Modeling the Raft Distributed Consensus Protocol in LNT

Hugues Evrard
Google
hevrard@google.com

Consensus protocols are crucial for reliable distributed systems as they let them cope with network and server failures. For decades, most consensus protocols have been designed as variations of the seminal Paxos, yet in 2014 Raft was presented as a new, "understandable" protocol, meant to be easier to implement than the notoriously subtle Paxos family. Raft has since been used in various industrial projects, e.g. Hashicorp's Consul or etcd (used by Google's Kubernetes). The correctness of Raft is established via a manual proof, based on a TLA+ specification of the protocol. This paper reports our experience in modeling Raft in the LNT process algebra. We found a couple of issues with the original TLA+ specification of Raft, which has been corrected since. More generally, this exercise offers a great opportunity to discuss how to best use the features of the LNT formal language and the associated CADP verification toolbox to model distributed protocols, including network and server failures.

1 Introduction

Consensus protocols enable distributed systems to cope with network and server failures via the state machine replication approach [14]. Most consensus protocols are designed as variations of Paxos [11], and they are all notoriously difficult to implement due not only to their inherent complexity, but also to the fact that they are typically presented in abstractions that are non-trivial to transcribe into an executable program. The Raft consensus protocol was designed with understandability in mind, and user studies indicate that it is easier to reason about than Paxos [13]. Several open-source implementations of Raft exist,[1] and the protocol is also used in some industrial projects like Hashicorp's Consul[2] or the etcd[3] key-value store used by Google's Kubernetes.[4] Raft is formally defined by its TLA+ specification [12], upon which the Raft authors presented a hand-written proof of the protocol safety.

In this paper, we discuss our experience in modeling Raft in the LNT formal language [1]. A first version of this model was written as a use-case for the Distributed LNT Compiler [3], and helped to identify two issues in the original TLA+ specification, which have been fixed since. We contributed a second version as a new model for the 2015 edition of the Model Checking Contest [8, 9]. For the first time, in this paper, we have the opportunity to discuss the whole specification in detail. Our goal is not so much to describe Raft internals as to discuss the process of developing the formal model of a distributed system using LNT.

An LNT model can be verified using CADP [4], which relies on explicit state-space exploration model checking techniques. The style in which an LNT model is written can have a dramatic effect on the size of its state space, possibly restricting the scope of achievable verification. Hence, we regularly mention which style choices were made with respect to their impact on the state space size.

[1] https://raft.github.io/#implementations
[2] www.consul.io/docs/internals/consensus.html
[3] github.com/etcd-io/etcd/tree/master/raft
[4] kubernetes.io/blog/2019/08/30/announcing-etcd-3-4/

A. Fehnker and H. Garavel (Eds.):
Models for Formal Analysis of Real Systems (MARS 2020)
EPTCS 316, 2020, pp. 15–39, doi:10.4204/EPTCS.316.2

© H. Evrard

The paper is structured as follows: Section 2 gives an overview of Raft, and Section 3 presents its LNT model. Section 4 discusses the style of the specification, and broadens to distributed systems modeling in general. Section 5 gives concluding remarks. The complete LNT specification of Raft is provided in Annex A.

2 The Raft consensus protocol

We give a brief overview of the consensus problem, and how Raft solves it. We aim at providing a general understanding of Raft structure, more details come in the following section where we present the LNT model.

2.1 Consensus for state machine replication

The state machine replication approach [14] enables a service to be made robust to failures by replicating the logic of the service, represented by a deterministic state machine, among several servers of a distributed system. When a service's client request—named *command* here—arrives, a server handles it. This server then replicates them to all servers, which store them into their local *log*. As long as all server logs stay coherent, they enable servers process the same client commands in the same order, such that all the replicated deterministic state machines reach the same states and return the same responses. Consensus is the mean by which we keep server logs coherent. This lets the system cope with server failures: the service is kept available via the servers that are still alive.

This approach relies on a consensus protocol whose role is to let all alive servers agree on which client commands to add to their log, and in which order, and to do so even in the presence of server and network failures. Raft is a consensus protocol that operates in two main phases: leader election, and log replication.

2.2 Raft leader election

In Raft, time is divided into *terms*, which are monotonically increasing and act as logical clocks [10]. The leader election phase of Raft aims at getting one server to be elected as the leader for the current term.

Servers can be in one of three states: *follower*, *candidate* or *leader*. Every server that is not a leader has a timeout after which, if it did not receive any message, it increments its own term index, moves into the candidate state, and sends vote requests to all other servers. A server that receives a vote request will grant its vote if: the candidate term is not lower than its own term, and the server has not already voted for some other server for the current term. This election mechanism is achieved with *RequestVote* requests and responses.

When a candidate for a given term is granted a majority of votes, it becomes the leader for this term and starts sending heartbeat messages to all other servers. These heartbeat messages let other servers know about the new leader, and other candidates step down to the follower state.

A leader regularly sends heartbeat messages to reset the timeout counter of its followers, and thus prevent new elections. Yet, if the leader server crashes or becomes isolated by a network partition, other servers stop receiving its messages and soon trigger a new leader election.

The timeout durations are chosen randomly within certain bounds, such that there is a high chance for an election to successfully establish a leader. The frequency at which heartbeat messages are sent is chosen accordingly, to minimize the opportunities of spurious elections while a leader is alive and able

to communicate with its followers. The Raft paper [13] features a proper discussion on the choice of relevant timeout duration to use in practical implementations.

Once a leader is elected, it becomes responsible for log replication.

2.3 Raft log replication

A leader handles client commands by adding them to its own log, and making sure to replicate these log updates among its followers. The log is made of *entries*, each entry containing a client command and the term of the leader that handled this command.

To achieve log replication, a leader exchanges *AppendEntries* requests and responses with its followers, to append one or more entries in their log. An entry must be replicated by a majority of servers before it can be considered *committed*, i.e. ready to be applied to the service state machine. Each server maintains a *commit index* that indicates the log index up to which entries are known to be committed.

Failures may trigger an election while the current leader still has some entries not yet replicated by a majority of servers. The new leader for the next term may append new entries that do not match the ones that were not yet committed. In such situations, the new leader must send AppendEntries requests that overwrite the tail of some followers log, effectively replacing the entries that are beyond the commit index by new ones that are imposed by the new leader.

When a leader is elected, it does not know the commit index of its followers. It may try to replicate entries whose indexes are too far beyond some of the followers commit index. In such cases, a follower can refuse an AppendEntries request: the leader then retries to replicate entries with a lower index until it catches up with the follower commit index.

