Modeling and Verification of Distributed Virtual time Algorithms

Dara Darling

Motivation

Distributed systems are an important part of our day-to-day life. The largest of which is the internet but also the ways that we communicate aren't too dissimilar to a distributed system. The benefits of such systems are to be able to parallelize processes, backup data, or allow people of vastly different geologic locations to communicate. Strictly speaking, a distributed system doesn't share a clock and doesn't share physical memory. However, many of the ideas of these systems can be expanded to refer to any sort of concurrent programming.

Concurrency is a devilishly difficult thing to debug. I have personally experienced much hardship in this area. For one, debuggers aren't designed to follow multithreaded processes. On the other hand, Spin can help show traces of specific behavior making certain issues easier to troubleshoot. Furthermore, I had the opportunity to talk with a professor about the theoretical side of Distributed systems so I figured that this would be the perfect system to explore in greater depth.

System

The basics of distributed stems starts by the description of a directed graph. Each node of the graph is a process and can do some amount of work. The edges are directed message passing channels between each node in the graph. Each process can experience a variety of events, a computation event which happens internally, a send event when a message is put into the out buffer of the process, a receive event for when a message goes from a process' in buffer into the process itself, as well as a few others we won't worry about for now. The most important of these are the send and receive events. This whole system can be described as an I/O automaton, and you can see a snapshot of each step as well as the transition between each of these snapshots. Overall, these lined up make an execution.

Distributed systems can be subdivided into synchronous and asynchronous systems. In fact, various aspects of distribution can be considered synchronous and asynchronous. With process and communication synchronicity, you can think of events as being queued up so that each new event is queued up and execute in a round. Each round the events get set up, run, and set up the next round. Broadcast is when a message is sent

to all of a process' neighbors at the same time, as apposed to point to point. Then sometimes the message channels are considered to have a first in first out scheme while in other systems, this may not be guaranteed.

The higher level of synchronization, the more guarantees you can have on your system though it may be harder or impossible to actually implement them in their entireties. Often that is why it is useful to simulate some of the capabilities of a synchronous system in an asynchronous one. Here we will explore the idea of virtual time which simulates global time. This can be used in many areas such as showing use in message passing synchronicity.

Lamport time

A system for creating a virtual clock. Each process has it's own clock, whenever an event happens on a process, the clock is incremented by one. If the event is a send event, it will send it's clock time as a time stamp along with it's message. If it is a receive event, it will compare it's own process time to the timestamp on the message, take the max of the two, increment by 1 and have that be the new clock. Ties can be dealt with a few ways but an easy way to deal with them is by taking whichever process has the smaller id as being the smaller number.

Setup

With these virtual clocks we want to see ways of testing their usefulness. They both are more or less by definition satisfying the three main ideas, if a message is received, it's send event happened before it was received. Then, if two events happen on the same process, they have the same order as they would in the original execution. Lastly, you must be able to create a total ordering from the partial ordering. For example, if two events have the same time, then consider the one with the lower process id to have happened first. Because of this, we want to find some ways to test the capabilities of these virtual clocks. While I spent quite some time trying to figure out the best way to do this, I realized that the Lamport Time paper describes it's use in it's original paper and Vector Clocks goes on to show how a similar algorithm fails with Lamport time but works with Vector Clocks. Specifically, they do this by exploring a Mutual exclusion problem.

Mutual Exclusion

Often in distributed systems, there is some sort of shared resource that each process wants access. Often these break when multiple processes are accessing it at the same time. In other words, the resource must be mutually exclusive with whichever process has access to it at this moment at a time. If the idea is to have each process go in the order that they make a request, and each process keeps a que so it knows when to go, you may run into a problem where you try running it without a virtual clock.

First off, it is assumed that the system doesn't have some physical clock to go off of so how a process knows which gets pushed up or down the que. What if you just put them on the queue on the order that the request is received. A process sees it's own request right away. Let's say that process 0 creates a request, adds it to it's que and sends it to process 1 to inform them of this request. Before that message is received, process 1 decides to create a request, put it on it's own que and sends out it's intent to use the shared resource. Depending on the implementation, this may allow both access at the same time or it may lead to a deadlock.

Since we saw that there are issues with our mutual exclusion algorithm, let's see how Lamport time fixes it. We see that the previous implementation has issues with not keeping things exclusive and with deadlock. I tested mutual exclusion by having a section in each process increment a variable, check the value of the variable, and then decrement the variable. At the check, if ever that variable == 2, then something has gone wrong, and two processes have reached the critical section. Since I was wanting to test with 3 processes sending 3 request, I could check for deadlock by having the requirement, eventually timesInCritSection will == 9.

Lamport Tests

With the previously mentioned algorithm of incrementing clock for each send and receive event as described above, we can do something special. Now we can tell whether something could have happened before. If you have two events, one with timestamp 7 and another with timestamp 12, you know that the timestamp 7 event couldn't have been influenced by the event with timestamp 12. We can use these timestamps to decide on which order processes should have access to the shared resource. This was the main section I intended to test in spin however I ran into issues and couldn't complete it. I was very much interested at looking at how it dealt with multiple requests from the same process since I think that can lead to issues though I couldn't tell for certain. It also would probably have issues with random send receive instead of first in first out message channels. You can look at more of the specifics for the design of this model down in the appendix or you can find it in ...\applied-formal-methods-final-project-hivemind\Spin\mutalOutline.txt.

