From Models to Implementations - Distributed Algorithms using Maude

Sam Stephens

May 2018

Abstract

Maude is an equational and rewriting logic specification tool. It allows a unique and simple way of specifying concurrent programs and lends itself nicely to verification. This senior thesis focuses specifically on patterns for creating distributed algorithms in Maude, and after applying these patterns to several classical algorithms, it builds up to the consensus algorithm Raft, which has not previous been implemented.

Maude has limited support for communication between processes on separate machines. This paper develops a "middleware" that enables a straightforward approach for transforming a model into an implementation, allowing correct-by-construction working implementations of distributed systems. The design and usage of this middleware will be examined, especially with the use of case studies. Ideally, this will allow easier development of future distributed systems in Maude, without having to worry about socket-level coding.

Senior Thesis

Submitted in partial fulfillment of the Bachelor of Science degree in Computer Science at the University of Illinois at Urbana-Champaign. Special thanks to my thesis advisor, Professor José Meseguer, for assisting me throughout the process.

The source code for this thesis, including the middleware and the case studies, can be found at: `http://maude.cs.illinois.edu/links/scsteph2-thesis-source.zip`

Contents

1 Introduction

1.1 Problem Description

The development of correct distributed systems can be challenging due to the difficulties with predicting all the non-deterministic interactions and because they are much harder to test than sequential systems. Maude, through the use of rewrite rules, provides a very high-level method to simply and intuitively model distributed systems. Furthermore, it can use model-checking to verify properties of such systems.

However, there is often be a gap between the formal model of a system and its implementation. The implementation may not faithfully implement the model and could introduce bugs. Consequently, the correctness guarantees from the model may not actually extend to the implementation as desired. Minimizing this gap is therefore valuable because it reduces the risk of bugs and allows the guarantees from the model to also apply to the implementation.

1.2 Technical Approach and Case Studies

Maude has built-in support for TCP/IP sockets. In principle, this means that it is possible to transform a Maude model of a distributed system into a distributed implementation. Some case studies that demonstrate this possibility have been developed [3, 4, 5], but Maude support for transforming formal Maude specifications into distributed systems is still quite limited and the process is not straight-forward. In particular, there are difficulties bridging the gap between the levels of communication by TCP/IP sockets and higher-level message communication between distributed Maude objects.

The main goal of this thesis is to significantly ease the difficulties of transforming from the model to a correct-by-construction distributed implementation by:

1. Developing a *middleware* for users that simplifies the implementation of message communication between Maude objects via TCP/IP sockets.
2. Presenting a *methodology* where
    - Distributed systems are first specified in Maude
    - Formal properties are then verified by model-checking, and
    - A distributed implementation of that system is semi-automatically derived from the model and the middleware.

A series of case studies help demonstrate this methodology, beginning with some well-known distributed algorithms and culminating with a new Maude specification and distributed implementation of a simplified version of the recent Raft consensus algorithm [7].

1.3 Outline

The rest of this thesis is organized as follows. Section 2 gives an overview of Maude and some of its patterns that will be used. Section 3 presents the design of the new middleware that facilitates a user in deriving a distributed implementation from a Maude model. Section 4 builds a sample client-server model, where a client request the factorial of a number, and the server returns it. Section 5 models and implements the two-phase-commit protocol, whereby a coordinator sends a commit to a multitude of cohorts based on how they respond to a vote. The case study in section 6 models, verifies, and builds a distributed implementation of the Raft consensus algorithm. Lastly, section 7 summarizes the methodology demonstrated in the case studies, and adds some concluding remarks and future directions. Appendix A works out in detail transforming a ring leader election protocol, showing each step of the methodology in detail.

2 Maude Overview

2.1 Rewrite Rules

A basic understand of Maude is already assumed, but a brief overview is given below. In Maude, a module defines syntax and semantics of terms. The syntax defines what expressions are valid and what types they evaluate to. The semantics of a module is defined by a combination of equations and rules that define how to simplify and evolve those terms. Both rules and equations can be marked as conditional, where they can only be applied if some condition is true. Modules can use other modules, making it easier to decompose tasks.

There are two important aspects of syntax - sorts and operators. A sort is a type of data. For example, some default sorts are `Bool`, `Nat`, or `String`, for boolean, natural numbers, or strings respectively. If one sort is a subsort of another sort, then any term of the former sort is also a term of the latter sort. This subsorting is the same notion of set inclusion for the corresponding set of terms. An operator in Maude has a name, some number of operands given by their sort, and a resulting sort. If there are no arguments, then that operator is also called a constant. An operator can also have additional attributes, such as associative, commutative, or have an identity element, which is useful for expressing some properties that cannot be defined with oriented equations. Checking the syntax of a term in Maude can be done with the `parse` command. As an example, a basic syntax of the natural numbers might include two sorts, naturals and non-zero naturals, as well as a constant 0, successor function, and addition. In Maude, this could be defined as:

```
fmod NAT-SYNTAX is
  sorts Nat NzNat .
  subsort NzNat < Nat .
  op 0 : -> Nat .
  op s : Nat -> NzNat .
  op + : Nat Nat -> Nat .
endfm
```

Equations are defined by writing some left-hand side term equals some right-hand side term. Each of these terms may contain some variables; however, there are some conditions to allow these equations to be executable. To evaluate a term using an equation, the term is matched to the left hand side of the equation, and the right hand side is substituted. To evaluate a term in a module, equations are applied until no more can be applied.

In general, it will also be important that the equations also satisfy the Church-Rosser property and are terminating, which is a key difference compared to rules. This essentially means that any term can be evaluated down to only a single unique term using the equations; these fully evaluated terms define the initial algebra of the functional module in the intuitive form of the so-called canonical term algebra. To compare two terms, such as if $s(s(0)) + s(s(0)) = s(s(s(0))) + s(0)$ (e.g. $2 + 2 = 3 + 1$), both sides are simplified completely and compared syntactically. Reducing a term with equations can be done with the red command. For example, the equations for the natural numbers defined before could be:

```
fmod NAT-EQ is
  inc NAT-SYNTAX .
  vars N M : Nat .
  eq N + 0 = N .
  eq N + s(M) = s(N + M) .
endfm
```

The last important aspect of modules are rules. The previous two examples are functional modules, defined with `fmod` and `endfm`. More general systems modules can contain both equations and rules and are defined with `mod` and `endm`. Rules are similar to equations, in that they describe how to modify a term. However, in general there are little restrictions put on rules, and they are *not* intended to capture equality in a system. Rules capture *transitions* between states, especially allowing them to capture non-determinism. A system module can have both rules and equations. Before a term is evaluated with a rule, it is fully simplified with equations. It is possible that multiple rules can be applied to a term, in which case one of them is chosen non-deterministically. To see some rewriting of a term, the `rew` command can be used. To search every possible rewriting, the `search` command can be used.

2.2 Concurrent Object Programming

There is a very useful pattern in Maude programming to capture concurrent object-based programming. Core Maude supports this pattern with the `CONFIGURATION` module in `prelude.maude`:

```
mod CONFIGURATION is
  sorts Attribute AttributeSet .
  subsort Attribute < AttributeSet .
  op none : -> AttributeSet  [ctor] .
  op _,_ : AttributeSet AttributeSet ->
    AttributeSet [ctor assoc comm id: none] .

  sorts Oid Cid Object Msg Portal Configuration .
  subsort Object Msg Portal < Configuration .
  op <_:_|_> : Oid Cid AttributeSet -> Object [ctor object] .
  op none : -> Configuration [ctor] .
  op __ : Configuration Configuration -> Configuration
    [ctor config assoc comm id: none] .
  op <> : -> Portal [ctor] .
endm
```

An object has an object id, class id, and some set of attributes. The attributes of a class need to be defined through operators that evaluate to `Attribute`; this is equivalent to defining the fields of an object. The `Msg` sort for messages is defined, but as with an `Attribute`, operators that evaluate to `Msg` need to be defined. A message is a way to enable objects to communicate with each other. A common syntax, and one used throughout this paper, is given by:

```
op msg_from_to_ : MsgCont Oid Oid -> Msg [ctor] .
```

Hence, a message is some contents of sort `MsgCont` that has a source and destination. A portal is a special kind of object that will be further described later. A set of objects and messages make up a configuration, also sometimes referred to as the object pool. Rules are defined on objects of a certain class id; typically, a rule will involve consuming or sending a message in the pool and possibly modifying some internal state. The `frew` (fair rewrite) command is the best choice to evolve objects because it ensures that no objects will be starved. It is very similar to the `rew` command, but attempts to apply rules fairly to all objects.