3 Modeling Raft in LNT

Our LNT model is based on the Raft paper [13], and the TLA+ specification of the protocol [12]. Raft contains several optional features, such as dynamically changing the number of servers, or compacting the log. Our model is restricted to the core of the protocol, i.e. leader election and log replication. For the sake of being comparable, we tried to stay close to the variable names used in the TLA+ specification. After a primer on LNT, we describe our Raft model in a top-down fashion.

3.1 LNT: a primer

LNT is a formal language inspired by the LOTOS [6] and E-LOTOS [7] process algebras. Its syntax is close enough to regular programming languages that most of the language should be understandable to someone experienced in programming. The detailed definition of the language is available in the LNT to LOTOS translator manual [1]. This primer focuses on illustrating the interaction between concurrent processes.

In LNT, concurrent processes interact via an *action* on a *gate* to achieve both synchronization and data exchange. Figure 1 illustrates LNT with a small example that defines three processes: `Calc` is a calculator, `User` is a user of this calculator, and `Main` is the top-level parallel composition of `Calc` and `User` processes. `Calc` loops on offering actions on either gate `Add` or `Mul`, via the non-deterministic choice operator `select`. Any action on one of these gates involves three natural numbers (built-in type `nat`), as imposed by the `CalcOp` *channel*. For both calculator operations, the first two naturals are the operands and the third one is the result. The `where` condition imposes a relation between the operands and the result, such that only the result corresponding to the operands is allowed in an action.

```
module primer is

channel CalcOp is
  (nat, nat, nat)
end channel

process Calc [Add, Mul: CalcOp] is
  var op1, op2, res: nat in
    loop
      select
         Add(?op1, ?op2, ?res)
           where res == (op1 + op2)
      [] Mul(?op1, ?op2, ?res)
           where res == (op1 * op2)
      end select
    end loop
  end var
end process

process User [Op: CalcOp] (a, b: nat) is
  var result: nat in
    Op(a, b, ?result)
  end var
end process

process Main [Add, Mul: CalcOp] is
  par Add, Mul in
     Calc[Add, Mul]
  || par
        User[Add](1, 2)
     || User[Mul](3, 4)
     end par
  end par
end process

end module
```

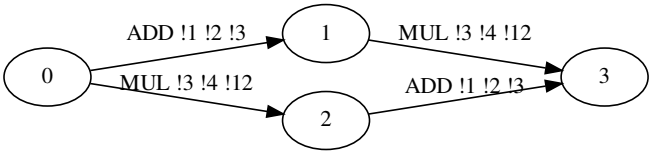

Figure 1: Source code and state space of a small Calculator example of LNT

`User` takes one of the calculator operations as a gate argument (between square brackets), its two operands as data arguments (between parentheses), and does an action on that gate to perform the operation. The value obtained in the `result` variable is imposed by the logic in `Calc`.

`Main` is the default process name for the model entry point. It uses the `par` operator to instantiate a parallel composition of one `Calc` process and two `User` processes. The inner `par` composition indicates that the two `User` processes run in parallel but do not synchronize on any gate. However, the outer `par` composition indicates that any action on gates `Add` or `Mul` must synchronize `Calc` and one of the two `User` processes. Another way to define which gates synchronize with which processes is to list gates before each process in a parallel composition. For instance, our parallel composition can alternatively be written as:

```
par
   Add, Mul → Calc [Add, Mul]
|| Add → User [Add] (1, 2)
|| Mul → User [Mul] (3, 4)
end par
```

The graph illustrates the state space of this example, as generated by CADP tools: it is represented as a labeled transition system (LTS) with the actions as labels—no other information is retained. We can see that both operations lead to relevant results, thanks to the `where` conditions. More importantly, this LTS contains all possible execution paths: as both users run in parallel, there is no order imposed on which one of them interacts first with the calculator. Hence, the LTS has a typical diamond shape that represents several possible execution paths leading to the same final state.

Now that we saw an overview of LNT, let's see how we use it to model a distributed system protocol like Raft, with a top-down approach.

3.2 Raft model parameters

The Raft protocol can be used between any number of servers, but our LNT model has to instantiate a specific Raft configuration among a fixed amount of servers. In the same vein, we need to bound the number of terms, and the number of client interactions, otherwise the state space would be unbounded. We define these three bounds as global configuration constants, which can be easily changed to obtain different configurations.

LNT does not have global constants *per se*. Annex A.1 illustrates how we can use functions with no arguments to define the `MaxTerm`, `MaxServerID` and `MaxClientInteraction` constants. Server IDs range from one to `MaxServerID`, included: the zero ID is reserved as a "nil" value, which will prove useful in Section 3.6.2. We also declare here the `NatArray` type, indexed by the server IDs, and the `Majority` function which returns the number of servers needed to get a majority.

3.3 Top-level parallel composition

In Annex A.2, the `Main` process is our model entry point. It defines the top-level parallel composition of three kinds of processes: `Network`, `Server` and `Clients`. The nested parallel composition is such that: an action on gate `Send` or `Recv` involves the network and one of the servers, an action on `Client` involves a server and a client, and an action on `Crash` involves a server. This enables servers to interact by sending and receiving messages to and from the network via `Send` and `Recv`, clients to interact with servers via `Client`, and server crashes to be monitored via `Crash`. The possible data exchange on these gates is typed via the `ID` and `Comm` channels declared here.

Each server receives its ID as an argument. The number of server instances in this parallel composition must match the `MaxServerID` constant. Changing the number of servers is a matter of adding or removing server instances here, and updating `MaxServerID` and the range of `NatArray` accordingly.

3.4 Interaction with clients

The `Clients` process in annex A.3 is a simple abstraction of how clients may interact with the servers. Neither the TLA+ specification nor our model include the details of a service finite state machine, such that we do not need to model client commands, or server responses to clients. Thus, client-server interaction boils down to an action on gate `Client`, where the only data is the ID of the leader with which the client interacts. This is done in a loop which role is to bound the number of client interactions.

3.5 Network communication

Raft claims to be robust to the reordering, loss or duplication of messages. Annex A.4 contains our model for such an unreliable communication network.

We start by defining a message `Payload`: Raft messages always contain the sender's current term, together with the request or response of a *remote procedure call*[5] (RPC), defined later. A `NetworkMessage` encapsulates the origin and destination server ID, along with a payload. A `NetworkMessageSet` is simply a set of network messages.

The `Network` process is responsible for transferring messages between servers, but it can also loose or duplicate messages. It interacts with servers through the `Send` and `Recv` gates, which are named from

[5] We use "remote procedure call" in this context to keep the naming coherent with the Raft paper.

a server point of view: the network actually obtains a message through an action on `Send`, and delivers it through an action on `Recv`.

