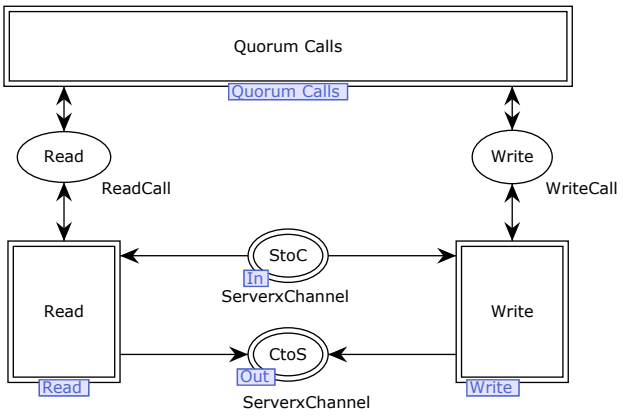

Fig. 3. The Clients module.

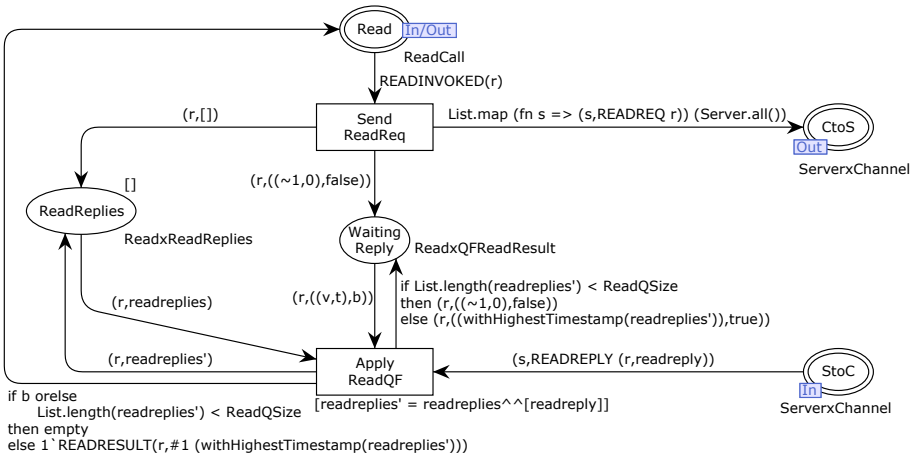

Fig. 4. The Read module.

a value `v'` to be written in the distributed storage together with a timestamp `t'`. The server compares the timestamp of the incoming write request with the timestamp `t` for the currently stored value `v`. If the timestamp of the incoming write request is larger, then the new value is stored on the server, and a write acknowledgement is sent back in a write reply to the client. Otherwise, the value stored remains unchanged and a negative write acknowledgement is sent to the client in the write reply. The handling of read request is modeled in a similar manner, except that no comparison is needed, and the server simply returns the currently stored value together with its timestamps.

5 Test Case Generation

The generation of test cases for the distributed storage system is based on the analysis of executions of the CPN model. Test cases can be generated for both the quorum functions and the quorum calls. The test cases generated for the quorum functions are unit tests, whereas the test cases generated for quorum calls are system tests consisting of concurrent and interleaved invocations of read and write quorum calls. In addition to the test cases, we also generate a *test oracle* for each test case to determine whether the test has passed.

5.1 Unit Tests for Quorum Functions

Test cases for the read quorum function can be obtained by considering occurrences of the ApplyReadQF transition (Fig. 4). When this transition occurs, the variable `readreplies'` is bound to the list of all replies that have been received from the servers so far, and which the quorum function is invoked on. In addition, we can use the implementation of the quorum function in the CPN model as the test oracle. This means that the expected result of invoking the quorum function can be obtained by considering the value of the token put back on place WaitingReply. The value of this token contains the result of invoking the quorum function in its second component. Occurrences of ApplyReadQF can be detected using either state spaces or simulations:

State-space based detection. We explore the full state space of the CPN model searching for arcs corresponding to the ApplyReadQF transition. Whenever an occurrence is encountered we emit a test case together with the expected result. In this case, we obtain test cases for all the possible ways in which the quorum function can be invoked in the CPN model.