2.3 Sockets and Buffering

Maude sockets are external objects that enable a Maude process to send and receive messages to and from an external environment. Rewriting is done using the `erew` command, for external rewrite. It is similar to the `frew` command,

except that it also allows messages to be exchanged with external objects. The *portal* object is a special object that must be in the object pool to allow external rewrites, and is defined as:

```
sort Portal .
subsort Portal < Configuration .
op <> : -> Portal [ctor] .
```

The sockets are IPv4 TCP, and come in both client and server variants. A basic overview is given here, although full details and examples are in the Maude manual [4]. Ignoring error handling, a client socket is created by first rewriting `createClientTcpSocket` with a specified address and port. Once the socket is created, this operator is removed and `createdSocket` is added to replace it. Data can be sent with `send` or received with `receive`, which are replaced with `sent` and `received` respectively when successful. The server socket is similar; however, it must listen for clients before data can be sent or received.

One major drawback with Maude sockets is that they are unbuffered. When a message is sent, it may be broken up into multiple packets. However, when the receive socket receives the data, one call to receive may only return some subset of the data. This can make them extremely difficult to use, as it requires knowing the end of the data being sent and reading multiple times until the end can be found. This is not always possible.

Instead, the Maude book introduces `BufferedSockets`, and is included in the provided code `buffered-socket.maude`. The code from the book had a bug in the `closedSocket` rule which is corrected in the provided code. Buffered sockets follow the exact same syntax as regular sockets, except that the first letter of each operator is capitalized, so `acceptClient` becomes `AcceptClient` and so on. Whenever a message is sent across a buffered socket, a `#` separator is appended to the end. When a buffered socket reads, it reads from the socket multiple times and stores the cumulative result until the `#` separator is found and the entire block of data is returned. As result, it is important that any data being sent does not contain this symbol. The middleware described in the next section exclusively uses buffered sockets because of the benefit of not worrying about data being split across multiple reads.

3 The Middleware

3.1 Overview

The middleware in the module `MW` in `mw/mw.maude` is a tool to make it as easy as possible to enable Maude objects on multiple machines to communicate with each other. The motivation behind this simplicity is to minimize the possibility of any errors being introduced. Originally, Mobile Maude (described in *All About Maude*) was considered. However, this requires understanding a much more complicated system and modifying the original code too much to be as useful. Instead, the middleware presented here is designed to be an object that

sits in the object pool in each Maude process and automatically consumes and sends messages that are destined for objects on other processes as well as read and introduce messages from other processes. Note that loading the middleware causes the trace to be enabled; this allows output to be shown to the console while it runs. Remove the line at the top that enables trace if not desired.

3.2 Design

A variety of design choices motivated the final implementation. A few of those are outlined here.

The first key decision was how to handle sending and receiving messages. Many network architectures can be modeled as a server and client, where the server starts listening for connections, a client connects, then the two communicate over that single connection. It would be straightforward and possibly reduce errors if there was a client version and a server version of the middleware. For the first couple examples, this would work fine. Instead, the middleware acts as both a client and a server. It initializes a server socket that listens for incoming connections and creates a new socket for each connection received, as well as create a client socket for each process it sends to. As a result, for two nodes to communicate, they require opening two connections. However, the middleware does not need to remember what nodes have connected to it, and the user does not have to differentiate between a server and client node.

A more extreme variation of the previous design is to not only create two connections between every pair of nodes, but also create a socket for every single message sent that lives only for the duration of the message. This requires an absolute minimal amount of storage by the middleware, since it does not even need to store any sockets. However, this does not give any added benefit to the user, and could possibly have too much overhead for systems with many messages.

Only strings can be sent over a socket, not generic message objects. So, there needs to be a way to convert from strings to messages and from messages back to strings. For this, the `QID-LIST-STRING` module from `mobile-maude-centrailzed.maude` is used, which can convert a string to a list of quoted identifiers and vice versa. To convert a message to a string, the `META-LEVEL` module is used to parse the message to a list of quoted identifiers, which are then converted to a string. A reverse process is used for the other direction:

```
eq to-string(M) = qidListString(metaPrettyPrint(
  upModule('MESSAGE, false), upTerm(M), none)) .
eq to-msg(S) = downTerm(getTerm(metaParse(
  upModule('MESSAGE, false), stringQidList(S), 'Msg)),
  error(S)) .
```

A module must be specified for the metalevel to use to parse. Here, the `MESSAGE` module is used, since messages are being converted. In general, there will be user specified objects that need to be converted, and somehow the middleware needs to know their syntax. To do this, the middleware will assume a

module called INTERFACE is already defined when it is loaded. This interface is required for specifying any user-defined syntax for messages; in particular, any syntax that is required to specify object ids (sort Oid) or message contents (sort MsgCont). The MESSAGE module protects this module, which is then protected by MESSAGE-EXT which handles conversion.

3.3 Implementation

As described before, the MESSAGE module protects an INTERFACE module that should already be loaded. This INTERFACE module should extend the CONFIGURATION module in the prelude that defines objects, configurations, and messages. This MESSAGE module with the QID-LIST-STRING module described before are used by the MESSAGE-EXT module that allows converting between messages. Lastly, the LOC module defines the sort Loc and the operator ip-port, which takes in an IP as a string and a natural number as a port to create a Loc. This Loc is used to define where a socket should be created to send a message.

The bulk of the interesting work is in the MW module, which defines the Mw class. The middleware has three attributes: obj-to-loc, loc-to-socket, and building-socket. obj-to-loc is a map from object ids to locations, and is static and specified by the user when they initialize the middleware. This map tells the middleware where the destination object is located. loc-to-socket is a map that takes in a location object and returns the Oid of a socket. This starts as empty, but as sockets are created to send messages, they will be stored here to avoid having to recreate them. Note that this two-map design allows for messages that are destined for different objects at the same location to reuse the same socket. Lastly, the building-socket attribute contains either a location or none, and indicates whether a socket is being built. For simplicity, only one socket can be constructed at a time.

Some of the rules are described below. They will use the following variable set:

```
var Motl : Map{Oid, Loc} .
var Mlts : Map{Loc, Oid} .
vars ServerSocket Socket Client O : Oid .
vars IP DATA : String .
var N : Nat .
var L : Loc .
var M : Msg .
var A : AttributeSet .
```

To initialize the middleware, the user must pass in a port to run it on and a mapping of objects to locations. It opens a server socket and starts as a Mw-no-server object, so no other rules apply until the server socket was opened successfully. It also creates a portal to allow communicating:

```
rl [init-mw] :
  init-mw(N, Motl) =>
  <>
  < mw : Mw-no-server | obj-to-loc : Motl, loc-to-socket : empty,
    building-socket : none >
  CreateServerTcpSocket(socketManager, mw, N, 5) .
```

The rules for accepting messages are relatively straightforward. This server socket continually accepts clients. When a client is accepted, it starts accepting for another client and receives on the client accepted. Note that buffered sockets are used, so one `Receive` is guaranteed to correspond to one message, regardless of message length. When data from a client is received, the data is converted to a message and added to the pool, and the middleware listens to the client for more messages:

```
rl [received] :
  Received(mw, Client, DATA)
  < mw : Mw | A > =>
  < mw : Mw | A >
  Receive(Client, mw)
  to-msg(DATA) .
```

The rules for sending messages are a little more complicated, since they need to keep track of open sockets. If a message `M` is in the pool, and the destination of that message is mapped to a location that maps to a socket, then that message is converted to a string and sent:

```
crl [consume-message-socket-exists] :
  M
  < mw : Mw | obj-to-loc : (O |-> L, Motl), loc-to-socket :
    (L |-> Socket, Mlts), A > =>
  < mw : Mw | obj-to-loc : (O |-> L, Motl), loc-to-socket :
    (L |-> Socket, Mlts), A >
  Send(Socket, mw, to-string(M))
  if (dest(M) == O) .
```

If no socket is being made and there is a message in the pool where the location of the destination does not have a socket built yet, then that socket starts being created:

```
crl [build-socket] :
  M
  < mw : Mw | obj-to-loc : (O |-> L, Motl), loc-to-socket : Mlts,
    building-socket : none, A > =>
  < mw : Mw | obj-to-loc : (O |-> L, Motl), loc-to-socket : Mlts,
    building-socket : L   , A >
  CreateClientTcpSocket(socketManager, mw, ip(L), port(L))
  M
  if (dest(M) == O) /\ (Mlts[L] == undefined) .
```

No other sockets will be created during this time. Once that socket is created, it is saved for future use:

```
rl [created-socket] :
  CreatedSocket(mw, socketManager, Socket)
  < mw : Mw | loc-to-socket : Mlts, building-socket : L, A > =>
  < mw : Mw | loc-to-socket : (Mlts, L |-> Socket),
    building-socket : none, A > .
```

4 Factorial Server

4.1 Overview

A typical setup is a client requesting data from a server. In this example, a client requests the factorial of a number from a server. If the server has already computed the result, it returns it. Otherwise, it computes the result, stores it, and returns it. The server starts with an empty memory. To start a server and two clients that are requesting 100!, run

```
frew init-client(1, 100, ser) init-client(2, 100, ser)
  init-server .
```

The server will only compute the result once.