This process is structured as an infinite loop on a non-deterministic choice of possible actions. The network has two alternatives to obtain a message: in the first one, the message is successfully stored in the bag of messages; in the second one, the message is obtained but not stored: this models message loss. There are also two alternatives for message delivery: in the first, the delivered message is effectively removed from the bag of messages; in the second, it is kept in the bag: this models message duplication. In both cases of message delivery, any message stored in the bag may be transmitted: this models message reordering.

3.6 Server

Annex A.5 presents the `Server` process, along with the declaration of a few types. The `ServerState` type illustrates how to define an enumerated type with the possible states of a server. The `Entry` type would in principle be made of both a term and a client command, yet as we do not model client commands (see section 3.4), only the term remains. Both `EntryList` and `NatSet` types are pretty self-explanatory: they represent a list of entries, and a set of naturals, respectively. The `LastTerm` function returns the term of the last entry in the server's log, or zero if this log is empty.

Raft servers may interact through two kinds of RPC, one for each of the protocol phases: the leader election is achieved by the *RequestVote* RPC, the log replication by the *AppendEntries* RPC. The `RPC` type defines the request and response messages for these two RPC. Note that the `matchIndex` field of `AppendEntriesResponse` is not mentioned in the Raft paper, but is present in the TLA+ specification.

The `Server` process models the behavior of a Raft server. After the initialization of its state variables, it follows a structure similar to the network process, i.e. an infinite loop on a non-deterministic choice between a few possible behaviors: hit a timeout, receive an RPC message, or interact with a client. On top of that, the server may crash at any time.

3.6.1 Timeout

When a leader times out, it sends AppendEntries requests to all its followers, whether to actually perform log replication or just as a heartbeat to prevent them to time out. This is achieved by calling the `BroadcastAppendEntriesRequest` sub-process, which is defined underneath the `Server` process.

If a server times out while not being a leader, it becomes an election candidate for a new term. As we bind our model on the number of terms servers may explore, we use the `stop` operator to disable progress on this `select` branch if our current term is already at the limit. Note that this only stops the server from triggering a new election, it can still perform behaviors of the other `select` branches, like RPC message handling, within this term.

If the server's term is still within bounds to be able to trigger a new election, then the server increments its term, becomes candidate, votes for itself and sends a vote request to all the other servers. In the TLA+ specification, a candidate does not vote for itself, but rather sends itself a vote request. We choose to have candidates vote for themselves because it is likely how a real implementation would behave, and it reduces the state space by having less vote requests being sent.

3.6.2 Message reception

A server receives a message from the network through an action on the `Recv` gate. Here, we start by declaring the variables needed to unpack message contents, all named with a "`m`" prefix to mirror the

TLA+ specification. Then, the action on gate `Recv` stores the message sender ID in `mFrom`, while the message payload is directly unpacked in `mTerm` and `mRPC`.

If the message is stale, i.e. the sender's term `mTerm` is smaller than our current term, then we do not bother handling it further. This is a difference with the TLA+ specification, which drops stale RPC responses, but does reply to stale RPC requests in order to let the sender quickly become aware of a new term. This behavior makes sense to minimize the time window during which a server is using an outdated term. Still, we can safely ignore stale requests, and here we do so to reduce the model state space.

If the message is not stale, then we check whether we should update our own term: this is the equivalent of the TLA+ specification `UpdateTerm` part. Updating to a higher term means falling back into the follower state, and also store zero as a "nil" server ID into `votedFor`, to indicate that we did not yet vote for anyone in this term. This is where having zero as a "nil" ID proves to be useful.

Finally, we use the pattern matching operator `case` to conveniently dispatch the RPC message according to its type. Each RPC message type has a corresponding handler, which receives the message contents extracted into variables via the matched pattern. An argument must be prefixed by "`!?`" if it is an "`in out`" argument that may be modified by the handler.

We briefly describe the handlers, whose names mirror the ones in TLA+. For a full discussion of why these behaviors safely implement a consensus, see the Raft original papers [12, 13].

RequestVoteRequest handler (annex A.6.1). The server checks whether to grant its vote or not based on the status of its log, and whether it has already voted for someone else or not.

RequestVoteResponse handler (annex A.6.2). The server stores the vote response, and proceeds to become leader if it received a majority of votes.

AppendEntriesRequest handler (annex A.6.3). The server checks whether its log is ready to receive the new entries. If not, it declines the request. Otherwise, it accepts the request and appends the entries, possibly by removing uncommitted entries at the tail of its log. The server commit index is updated to the leader commit index, and a match index is computed to let the leader know up to which index this server logs is matching the leader's one. While the AppendEntries RPC is designed to enable the update of several entries in one call, our model reflects the TLA+ specification choice of restricting updates to one entry at a time. Hence, an AppendEntries request is either a heartbeat message with zero entries, or a request to update a single entry, as verified with an assertion.

AppendEntriesResponse handler (annex A.6.4). The server keeps track of its followers' current matching indexes in the `nextIndex` and `matchIndex` arrays, which are updated accordingly to the AppendEntries response. Then, it tries to increase its commit index based on whether a majority of servers now agree on sharing the same entry at a higher log index.

3.6.3 Client interaction

The last alternative in the server main loop (back to annex A.5) is to interact with a client. Only a leader may interact with clients, hence the action on gate `Client` is guarded by a check on the current server state. Remember that client commands are not part of our model, so a client interaction results in a new entry containing only the leader's current term.

The AppendEntries requests related to this new log entry will be sent after a timeout (see section 3.6.1). One may expect a real implementation to send an AppendEntries request right after a client interaction, yet we know that at least in the case of Consul, the leader actually waits for its timeout, to be able to batch an update of all the client interactions that may happen during a leader timeout duration. This choice may be detrimental to latency, but beneficial to throughput.

3.6.4 Crash

Servers may crash at any time. The careful reader will have noticed that the whole main loop is encapsulated in a `disrupt` statement, which enables an action on gate `Crash` to happen at any point during the server execution. One may think that the crash could simply be one of the `select` alternative, but this would not enable a crash to happen e.g. in the middle of a heartbeat broadcast. This illustrates how convenient is the `disrupt` operator to model behaviors that may pertubate others at any point.

3.7 Issues with the original TLA+ specification

Modeling Raft in LNT highlighted two issues with the original TLA+ specification.

The first[6] is a minor typo: a "`matchIndex'`" which should have been a "`matchIndex`", without the apostrophe that denotes a variable's next state in TLA+. This issue in `AdvanceCommitIndex` was minor and made no practical difference.

The second[7] issue is related to a missing server state change: candidates that received a heartbeat message from the leader who won the term election would not step down to become followers. In practice, a candidate behaves almost like a follower, it just will not grant its vote to any other server, so this issue did not impact Raft manual proof. Yet this proves that discrepancies between the intended behavior and the formal specification can appear by accident.