Simulation-based detection. We run a simulation of the CPN model and use the monitoring facilities of the CPN Tools simulator to detect occurrences of the ApplyReadQF transition and emit the corresponding test cases. The advantage of this approach over the state-space based approach is scalability, while the disadvantage is potentially reduced test coverage.

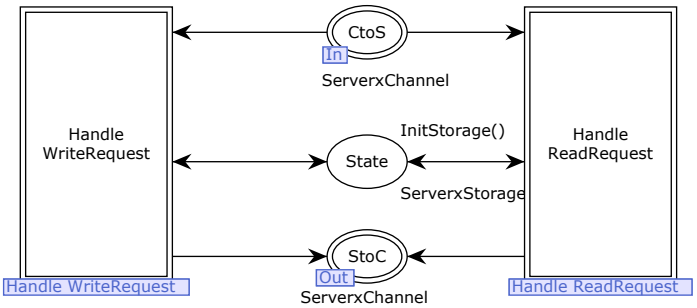

Fig. 5. The Server module.

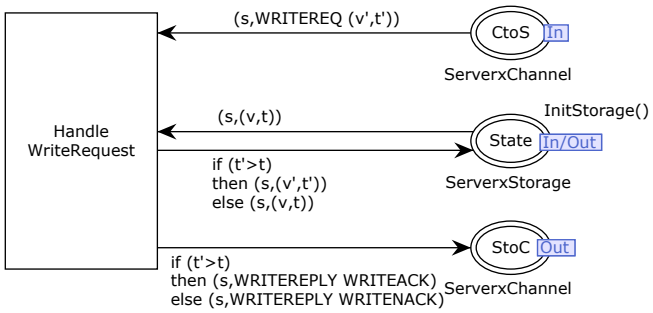

Fig. 6. The HandleWriteRequest module.

Test cases are generated based on detecting transition occurrences. This is done in a uniform way for both detection approaches. Specifically, we rely on a *detection function*, which must evaluate to true whenever a transition occurrence is detected. When this happens, a *generator function* is invoked to generate the actual test case.

The generated test cases and the expected results are exported as XML. In our initial work, we have used a custom XML format. As part of future work, we will investigate the use of a general-purpose XML format for test cases. The listing below gives an example of how a test case for the read quorum function is represented.

```
<Testcase>
   <CaseName>ReadQFTest1</CaseName>
   <Value>
     <ContentTest>
       <ValTest>0</ValTest><TsTest>0</TsTest>
     </ContentTest>
     <ContentTest>
       <ValTest>42</ValTest><TsTest>1</TsTest>
     </ContentTest>
   </Value>
   <ContentExpect>
     <ValExpect>42</ValExpect><TsExpect>1</TsExpect>
   </ContentExpect>
   <QuorumExpect>true</QuorumExpect>
</Testcase>
```

In this example, the test cases for the read quorum function corresponds to two replies (one with value 0 and timestamp 0, and one with value 42 and timestamp 1). With three servers, this constitutes a quorum, and the value returned from the quorum function is therefore expected to be 42 with the timestamp of 1.

5.2 System Tests of Quorum Calls

The generation of test cases and expected results is based on the submodule of the QuorumCalls substitution transition (Fig. 3). This module acts as a test driver for the system by specifying scenarios for read and write quorum calls to the underlying quorum system. By varying this module, it is possible to generate different scenarios of read and write calls.

Fig. 7 shows an example of a test driver in which the client executes one read and one write call (concurrent or in any order). Upon completion of these two calls, a new read call is made. The different Invoke transitions represent invocations of read and write quorum calls. Each call has a unique identifier (1, 2, and 3) for identifying the call. The write call also has a value (in this case 7) to be written to the distributed storage.

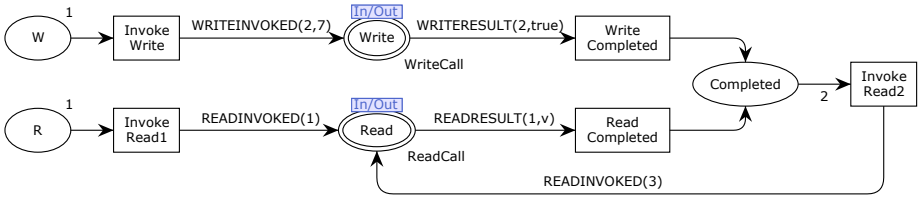

Fig. 7. The QuorumCalls module.

