

Timed Distributed Algorithm Analysis and Verification

Justin Loew

University of Illinois at Urbana-Champaign

Siebel Center for Computer Science

Urbana, IL 61801

jloew2@illinois.edu

December 14, 2017

Abstract

This is a case study of the verification of selected properties of Lamport’s distributed mutual exclusion algorithm using the Maude tool.

1 Introduction

We’re designing a device with multiple components that operate asynchronously. The components may exist at the block level within the same die, at the board level in separate subsystems, or anywhere in between. We have some guarantee of reliability; if one subsystem fails, the entire device is said to have failed, and we do not need to account for any redundancies. These separate subsystems must cooperate to share some resource, which may be as simple as a shared non-volatile RAM (NVRAM) or as complex as a wireless transmitter.

Without some sort of coordination, these components may access the shared resource in ways that produce undesired, unpredictable, or even unsafe ways. It is imperative that access to the shared resource be arbitrated in a manner which can be proven safe.

There are many tools and techniques available to model distributed systems and formally verify various invariants and properties. We take a look at one such example, Maude. Specifically:

- We create a model of Lamport’s distributed mutual exclusion algorithm.
- We specify a few safety- and correctness-critical invariants of the algorithm.
- We prove the same invariants using reachability logic techniques.

In Section 2 we summarize the foundational work in the synchronization of distributed systems, from which we build our model. In Sections 3 and 4 we describe and present a formal model of the system. In Section 5 we describe our steps taken in verifying the stated property of the system. And in Section 6 we describe our plans for expanding upon this work moving forward.

2 Background

Lamport’s bakery algorithm [1] is better known than Lamport’s distributed mutual exclusion algorithm. We focus on a system of nodes sending messages to each other asynchronously.

Lamport timestamps Mutual exclusion means no two accesses occur at the same time. This implies that one access event happens before the other. Determining which event happens first is trivial with synchronized clocks by simply comparing timestamps. However, synchronizing clocks amongst all nodes in a distributed system is a much more difficult task. (In fact, it’s impossible if nodes are permitted to fail [3].) In order to determine an ordering of (unspecified) events in a distributed system, we need a way to compare which event happens before the other. Lamport timestamps [2] are commonly used as logical clocks to define a partial ordering on distributed events. A Lamport timestamp can be thought of as a sort of event counter; the system works as follows. Each node in the system has an initial timestamp of 0. Whenever an event takes place at a node, the node adds 1 to its timestamp. When a node sends a message, it adds 1 to its timestamp, tags the message with the updated timestamp, and sends it. To receive a message, a node first sets its own timestamp to either itself or the timestamp in the incoming message, whichever is greater. The node then increments its timestamp, then processes the message.

Lamport’s distributed mutual exclusion algorithm Synchronization is achieved at each node using a request queue and message passing. Each node maintains a queue of pending requests, sorted by the Lamport timestamp associated with each request. A node is permitted to access the shared resource and enter its critical section only if the node is the originator of the request currently at the head of its queue. When a node decides that it wants access to the shared resource, it first creates a new request and adds it to its own queue. It then sends a message to each other node in the system to inform them of the new request. All other nodes then reply with an acknowledgment of the new request. Once all replies have been received, the request is eligible for execution on its originating node (once it reaches the head of the queue, of course). Upon completion of its critical section, the node relinquishes use of the shared resource by sending a message to each other node in the system notifying them that the current request has been completed; no acknowledgment is sent for this notification. All nodes then remove the request from their queues.

Related Work Sedletsky et. al. [5] have completed a formal verification of the mutual exclusion and liveness properties of the Ricart-Agrawala algorithm [4] (an improved version of Lamport’s distributed mutual exclusion algorithm) for an arbitrary number of nodes using the Stanford Temporal Prover (STeP) software package.

