Ira A. Fulton Schools of Engineering
Arizona State University
Engineering | School of Computing, Informatics, and Decision Systems Engineering

CSE 531: Distributed and Multiprocessor Operating Systems

Logical Clock Written Report

By Hannah Ajoge

Introduction

Theoretical investigations of multiprocessor systems usually treat processors as being distributed spatially[1]. Distributed applications usually need to maintain logical time in an asynchronous message-passing system. This is achieved with the use of Logical clocks. These Logical clocks track causality which helps to determine the extent with which the past execution could possibly affect actions at any process [2]. As a result, Logical time and causality are critical in the design of distributed applications. The most noteworthy of Logical clocks are Lamport timestamps (monotonically increasing software counters), Vector clocks (they allow for partial ordering of events in a distributed system), Version vectors (which order replicas based on updates in an optimistic replicated system) and Matrix clocks (an extension of vector clocks with information about other processes' views of the system) [3].

Among the different types of Logical clocks, Lamport's logical clock is considered as a cornerstone in the parallel and distributed computing areas; as such various problems in multiprocessor systems have been treated with Lamport's logical clock [1]. The idea is that given a pair of operations, there should be an obeying of some logical time order obtained by logical clock (e.g. processor order or execution order). Thus, the logically later operation should observe and proceed accordingly by the effort of the logically earlier operation. Based on the logical time orders between all pairs of conflicting operations in a system, the final execution result of the system is determined. Thus the logical order information or the happened-before relationship, "if perfect, can reveal some intrinsic features of parallel and distributed computing without a global lock which was considered hard-to-achieve"[2]. Because of imperfection, extensions of Lamport's Logical clock has come into existence. An example is a tool by Tong *et al.* 2018. The tool extends Lamport's logical clock to in addition to preserving causal (happened-before) relationships, to also maintain "predicted application time by incorporating non-unit computation time and non-unit communication time" [4]. With Tong's tool each network configuration has one logical clock. According to Tong *et al.*: "The logical time for each event represents the predicted time for the event for the given network configuration. The tool assumes a starting event such as the default MPI_Init and computes the timing for all following events." This application/extension of the Lamport's Logical clock provided notable results like: (1) Fast classification ability to predict execution times for many network configurations in a single run. (2) Multiple-prediction capability

which made previously computationally intensive analysis possible. (3) Ability to analyze applications for their sensitivity to compute speed, latency and bandwidth. (4) Better understanding of application performance limiting factors.

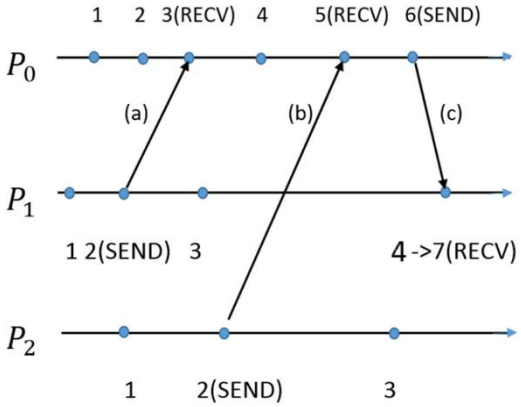

Fig 1: Example of Lamport's Logical Clock [adopted from Z. Tong, S. Pakin, M. Lang and X. Yuana, 2018].

Problem Statement

Modern banks have many branches in different cities in a country or all over the world. Therefore, there could be multiple transactions on the same account in different branches/locations. This means that bank database server performance should be kept optimal [5]. Distributed banking systems provide an alternative to centralized bank-based systems; and in 2017, distributed banking was projected to reduce the world's unbanked population by more than a third [6].

This project is an extension of the first project(gRPC). The intention is to implement a Lamport's clock algorithm which is built on a distributed banking system in which specific customer communicate with specific branch and carries out transactions (query on balance, withdrawal and deposit). This specific branch to customer communication is supposed to be unique ID based. As each customer update a specific replica branch, there should be a system for that replica to update other replica for efficient distributed banking. At the end of the day, it is expected that the overall bank balance is the same across branches. This branch to branch communication is the part of the project that will benefit from the implementation of the Lamport Logical clock algorithm.

Goal

The goal of this project is to enable the implementer/programmer/student implement a Lamport's logical clock on a distributed banking system that is implemented using gRPC API. The implementation should fulfill the above problem statement. As a result of the implementation, the specific goals of the project is to ensure that the student is able to:
a. Understand the basics concept of Lamport's logical clock and also implement the algorithm.

b. As such implement interconnected sub-interfaces in the correct locations (emphasis on the branch to branch communications concerning deposit and withdrawal of funds).
c. Execute the specific order of six interconnected sub-interfaces, that update the local clock of the Branch processes.