To make test case generation independent of the particular test driver module, we exploit that the read and write quorum calls made during an execution of the CPN model, can be observed as tokens on the Read and Write socket places (see Fig. 3). When there is a `READINVOKED(i)` token on place READ for some integer `i`, it means that a read quorum call identified by `i` has been invoked. When the read quorum call has terminated, there will be a token with the color `READRESULT(i,v)` present on the place Read, where `v` is the value read by the call. The invocation and termination of write quorum calls can be detected in a similar manner by considering the tokens with the colors `WRITEINVOKED(i,v)` and `WRITERESULT(i,b)` on the place Write (Fig. 3), where the boolean value `b` denotes whether the value `v` was written or not.

Based on this, we can generate test cases in XML format specifying both the concurrent and sequential execution of read and write calls. The listing below shows an example where first a read and write is initiated and upon completion of these two calls, a new read call is initiated.

```
<RWTest>
  <Function>RWTest</Function>
  <Testcase>
    <CaseName>RWTest1</CaseName>
      <QuoCallMainROpers>
        <OperationName>WRITE</OperationName>
        <OperationValues>
          <Value>7</Value>
        </OperationValues>
          <QuoCallOtherROpers>
            <OperationName>READ</OperationName>
            <OperationValues>
              <Value>7</Value>
              <Value></Value>
            </OperationValues>
          </QuoCallOtherROpers>
      </QuoCallMainROpers>
      <QuoCallMainROpers>
        <OperationName>READ</OperationName>
        <OperationValues>
              <Value>7</Value>
        </OperationValues>
      </QuoCallMainROpers>
  </Testcase>
</RWTest>
```

We handle concurrent executions by nesting the read and write `operation` tag as illustrated above. In addition, we interpret a new operation positioned after the completion of a call `operation` end tag as the operation should not be started until after the termination of the call. For write calls, we use the value tag to be specify the value to be written, and for read calls, we use the value tag to describe the permissible value (see next section) returned by read calls for the test case. The absence of a value between value tags indicates that the result could be null - corresponding to the case where no value have yet been written into the storage.

It should be noted if the CPN model specifies that a read and write call may execute concurrently (independently), but happened to be executed in sequence in a concrete execution of the CPN model (e.g., first the read executes and completes and then the write executes an completes), then that will be specified as a sequential test case in the XML format. This is not a problem as the CPN model captures all the possible executions and hence there will be another execution of the CPN model in which the read and the write are running concurrently.

5.3 Test Oracle for System Tests

Checking that the result of an execution with read and write quorum calls is as expected is more complex than for quorum functions. This is because the result of concurrently executing read and write calls will depend on the order in which

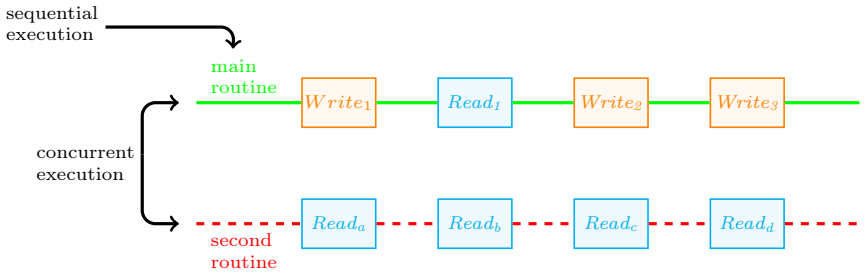

Fig. 8. An example of concurrent and sequential execution of quorum calls.

messages are sent and received. Fig. 8 shows an example test case in which there are two routines (threads of execution) that concurrently execute read and write quorum calls. When $Write_1$ and $Read_a$ are initialized and executed concurrently, the returned result of $Read_a$ could be the old value in the servers before $Write_1$ writes a new value to servers, or the returned result of $Read_a$ could be the value already written by $Write_1$. The same situation for $Write_2$ and $Read_c$, since they are executed concurrently, the returned value of $Read_c$ could be the value written by $Write_1$ or $Write_2$.