3 Modeling Approach

In order to determine what our model will look like, we must first decide on our operating conditions. We adopt the following assumptions:

- **Membership set stability** Nodes can never fail, and no new nodes join the system during execution.
- **Lossless message transmission** Inter-node messages are never lost or corrupted, though they may be delayed an arbitrary amount of time. Note that this means that messages may be sent in a different order than they are received.
- **Atomic intra-node transitions** Each node is able to update its own state atomically. This is representative of the common case of a device which can use locks, queues, or other local synchronization mechanisms to ensure one message is processed completely before the next message is handled.
- **Sanity of Scale** This is leaning towards pedantry, but for the sake of completeness: a node will never run out of memory; no timestamp will grow too large, and nodes are able to comprehend the number of nodes in the system.
- **A One Track Mind** Each node completes execution of the critical section of its current task before requesting another task. This is an artifact arising from our current implementation related to how `ack` messages are managed; we intend to remove the need for this assumption in the future.

4 Model Implementation

We now take our first look at the design of the model.¹ For the curious, the full source code is available at <https://gitlab.engr.illinois.edu/jloew2/lamport-modeling-and-verification>.

At the top level, we have a system of nodes, with messages being sent back and forth between them. We model this syntactically as a “soup” of associative and commutative objects and messages (an “AC soup”). To send a message, a rewrite rule is simply added to create a new message in the AC soup.

Three types of messages are in use, just as in the Lamport algorithm’s description: a `request` to access the shared resource and enter the critical section, an `ack` to confirm receipt of a resource request, and an `exeDone` message to let other nodes know when a node has finished executing its critical section.

Next (and perhaps most significant) is the representation of a node. A node must contain several elements:

- A unique identifier used to refer to each node; we use a natural number. This is used when addressing messages, similar to an IP address.
- A Lamport timestamp `ts`, to create a partial order on the events; more on this shortly.
- Three queues:
 - A `runQueue` containing the list of pending tasks for all nodes; this is the queue explicitly mentioned in the above description of Lamport’s distributed mutual exclusion algorithm.

¹This design description is accurate as of commit 0b88e514.

- An `ackQueue` as a temporary storage area for received requests for access to the shared resource. When a node receives a `request` message, it adds an entry to its `ackQueue` before sending its `ack`. This is done to model asynchronicity in the reply.
- An `exeDoneQueue`. Again, this enables a node to send an `exeDone` message asynchronously after the node completes the critical section of a running task.
- An execution state `exe` to keep track of whether the node is currently executing the critical section of a task.
- A count `stimuli` of the number of new tasks the node can spawn. This number is fixed at the time of the node’s creation, and is present in order to provide a bound on the number of possible states of the system. It is an implementation artifact which arose due to time constraints and it has no basis in reality. Its presence makes us uncomfortable because it is an artificial limitation on the execution of the system. We are aware of techniques to model and verify the same properties without resorting to such limitations, and its removal is a priority in our future work.
- A `numPendingAcks` count which represents the number of acks a node has yet to receive before it can enter the critical section of its task. In the future, we expect that associating this count with each task in a node’s queue instead of per-node will enable nodes to request multiple tasks at once.
- `everyOtherNodeID`, a list of every other node’s identifier. This is another artifact arising from our implementation of the model, but significantly less limiting than certain other artifacts; it is used in a similar manner to an IP routing table in order to broadcast messages.

Each task is fairly simple; it contains the ID of the node executing the task and the Lamport timestamp which is used to place the task in the correct sorted position in each node’s `runQueue`.

Finally, each timestamp contains a simple counter used as a Lamport timestamp, and the unique node ID of its “owner” node. The node ID is only used to break ties when comparing timestamps in a globally stable manner.

A summary of the types used is given in Figure 4.

5 Verification of the Mutual Exclusion Property

In order to verify a mutual exclusion property, we must first define the property. We define mutual exclusion to mean no two nodes execute a critical section at the same time. More formally, we define a simple predicate `bothInCritSec` which does just what its name implies: it determines whether two given nodes are both executing their critical sections at the same time.