Setup

This project was executed in Python using gRPC framework which is based on Protocol Buffers language (proto3) and runs over HTTP/2. All the implementation and testing was carried out in Ubuntu 20.04.2 LTS using Visual Studio Code. There is also an assumption that any person who wish to run the code will have the appropriate libraries.

Implementation Processes

This project is an extension of the previous project. But since it is also stands alone and possibly graded by someone else, I will provide as much as possible, a summarized form of the implementation, which borrows heavily from the previous project. The project was implemented as following in a single folder which I named HannahAjoge_LamportClock_Project_Code:

A. **Definition of service in a .proto file:** Here I implemented my .proto file, which I named as example.proto. As expected this file consisted of a bunch of definition to describe the service that the server provides. First, I described the syntax to use specific protocol buffer version 3 (proto3). proto3 is recommended over the default proto2, because proto3 allows the use of the full range of gRPC-supported languages, as well as avoids compatibility issues with proto2/3 clients-servers communication [7]. I did not specify a package, since I have only a single .proto file in the folder. Secondly, I defined a service named RPC (taking hint from the class in the given Branch.py file) which defines the interaction of the servers with the clients. Unlike in the previous project where I had a single RPC, in this project, I felt two RPCs will be better to enable better implementation of the logical clock, especially in respect to the branch to branch communication. As previous I had an RPC called MsgDelivery (for the Customer to Branch communication) and an additional RPC called ClockUpdate (to handle the Branch to Branch communication). Finally, I implemented the message schema's data types to be structured as strings. The idea here, is that the Customer sends a string consisting of a dictionary of transaction requests. The Branches also sent messages in form of strings.

B. **Automatic generation of client and server stubs for service using protocol buffer compiler:** The result was two files - example_pb2.py (containing generated protocol buffer code) and example_pb2_grpc.py (containing client and server classes corresponding to protobuf-defined services). I executed the following command from within the folder in the terminal: *python -m grpc_tools.protoc -I./ --python_out=. --grpc_python_out=. example.proto*

C. ***Input test file:*** The provided test input was copied and pasted into a file and saved as "input.json". This json file contains a list of Customers and Branch processes.

D. **Implementing the servers (Branches):** The aim here is to ensure a happened-before relationship of multiprocessed events. The Branches accept a message from Customer and utilize a class containing four methods to process the request. Based on the clock timestamp on the strings of events, the local clock is updated. Since the aim of the project here is the implementation of logical clock and not the balance, I did not generate balances. The Branch execute the transaction, produce a log of events and the logical timestamps and replies with a string of successful execution to the Customer. The four methods of the Branch class, ensures that the six interconnected sub-interfaces are implemented in the right order and appropriately:

a) The eventRequestExecute method handles how a message from the Customer is handled before propagation. That is the Event_Request and Event_Execute sub-interfaces. The appropriate events are generated with time stamps and logged into a file (which is generated using self.branchfilename). Here what is important is the Customer and the local time of the Branch. To keep things consistent with the expected output file, it is assumed that the Customer local clock is 1 and the Branch local clock is also at inception. Thus, the local two-step operations which has no influence from another Branch is enforced. The resulting subevents which is logged along with the timestamps meets expectation. Which is that the Branch calculate the maximum of the local clock and the Customer's clock and increment by 1 and to assign that to Event_Request events. Similarly, the subsequent event (Event_Execute) is implemented after another clock increment.

b) The ClockUpdate method handles the events in a Branch that receives propagated message, that is the Propagate_Request and Propagate_Execute sub-interfaces. The receiving Branch compares the clock on the received message (Propagate_Request; which has the clock time from the sending Branch) to its local time. It gets the maximum of the 2 and increment it by 1. In the second step, the Branch increment the now local time by 1 before sending the message back to Branch that propagated the message. The first step (Propagate_Request)'s result and that of the second (Propagate_Execute) is logged into a file (self.branchfilenameP).

c) The propagateRequest method handles the events in a Branch that propagate the message, and it involves the result of the propagated message, that is the Propagate_Response and Event_Response sub-interfaces. The propagating Branch send an Propagate_Request message along with it's local timestamp to a list of Branches, other than itself (which is generated using self.others). The Branch then awaits for the other Branches to receive the messages, execute the ClockUpdate method and return a Propagate_Response event. The Branch on receiving the Propagate_Response message, increment the clock based on the maximum between the recieved message and local time. Once the Branch receives all the messages back, the now local time is incremented by 1. This time stamp is added to an Event_Response event. Both the Propagate_Response and Event_Response are logged into a file (which is generated using self.branchfilenameR). Multiprocessing lock ensures a happened-before relationship.

d) The MsgDelivery method puts it all together. It ensures that request from the client is appropriately executed by all the methods in the right order to ensure happened-before relationship.
e) In addition to the Branch class, a function (creatServer) ensures the servers are actively listening to the appropriate ports. The servers listen on specific ports and wait for request from specific clients so that the right responses are sent back to the clients. The specific port is calculated to be specific for a specific customer by ensuring that the port is equal to 50047 plus the unique branch ID. To run the servers, I execute the command "*python Branch.py*" in the terminal. In the given example case this resulted in three servers listening on ports 50048, 50049 and 50050. On receiving a request from the appropriate client, the server processes the request and respond as expected.
f) Finally, I ensured that running the server is done graciously. I ensured that on running the Branch.py file, any preexisting output file log folders are deleted to ensure that the server starts on a clean slate. Also, I ensured multiprocessing.