This means that if we execute (simulate) the CPN model with a test case containing concurrent read and write quorum calls, then the result returned upon completion of the calls may be different if we execute the same test case against the Go implementation. The reason is that we cannot control in what order the messages are sent and delivered by the underlying gRPC library, i.e., due to non-determinism in the execution. When we apply a state-space based approach for extracting the test cases, e.g., for the quorum function, then we can compute all the possible legal outcomes of a quorum call since the state space captures all interleaved executions. In contrast, we cannot obtain all legal values when extracting test cases from a single execution of the CPN model.

The first step towards constructing a test oracle is to characterize the permissible values of a read quorum call. These are:

1. the initial value of the storage in case no writes were invoked before the read was invoked, or;
2. the value of the most recent write invoked but not terminated prior to the read call (if any) or;
3. the value of the most recent write that has terminated prior to invocation of the read or;
4. the value of a write that was invoked between the invocation and completion of the read.

The above can be formally captured in the stateful automaton shown in Fig. 9 (left), which can be used to monitor the global correctness of the distributed storage. The four events are shorthands for the abstract tokens per client-request observed in the model, `READINVOKED(i)` etc.

On a read call RI_i, any pending write $WI(c)$ observed since the last write-return $WI(c)$ is a potential read-result. We abuse notation from alternating automata with parametrized propositions [14] to capture that on a read invocation, we remain in the initial state and collect further input for a new instance of the monitor with the same current state (indicated by the dashed line) for subsequent read-invocations.

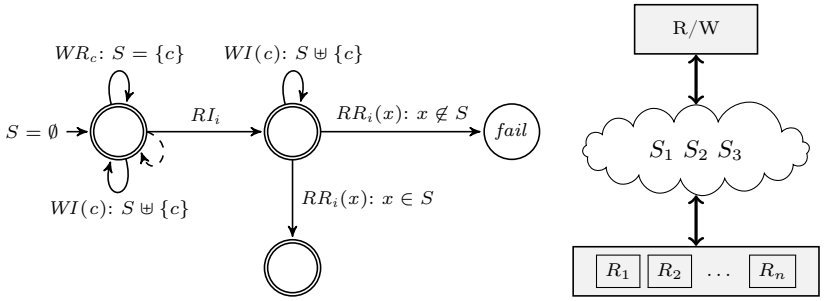

Fig. 9. Read-write automaton (left) and monitor deployment (right).

In order to obtain a test oracle which can be used in state space-based and simulation-based test case generation, we use the above automaton to perform run-time verification of the Go implementation when executed on the test cases derived from the CPN model. Specifically, our test adaptor implements a *run-time monitor* corresponding to the above automaton in order to keep track of the invoked and terminated write calls and thereby determining whether a value returned from a read call is permissible. Our test framework currently runs all clients (the single writer and multiple readers) within a single Go process. This allows us to directly call into the monitor *before* the client sends the fan-out messages to servers, and *after* the quorum function returns the resulting quorum value, to check the result of the read request for plausibility against the permitted values specified above. This corresponds to monitoring *all* calls and returns in a particular deployment, i.e., correlating read calls and returns of all clients in the system against those of the writer in the shaded area of Fig. 9 (right).

6 Testing the Distributed Storage Implementation

We have developed the QuoMBT test framework in order perform model-based testing of quorum-based systems implemented using Gorums. Fig. 10 gives an overview of the framework which consists of CPN Tools and a test adapter. CPN Tools is used for modeling and generation of test cases and oracles as explained in §4 and §5. The generated test cases are written into XML files by CPN Tools, and then read by the reader of the test adapter, as shown in Fig. 10. The reader

feeds the test cases into the distributed storage and each test case is executed with the provided test values as inputs. The tester included in the test adapter compares the test oracle's output against the output of each test case in order to determine whether the test fails or succeeds.

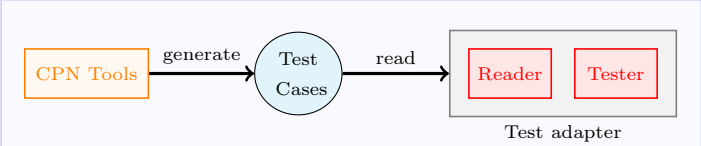

Fig. 10. The QuoMBT test framework.