4.2 Modeling

The original code can be found in `fact/original.maude`, and contains both the client and server code. The client code only has one rule, for initialization. The server uses the modules `MEM` and `FACTORIAL` to store the results and compute the factorial respectively. `MEM` defines the sorts `Pair` and `Mem`. A `Pair` is a mapping of one natural to another, and `Mem` is a associative and commutative set of pairs with identity `nomem`. It has two rules, `[cache-result]` and `[send-response]`.

4.3 Distributed Implementation

First, the server and client code are split from each other, and placed into `ex-server.maude` and `ex-client.maude`. While not strictly necessary, this makes the example slightly more realistic, since a client and server running on separate machines should not require the other to operate. Also, the `MESSAGE` module is removed, since the code would be replicated in the middleware.

Next, the `INTERFACE` module is created in `interface.maude`. This module is responsible for including any syntax that may go into a message being sent through the middleware, in particular, object ids and message contents. It must extend `CONFIGURATION` and define the sort `MsgCont`:

```
mod INTERFACE is
  ex CONFIGURATION .
  pr NAT .

  sort MsgCont .
  op i : Nat -> MsgCont [ctor] .
  op cli : Nat -> Oid [ctor] .
  op ser : -> Oid [ctor] .
endm
```

Lastly, the extra modules `SERVER-EXT` and `CLIENT-EXT` are added to include equations to initialize the middleware with the server and client. For this example, there are two clients running on ports `8888` and `8890`, while the server runs on port `8889`. All of the examples will only be shown running on localhost for ease of demonstrating, but have also been tested on separate computers. The server needs to know the location of both the clients, while the clients only need to know the location of the server:

```
mod SERVER-EXT is
  pr SERVER .
  pr MW .
  op init : -> Configuration .
  eq init = init-server init-mw(8889,
   (cli(1) |-> ip-port("localhost", 8888),
    cli(2) |-> ip-port("localhost", 8890))) .
endm
```

```
mod CLIENT-EXT is
  pr CLIENT .
  pr MW .
  op init1 : -> Configuration .
  op init2 : -> Configuration .
  eq init1 = init-client(1, 100, ser)
    init-mw(8888, ser |-> ip-port("localhost", 8889)) .
  eq init2 = init-client(2, 100, ser)
    init-mw(8890, ser |-> ip-port("localhost", 8889)) .
endm
```

To run this example, start three Maude processes on the same machine. In one of them, load ex-server, and in the other two load ex-client. In the server, run `erew init`, and in one of the clients run `erew init1` and in the other `erew init2`.

5 Two-Phase Commit

5.1 Overview

Two-phase commit, or 2PC, is a commit procedure for a network that ensures a change is made to a system only if every node can accept it. There is a single

coordinator and multiple cohorts. During the voting phase, the coordinator sends a request to all of the cohorts to vote yes or no, and each of the cohorts responds. During the commit phase, either the coordinator sends a rollback request to all the nodes, or tells the nodes to commit the changes. In either case, the cohort responds once it has done the appropriate action. This example is mostly adapted from the *Formal Modeling and Analysis of Distributed Systems* reference.

5.2 Modeling

The model is found in `2pc/original.maude`. There are four modules: `INTERFACE`, `COORDINATOR`, `COHORT`, and `2PC`. While there are quite a few rules, they are straightforward and commented with their task. The `INTERFACE` module contains the types of messages that can be sent as well as a special `multicast` message. The possible message contents are `QueryCommit`, `VoteYes`, `VoteNo`, `Commit`, `Rollback`, and `Ack`. The first three are used for the voting stage and the latter three are used for the commit stage. Multicasting is a useful pattern where an object can generate a message for a variety of nodes at once by evaluating out to a set of messages for each object in the destination set. Originally, it was going to be a a part of the middleware tool, but this would have required the user to understand the middleware specification more than desired. To define multicasting, a `OidSet` is also defined:

```
sort OidSet .
subsort Oid < OidSet .
op multicast_from_to_ : MsgCont Oid OidSet -> Configuration .
op none : -> OidSet [ctor] .
op _,_ : OidSet OidSet -> OidSet [ctor assoc comm id: none] .
var M : MsgCont .
vars O O2 : Oid .
var OS : OidSet .
eq multicast M from O to none = none .
eq multicast M from O to (O2 , OS) = (msg M from O to O2)
  multicast M from O to OS .
```

The `COORDINATOR` module defines the `Coord` class. A coordinator can be in six phases: `Start` for initialization, `CommitRequest` during the voting phase, `CommitSend` and `RollbackSend` for the commit phase, and `Failed` and `Success` for after the commit phase. Other than phase, a coordinator has the attributes `cohorts` and `receivedACK`. The attribute `cohorts` is just the set of cohort object ids, and `receivedACK` is the set of cohort object ids that have voted yes or sent an acknowledgement.

The `COHORT` module defines the `Cohort` class. The only attribute is `DB` which can be `Start`, `Prep`, or `Commit`. The database starts in the `start` state, transitions to `prep` after the voting phase, then either transitions to `start` or `commit` depending on whether the coordinator decided to rollback or commit.

5.3 Verification

There are two basic properties to check. First, it should be impossible for a cohort to end in a `Prep` state, and it should be impossible for two cohorts to ever be in a `Commit` and `Start` state simultaneously. These can be checked respectively by ensuring the following commands have no solutions:

```
search [1] init(3) =>! C:Configuration
  < O:Oid : Cohort | DB : Prep > .
search [1] init(3) =>* C:Configuration
  < O:Oid : Cohort | DB : Commit >
  < O2:Oid : Cohort | DB : Start > .
```

5.4 Distributed Implementation

As in the factorial example, the first step is to split the code into the cohort and coordinator code, which are in `cohort.maude` and `coord.maude`. Next, the `INTERFACE` is extracted out into its own file. In it, only the operators required for parsing a message are included:

```
mod INTERFACE is
  ex CONFIGURATION .
  pr NAT .
  sort MsgCont .
  op cohort : Nat -> Oid [ctor] .
  op coord : -> Oid [ctor] .
  ops QueryCommit VoteYes VoteNo Commit Rollback Ack :
    -> MsgCont [ctor] .
endm
```

The multicast code was moved into the coordinator file, since only it used multicast. The syntax definition for a message was deleted, since that is defined in the middleware. The `2PC` module to handle initialization was also deleted, since each file needs to be initialized differently. To use the interface, both the `coord` and `cohort` files needs to load the`interface` file. After that, the middleware should be loaded. Any modules that use the message syntax should protect the `MESSAGE` module.