4 Discussion on the modeling of distributed systems

While our model is specific to the Raft protocol, it invites some remarks on the act of specifying distributed systems in general. In this section, we discuss various aspects of our experience with developing the formal model of a distributed system.

4.1 Iterative development with LNT and CADP

Based on our experience with Raft, we argue that LNT and CADP offer a good environment to iteratively develop a formal specification of a distributed system. With regular programming, the ability to experiment with a quick feedback loop can greatly improve the development experience. In practice, this can be achieved with fast modify-compile-execute cycles, or by using interpreted languages which offer read-eval-print-loop interfaces. We consider that writing a formal specification is close to writing a program, except that we operate with a formal language. Having a quick iteration feedback loop is still relevant in a formal context.

With this in mind, the following features of LNT and CADP have been very helpful. First, the LNT syntax is close to mainstream programming languages. This cannot be said of numerous other formal languages, which roots in theoretical computer science often lead to an exotic syntax. Second,

[6] `github.com/ongardie/raft.tla/blob/34cdd49d22615426ea00a6605b95be57b3cab49a/raft.tla#L478`

[7] `groups.google.com/g/raft-dev/c/yu-w0Ux-gnA/m/VsM49xpFPwcJ`

LNT is a strongly typed language. Type checking can catch many issues as early as compilation time. Beside type errors, CADP is also able to statically report useful warnings about e.g. unused variables, or unreachable actions. Not all formal languages are typed, and in particular TLA+ is not typed. Third, LNT features an `assert` statement, which will stop state-space generation with an error if violated. This enables to quickly spot some bugs, without having to wait for a complete state-space generation. Fourth, although not demonstrated here, LNT parallel composition enables to synchronize an arbitrary number of processes (two or more) on a single action. Many languagues can only express process interaction between a pair of processes, such that any kind of barrier synchronization between more than two processes requires a protocol built on top of pair-wise interaction. The ability to synchronize an arbitrary number of processes can prove very useful in some models [5], and lead to smaller state-space by factoring barrier synchronization protocols into a single action.

Fifth, CADP can generate state spaces of a given model in both implicit and explicit forms. An implicit state space is effectively a dynamic library offering relevant primitives to generate the list of reachable states from a given state. An explicit state space is the actual LTS containing all the states and transitions of a model, obtained by exhaustively exploring all reachable states from the model initial state. Generating the explicit state space can take a significant time, which may hinder the iterative development process. The implicit state space is typically generated within seconds, and can then be explored manually to check e.g. whether a given execution path is reachable. This can prove very handy when trying modifications on the model being developed.

4.2 Generic skeleton for distributed systems

A distributed system can arguably be defined as *node* processes that interact via some sort of network. In that sense, LNT models of distributed systems are very likely to have a top-level parallel composition similar to ours, i.e. a series of nodes (here, the Raft servers) communicating over a network.

Our `Network` process is already generic: it only knows about a message origin, destination, and payload. What the payload actually contains can be easily adapted to the needs of other protocols.

While our `Server` process is specifically implementing Raft, we argue that its general structure can be used as a template for distributed system nodes. This structure is expressed in the `GenericNodeSkeleton` process of annex B.1. The node initializes its state, then enters a main loop where it reacts to the reception of messages, or other local events. The `disrupt` operator lets us easily model a crash at any time. Note that such generic structures have already been captured by established distributed system frameworks, like the `gen_server`[8] module in Erlang.

4.3 A library of network models

The genericity of our network process means that it can be easily replaced by other ones, with different communication behaviors. As an example, annex B.2 presents the `ReliableNetwork` process through which messages may be re-ordered, but neither lost nor duplicated. This alternative network model is as simple to use as dropping it in place of the `Network` process in the top-level parallel composition.

This ability to swap network models brings up two major benefits. First, using a reliable network helps in iteratively developing a model. A reliable network will generate smaller state spaces, because its behavior is a subset of the unreliable one, which leads to less possible message interactions between nodes. As an example, consider a Raft configuration of two servers, one term and one client interaction: with the unreliable `Network` process, this configuration leads to an LTS of 11,862,015 states and

[8] `https://erlang.org/doc/design_principles/gen_server_concepts.html`

74,821,042 transitions. With the `ReliableNetwork` process, the state space is down to 22,311 states and 108,176 transitions, that is three and two orders of magnitude smaller, respectively. Thus, a reliable network helps to get a quick development feedback loop by being able to generate smaller state spaces. While those state spaces are not relevant for a proper verification, they can still be used as a proxy: if an issue is found with the reliable network, then it will appear with the unreliable network as well.

Second, being able to swap network models enables to test a system under a whole spectrum of network behaviors. For instance, some network models could capture the guarantees offered by Unix sockets in TCP or UDP mode. Others could model the communication guarantees of the run-time of some specific distributed programming languages. We could then have a whole library of network models, such that the same distributed system can be easily verified against each of them. This would help to gain confidence on which requirements are actually needed for a real implementation. It could also help in making some existing distributed algorithms become robust to some communication failures (e.g. message duplication) that were not considered in its original design, by verifying the algorithm against a less reliable network model and then improving the algorithm until it supports this alternative network model.

4.4 Keep the state space minimal

When editing an LNT specification, it is recommended to try and keep the state space as small as possible, while still exploring all relevant execution paths. This balance can be risky when, for the sake of minimizing the state space, the model restricts some execution paths that may well lead to some issues in the real implementation. Here, we discuss a couple of techniques we used.

First, events surfaced as actions on gates should be kept to a minimum. In the state space LTS, only the actions appear as transitions, so the less actions there are, the smaller the state space. Still, there should be enough actions to analyze the state space and look for issues. For distributed systems, a rule of thumb seems to be to surface events related to either inter-process communication, or outstanding local events. In Raft, communication events happen on gates `Send` and `Recv`, while actions on gates `Client` and `Crash` are, as long as the protocol is concerned, local to a Raft server. One could argue that actions on `Client` are not strictly necessary to surface, since we could infer whether a leader had a client interaction by observing its AppendEntries requests. Still, we choose to model them as explicit actions because it makes it easier to reason about the resulting state space.

Second, some of the action interleavings may be avoided by forcing a specific order of execution. For instance, in our Raft model a leader broadcasts AppendEntries requests in the order of server IDs (e.g. leader 1 would send requests to follower 2, and then follower 3). If we wanted to be truly exhaustive, we should model this broadcast as sending requests in any order (e.g. leader 1 could send to either follower 2 or 3, and then to the other one). Yet we argue that exploring all these sending orders is irrelevant here, because of the asynchronous nature of communications. What matters is that the receiving order is not fixed, and in our case all possible receiving orders will be explored thanks to the network process which buffers messages and delivers them in any order. Based on this reflection, we decided that it was acceptable to impose the sending order, and thus reduce the state space along the way.