6.1 The Test Adapter: Reader and Tester

The *Reader* and the *Tester* are both implemented in the Go programming language. The *Reader* can read three kinds of XML files for testing read quorum functions, write quorum functions, and quorum calls. We could have generated Go-based table-driven tests, which is already supported by the Go standard library. However, we chose to use an XML-based format for the generated test cases to enable reuse of the test generator across programming languages.

The *Reader* uses Go's *encoding/xml* package, which makes it easy to define mappings between Go structs and XML elements. Listing 1.1 shows the Go structs defined for test cases for the ReadQF quorum function. In order to map XML content into these structs, each field has an associated XML tag, which is used by Go's XML decoder to identify the field to populate with content from the XML. As shown Listing 1.1, the values for testing the ReadQF quorum function include a string value (*ValTest*) and a timestamp (*TsTest*). The expected values of the test include the expected value (*ValExpect*) and expected timestamp (*TsExpect*).

```
// ReadQFTest contains the test function name and test cases for ReadQF.
type ReadQFTest struct {
  FunctionName string     `xml:"Function"`
  TestCases    []ReadQFTC `xml:"Testcase"`
}
// ReadQFTC contains the test case name, test replies array, and
// the expected value and whether or not this represents a quorum.
type ReadQFTC struct {
  CaseName       string         `xml:"CaseName"`
  Replies        []*Value       `xml:"Value>ContentTest"`
  ExpectedVal    *ContentExpect `xml:"ContentExpect"`
  ExpectedQuorum bool           `xml:"QuorumExpect"`
}
// Value contains the test content for ReadQF.
```

```
type Value struct {
  ContentTest
}
// ContentTest contains the test value and test timestamp for ReadQF.
type ContentTest struct {
  ValTest string `xml:"ValTest"`
  TsTest  int64  `xml:"TsTest"`
}
// ContentExpect contains the expected value and timestamp for ReadQF.
type ContentExpect struct {
  ValExpect string `xml:"ValExpect"`
  TsExpect  int64  `xml:"TsExpect"`
}
```

Listing 1.1. Test data structures for ReadQF quorum function.

Finally, we have implemented the *Tester* using the *testing* package provided by the Go standard library. Go's testing infrastructure allows us to simply run the `go test` command to execute our generated tests, which will provide pass/-fail information for each test case. In addition, this test infrastructure can also provide code coverage.

6.2 Distributed Storage Under Test

To test our distributed storage, we have implemented a test adapter that can execute both the unit tests for user-defined quorum functions and system level tests for quorum calls. The unit tests for read and write quorum functions can be performed without running any servers, while the system level tests for quorum calls requires a set of running servers to test the complete system, including parts of the Gorums framework. When testing the distributed storage, we distinguish between quorum functions and quorum calls, because quorum functions are defined by developers when implementing their specific abstractions, whereas quorum calls are provided generally by the Gorums library. This separation also provides a modular approach to testing.

Our test adapter implements a Go-based *Tester* for testing quorum functions. We simply iterate through the test cases obtained from the *Reader*, invoking the ReadQF and WriteQF functions with the test values, and compare the results against the test oracle.

When testing the quorum calls, the servers shown in Fig. 1 must be started first. Then, the test adapter starts a client so that it can execute the quorum calls. The test value, obtained from XML files, for each write quorum call is written to servers by calling the write quorum call, and for each read quorum call, the value returned by the servers will be captured by the *Tester* to compare against the test oracle. For each write quorum call, the tests only check if it returns an acknowledgment from servers.

The non-trivial part of the test case execution is the concurrent and sequential executions of read and write quorum calls. Fig. 11 illustrates the detailed

implementation of quorum calls under test. The testing function for quorum calls run through each test case read from the *Reader*. For the run of each test case, the write and read quorum calls can be executed both sequentially and concurrently depending on the test driver used. For the sequential executions, the decision to execute write or read calls is made according to their sequences in the XML files generated by CPN Tools. For the concurrent executions of write and read quorum calls, the test execution makes use of goroutines provided by the Go programming language. Therefore, in Fig. 11, within each run of test cases, a write or read quorum call (shown by solid line arrows) is executed based on their sequence in the XML files. Meanwhile, there are other read calls, shown as dashed line arrows in Fig. 11, that can be executed concurrently with the running write or read quorum call shown by solid line arrows. After executing each test cases, the returned values of quorum calls are collected.