Lastly, the middleware needs to be initialized. The original file used a `2PC` module that handled initialization. Instead, the `COORDINATOR-INIT` and `COHORT-INIT` modules were defined in their respective files. They initialize the coordinator on port `8800` and three cohorts on ports `8801`, `8802`, and `8803`:

```
mod COORDINATOR-INIT is
  pr MW .
  pr COORDINATOR .

  op cohorts : -> OidSet .
  eq cohorts = cohort(1), cohort(2), cohort(3) .
  op cohorts-map : -> Map{Oid, Loc} .
  eq cohorts-map = cohort(1) |-> ip-port("localhost", 8801),
    cohort(2) |-> ip-port("localhost", 8802),
    cohort(3) |-> ip-port("localhost", 8803) .
  op init-coord : -> Object .
  eq init-coord = init-mw(8800, cohorts-map)
    < coord : Coord | phase : Start, cohorts : cohorts,
      receivedACK : none > .
endm
```

```
mod COHORT-INIT is
  pr MW .
  pr COHORT .
  var N : Nat .
  op init-cohort : Nat -> Object .
  eq init-cohort(N) =
    init-mw(8800 + N, coord |-> ip-port("localhost", 8800))
    < cohort(N) : Cohort | DB : Start > .
endm
```

The coordinator is started by rewriting `init-coord`. A cohort is started by rewriting `init-cohort(N)`, where `N` ranges from `1` to `3`. All cohorts must be started before the coordinator, as the coordinator will try to send them messages.

6 Raft

6.1 Overview

The most interesting example to consider is Raft, a distributed consensus algorithm recently proposed by Ongaro and Ousterhout. Essentially, a complete network of nodes wants to maintain some consistent internal state machine, such as a database. A command sent to any of the nodes should be replicated across the servers. Raft has never been specified or verified in Maude, and was originally designed to be a simpler and easier to understand variation of Paxos, making it a prime candidate to try to specify in Maude. In this specification, some assumptions were made to make Raft easier to model and verify, which will be described later. More details, including design choices and proof of correctness, can be found in the paper "In Search of an Understandable Consensus Algorithm". The three key sections are leader election, log replication, and safety.

Every node is able to communicate with every other node and is either a leader, a follower, a candidate, or an offline node. Each node maintains a log, or ledger, which is a list of index, term, and command triplets. In Maude, this is modeled as:

```
sorts Command Entry Ledger .
subsort String < Command .
subsort Entry < Ledger .
op entry : Nat Nat Command -> Entry [ctor] .
op empty : -> Ledger [ctor] .
op _;_ : Ledger Ledger -> Ledger [ctor assoc id: empty] .
```

The index and terms are monotonically increasing counters from 0. Index increases with each entry added, and term is only increased when an election occurs. Each message between nodes is tagged with the term of the sender, so out-of-date nodes and messages can be identified. Essentially, whenever a node receives a message from a leader with a higher term, it immediately becomes a follower. In normal operation, the leader periodically sends heartbeat messages out to all the other nodes, including the index and term of the leader's most recent entry. The follower responds with whether it already has that entry or not. If not, the leader and follower communicate until they find the last entry where they agree, and the leader sends all the entries past that point and the follower overrides existing entries until they are in sync.

When the leader goes offline, one of the follower nodes will timeout and start an election by increasing its term, becoming a candidate, voting for itself, and sending a vote request to all the other nodes. During any term, a node will only vote once, on a first-come-first-serve basis. The last entry log is sent with this vote request, since there is a requirement that a node will only vote for another node if the candidate's log is at least as up to date as the voter's log. A log is more up to date than another log if its latest entry has a higher term or has the same term and a higher index. In Maude:

```
op compare-log : Entry Entry -> Bool .
eq compare-log(E1, E2) = term(E1) < term(E2) or
  (term(E1) == term(E2) and index(E1) <= index(E2)) .
```

A candidate wins the election if it receives a majority of votes. If it starts receiving heartbeats from a leader with a higher term, it becomes a follower. If it times out, and does not receive a majority of votes such as by a split vote, then it will start a new election.

Once an entry of the current term is contained on a majority of nodes, it can be shown that it will always remain in the network, due to the way elections are handled. The leader should try to identify such entries and commit them, so they are applied to the state machine and their result returned if necessary. During the heartbeat process, the latest message in each follower log is known, and so entries in a majority of nodes can also be known. This furthest committed entry information is passed to the followers in the heartbeat, so the followers

can apply it and all previously uncommitted commands to their state machine as well.

6.2 Simplifications

The complete raft specification relies heavily on the use of timing. Every node contains an election timer, and if that timer ever reaches zero, it will start a new election by notifying all the the other nodes and voting for itself. This timer resets every time the node receives a heartbeat message from the leader. So, if the leader goes down, a node will start an election.

Raft mitigates the risk of multiple nodes starting elections and voting for themselves by randomly selecting the timer duration. This immediately poses two difficulties for Maude: randomization and timing. Maude has some ways to model timing, but no built-in way to use an actual timer needed for implementation. Similarly, Maude can generate random numbers, but this poses a massive difficulty for verification, since it is not feasible to search every possible random result. Additionally, randomized election timeouts help avoid split votes. If there are two candidates for a term, the vote could be split, and neither win. To limit the risk of the split vote repeating, each candidate waits a random amount of time before deciding to start another election. This is difficult to model for the same reason as before.[1]

To circumvent this, there is no heartbeat system in the model, and only one candidate is possible at a time. Instead, when a leader goes offline, it will notify its neighbor to start the election process. If that neighbor fails to become leader, it will tell the next neighbor, and so on. There are other drawbacks to removing the heartbeats, which were also responsible for syncing log entries of followers and specifying entries to commit. Instead, when a leader adds an entry to its log, it immediately sends its entire logs to all the followers, and the follower simply sets it log to be the one sent. The heartbeat system would eventually sync up the logs, just with more messages of smaller size. The large number of messages would cause verification slowdown, so this system works more effectively. The leader waits to hear back from all the nodes before deciding whether to commit. If a majority of nodes respond affirmatively, indicating that they updated their log, the leader stops waiting and sends a commit message to all other nodes. Otherwise, it repeats until a majority of nodes respond affirmatively.

Removing dropped messages is the last key abstraction. If a node goes down and cannot respond to messages, this will be learned by the system due to time. For example, the leader will know that a follower went down because it is not responding to heartbeats. To remove this time aspect, instead when a node goes down, it will respond in a negative way and will not update its state. For example, if an offline node is told to become leader, it will simply pass the

[1] Modeling time and randomization in Maude is possible using probabilistic rewrite rules, described in [1]. It is also possible to perform statistical model-checking verification for Maude modules with probabilistic feature using the PVeStA tool [2]. However, the generation of distributed implementations from their probabilistic Maude specifications is outside the scope of this Thesis.

message to its neighbor. If it is asked to vote in an election, it just votes no.

6.3 Modeling

Under these simplifying assumptions, Raft is then modeled in Maude. This section will lay out many of the important aspects. The full specification of the provided code can be found in `raft.maude` . In total, there are ten modules. These define the nodes and a special object called the client that is responsible for coordinating the deaths of nodes and sending queries. While the nodes can communicate as a complete graph, they also have connections to one of their neighbors to form a ring, which will be useful for sending a message to exactly one recipient. For example, no matter which node receives a query from the client, it will pass the query around until the leader receives it and applies it. This section will go over these modules and highlight a couple of the rules.

The `LEDGER` module defines the sorts `Command`, `Entry`, and `Ledger`. Here, a command is just a string, although in the future it could represent a more complicated object, such as a database command. An entry is a triplet of a term, index, and a command, and a ledger is a sequence of entries. This module also defines some helper functions for accessing parts of the ledger and entry.

The `MESSAGE` module defines the kinds of messages that can be sent to each other with the `MsgCont` sort. `Query`, `Die`, or `Live` comes from the client and tells a node to add a command, go offline, or come back online. The rest of the commands are tagged with the term of the sender. `SetLog` goes from leader to follower and tells a follower to update its log, which then responds with `SetLogResponse`. `Commit` tells a follower to commit all entries up to a specified entry. `BecomeLeader` tells a node to become a leader for a specified term. A candidate sends `RequestVote` out to all nodes, which then responds with `Vote`. The `MULTICAST` module is a helper module to send a message to multiple recipients at once.

The `NODE` module defines class names, object names, and attributes of all the nodes, as well as an equation for cleaning up older messages. The class names are `FollowerNode`, `LeaderNode`, `OfflineNode`, and `CandidateNode`, and the only object names are `node(N)` for any natural number `N`. The class of an object changes as the state of the object changes. Each node has some attributes that it maintains; note that some are not relevant unless in the right state. The `currentTerm` attribute stores the current term for the node. The attributes `log` and `committed` store the current ledgers that have been received and committed respectively. Here, `committed` is also serving as a kind of state in a basic state machine, where the only operation is to add an entry. The `neighbors` attribute store all other nodes in the graph. The `waiting` attribute is a `Bool` indicating whether the node is waiting for some responses back, such as an election result. The attribute `next-neighbor` is the neighbor of the node when the nodes are treated as a ring. Lastly, the attributes `majority`, `number-neighbors`, `number-yes`, and `number-response` all keep track of any election or vote results.