Given a correctly specified system, Maude makes proving an invariant from a predicate simple. We hand Maude an initial state and ask it to search the reachable state space for a state satisfying a given predicate. For example, we can set up two nodes, then ask Maude to search for a state where both node 1 and node 2 are in their critical section at the same time using our `bothInCritSec` predicate. Code for the predicate and the search is shown in Figure 4. The output² of the search

²Output extracted from the output produced by running `tests.maude` at commit `eef834a8`.

```

--- Seen below:
--- Nat = Natural number
--- Oid = Object ID (used as a Node ID)
--- Cid = Class ID (part of a larger framework; we just use Node)
--- Msg = Message passed between nodes

mod TIMESTAMP is
  protecting NAT .
  including CONFIGURATION .
  sort Timestamp .
  subsort Nat < Oid .
  op @_ : Oid Nat -> Timestamp [ctor] .
  op #_ : Timestamp Timestamp -> Timestamp . --- update timestamp, breaking ties
  op #_ : Timestamp -> Timestamp . --- unary equivalent
  op ts : Timestamp -> Nat . --- get time val
  op referenceFrame : Timestamp -> Oid . --- get owner's reference frame
  op _< : Timestamp Timestamp -> Bool .

mod TASK is
  protecting TIMESTAMP .
  including CONFIGURATION .
  --- NETaskQueue is a non-empty TaskQueue
  sorts Task NETaskQueue TaskQueue .
  subsort Task < NETaskQueue < TaskQueue .
  --- executor, timestamp
  op initTask : Oid Timestamp -> Task .
  op initTask : Oid -> Task .
  op executor : Task -> Oid .
  op ts : Task -> Timestamp . --- timestamp
  op _< : Task Task -> Bool .

mod NODE is
  protecting TASK-QUEUE .
  protecting TIMESTAMP .
  protecting NAT .
  including CONFIGURATION .
  sort ExecState .
  subsort Nat < Oid .
  ops CritSec NotCritSec : -> ExecState .

  --- Node + Node attributes
  op Node : -> Cid [ctor format (rn o)] .
  op ts :_ : Timestamp -> Attribute [ctor gather (&)] . --- Lamport timestamp
  op runQueue :_ : TaskQueue -> Attribute [ctor gather (&)] .
  op ackQueue :_ : TaskQueue -> Attribute [ctor gather (&)] .
  op exeDoneQueue :_ : TaskQueue -> Attribute [ctor gather (&)] .
  op exe :_ : ExecState -> Attribute [ctor gather (&)] .
  op stimuli :_ : Nat -> Attribute [ctor gather (&)] .
  op numPendingAcks :_ : Nat -> Attribute [ctor gather (&)] .
  op everyOtherNodeID :_ : OidSet -> Attribute [ctor gather (&)] .

  --- Format: destination, timestamp, task
  ops request ack exeDone : Oid Timestamp Task -> Msg [ctor format (gn o)] .

```

Figure 1: A summary of the type definitions used in the model.

command is shown in Figure 4. Since no solution is found, there does not exist a reachable state satisfying the `bothInCritSec` predicate, achieving our proof goal.