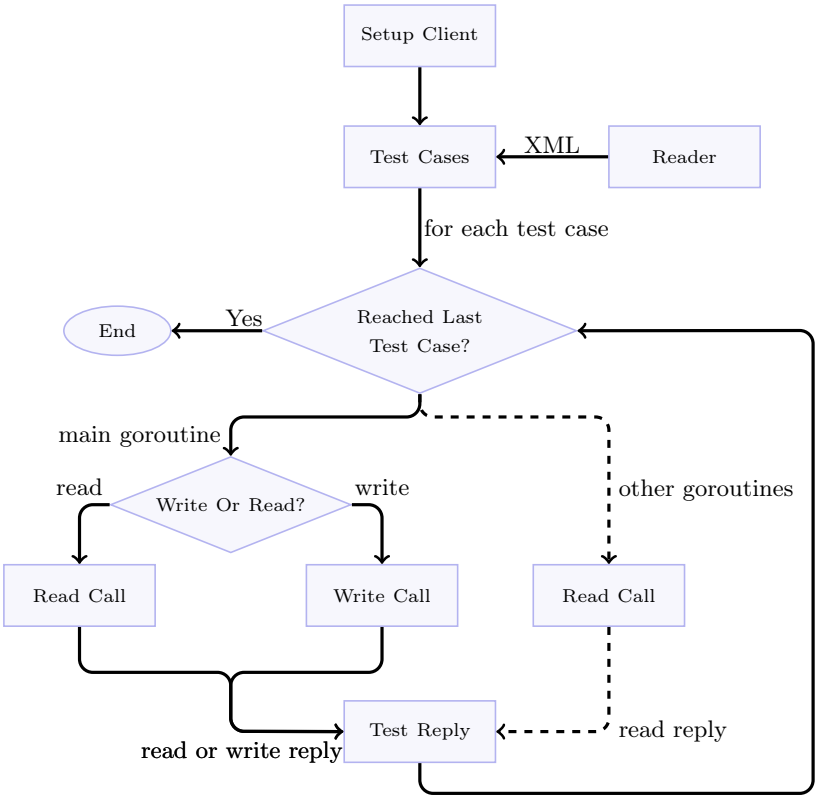

Fig. 11. Flow-chart of test case execution for quorum calls.

6.3 Experimental Results

To perform an initial evaluation of our model-based test case generation, we consider the code coverage obtained using different test drivers. The Go toolchain includes a coverage tool which we have used to measure statement coverage.

Table 1 summarizes the experimental results obtained using different test drivers. We consider the following test drivers: one read call (RD), one write call (WR), a read call followed by a write call (RD;WR), a write call followed by a read call (WR;RD), a read and a write call executed concurrently (WR||RD), a read and a write call executed concurrently and followed by a read call ((WR||RD);RD). The table shows the number of nodes/arcs in the state space of the CPN model with the given test driver, the state space generation time (TM), the number of test cases generated for quorum calls (QC), the number of test cases generated for quorum functions (QF). The test case generation time was less than one second in all cases.

For the test case execution, we show the code coverage (in percentage) that was obtained for the system level and unit tests. The results show that the statement coverage for read (RD-QF) and write (WR-QF) quorum functions is 100 % for both system and unit tests, as long as both read and write calls are involved. The statement coverage for read (RD-QC) and write (WR-QF) quorum calls is up to 84.4 %. For the Gorums library as a whole, the statement coverage reaches 40.8 %. Note that the Gorums library contains all code generated by Gorums' code generator, including gRPC code, various auxiliary functions and logic for the quorum calls. Much of this code is actually not used in the critical path of the quorum calls. The total number of lines of code for the system under test is approximately 2200 lines, which include generated code by Gorums' code generator (around 2000 lines), server code (around 130 lines), client code (around 80 lines) and the code for quorum functions (around 60 lines). The test case execution time was less than one second in all cases.

Table 1. Experimental results – test case generation and code coverage.