`LEADER` defines a leader node. Primarily, a leader receives queries, sends

them to the followers, gets a response, then commits. The query rule is shown below. When a leader receives a query and is currently not waiting for any election results, it increases its index by 1, adds its entry to its log, broadcasts the `SetLog` message to its neighbors, then goes to waiting until it hears back:

```
crl [query-leader] :
  (msg Query(C) from cli to lea)
  < lea : LeaderNode | currentTerm : term, log : led ,
    neighbors : fols, waiting : false, number-yes : N1,
    number-response : N2, AS > =>
  < lea : LeaderNode | currentTerm : term, log : led',
    neighbors : fols, waiting : true , number-yes : 1 ,
    number-response : 0 , AS >
  (multicast SetLog(term, led') from lea to fols)
  if led' := led ; entry(index(head(led)) + 1, term, C) .
```

When the client tells the leader to die, it responds with `DIE` to let the client know it is offline, becomes offline, and tells the next node to try to become a leader:

```
rl [die] :
  (msg Die from cli to lea)
  < lea : LeaderNode  | currentTerm : term,
    next-neighbor : fol, AS > =>
  < lea : OfflineNode | currentTerm : term,
    next-neighbor : fol, AS >
  (msg Die from lea to cli)
  (msg BecomeLeader(term + 1) from lea to fol) .
```

`OFFLINE` defines the rules for a node that is offline. Importantly, it never updates its internal state. It deletes any unneeded message, responds in the negative for any requests, and passes any queries or leader requests forward to its neighbor.

`FOLLOWER` is the most usual state and also contains the most rules. It passes on any query to its neighbor, so a query will eventually make it to the leader. When it gets a `SetLog` from the leader, it sets its log ledger and returns true. When it receives a `Commit` message, it updates its committed ledger if necessary, by committing all entries in its long up to the received entry. If the log does not contain the entry or the committed log already contains the entry, then it does nothing (except update the term if needed):

```
crl [cant-commit-follower] :
  (msg Commit(new-term, E) from lea to fol)
  < fol : FollowerNode | currentTerm : term,
    log : led, committed : comled, AS > =>
  < fol : FollowerNode | currentTerm : new-term,
    log : led, committed : comled, AS >
  if (new-term >= term) and ((not contains(led, E)) or
    (contains(comled, E))) .
```

Because a leader can go offline and come back as a follower in the same term, it needs to delete any excess `SetLogResponse` or `Vote` messages that might remain in the pool. A follower can become a candidate and broadcast out request votes. Lastly, when a follower receives a `ReqestVote` from a candidate, it votes by comparing the latest entry sent to it with the latest entry in its own log:

```
crl [follower-vote] :
  (msg RequestVote(new-term, E2) from lea to fol)
  < fol : FollowerNode | currentTerm : term,
    log : led ; E1, AS > =>
  < fol : FollowerNode | currentTerm : new-term,
    log : led ; E1, AS >
  (msg Vote(new-term, compare-log(E1, E2)) from fol to lea)
  if (new-term >= term) .
```

Next, the `CANDIDATE` module defines the rule for a candidate. When created, a candidate will send out request votes to all nodes. It tallies the votes that it receives. Once all the votes are collected, if a majority voted affirmatively, then the candidate becomes a leader and sends a `SetLog` to all nodes to get them up to date. If it fails, then it notifies its neighbor to start the election process. If it is sent `DIE` by the client, then it goes offline and tells the next node to become a leader.

`NODE-INIT` contains all the equations and syntax needed for initializing a ring of nodes and an individual node. Reducing `node-init(i, k)` initializes a node of index `i` in a ring of `k` total nodes. Reducing `init-ring(k)` initializes a ring of k nodes by initializing each node in the ring:

```
op init-ring : Nat -> Configuration .
op init-ring : Nat Nat -> Configuration .
eq init-ring(N) = init-ring(0, N) .
eq init-ring(N, N) = none .
eq init-ring(N1, N2) = init-node(N1, N2)
  init-ring(N1 + 1, N2) [owise] .
```

Four helper functions are used to define a node: `make-neighbors`, `add-one-modulo`, `get-majority`, and `get-minority`. The function `make-neighbors` generates a set of all other node object ids in the pool. The function `add-one-modulo` adds one to the index of the node modulo the total number of objects, which is used to make the ring structure. The function `get-majority` just returns what a majority means for some number, so returns 3 for 5, 4 for 7, and so on. Lastly, the function `get-minority` returns a minority for a number, so 2 for 5, 3 for 7, and so on. When a node is initialized, it starts in term 0 as a follower with a dummy entry at the start of it log and committed ledgers.

The last module `SIMUL` defines a client object that handles sending requests to the network and coordinating node deaths, since only a minority of nodes should be offline at any time. To allow the execution to terminate, it also

contains a `gas` attribute that stores how many more nodes can fail and come back online during the lifetime of the execution. A client maintains what a minority of nodes is, what object ids are currently known alive, and what object ids are currently known dead. At any point, the client can tell a known alive node to die, and a known dead node to come online. Once it receives confirmation from the object that it has updated its state, it marks the object as known dead or alive correspondingly. If a minority of nodes are already not known alive, it will not kill any additional nodes. Every time a node is told to go offline, `gas` is decreased by 1, and no more nodes will be sent offline if `gas` is 0. When the client is initialized, three queries (`"cmd1"`, `"cmd2"`, and `"cmd3"`) are sent to node 0 at the same time, as well as a request for node 0 to become leader:

```
op init-client : Nat Nat -> Configuration .
eq init-client(s N, g) =
  (msg BecomeLeader(1) from client to node(0))
  (msg Query("cmd1") from client to node(0))
  (msg Query("cmd2") from client to node(0))
  (msg Query("cmd3") from client to node(0))
  < client : Client | gas : g, num : get-minority(s N),
    live : make-neighbors(s N, N), fail : none > .
```

To initialize the model, the client is initialized and the ring is initialized. The first parameter is the number of nodes and the second is the amount of gas.

```
op init : Nat Nat -> Configuration .
eq init(N, g) = init-client(N, g) init-ring(N) .
```

One additional step was taken, called abstraction by invisible transitions, where some subset of rules are replaced with equations to reduce the number of states [6]. This will allow fewer states to be explored during search, but requires proving that the modification will not hide any important details. During rewriting, all terms are reduced with equations before any rules are applied. Ideally, rules capture any non-deterministic aspect, since a term can be rewritten using any matching rule. This comes at the cost of additional states during the search in verification.

For example, one rule is that a node will simply delete any message intended for it that is from a previous term. A rule will only apply to an object and message if the term of the object is at most the term of the message. But because the term of any object is increasing, a message from a previous term will never be relevant, and so can safely be deleted. If this consumption is modeled by a rule, it introduces non-determinism where it is not necessary, since it does not really matter when the message is consumed as its consumption will always be side effect free. Making this consumption an equation reduces the number of states, since every combination and order of message consumption does not need to be considered. The expression `get-term(cont(M))` gets the contents of a message M and then gets the term number of those contents:

```
ceq
  (M)
  < O : C | currentTerm : N, AS > =
  < O : C | currentTerm : N, AS >
  if N > get-term(cont(M)) /\ dest(M) == O .
```

6.4 Verification

Verification in Maude is relatively straightforward using the `search` command, and this simplicity is the motivation behind using Maude. With proper verification, the middleware allows a simple, correct by construction implementation. More extensive verification is certainly possible than what is done here. This is mostly to demonstrate the concept. The `search` command will perform a breadth-first-search from the starting state to a matching end state that satisfies some condition. Verification is not only desired for certain Raft guarantees, but also as a helpful debugging tool.