Most of the work is in defining a valid model specification; the actual proof obligations are quite simple and elegant, requiring only three commands spread across 6 lines of code.

```

--- NODE:
--- destination, timestamp, task
ops request ack exeDone : Oid Timestamp Task -> Msg [ctor format (gn o)] .
ops sendRequestToAllOtherNodes sendExeDoneToAllOtherNodes :
  Task Timestamp OidSet -> Configuration .

vars N M : Oid . vars I TNC PAQ : Nat .
vars T U : Task . var Q R : TaskQueue . var AQ : NETaskQueue .
var X : ExecState . var A : AttributeSet .
var ENID EONID : OidSet .
vars TM TM2 : Timestamp . --- timestamps

rl [send-request] :
  < N : Node | stimuli : s(I) , runQueue : Q , numPendingAcks : PAQ ,
    everyOtherNodeID : EONID , ts : TM , A >
=>
  < N : Node | stimuli : I , runQueue : insert(Q, initTask(N, # TM))
    , ts : # TM , numPendingAcks : sizeof(EONID) ,
    everyOtherNodeID : EONID , A >
  sendRequestToAllOtherNodes(initTask(N, # TM), # TM, EONID)
  if PAQ == 0 .

rl [receive-request] :
  < N : Node | ackQueue : Q , ts : TM , A >
  request(N, TM2, T)
=> < N : Node | ackQueue : insert(Q, T) , ts : TM2 # TM , A > .

rl [send-ack] :
  < N : Node | ackQueue : T AQ , ts : TM , A >
=> < N : Node | ackQueue : AQ , ts : # TM , A >
  ack(N, # TM, T) .

rl [recv-ack] :
  < N : Node | numPendingAcks : s(PAQ) , ts : TM , A >
  ack(N, TM2, T)
=> < N : Node | numPendingAcks : PAQ , ts : TM2 # TM , A > .

rl [begin-using-resource] :
  < N : Node | exe : NotCritSec , numPendingAcks : PAQ , runQueue : T Q ,
    ts : TM , A >
=> < N : Node | exe : CritSec , numPendingAcks : PAQ , runQueue : T Q ,
    ts : # TM , A >
  if PAQ == 0 /\ executor(T) == N .

rl [end-using-resource] :
  < N : Node | exe : CritSec , runQueue : T Q , exeDoneQueue : R , ts : TM , A >
=> < N : Node | exe : NotCritSec , runQueue : Q , exeDoneQueue : R T ,
    ts : # TM , A > .

rl [send-exeDone] :
  < N : Node | exeDoneQueue : T Q , everyOtherNodeID : EONID , ts : TM , A >
=> < N : Node | exeDoneQueue : Q , everyOtherNodeID : EONID , ts : # TM , A >
  sendExeDoneToAllOtherNodes(T, # TM, EONID) .

rl [recv-exeDone] :
  < N : Node | runQueue : Q T R , ts : TM , A >
  exeDone(N, TM2, T)
=> < N : Node | runQueue : Q R , ts : TM # TM2 , A > .

```

Figure 2: A listing of the rules used to describe valid node state transitions.

```

--- Predicates
mod NODE-PREDS is
op bothInCritSec : Oid Oid -> Prop [comm] .
vars N M : Oid . vars A B : AttributeSet . var C : Configuration .

eq < N : Node | exe : CritSec , A >
  < M : Node | exe : CritSec , B >
  C |= bothInCritSec(N, M) = true .
eq C |= bothInCritSec(N, M) = false [owise] .

-----
--- Actually prove the invariant ---
-----

--- No two nodes are ever in their critical section at the same time
search initSystemWith 2 nodes, 2 stimuli =>* C:Configuration such that
  C:Configuration |= bothInCritSec(1, 2) .

```

Figure 3: A formal definition of the predicates used.

```

=====
search in NODE-TEST : initSystemWith 2 nodes,2 stimuli =>* C:Configuration such that C:
  Configuration |=
    bothInCritSec(1, 2) = true .

No solution.
states: 1  rewrites: 20 in 0ms cpu (0ms real) (714285 rewrites/second)
Bye.

```

Figure 4: A formal definition of the predicates used.

6 Future Work

In addition to the tidbits of future work mentioned above, we have several items on the agenda:

- *Correctly* prove the mutual exclusion property. Our proof only finishes executing when run before Lamport timestamps were implemented. The results shown above are for demonstration purposes only, and prove the correct invariant for a variation of the desired Lamport algorithm instead of the algorithm as actually described above. We have a model of the correct algorithm, but the proof seems to be non-terminating; we intend to merge the two in the near future.
- Prove more invariants, of course. Both for correctness and for practice, we intend to prove the following invariants (in increasing order of expected complexity):
 - Each node has a unique node ID.
 - Lamport timestamps only ever increase.
 - The system never deadlocks.
 - The system never livelocks.

- Forward progress: every task will eventually complete.
 - While a node is executing its critical section, the head of its queue always contains its own node ID.
- Explore proving the above invariants using LTL syntax in Maude.

References

- [1] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.
- [2] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.
- [3] Nancy A. Lynch Michael J. Fischer and Michael S. Paterson. Impossibility of distributed consensus with one faulty process, 1982.
- [4] Glenn Ricart and Ashok K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM*, 24(1):9–17, January 1981.
- [5] Ekaterina Sedletsy, Amir Pnueli, and Mordechai Ben-Ari. *Formal Verification of the Ricart-Agrawala Algorithm*, pages 325–335. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.