Test Driver	**Test case generation**					**Test case execution**				
						System			**Unit**	
	Nodes	**Arcs**	**TM**	**QC**	**QF**	**Gorums Library**	**QCs RD**	**QCs WR**	**QFs RD**	**QFs WR**
RD	39	74	<1	1	3	24.6	84.4	0	100	0
WR	39	74	<1	1	3	24.6	0	84.4	0	100
RD;WR	444	1073	<1	1	7	39.1	84.4	84.4	100	100
WR;RD	615	1376	<1	1	12	40.8	84.4	84.4	100	100
WR||RD	5,119	16,677	6	6	17	40.8	84.4	84.4	100	100
(WR||RD);RD	21,020	59,647	53	6	17	40.8	84.4	84.4	100	100

For our system level tests, the statement coverage of quorum calls and Gorums are lower than the coverage for quorum functions. We have conducted a

code inspection, which shows that the statements currently not covered in the Gorums library is code related to error handling. Our initial test case generation presented in this paper does not consider failures and error conditions. Hence, with the current testing model, we cannot expect to obtain a higher coverage.

7 Related Work

Research into model-based testing for distributed systems and cloud applications is not new. In Saifan and Dingel's survey [12], they provide a detailed description of how model-based testing is effective in testing different quality attributes of distributed systems, such as security, performance, reliability, and correctness. The authors also classified model-based testing based on different criteria and compared several model-based testing tools for distributed systems according to this classification. This comparison, however, did not include CPN-based tools, nor did they consider quorum-based distributed systems.

Until now, there also has been relatively few applications of CPNs for model-based testing and test cases generation. Watanabe and Kudoh [17] propose two CPN-based test suite generation methods for distributed systems, referred to as the CPN Tree (CPT) method and the CPN Graph (CPG) method. Their method does not directly address a particular way in deriving a CPN model for a distributed system, nor do they give any particular guarantees on achieved coverage for their methods.

Xu [18] presents the Integration and System Test Automation (ITSA) tool which supports test code generation for languages such as Java, C/C++, and C#. The ITSA tool also uses the state spaces of the testing model to generate and select test cases. To obtain concrete test cases with input data, the tool relies on a separate model-implementation mapping. In contrast, we obtain the input data for the quorum functions and calls directly from the data contained in the testing model. The ITSA approach also includes test selection techniques and metrics in order to prune the number of test cases. For testing the distributed storage implementation considered in this paper there was no need for test selection techniques. However, when considering more complex quorum system this will likely be needed also in our approach.

Lima and Faria [10] use timed event-driven CPNs to generate test cases for distributed systems. They do not use CPNs as a direct interface to the user, but generate them from UML sequence diagrams. A CPN-based test generation approach has been proposed by Liu *et al.* [11]. The approach consists of conformance testing oriented CPN (CT-CPN) as the basic models, a new PN-ioco relation to specify the meaning for an implementation to conform to its specifications, and the test case generation algorithm for simulating the CP-CPN model. For the test case generation algorithm, the authors only considered simulation-based test case generation for the simplified file downloading protocol system. However, in our paper, we also consider state-space based test case generation.

8 Conclusions and Future Work

The main contribution of our work has been to establish an infrastructure consisting of a CPN modeling approach, test case generation algorithms, and a test case execution framework, which can be used to validate quorum-based systems implemented using the Gorums library. Our initial experiments with this infrastructure on a distributed storage system have been promising in that we have obtained a relatively good code coverage even with simple test drivers and a small number of test cases.

An important attribute of our approach is that the CPN testing model has been constructed such that it can serve as a basis for model-based testing of *other* quorum-based systems. In particular, it is only the modeling of the quorum calls on the client and server side that are system dependent. To experiment with different quorum functions for a given quorum system, it is only the implementation of the quorum functions in Standard ML that needs to be changed. The state space and simulation-based test case generation approach is independent of the particular quorum system under test.

Model-based testing can be used to test a system either by connecting a model (acting as a test driver) directly to an instance of the running system, or, as we do in this paper, generate test case offline and execute these test cases against the system. The main challenge related to this, is how to handle non-determinism during test case execution. In our current approach, we have addressed this by using monitors known from the field of run-time verification.