For example, one basic desired property is that in any state reached from a starting state, if there is a message in the soup, more rewrites should be possible. This is not a special requirement for Raft, but instead something that should be true for this implementation. The following command checks this:

`search [1] in SIMUL : init(3, 1) =>! M:Msg C:Configuration .`

`[1]` causes the search to stop after it finds any matching state, rather than finding all of them. `SIMUL` is the name of the module being tested. `init(3, 1)` is the starting state, and expands out to a Raft configuration containing three nodes where 1 failure is allowed to occur at any point in time. The exclamation in `=>!` indicates that only final states should be considered. Lastly, a configuration will match with `M:Msg C:Configuration` if there is any message in the state. This takes a couple hours to run on a regular desktop computer, but returns that there are no solutions. While developing the implementation, the first few times this ran, solutions were found, indicating bugs in the preliminary versions. The `show path` command in Maude with the state number for the solution will show the rewrites that caused that state to be reached, making it a very useful debugging tool. This particular example of model-checking verification helped find messages that were not being consumed correctly.

The three key properties to verify are election safety, log matching, and state machine safety. Note that each of these are being verified in the `init(3, 1)` initial state described before, where there are three nodes with 1 failure possible. While larger states give more trust that the system is correct, due to the state space explosion caused by concurrency the runtime becomes unwieldy. Election safety checks that there can only be one leader at a time, and is simply checked by looking for any configuration (final or not) with two leader objects:

```
search [1] init(3, 1) =>* C:Configuration
  < O1:Oid : LeaderNode | AS1:AttributeSet >
  < O2:Oid : LeaderNode | AS2:AttributeSet > .
```

As desired, this returns no matches. Log matching checks ensures if two

logs are the same at some index and term, then all logs up to that point are equal. State machine safety ensures that once a particular entry is committed, no other node can commit a different command for that entry and term. Log matching can be checked with the following command, and replacing "log" with "committed" can check state machine safety:

```
search [1] init(3, 1) =>* C:Configuration
  < O1:Oid : C1:Cid | log : L11:Ledger ;
    entry(ind:Nat, term:Nat, C1:Command) ; L12:Ledger,
    AS1:AttributeSet >
  < O2:Oid : C2:Cid | log : L21:Ledger ;
    entry(ind:Nat, term:Nat, C2:Command) ; L22:Ledger,
    AS2:AttributeSet >
  such that C1:Command =/= C2:Command or
            L11:Ledger =/= L21:Ledger .
```

In every state, this checks for every pair of objects and for every matching index and term in their log whether the command for that entry is different or whether any previous entry is different. Because no solutions are found, log matching is satisfied. Swapping `log` with `committed` shows state machine safety as well.

6.5 Distributed Implementation

Given the verified model, very few steps are actually required to turn this into a distributed implementation when using the middleware. Even though the model is quite complex, no more actual work is needed to make it use the middleware, as long as the original source is designed well. Copy the model into its own file, `node.maude`, and apply the steps.

The first step is extracting out the interface. The syntax required for serializing a message - the contents, labeled `MsgCont`, as well as object ids - are put in their own file `interface.maude` in a module `INTERFACE`, and the original lines are deleted. `load interface` is added to the top of the file. Because the syntax was deleted from the original modules, any module using the syntax previously needs to add `pr INTERFACE .` to continue using the syntax. It is safe to protect the syntax, since no equations or rules are declared in the interface.

Next, the middleware needs to be initialized. Add `load ../mw/mw` (or whatever the path to the middleware file is) to the top of the file. Make sure it is loaded after the interface, since the middleware requires the interface module to be loaded already. The middleware needs to be initialized at the time that the object is initialized. Recall that the middleware requires a port for the server to run on, and then a mapping of object ids to locations, given by IP address and port numbers. Rather than hard-coding the mapping, it will be easier to define helper functions that implement these maps. Port `8700` will be the port where the client runs, and `8800 + N` will be the port that node `N` runs on. The module `IP-MAP` will define `remove-from`, that takes an object and map and removes the object from the map, and `get-ip-map`, which takes a natural for the number of

nodes and returns the map where the client maps to port 8700 and each node maps to its corresponding port on localhost:

```
mod IP-MAP is
  pr MW .
  var N : Nat .
  var O : Oid .
  var M : Map{Oid, Loc} .
  var L : Loc .
  op remove-from : Oid Map{Oid, Loc} -> Map{Oid, Loc} .
  eq remove-from(O, (M, O |-> L)) = M .
  op get-ip-map : Nat -> Map{Oid, Loc} .
  eq get-ip-map(0) = client |-> ip-port("localhost", 8700) .
  eq get-ip-map(s N) = get-ip-map(N), node(N) |->
    ip-port("localhost", 8800 + N) .
endm
```

The middleware should not try to send objects to its own IP, and so this object should be removed from the map. So, for a node with index `i` and total nodes `N`, the middleware should be initialized as:

```
init-mw(8800 + i, remove-from(node(i), get-ip-map(N)))
```

Similarly, for a client for `N` nodes, it should be initialized as:

```
init-mw(8700, remove-from(client, get-ip-map(N)))
```

Lastly, the `init-node` and `init-client` code is modified to also create the middleware. The unneeded code for `init-ring` is also removed. To test running locally, start multiple Maude processes with `node.maude` loaded. In three of them, execute `erew init-node(N, 3)`, where `N` goes from 0 to 2. Note that the nodes have to be initialized first, so the client can send messages. In one of them, after the nodes are initialized, run `erew init-client(3, 2)`, to start a client that can communicate with the three nodes and allows two nodes to fail. The sockets should send and receive messages until no more rewrites are possible, at which point they will each contain some log and committed entries that are consistent with each other.

7 Summary and Conclusion

7.1 Summary of Methodology

Creating a semi-automatic methodology for transforming a Maude model into a distributed implementation is important to minimize the number of implementation decisions the user has to make which can introduce bugs. While each model will presents its own challenges when converting to a distributed implementation, the steps can broadly be categorized as follows:

1. Choose a distributed algorithm to implement. Keep in mind that those with randomness or timing may be more difficult to verify. Consider

changing certain aspects of the algorithm to reduce the number of states, such as reducing the number of messages sent.

2. Formally define the model of the distributed algorithm in Maude. Follow the concurrent object programming pattern by extending `CONFIGURATION`, where objects communicate using messages to each other. For simplicity, ensure messages are formed by the operator `msg_from_to_` or `msg_to_`where the contents of a message have sort `MsgCont`, as this will help when adding the middleware later. It will also be useful to define a separate module for initializing any necessary objects. Placing this in its own module will help because this code is likely to be changed the most when creating the distributed implementation. Avoid the use of `#` in anything that may be a part of a message, as this can break the middleware because of its use of buffered sockets.

3. Verify important properties of the model using model-checking in Maude. This can be done using the `search` command, which searches from some starting state for some final or intermediate state that satisfies some property. For verifying LTL temporal logic properties, it is likewise possible to use the Maude LTL model-checker [4]. For models with large numbers of states, consider changing initialization parameters to reduce the number of states. For example, reduce the number of nodes in the network, or for models simulating failure, reduce the number of failures that can occur.

4. Optionally, split the source code into separate files so that each process will only have the rules and operators that it requires. For example, in a client-server architecture, split the client code and server code into their own files. This will allow each one to be modified independently more easily.

5. Extract the interface into its own file, typically called `interface.maude`. This interface should define the sort `MsgCont` and include all syntax necessary to parse any message. It should also extend the CONFIGURATION module, which defines the `Oid` sort. Hence, all operators that may be included in a message content or object id should be moved to this module. Modify the existing code to load this file and include the module where necessary.

6. Load the middleware file after the interface is loaded. This middleware file defines a `MESSAGE` module that defines the message syntax that should be used. In step 2, the model should have defined the operators `message_from_to_` and/or `message_to_` for objects to use to send messages to each other. However, the middleware code also defines these operators, so these previously defined operators should be deleted to avoid confusion, and any modules that use this syntax should protect the `MESSAGE` module that was loaded with the middleware. This step should be done for each file if the source code is split over multiple files.

7. Rewrite the initialization code to also initialize the middleware. In the initialization module, protect the module `MW`. The middleware needs to be initialized with the port it should listen on and a map from object ids to locations, defined by an IP and a port. To test locally, just use the IP of `"localhost"` for each object.

8. To run the distributed implementation, start a Maude process for each node in the network. Load the file containing the code by using `load filename`. Use the `erew` command to rewrite the initialization state that was defined. Make sure the nodes are started in an order so that no node tries to communicate with a node that is not started yet, such as a client trying to connect to a server that is not running, as this will cause errors.