This example demonstrates how it is sometimes acceptable to impose an execution order that reduces the state space, but one should be genuinely careful about deciding when it is safe to do so. It is very easy to restrict the state space in a way that removes some of the possible execution paths that are actually problematic. These decisions depend not only on the system being modeled, but also on the kinds of verification that are performed on the state space. The obliteration of some execution paths may be irrelevant to a given verification property, but crucial to an other. Therefore, great care should be taken

when using this technique.

4.5 Working around unreliable communication

Could it be possible to safely reduce the state space by bypassing the modeling of unreliable communication altogether? Many distributed system protocols or algorithms are designed to cope with unreliable network, using well-known techniques. For instance, making messages idempotent is a classic way to cope with message duplication: receiving a message once or several times does not make a difference. If we take for granted that nodes are robust to any message duplication, why bother modeling message duplication in our specification? Removing message duplication from our model would help to reduce the state space by a significant amount.

In the case of our Raft model, messages are not only idempotent, but servers also discard any message that has a lower term than the server's current term. Thus, besides removing message duplication from our network model, we could go further and make the network track the current term of each server and drop all buffered outdated messages that would be dropped by the destination server anyway. This could reduce the state space by removing some `Recv` actions.

These considerations could pave the way to analyze state spaces of configurations with higher bounds on the number of servers, client interactions and terms, hence gaining more confidence in the verification. Yet, it is also very easy to overlook how such bypassing may cut away corner-cases that are relevant to expose bugs that could happen in real implementations. In other words, we are back to this subtle question: on which grounds can we be certain that it is safe to bypass some behaviors? Such decisions would be best based on some formal verification or proof of specific well-known solutions, such that we can safely put those bypasses in place. For instance, a proof that message duplication can be safely bypassed in modeling if all messages are idempotent would enable to use this technique in any distributed system model.

4.6 Formal model and implementation: bridging the gap

Several Raft implementations now exist, but how can we know whether they are correct? Some tools specialize in testing distributed systems. In recent years, the Jepsen[9] tool has been used to analyze more than twenty distributed systems, finding issues in almost all of them, including systems that are based on Raft, like etcd.[10]

More generally, the question is how to bridge the gap between the formal model of a protocol and its actual implementation. TLA+ does not have a code-generation feature. Other projects, like Verdi [15], are explicitly designed to generate code from the very same specification on which a proof is conducted. As a matter of fact, Verdi has been used to model Raft and obtain an implementation directly from the model [16]. In the model-checking world, the Distributed LNT Compiler [3] enables the generation of a distributed implementation (in C) of an LNT model. Yet, its protocol to implement synchronization on gates assumes a reliable message passing between processes, so it is not best suited to generate implementations of distributed systems. Still, it would be possible to specialize this tool and have the network actions inthe model being replaced by actual call to e.g. Unix socket primitives. Section 6 of [2] conducts a performance comparison between Consul and the code generated by DLC from an LNT Raft specification: the DLC version can process 1000 client commands on a cluster of 3 servers in 2.3 seconds, where Consul requires 0.5 second. DLC is slower than Consul in the general case, but

[9] `https://jepsen.io`

[10] See etcd analysis published on January 2020: `https://jepsen.io/analyses/etcd-3.4.3`

Consul also features an optimization that buffers client commands on the leader to treat them in batch to favour throughput over latency, whereas the DLC version triggers an *AppendEntries* RPC round for each client command. In any case, this study reports that DLC-generated implementations can achieve inter-LNT-process communication over TCP between separate machines in less than half a millisecond.

5 Conclusion

We gave an overview of the Raft consensus protocol followed by a presentation of how we specified it in LNT. Our model clearly separates the behavior of Raft servers on one side, and the behavior of the network on the other. Having the network as a separate, generic entity enables not only to reuse this network model to be reused for other systems, but also to be replaced with other network behaviors. This can help during the development of a model, to keep the state spaces from growing too big too early. Our model was written to be comparable with the original TLA+ specification of Raft, in which we found two issues.

We then conducted a discussion on the formal specification of distributed systems in general. As opposed to proof systems, we argue that LNT and CADP offer an iterative development environment that provides a quick feedback loop. We also discussed various techniques for safely reducing the state space of a model, while still capturing all the execution paths relevant for verification.

Our Raft model only contains the core of the protocol, and a possible future work would be to extend it to cover optional features, like dynamic configuration changes or log compaction.

Acknowledgements

The author would like to thank the MARS workshop editors and reviewers for their valuable feedback. Special thanks also goes to Hubert Garavel and Frdric Lang from the Inria CONVECS team for their support in running some experiments.

References

[1] David Champelovier, Xavier Clerc, Hubert Garavel, Yves Guerte, Christine McKinty, Vincent Powazny, Frédéric Lang, Wendelin Serwe & Gideon Smeding (2017): *Reference Manual of the LNT to LOTOS Translator (Version 6.7)*. Available at `http://cadp.inria.fr/publications/Champelovier-Clerc-Garavel-et-al-10.html`. INRIA/VASY and INRIA/CONVECS, 130 pages.

[2] Hugues Evrard (2015): *Génération automatique d'implémentation distribuée à partir de modèles formels de processus concurrents asynchrones*. Theses, Université Grenoble Alpes. Available at `https://hal.inria.fr/tel-01215634`.

[3] Hugues Evrard (2016): *DLC: Compiling a Concurrent System Formal Specification to a Distributed Implementation*. In Marsha Chechik & Jean-François Raskin, editors: *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, *Lecture Notes in Computer Science* 9636, Springer, pp. 553–559, doi:10.1007/978-3-662-49674-9_34.

[4] Hubert Garavel, Frédéric Lang, Radu Mateescu & Wendelin Serwe (2013): *CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes*. *Springer International Journal on Software Tools for Technology Transfer (STTT)* 15(2), pp. 89–107, doi:10.1007/s10009-012-0244-z. Available at `http://cadp.inria.fr/publications/Garavel-Lang-Mateescu-Serwe-13.html`.

[5] Hubert Garavel & Wendelin Serwe (2017): *The Unheralded Value of the Multiway Rendezvous: Illustration with the Production Cell Benchmark*. In Holger Hermanns & Peter Höfner, editors: *Proceedings of the 2nd Workshop on Models for Formal Analysis of Real Systems (MARS'17), Uppsala, Sweden, Electronic Proceedings in Theoretical Computer Science* 244, pp. 230–270, doi:10.4204/EPTCS.244.10.