The work presented in this paper opens up several directions of future work. We have obtained good coverage results on the quorum functions and calls with the current testing model which encodes a *fair weather scenario*, i.e., it generates test cases where the environment behaves flawlessly by only doing reordering of messages through non-determinism and interleaving in the model. In order to increase coverage of the Gorums library as a whole, we need to test the quorum calls under adverse conditions, such as network errors and server failures. This will require extensions to the model, e.g. injecting erroneous values or generating timeouts. Thus, the recorded test cases must also record the particular scenarios in system tests such that the environment can replay the conditions. Conversely, with the current model, an intermittent failure in the test environment during system testing may be reported as test failures, as they are likely to produce a result diverging from the recorded test output.

Our current solution uses the CPN model to generate test cases and record the correct response from the quorum function. The global monitor presented in §5 independently specifies safe behavior in the form of correct read calls. Instead of the automaton, a different formal specification logic for (distributed) systems could have been used, e.g. Scheffel and Schmitz's distributed temporal logic [13]. Their three-valued logic would allow us to adequately capture that the monitor has neither detected successful nor failed completion.

To evaluate the generality of our modeling and test case generation approach, we need to apply it to other and more complex quorum-based systems. This will challenge the limits of state space-based generation of test cases. It, therefore,

becomes important to investigate the test coverage that can be obtained with simulation versus the test case coverage that can be obtained with state spaces. We anticipate that this will motivate work into techniques for on-the-fly test case generation during state space exploration.

References

1. H. Attiya, A. Bar-Noy, and D. Dolev. Sharing Memory Robustly in Message-passing Systems. *J. ACM*, 42(1):124–142, Jan. 1995.
2. CPN Testing Model for Gorum-based Distributed Storage. http://home.hib.no/ansatte/lmkr/DistributedStorage.cpn. April, 2017.
3. Google Inc. gRPC Remote Procedure Calls. http://www.grpc.io.
4. Google Inc. Protocol Buffers. http://developers.google.com/protocol-buffers.
5. K. Jensen and L. Kristensen. Coloured Petri Nets: A Graphical Language for Modelling and Validation of Concurrent Systems. *Communications of the ACM*, 58(6):61–70, 2015.
6. Jepsen. Distributed Systems Safety Analysis. http://jepsen.io.
7. L. Kristensen and V. Veiset. Transforming CPN Models into Code for TinyOS: A Case Study of the RPL Protocol. In *Proc. of Intl. Conf. on Application and Theory of Petri Nets and Concurrency*, volume 9698 of *LNCS*, pages 135–154, 2016.
8. L. Lamport. The Part-time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.
9. T. E. Lea, L. Jehl, and H. Meling. Towards New Abstractions for Implementing Quorum-based Systems. In *37th IEEE Intl. Conf. on Distributed Computing Systems (ICDCS)*, 2017. To appear.
10. B. Lima and J. P. Faria. An Approach for Automated Scenario-based Testing of Distributed and Heterogeneous Systems. In *10th Intl. Joint Conf. on Software Technologies (ICSOFT)*, pages 241–250. SciTePress, July 2015.
11. J. Liu, X. Ye, and J. Li. Colored Petri Nets Model Based Conformance Test Generation. In *IEEE Symp. on Computers and Communications (ISCC)*, pages 967–970. IEEE, 2011.
12. A. Saifan and J. Dingel. Model-based Testing of Distributed Systems. Technical Report 548, School of Computing, Queen's University, Canada, 2008.
13. T. Scheffel and M. Schmitz. Three-valued Asynchronous Distributed Runtime Verification. In *Twelfth ACM/IEEE Intl. Conf. on Formal Methods and Models for Codesign (MEMOCODE)*, pages 52–61. IEEE, 2014.
14. V. Stolz. Temporal Assertions with Parametrized Propositions. *Journal of Logic and Computation*, 20(3):743–757, 2010.
15. M. Utting, A. Pretschner, and B. Legeard. A Taxonomy of Model-based Testing Approaches. *Software Testing, Verification and Reliability*, 22:297–312, 2012.
16. M. Vukolic. *Quorum Systems: With Applications to Storage and Consensus*. Morgan and Claypool, 2012.
17. H. Watanabe and T. Kudoh. Test Suite Generation Methods for Concurrent Systems Based on Coloured Petri Nets. In *Software Engineering Conference*, pages 242–251. IEEE, 1995.
18. D. Xu. A Tool for Automated Test Code Generation from High-level Petri Nets. In *Proc. of ICATPN'2011*, volume 6709 of *LNCS*, pages 308–317. Springer, 2011.