See appendix A for an example of a model of ring-leader election trasformed to a distributed imlementation in detail following these steps. There are some other useful ideas to keep in mind:

- Rework the algorithm to avoid timing and randomization if it becomes a verification issue. Even though Raft makes use of timing and randomization, this thesis developed a Raft model without the use of either by making some observations about their role and changing the algorithm to not require them.

- Use port numbers that correspond to object ids. For many of these examples, each object was given an index as some natural number. For example, when running locally, the port number for object with index `N` would be `8800 + N`. This ensures that multiple objects do not try to open the same port.

- Use verification also as a debugging tool. While a distributed algorithm may have certain guarantees for correctness, a particular model of that algorithm may have additional properties that should be true. For example, the Raft model in this thesis had the property that there should be no messages in the pool in any final state, even though Raft itself makes no such guarantee. By failing to verify this property in preliminary models, bugs in those models were exposed. The Maude command `show path` *`state-number`* can show the sequence of rewrites and states that led to some state, further helping debugging.

- Multicasting is a useful pattern where one objects sends a message to a set of nodes. Even though the message operator defined in the middleware can only send a message to one destination, it is a common case that an object wants to send a message to multiple nodes at once. To do this, create an operator `multicast_from_to_` where the destination is a set of nodes, and include equations that cause this to expand to a set of messages.

- Reduce the number of states by using abstraction by invisible transitions. Rules are intended to capture non-determinism in a system, although

sometimes they may cause non-determinism to appear in a system where it is not necessary. For example, in the Raft case study, all nodes should delete messages from older terms. But because term number in nodes is increasing and the term number of a particular message never changes, replacing the rule to delete these older messages with a similar equation does not actually change the outcome of the model, and making this change reduces the number of states since the degree of non-determinism in the model is reduced.

7.2 Future Considerations

For the future, more features of the middleware can be created. For example, the middleware could be expanded to include more than just TCP sockets, such as files. Other objects could directly query the middleware, such as getting a list of all objects that it knows about. The middleware could become less static, so if an object is created in a new location, it can create a new socket for it. If an object moves from one process to another, the middleware could be updated with its new location. Middlewares on different processes could communicate about what objects they know about, to avoid each location having to be initialized with the location of all objects. The middleware could be smarter about how it sends messages, such as sending multiple messages at once or avoiding trying to send a message that has a destination of the location of the middleware.

Additionally, new distributed systems can be implemented, or the Raft model shown here could be greatly extended. Due to not having used any real-time or probabilistic features in the Maude model, the Raft model is somewhat limited. For example, the leader election process is greatly simplified, and issues such as split votes are not considered at all. Developing distributed systems that use randomization or timing would help create more realistic and usable implementations.

7.3 Conclusion

This thesis serves as an introduction to modeling, verifying, and implementing distributed algorithms in Maude. It is essential to verify distributed algorithms, and modeling in Maude and then using the methodology and middleware described here to transform the model into a distributed implementation minimizes the risk of bugs being introduced. Through four case studies, the patterns and design of the middleware were demonstrated, as well as patterns of creating distributed systems in Maude.

References

[1] Gul Agha, José Meseguer, and Koushik Sen. PMaude: Rewrite-based specification language for probabilistic object systems. *Electr. Notes Theor. Comput. Sci.*, 153(2):213–239, 2006.

[2] M. AlTurki and J. Meseguer. PVeStA: A parallel statistical model-checking and quantitative analysis tool. in Proc. CALCO 2011, Springer LNCS 6859, 386–392, 2011.

[3] Musab AlTurki and José Meseguer. Dist-Orc: A rewriting-based distributed implementation of Orc with formal analysis. In Peter Csaba Ölveczky, editor, *Proc. 1st Intl. Workshop on Rewriting Techniques for Real-Time Systems, RTRTS 2010*, volume 36 of *Electronic Proceedings in Theoretical Computer Science*, pages 26–45, 2010.

[4] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All About Maude - a High-performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic*. Springer-Verlag, Berlin, Heidelberg, 2007.

[5] Jonas Eckhardt, Tobias Mühlbauer, José Meseguer, and Martin Wirsing. Semantics, distributed implementation, and formal analysis of KLAIM models in maude. *Sci. Comput. Program.*, 99:24–74, 2015.

[6] A. Farzan and J. Meseguer. State space reduction of rewrite theories using invisible transitions. In *Proc. AMAST'06*, volume 4019 of *LNCS*, pages 142–157, 2006.

[7] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association.

A An Example of the Methodology: Ring Leader Election

To fully demonstrate the methodology, this section works out the 8 steps outlined in the conclusion for transforming from a Maude model to a distributed implementation.

Step 1: Choose a distributed algorithm to implement. For this, ring Leader Election is used, based on an exercise from the *Formal Modeling and Analysis of Distributed Systems* reference. Leader election involves a network of nodes attempting to designate one of them as a leader, usually satisfying some criteria. This is a common mechanism in distributed networks, and a different one was looked at in Raft. In ring leader election, each node only knows about one other neighbor. Each node has a value associated with it, and the node with the highest value should become leader and let all other nodes know.

Step 2: Formally define the model of the distributed system in Maude. This model can be found below, or in the provided code in `leader/original.maude`. In ring leader election, one of the nodes receives a message to start the election, which passes its own identifier and value to the next node. Each node compares

the value to its own value, and decides to propagate the previous candidate or itself. After the first pass around the ring, it passes a second time around the ring, and each node updates its known leader with the leader passed around.

The `NODE-SYNTAX` and `NODE-RULES` modules define the `Node` class. A node can be in one of the phases `start`, `waiting`, or `finished`. A node is in the `start` phase before it receives the first pass, in `waiting` after it receives the first pass, and in `finished` after it receives the second pass. It has three other attributes: `value`, `leader`, and `next`. The attribute `value` is the value being compared for leader, `leader` is the object id of the expected leader, and `next` is the next node in the ring. The `RANDOM` module is also protected since it is used to generate a random value for each node. The `[start-election]` rule causes the node to pass around the `best` message with its own value and id. The `[vote-best]` and `[vote-self]` rules cause a node to propagate the candidate around, this candidate being either the existing candidate or itself if its value is higher. After the first pass, the `[propagate-results]` rule causes a node to share the best leader with all the other nodes.

```
mod NODE-SYNTAX is
  ex CONFIGURATION .
  pr NAT .

  sort MsgCont .

  op node : Nat -> Oid [ctor] .
  op best : Oid Nat -> MsgCont [ctor] .
  op startelection : -> MsgCont [ctor] .
  op Node : -> Cid [ctor] .
  op msg_to_ : MsgCont Oid -> Msg [ctor] .

  sort Phase .
  op start : -> Phase .
  op waiting : -> Phase .
  op finished : -> Phase .

  op next :_ : Oid -> Attribute [ctor] .
  op value :_ : Nat -> Attribute [ctor] .
  op leader :_ : Oid -> Attribute [ctor] .
  op phase :_ : Phase -> Attribute [ctor] .

  op init : Nat -> Configuration .
endm
```

```
mod NODE-RULES is
  pr NODE-SYNTAX
  vars O O2 O3 : Oid .
  var A : AttributeSet .
  vars N N2 : Nat .
  var P : Phase .
  --- start the election process
  rl [start-election] :
      (msg startelection to O)
      < O : Node | next : O2, value : N, phase : start, A > =>
      < O : Node | next : O2, value : N, phase : waiting, A >
      (msg best(O, N) to O2) .

  --- once the election is over, just propagate the
  --- result and send the message to the next node
  rl [propagate-results] :
      (msg best(O, N) to O2)
      < O2 : Node | next : O3, phase : waiting, leader : O2, A >
      =>
      < O2 : Node | next : O3, phase : finished, leader : O, A >
      (msg best(O, N) to O3) .

  --- if the previous candidate is better, propagate
  crl [vote-best] :
      (msg best(O, N) to O2)
      < O2 : Node | next : O3, value : N2, phase : start, A > =>
      < O2 : Node | next : O3, value : N2, phase : waiting, A >
      (msg best(O, N) to O3)
      if N > N2 .

  --- if self is better, propagate self
  crl [vote-self] :
      (msg best(O, N) to O2)
      < O2 : Node | next : O3, value : N2, phase : start, A > =>
      < O2 : Node | next : O3, value : N2, phase : waiting, A >
      (msg best(O2, N2) to O3)
      if N <= N2 .
endm
```

```
mod NODE-INIT is
  pr NODE-RULES .
  pr RANDOM .
  op init-node : Nat Nat -> Configuration .
  ---- initialize node N pointing to node N2
  eq init-node(N, N2) =
    < node(N) : Node | next : node(N2), value : random(N),
      phase : start, leader : node(N) > .

  op init-ring : Nat -> Configuration .
  op init-ring-helper : Nat Nat -> Configuration .

  eq init-ring(N) = (msg startelection to node(0))
    init-ring-helper(0, N) .
  ceq init-ring-helper(N, N2) = init-node(N, N + 1)
    init-ring-helper(N + 1, N2) if N < N2 .
  eq init-ring-helper(N, N) = init-node(N, 0) .
endm
```