Take Away

A lot of the constraints were able to be strictly followed since, in theory, we don't have to deal with the overflow of integers with logical systems. We also can't go over an infinite trace in finite time. Because of that, I tried switching gears to verifying systems that I thought would fail if and only if the infinite system would fail but I was unable to do even that in the timeframe.

Acknowledgement

Thank you to Professor Soma Chaudhuri for helping me understand the intricacies of distributed systems and the papers that describe them.

Bibliography

1. Lamport, Leslie. “Time, Clocks, and the Ordering of Events in a Distributed System.” Communications of the ACM 21, no. 7 (July 1978): 558–65. https://doi.org/10.1145/359545.359563.

2. Mattern, Friedemann. “Virtual Time and Global States of Distributed Systems.” Essay. In *Parallel and Distributed Algorithms*. North-Holland, n.d.

3. Holzmann, G.J. “The Model Checker Spin.” *IEEE Transactions on Software Engineering* 23, no. 5 (May 1997): 279–95. https://doi.org/10.1109/32.588521.

4. Rozier, Kristin. “Applied Formal Methods Homework 3: Classifying Specifications and Explicit-State Model Checking with Spin.” IowaStateAerospaceCourses-Rozier, 2024. https://github.com/IowaStateAerospaceCourses-Rozier/homework-3-spin-Migs657.git.

5. Netymk, Albert. “Linear-Time Temporal Logic (LTL) Formula in Spin.” albertnetymk’s notes, July 20, 2020. https://albertnetymk.github.io/2020/07/18/ltl/.

6. “Basic Spin Manual.” Basic spin manual. Accessed December 17, 2024. https://spinroot.com/spin/Man/Manual.html.

7. Lynch, Nancy A. *Distributed algorithms Nancy A. Lynch*. San Francisco, Calif: Kaufmann, 1996.

8. Attiya, Hagit, and Welch. *Distributed computing: Fundamentals, simulations, and advanced topics*. Wiley-Interscience, 2004.

Appendix

general Lamport necessary guaranties

C1. if a and b and events in process P_i, and a comes befor b, then C_i<a> < C_i<b>

C2. if a is the sending of a message by process P_i and b is the receipt of the message by process P_j, then C_i<a> < C_j<b>.

IR1.each process P_i increments C_i between any two successive events

IR2.

a. if event a is sending of message m by process P_i, then the message m contains a timestamp T_m = c_i<a>.

b. upon recieving a message m, process P_j sets C_j = MAX(C_j,T_m)+1

break ties with arbitrary total ordering. if C_i = C_j the C_i has higher priority iff i < j.(and two processes cna't have same index)

possible points of failure

multiple processes in the critical section

must be in the proper order

specifically in the order they are made but I need a definition of what they mean by MADE

not every recource is granted

system definition

fixed collection of processes

shre single recource

Only one process can use the resource at a time must synchronize to avoide conflict

award a process to use recource if it meats conditions

if a process that has been granted must release it befor being granted to another

Different request must be granted in the order they are MADE

if every process which is granted a resource releases it, every request is eventually granted

assume that a process already has the recource at the beggining

implement rules IR1, and IR2(though not for the failed on)

each process maintains it's own que(no other process can see the other's ques)

inial configureation of the que T_0:P_0 requests recource

so at timestamp 0 P_0 requested acess.

T_0 must be less than the inital value of ay clock so set all inital clocks to 1

use a global variable as the shared recource

events that can happen

request

P_i send "T_m:P_i request" to all Processes(including to self/puts own request on que)

recieve request

P_j recieves the message "T_m:P_i request" puts it on it's request que and sends a timestamped

acknowledgment to P_i(may be unceccicary if P_j already sent P_i a message with time stampt greater than T_m)

release recource

P_i release. remove T_0:P_i from queue and sends a timestamped message "T_m(after):P_i release" to all processes

recieve release

P_j recieves P_i release recource message, remove any T_mLP_i requests resource message from it's request que

P_i granted resource if the following is satisfied

i. T_m:P_i requests resource message in head of it's request que

and

ii. recieve acknowledge from all others greater than the activity

algorithm steps(each rule is considered single event (implemented as automic and incrementing clcok))

1. To request resource, P_i sends the message T_m:P_i requests resource to every other process, and puts that message

on it's request queue where T_m is the timestamp of the message(isn't it possible for two timestap requests to be the same number?)

assumptions

any two processes P_i, P_j the message sent from P_i to P_j is received in the same order sent from P_i(assume fifo que)

assume every message is eventually recieved (assume reliable message chanel)

assume complete graph so each process can send messages to every other process

assumtion I added

at least for now, say that a process can't have multiple requests pending

each can only request once

interesting other inquaries

have centeral scheduling process which grants requests won't work without other assumptions being made

try with out the fifo que

try without reliable message channel

above two can be avoided by introducing message numbers and messsage acknowledgment protocols.

try without complete graph(though I'd have to redefine the problem or allow for message forwarding I think)