[6] ISO/IEC (1989): *LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*. International Standard 8807, International Organization for Standardization – Information Processing Systems – Open Systems Interconnection, Geneva.

[7] ISO/IEC (2001): *Enhancements to LOTOS (E-LOTOS)*. International Standard 15437, International Organization for Standardization – Software and systems engineering, Geneva.

[8] F. Kordon, H. Garavel, L. M. Hillah, F. Hulin-Hubard, A. Linard, M. Beccuti, A. Hamez, E. Lopez-Bobeda, L. Jezequel, J. Meijer, E. Paviot-Adet, C. Rodriguez, C. Rohr, J. Srba, Y. Thierry-Mieg & K. Wolf (2015): *Complete Results for the 2015 Edition of the Model Checking Contest*. http://mcc.lip6.fr/2015/results.php.

[9] Fabrice Kordon, Hubert Garavel, Lom-Messan Hillah, Emmanuel Paviot-Adet, Loïg Jezequel, César Rodríguez & Francis Hulin-Hubard (2016): *MCC'2015 - The Fifth Model Checking Contest*. *Trans. Petri Nets Other Model. Concurr.* 11, pp. 262–273, doi:10.1007/978-3-662-53401-4_12.

[10] Leslie Lamport (1978): *Time, Clocks, and the Ordering of Events in a Distributed System*. *Commun. ACM* 21(7), pp. 558–565, doi:10.1145/359545.359563.

[11] Leslie Lamport (1998): *The Part-Time Parliament*. *ACM Trans. Comput. Syst.* 16(2), pp. 133–169, doi:10.1145/279227.279229.

[12] Diego Ongaro (2014): *Consensus: Bridging Theory and Practice*. Ph.D. thesis, Stanford University. Available at `https://purl.stanford.edu/qr033xr6097`.

[13] Diego Ongaro & John K. Ousterhout (2014): *In Search of an Understandable Consensus Algorithm*. In Garth Gibson & Nickolai Zeldovich, editors: *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, USENIX Association, pp. 305–319. Available at `https://raft.github.io/raft.pdf`.

[14] Fred B. Schneider (1990): *Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial*. *ACM Comput. Surv.* 22(4), pp. 299–319, doi:10.1145/98163.98167.

[15] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst & Thomas E. Anderson (2015): *Verdi: a framework for implementing and formally verifying distributed systems*. In David Grove & Steve Blackburn, editors: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, ACM, pp. 357–368, doi:10.1145/2737924.2737958.

[16] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst & Thomas E. Anderson (2016): *Planning for change in a formal verification of the raft consensus protocol*. In Jeremy Avigad & Adam Chlipala, editors: *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, January 20-22, 2016*, ACM, pp. 154–165, doi:10.1145/2854065.2854081.

A The LNT specification of Raft

A.1 Global configuration parameters

```
-- Global parameters to bound the system

-- Maximum term to be allowed
function MaxTerm: nat is
  return 3
end function

-- Maxium number of client interaction
function MaxClientInteraction: nat is
  return 3
end function

-- Server IDs are within 1..MaxServerID included
-- Zero is reserved as the "nil" server ID
function MaxServerID: nat is
  return 3
end function

-- The range of NatArray must be 1..MaxServerID
type NatArray is
  array[1 .. 3] of nat
end type

-- Majority returns the number of servers needed to get a majority
function Majority : nat is
   return (MaxServerID div 2) + 1
end function
```

A.2 Top-level parallel composition

```
-- For action that only exchange a server ID
channel ID is
  (nat)
end channel

-- For general communication between servers
channel Comm is
  (nat, nat, Payload)
end channel

-- Entry point: parallel composition of Network, Servers and Client
process Main [Send, Recv: Comm, Client, Crash: ID] is
  par
     Send, Recv -> Network[Send, Recv]
  || Client -> Clients[Client]
  || Send, Recv, Client ->
       par
          -- The number of servers here must reflect MaxServerID
```

```
        Server[Send, Recv, Client, Crash](1)
      || Server[Send, Recv, Client, Crash](2)
      || Server[Send, Recv, Client, Crash](3)
      -- Additional servers can be added with e.g.:
      -- || Server[Send, Recv, Client, Crash](4)
      end par
  end par
end process
```

A.3 Clients

```
-- Clients bounds the number of client interactions
process Clients [Client: ID] is
  var n: nat in
    for n := 0 while n < MaxClientInteraction by n := n + 1 loop
      Client(?any nat)
    end loop
  end var
end process
```

A.4 Network

```
-- Payload encapsulates the message contents
type Payload is
  Payload(mTerm: nat, rpc: RPC)
  with "get"
end type

-- NetworkMessage is a message Payload, plus origin and
-- destination which correspond to ServerID's
type NetworkMessage is
  NetworkMessage(orig, dest: nat, payload: Payload)
  with "get"
end type

type NetworkMessageSet is
  set of NetworkMessage
  with "element", "length", "remove"
end type

-- Network models an unreliable communication network.
-- Messages can be lost, duplicated, and reordered.
process Network [Send, Recv: Comm] is
  var
    n: nat,
    orig: nat,
    dest: nat,
    payload: Payload,
    msg: NetworkMessage,
    msgBag: NetworkMessageSet
  in
    msgBag := {}; -- start with no message in the bag
```

```
    loop
      select
        -- Obtain a message, and store it in the bag
        Send(?orig, ?dest, ?payload);
        msgBag := insert(NetworkMessage(orig, dest, payload), msgBag)
      []
        -- Message loss: obtain a message, but do not store it
        Send(?any nat, ?any nat, ?any Payload)
      []
        -- Transmit any message from the bag, then remove it from the bag
        -- Note: by picking up any message from the bag, we model reordering
        n := any nat where (n > 0) and (n <= length(msgBag));
        msg := element(msgBag, n);
        Recv(msg.orig, msg.dest, msg.payload);
        msgBag := remove(msg, msgBag)
      []
        -- Message duplication: transmit it, but do not remove it from the bag
        n := any nat where (n > 0) and (n <= length(msgBag));
        msg := element(msgBag, n);
        Recv(msg.orig, msg.dest, msg.payload)
      end select
    end loop
  end var
end process
```