I did not implement methods to specifically work on either deposit, withdrawal or queries. Instead ensure that based on the content of the message a Branch receive from a Customer or another Branch, the Branch choose how to implement the specific subevent. In some situation, like in the case of a query the Branch may not execute a subevent as appropriate.

E. **Implementing the client (Customer):** Previously I stripped the Customer class of its createStud method. My recent better knowledge of how class is implemented in Python, enabled me to stick more closely to the given Customer Python file class structure. I updated the createStub and the executeEvents methods which were for both creating the Customer stub and sending out events to the branches. Summarily, the following is achieved in the createStub and executeEvents methods respectively:

a) gRPC channel that is specific for branch-customer transactions is opened. Just as in the server side, the calculation involves adding 50047 to the unique ID to ensure the appropriate server receives the request. With the channel, a stub is created.
b) To send out requests to the appropriate branch, a valid request message is created and then a call is made.

Finally, I ensured that on running the Customer.py file, the "input.json" file is read and process for sending request to the appropriate server. Then response and intermediary log files of events and timestamps are processed to the expected format and stored in a file ("output.json"). In addition to the requirement I added the two following operations:

a) That after each request has been responded to and written to the output file. The client should shut down the corresponding servers.

b) In addition to shutting down the servers (Branches) the client also remove intermediary outputs (the directories of log files produced by the server which keeps details of subevent executions and their timestamps) to produce

Results

On executing the implementation process explained in the preceding section, an output file (output.json) with the expected output is generated. I wish to succinctly explain the overall results and the justification as following:

a) Using the implemented Branch.py, Customer.py, example.proto and the appropriate input file (input.json); a distributed banking system was built. Most importantly a software/counter based Lamport's logical clock was implement. This system ensured that specific customer communicate with specific branch and carries out transactions (query on balance, withdrawal and deposit) appropriately based on unique IDs; and most importantly log the subevents and the timestamps.

b) Using three methods implement two each of the subevents, I was able to produce a successful result which ensures a happened-before relationship. Pivotal to this accomplishment is the use of multiprocessing, locks and an efficient Branch to Branch communication. The Branch to Branch communication followed the Lamport's clock algorithm and served as both client (propagating server) and server (receiving Branch) as needed. This is in agreement with the concept of multi-tiered architectures of the centralized distributed system organization, in which an application server can act as both client and server [8].

c) My implementation's algorithm was more geared generally to maintaining the needed happened-before relationship over the type of subevent or the type of transaction. As such I was able to implement my succinct four methods in the Branch to execute the six type of sub-events and the three types of transactions appropriately.

d) The implementation ensured that the branches are the servers and the customers are the clients in a bidirectional distributed network communication. Finally, to mimic the real life scenario of exiting connection with bank branches once a client has successfully completed their transactions, each client stub terminates the server after writing the expected result to the output.json file.

References

1. Chen, Yunji, Tianshi Chen, and Weiwu Hu. "Global Clock, Physical Time Order and Pending Period Analysis in Multiprocessor Systems." (2009): n. pag. Print.
2. Kshemkalyani, Ajay D. "The Power of Logical Clock Abstractions." Distributed computing 17.2 (2004): 131–150. Web.
3. Wikipedia contributors. "Logical clock." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 30 Mar. 2021. Web. 18 Apr. 2021.
4. Tong, Zhou et al. "Fast Classification of MPI Applications Using Lamport's Logical Clocks." Journal of parallel and distributed computing 120.C (2018): 77–88. Web.
5. Nugroho, Agus Cahyo, and Mychael Maoeretz Engel. "Hybrid Distributed Application in Banking Transaction Using Remote Method Invocation." Telkomnika 17.5 (2019): 2208–. Web.
6. Fan, Wenjun et al. "Blockchain-Based Distributed Banking for Permissioned and Accountable Financial Transaction Processing." 2020 29th International Conference on Computer Communications and Networks (ICCCN). IEEE, 2020. 1–9. Web.

7. The gRPC Documentaion. https://grpc.io/docs/what-is-grpc/introduction/
8. Arizona Sate University Spring B 2021 CSE 531: Distributed and Multiprocessor Operating Systems lecture notes.