Step 3: Verify important properties of the model using model-checking in Maude. There are three important properties to check. First, in any final state, no node should be in the waiting state. Next, all nodes should agree on the leader. Lastly, there should not be a node with a higher value than the leader. As usual, each of these should have no solutions.

```
search init-ring(5) =>! C:Configuration
  < O:Oid : Node | phase : waiting, A:AttributeSet > .

search init-ring(5) =>! C:Configuration
  < O1:Oid : Node | leader : L1:Oid, A1:AttributeSet >
  < O2:Oid : Node | leader : L2:Oid, A2:AttributeSet >
  such that L1:Oid =/= L2:Oid .

search init-ring(5) =>! C:Configuration
  < L1:Oid : Node | leader : L1:Oid, value : N1:Nat ,
    A1:AttributeSet >
  < O2:Oid : Node | value : N2:Nat, A2:AttributeSet >
  such that N2:Nat > N1:Nat .
```

Step 4: Optionally, split the source code into multiple files. This is not necessary, as there is only one type of node. If, for example, this were a client-server architecture, splitting this code might be useful so the client and server code are in separate files.

Step 5: Extract the interface into its own file. For this, the file `interface.maude` is created. Any syntax that may go into parsing a message must be moved from the source file to this interface file. For this example, the operators to move are `node`, `best`, and `startelection`. So, the contents of `interface.maude` are:

```
mod INTERFACE is
  ex CONFIGURATION .
  pr NAT .
  sort MsgCont .

  op node : Nat -> Oid [ctor] .
  op best : Oid Nat -> MsgCont [ctor] .
  op startelection : -> MsgCont [ctor] .
endm
```

The modules `NODE-RULES` and `NODE-INIT` can stay unchanged, but the following code should replace the `NODE-SYNTAX` module:

```
load interface
mod NODE-SYNTAX is
  pr INTERFACE .
  pr NAT .

  op Node : -> Cid [ctor] .
  op msg_to_ : MsgCont Oid -> Msg [ctor] .

  sort Phase .
  op start : -> Phase .
  op waiting : -> Phase .
  op finished : -> Phase .

  op next :_ : Oid -> Attribute [ctor] .
  op value :_ : Nat -> Attribute [ctor] .
  op leader :_ : Oid -> Attribute [ctor] .
  op phase :_ : Phase -> Attribute [ctor] .

  op init : Nat -> Configuration .
endm
```

Step 6: Load the middleware file. In the source file, the middleware should be loaded, and the message syntax of the middleware defined in `MESSAGE` should replace the message syntax used by the model. Hence, the lines `load ../mw/mw` (or whatever the path to the middleware is) and `pr MESSAGE .` should be added, and the line `op msg_to_ : MsgCont Oid -> Msg [ctor] .` should be deleted, since the rules should use the operator with the same name defined by the `MESSAGE` module. If this example used a different syntax for messages, this syntax would need to be changed in each rule to follow the syntax defined in the middleware. However, because the middleware and this example use the exact same syntax for messages, conveniently none of the rules need to change. The interface file and the modules `NODE-RULES` and `NODE-INIT` can remain unchanged, but the syntax module should be replaced with:

```
load interface
load ../mw/mw
mod NODE-SYNTAX is
  pr INTERFACE .
  pr MESSAGE .
  pr NAT .

  op Node : -> Cid [ctor] .

  sort Phase .
  op start : -> Phase .
  op waiting : -> Phase .
  op finished : -> Phase .

  op next :_ : Oid -> Attribute [ctor] .
  op value :_ : Nat -> Attribute [ctor] .
  op leader :_ : Oid -> Attribute [ctor] .
  op phase :_ : Phase -> Attribute [ctor] .

  op init : Nat -> Configuration .
endm
```

Step 7: Rewrite the initialization code to also initialize the middleware. For this example, node `N` will run on `localhost` on port `8100 + N`. Each middleware only needs to know the location of the next object in the ring. Because it is no longer necessary to initialize a ring, the code for initializing a ring in `NODE-INIT` can be deleted. The code for `init-node` is otherwise mostly unchanged, except for also initializing the middleware. So, the `NODE-INIT` module is replaced with:

```
mod NODE-INIT is
  pr NODE-RULES .
  pr RANDOM .
  pw MW .
  op init-node : Nat Nat -> Configuration .
  ---- initialize node N pointing to node N2
  eq init-node(N, N2) =
    init-mw(8100 + N, node(N2) |->
      ip-port("localhost", 8100 + N2))
    < node(N) : Node | next : node(N2), value : random(N),
      phase : start, leader : node(N) > .
endm
```

At this point, the source code is finished, and can also be found in `leader/node.maude`. The contents of this file should be:

```
load interface
load ../mw/mw
mod NODE-SYNTAX is
  pr INTERFACE .
  pr MESSAGE .
  pr NAT .

  op Node : -> Cid [ctor] .

  sort Phase .
  op start : -> Phase .
  op waiting : -> Phase .
  op finished : -> Phase .

  op next :_ : Oid -> Attribute [ctor] .
  op value :_ : Nat -> Attribute [ctor] .
  op leader :_ : Oid -> Attribute [ctor] .
  op phase :_ : Phase -> Attribute [ctor] .

  op init : Nat -> Configuration .
endm
```

```
mod NODE-RULES is
  pr NODE-SYNTAX
  vars O O2 O3 : Oid .
  var A : AttributeSet .
  vars N N2 : Nat .
  var P : Phase .
  --- start the election process
  rl [start-election] :
      (msg startelection to O)
      < O : Node | next : O2, value : N, phase : start, A > =>
      < O : Node | next : O2, value : N, phase : waiting, A >
      (msg best(O, N) to O2) .

  --- once the election is over, just propagate the
  --- result and send the message to the next node
  rl [propagate-results] :
      (msg best(O, N) to O2)
      < O2 : Node | next : O3, phase : waiting, leader : O2, A >
      =>
      < O2 : Node | next : O3, phase : finished, leader : O, A >
      (msg best(O, N) to O3) .

  --- if the previous candidate is better, propagate
  crl [vote-best] :
      (msg best(O, N) to O2)
      < O2 : Node | next : O3, value : N2, phase : start, A > =>
      < O2 : Node | next : O3, value : N2, phase : waiting, A >
      (msg best(O, N) to O3)
      if N > N2 .

  --- if self is better, propagate self
  crl [vote-self] :
      (msg best(O, N) to O2)
      < O2 : Node | next : O3, value : N2, phase : start, A > =>
      < O2 : Node | next : O3, value : N2, phase : waiting, A >
      (msg best(O2, N2) to O3)
      if N <= N2 .
endm
```

```
mod NODE-INIT is
  pr NODE-RULES .
  pr RANDOM .
  pr MW .
  op init-node : Nat Nat -> Configuration .
  ---- initialize node N pointing to node N2
  eq init-node(N, N2) =
    init-mw(8100 + N, node(N2) |->
      ip-port("localhost", 8100 + N2))
    < node(N) : Node | next : node(N2), value : random(N),
      phase : start, leader : node(N) > .
endm
```

Step 8: Run the distributed implementation. To run the distributed implementation of ring leader election with four nodes, open four separate processes in the same directory as `node.maude`. In the first one, execute:

```
load node .
erew init-node(0, 1) .
```

In the second,

```
load node .
erew init-node(1, 2) .
```

In the third,

```
load node .
erew init-node(2, 3) .
```

In the last,

```
load node .
erew (msg startelection to node(3)) init-node(3, 0) .
```

It should even be visually apparent that messages are being sent around the ring twice from the terminal output. Make sure the first three processes are running before the fourth is executed, as otherwise a node may try to communicate with a node that is not online, causing errors.