A.5 Server

```
type ServerState is
  Follower,
  Candidate,
  Leader
  with "==", "!="
end type

-- Entry content is restricted to a term, we do not model client commands
type Entry is
  Entry(term: nat)
  with "get"
end type

type EntryList is
   list of Entry
   with "element", "append", "empty", "length"
end type

type NatSet is
   set of nat
   with "length"
end type

-- RPC models the remote procedure calls of Raft
-- Each of RequestVote and AppendEntries has a request and a response message
```

```
type RPC is
  RequestVoteRequest(lastLogIndex, lastLogTerm: nat),
  RequestVoteResponse(voteGranted: bool),
  AppendEntriesRequest(prevLogIndex, prevLogTerm: nat, entries: EntryList,
                       leaderCommit: nat),
  AppendEntriesResponse(success: bool, matchIndex: nat)
end type

-- LastTerm returns the term of the last log entry
function LastTerm(log: EntryList) : nat is
  case log in
    {} → return 0
  | any → return get_term(element(log, length(log)))
  end case
end function

-- Server models the behavior of a Raft server
process Server [Send, Recv: Comm, Client, Crash: ID] (self: nat) is
  var
    -- server internal state
    state: ServerState,
    currentTerm: nat,
    log: EntryList,
    commitIndex: nat,
    -- related to RequestVote RPC
    votedFor: nat,
    votesGranted: NatSet,
    -- related to AppendEntries RPC
    nextIndex: NatArray,
    matchIndex: NatArray
  in

    -- This disrupt wraps the whole main loop to model a Crash at any time
    -- See the actual Crash action at the bottom of the process
    disrupt

      -- state initialization
      state := Follower;
      currentTerm := 0;
      log := {};
      commitIndex := 0;
      votedFor := 0;
      votesGranted := {};
      nextIndex := NatArray(0);
      matchIndex := NatArray(0);

      -- main loop
      loop
        select

          -- Timeout
          if state == Leader then
```

```
    BroadcastAppendEntriesRequest[Send](self, currentTerm,
                                        commitIndex, log, nextIndex)
  else
    if currentTerm >= MaxTerm then
      -- Do not progress further
      stop
    else
      -- Trigger election for a new term
      currentTerm := currentTerm + 1;
      state := Candidate;
      -- Diff with TLA+ spec: we do vote for ourselves directly
      votedFor := self;
      votesGranted := {self};
      -- Broadcast vote requests
      var n: nat in
        for n := 1 while n <= MaxServerID by n := n + 1 loop
          if (n != self) then
            Send(self, n,
                 Payload(currentTerm,
                         RequestVoteRequest(length(log),
                                            LastTerm(log))))
          end if
        end loop
      end var
    end if
  end if

[]

  -- Receive message
  var
    -- Variables to unpack message content
    mFrom: nat,
    mTerm: nat,
    mRPC: RPC,
    -- RequestVote
    mLastLogIndex: nat,
    mLastLogTerm: nat,
    mVoteGranted: bool,
    -- AppendEntries
    mPrevLogIndex: nat,
    mPrevLogTerm: nat,
    mEntries: EntryList,
    mLeaderCommit: nat,
    mSuccess: bool,
    mMatchIndex: nat
  in

    -- Network interaction to receive a message
    Recv(?mFrom, self, ?Payload(mTerm, mRPC));

    if mTerm < currentTerm then
```

```
        -- Drop stale message
        -- Diff with TLA+ spec: drop any message, where in TLA+
        -- the server would still reply to stale RPC *requests*
        null
      else

        -- This is the "UpdateTerm" part of the TLA+ spec
        if mTerm > currentTerm then
          currentTerm := mTerm;
          state := Follower;
          votedFor := 0 -- Value zero is the nil server ID
        end if;

        -- Dispatch to the relevant handler using pattern matching
        case mRPC in
          RequestVoteRequest(mLastLogIndex, mLastLogTerm) →
            HandleRequestVoteRequest[Send](self, currentTerm, LastTerm(log),
                                           mLastLogTerm, length(log),
                                           mLastLogIndex, mFrom, !?votedFor)
        | RequestVoteResponse(mVoteGranted) →
            eval HandleRequestVoteResponse(log, !?nextIndex, !?matchIndex,
                                           mFrom, !?state, mVoteGranted,
                                           !?votesGranted)
        | AppendEntriesRequest(mPrevLogIndex, mPrevLogTerm,
                               mEntries, mLeaderCommit) →
            HandleAppendEntriesRequest[Send](self, mFrom, currentTerm,
                                             mPrevLogIndex, mPrevLogTerm,
                                             mEntries, mLeaderCommit,
                                             !?commitIndex, !?log)
        | AppendEntriesResponse(mSuccess, mMatchIndex) →
            if state == Leader then
              eval HandleAppendEntriesResponse(currentTerm, mFrom, mSuccess,
                                               mMatchIndex, log, !?nextIndex,
                                               !?matchIndex, !?commitIndex)
            end if
        end case
      end if
    end var

  []

    -- Client interaction
    Client(self) where state == Leader;
    log := append(Entry(currentTerm), log)

  end select
end loop

by
  -- Crash may happen at any time
  Crash(self)
end disrupt
```

```
  end var
end process

-- Helper process to broadcast AppendEntries requests
process BroadcastAppendEntriesRequest [Send: Comm] (self: nat,
                                                    currentTerm: nat,
                                                    commitIndex: nat,
                                                    log: EntryList,
                                                    nextIndex: NatArray) is
  var
    n, prevLogIndex, prevLogTerm, lastEntry : nat,
    entries: EntryList
  in
    for n := 1 while n <= MaxServerID by n := n + 1 loop
      if (n != self) then
        prevLogIndex := nextIndex[n] - 1;
        if prevLogIndex > 0 then
          prevLogTerm := get_term(element(log, prevLogIndex))
        else
          prevLogTerm := 0
        end if;
        lastEntry := min(length(log), nextIndex[n]);
        entries := SubSeq(log, nextIndex[n], lastEntry);
        Send(self, n,
             Payload(currentTerm,
                     AppendEntriesRequest(prevLogIndex, prevLogTerm,
                                          entries, commitIndex)))
      end if
    end loop
  end var
end process
```

A.6 RPC handlers

A.6.1 HandleRequestVoteRequest

```
process HandleRequestVoteRequest [Send: Comm] (in self: nat,
                                               in currentTerm: nat,
                                               in lastLogTerm: nat,
                                               in mLastLogTerm: nat,
                                               in lengthLog: nat,
                                               in mLastLogIndex: nat,
                                               in mFrom: nat,
                                               in out votedFor: nat) is
  var logOK: bool, voteGranted: bool in

    logOK := (mLastLogTerm > lastLogTerm) or
             ((mLastLogTerm == LastLogTerm) and (mLastLogIndex >= lengthLog));

    voteGranted := logOK and ((votedFor == 0) or (votedFor == mFrom));
```

```
      if voteGranted then
        votedFor := mFrom
      end if;

     Send(self, mFrom, Payload(currentTerm, RequestVoteResponse(voteGranted)))

    end var
  end process
```

A.6.2 HandleRequestVoteResponse

```
  function HandleRequestVoteResponse (log: EntryList,
                                      in out nextIndex: NatArray,
                                      in out matchIndex: NatArray,
                                      voter: nat,
                                      in out state: ServerState,
                                      mVoteGranted: bool,
                                      in out votesGranted: NatSet) is
    if (state == Candidate) and mVoteGranted then
      votesGranted := insert(voter, votesGranted);

      if length(votesGranted) >= Majority then
         -- This is the "BecomeLeader" part of the TLA+ spec
         state := leader;
         nextIndex := NatArray(length(log) + 1);
         matchIndex := NatArray(0)
      else
        use nextIndex;
        use matchIndex
      end if
    end if
  end function

  ------------------------------------------------------------
  -- Helpers

  function SubSeq (e: EntryList, in var i: nat, in var j: nat): EntryList is
     var n: nat, result: EntryList in
       result := {};
       -- restrict indexes
       if i == 0 then
          i := 1
       end if;
       j := min(j, length(e));
       for n := i while n <= j by n := n + 1 loop
         result := append(element(e, n), result)
      end loop;
      return result
     end var
  end function
```

A.6.3 HandleAppendEntriesRequest

```
process HandleAppendEntriesRequest [Send: Comm] (self: nat,
                                                 leaderId: nat,
                                                currentTerm: nat,
                                                mPrevLogIndex: nat,
                                                mPrevLogTerm: nat,
                                                entries: EntryList,
                                                mLeaderCommit: nat,
                                                in out commitIndex: nat,
                                                in out log: EntryList) is
  var
    logOK : bool,
    index, matchIndex : nat,
    entry: Entry
  in

    logOK := (mPrevLogIndex == 0) or
               ((mPrevLogIndex > 0)
                 and (mPrevLogIndex <= length(log))
                 and (mPrevLogTerm == get_term(element(log, mPrevLogIndex))));

    -- Diff with TLA+ spec: we don't need to test if the request
    -- was stale (outdated term), because we drop any stale message
    -- before dispatching to handlers

    if not(logOK) then
      -- Decline request
      Send(self, leaderId,
           Payload(currentTerm, AppendEntriesResponse(False, 0 of nat)));
      -- commitIndex is an "in out" variable, we must use it in all execution paths
      use commitIndex
    else
      -- Accept request
      index := mPrevLogIndex + 1;
      if empty(entries) then
        -- Requests with no entry are heartbeat messages
        matchIndex := mPrevLogIndex
      else
        -- We mirror the TLA+ spec logic which limits to 1 entry at a time
        assert(length(entries) == 1);
        entry := element(entries, 1);
        if (length(log) >= index) and (get_term(element(log, index)) != entry.term) then
          -- Conflict, remove our log tail
          log := SubSeq(log, 1, index - 1)
        end if;
        if length(log) == mPrevLogIndex then
          -- Entry not yet in the log, append it
          log := append(entry, log)
        end if;
        matchIndex := mPrevLogIndex + 1
      end if;
      commitIndex := mLeaderCommit;
      Send(self, leaderId,
```

```
            Payload(currentTerm, AppendEntriesResponse(True, matchIndex)))
    end if
  end var
end process
```

A.6.4 HandleAppendEntriesResponse

```
function HandleAppendEntriesResponse (currentTerm: nat,
                                      mFrom: nat,
                                      success: bool,
                                      mMatchIndex: nat,
                                      log: EntryList,
                                      in out nextIndex: NatArray,
                                      in out matchIndex: NatArray,
                                      in out commitIndex: nat) is
  if success then
    nextIndex[mFrom] := mMatchIndex + 1;
    matchIndex[mFrom] := mMatchIndex
  else
    nextIndex[mFrom] := max(1 of nat, nextIndex[mFrom])
  end if;

  eval AdvanceCommitIndex(currentTerm, log, matchIndex, !?commitIndex)
end function

------------------------------------------------------------
-- Helpers

-- MajorityAgree returns true if a majority of servers have the same
-- entry at the given index
function MajorityAgree (index: nat, matchIndex: NatArray): bool is
  var n, nbagree: nat in
    -- start nbagree at 1 since the current leader always agree,
    -- but its own matchIndex is zero
    nbagree := 1;
    for n := 1 while n <= MaxServerID by n := n + 1 loop
      if index <= matchIndex[n] then
        nbagree := nbagree + 1
      end if
    end loop;
    return nbagree >= Majority
  end var
end function

-- MaxAgreeIndex returns the higher index for which servers agree
function MaxAgreeIndex (lengthLog: nat, matchIndex: NatArray): nat is
  var index: nat in
    index := lengthLog;
    while not(MajorityAgree(index, matchIndex)) loop
      index := index - 1
    end loop;
    return index
```

```
  end var
end function

function AdvanceCommitIndex (currentTerm: nat,
                             log: EntryList,
                             matchIndex: NatArray,
                             in out commitIndex: nat) is
  var index: nat in
    index := MaxAgreeIndex(length(log), matchIndex);
    if (index > 0) and (get_term(element(log, index)) == currentTerm) then
      commitIndex := index
    else
      use commitIndex
    end if
  end var
end function
```

B General distributed system modeling

B.1 Node skeleton

```
process GenericNodeSkeleton [Send, Recv, Crash, Other, ...] (...) is
  var
    ... -- Declare node state variables
  in
    disrupt
      ... -- Initialize state variables
      loop
        select
          -- Handle message reception
          Recv(...);
          ...
          Send(...) -- May trigger sending of other messages
        []
          -- React on other events: timeouts, local sensor reading, etc
          Other(...);
          ...
          Send(...)
        []
          ...
        end select
      end loop
    by
      Crash() -- Crash at any time
    end disrupt
  end var
end process
```

B.2 Reliable Network

```
process ReliableNetwork [Send, Recv: Comm] is
  var
```

```
    n: nat,
    orig: nat,
    dest: nat,
    payload: Payload,
    msg: NetworkMessage,
    msgBag: NetworkMessageSet
  in
    msgBag := {};
    loop
      select
        -- Obtain a message, and store it in the bag
        Send(?orig, ?dest, ?payload);
        msgBag := insert(NetworkMessage(orig, dest, payload), msgBag)
      []
        -- Transmit a message from the bag, in no particular order
        n := any nat where (n > 0) and (n <= length(msgBag));
        msg := element(msgBag, n);
        Recv(msg.orig, msg.dest, msg.payload);
        msgBag := remove(msg, msgBag)
      end select
    end loop
  end var
end process
```