CSE 531: Distributed and Multiprocessor Operating Systems

Definition

This section covers all the definitions specific to the context of this project' implementation.

Location: the location where a replica of an object belonging to a client is located.

Problem statement

Case analysis

The 3 goals of a distributed system are communication, coordination and synchronization.

Previously, project 1 goal was to build the abstract level of a banking distributed system, the communication goal. The system allowed the clients to communicate with the bank through the branches without concerning how, where or when the interactions occurred. The system placed the balance replicas across its network to improve the performance. The replicas came with the reliability issue. Their values differed when multiple accesses from the clients modified the replicas in different nodes.

Project 2 solved the issue partially by implementing Lamport's logical algorithm and aligned with achieving the coordination goal. The implementation gives the system a mechanism to compare the logical orders of the events among the nodes. So far, the system built upon lacks the enforcement policy in event orderings in different processes.

The policy dictates what event a process must complete first before another. It is a contract between the system and its applications. It guarantees a set of results is certain as opposed to the disordering among the processes. Project 3 aims to guarantee the certainty in

the uncertainly possible outcomes by granting a temporary exclusive access to the processes to the shared resources. Thus, project 3 is to complete the synchronization goal.

Problem statement

The bank wants to guarantee a set of certain results to the clients when they behave according to the rules defined by its enforcement policy.

Goal of the problem statement

Continuing from the projects 1 and 2, this project uses the agile project management methodology. The agile method breaks down the project goals from an epic theme into more granular levels which help the developers to have a clear scope of what should be done.

Epic theme

As the bank, it wants the clients to see consistent results of their inputs and outputs as they interact with different branches.

Business goals

- As a client, they want to see the most recent results of their actions regardless of how or where they happen.
- As a client, they want their interactions with the branch to be processed in a sequential order regardless of how or where they happen.

Technical goals

- As a designer, they want to enforce the monotonic write consistency in the system.
- As a designer, they want to enforce the read-your-writes consistency in the system.

Technical subgoals

- As a developer, they want the events to be entered sequentially according to the client's order of events as opposed to the system's order of events.
- As a developer, they want the results of the events to be seen by the clients immediately after the clients enter the events regardless of how or where they happen.

Assumptions

For the client-centric model to work, the project assumes the followings according to van Steen and Tanenbaum (p.378, Distributed systems, 2020):

- The client data is lack of simultaneous updates
- Conflicts can be easily resolved within a client process

Relevant technologies for the setup and their versions

The relevant technologies have not changed since the previous project

Set up (Technology Requirements)

- Built on: Windows 10 (19042.867)
- Tested on: Windows 10 (19042.867) and Ubuntu 20.10
- Programming language: Python v3 (3.8.5)
  Libraries:

Package	Version
grpc	1.31.0
grpcio	1.36.1
grpcio-tools	1.36.1
protobuf	3.15.6
MessageDelivery_pb2	1.31.0 (grpc generated)
MessageDelivery_pb2_grpc	1.31.0 (grpc generated)

logging	0.5.1.2
time	3.8.5 (Python built-in)

Implementation processes

Replica management

The consistency model

The project requirements suggest implementing the client-centric consistency model of monotonic writes and read-your-writes. The model is a ‘cheap’ form of data-centric consistency, such as eventual consistency and entry consistency (p.378, van Steen and Tanenbaum, Distributed systems, 2020). In other words, the eventual and entry consistency implementations focus on each client object’s data rather than the data consistency in system wide.

To implement the client-centric consistency model, the project considers that data stored inside a branch object are replicas of the clients’ information. A client interacts with the system as a whole but the client actually interacts with the closest branch (or also known as a datastore) to perform (RPC) operations. The datastore performs the client’s library functions rather than its own.

Semantics

The following semantics relate to the algorithmic implementation of the consistency model and help the readers of this documentation to understand how the program does what it does. They also set rules on how the system perceives an interaction between itself and a client.

- **A process** is considered a client object.
- **A location** is the logical location of the data of the clients stored inside a branch object.
- **A session** is a time period of how long an interaction between a client and the bank lasts.

Assumptions

The ambiguities in the project specification requirements create roadblocks during the implementation process. The followings are assumed during the implementation:

- A process of a client object accessing multiple locations during a given time period is considered one session. In order words, each client object can't have more one session during a given time period.
- Although a client object can access multiple locations during a given time period, it can only request sequential events. This means that for example, a client object can't withdraw and deposit money at the same time. It can only withdraw then deposit or deposit and then withdraw. In other words, no concurrent request is allowed.
- Although the eventual consistency is achieved by design in project 1, the project specification requirements don't mandate how long the total propagations should take. The project assumes all the client's most recent update requests will have been propagated to the other replicas after one session. This was assumed because if the project is not given the specifications and the data is not propagated after a client starts

the second session, the closest node has to perform invalidation checks which send a broadcast to all other nodes will hinder the performance of the system.

Replica management

Client cache

Previously, the balance replicas were multiplied across the branches to improve the performance. This project introduces a balance replica, write history and read history stored in the client cache. The new introduction comes with 2 benefits. The first benefit is that it improves access time. During a live session, if a client decides to query a balance multiple times, the client object would only have to access the local data store to fetch the result and perform an invalidation check once on the server rather than sending multiple requests to the nodes. Secondly, the history of the latest read and write operations are sent to a branch datastore to determine the priority order of the requests pertaining to a specific client.

Content distribution

The client-centric model will without a doubt be followed by the client-based distribution protocol. In this protocol, the client initiates the content validity check. The protocol is good since each client object has its own balance and the updates are less frequent (p392, van Steen and Tanenbaum, Distributed systems, 2020). Whereas the server-based protocol would require the system to keep track of all the client caches and push the updates as soon as a new one is entered. The computational power of cache tracking would take a toll on the system performance. Additionally, the project also considers the hybrid protocol, the lease-based distribution. Nevertheless, more project specification details are needed to justify for its complex implementation.

Implementation results

ID generation

Thanks to the accounting method implemented in project 2, all the events have been recorded properly by using Lamport's logical clock and pandas dataframe. In this project, each object in the write/read set is identified by 3 factors: the unique client id, its unique event id (eid) and the process id, previously also known as the branch id (pid) or dest in this project. These factors create a unique identification. To determine whether an event is propagated these factors will be matched against the accounting object, clock_event stored in a branch. Since these are chained events, any other event happens only either before or after the chain. Otherwise

client_events

This is an accounting object that stores all events generated by the client sequentially.

It takes in a list of clock_event.

client_event

This is an accounting object that stores an event requested by the client object.

It has 4 parameters: clientid(cid),processid(pid, also known as dest), eventid(eid) and name(also known as interface).

Enforcing monotonic write

The project enforces monotonic write by sending the writeSet list to the ‘dest’ branch. The branch will check the id comparing its accounting clock_events. If the clock_events variable

in the branch is behind the id. I will halt the current write process to wait for the others to propagate, else it's going to continue.

Enforcing read-your-writes

The project enforces read-your-writes by sending the writeSet list to the 'dest' branch. The branch will check the id comparing its accounting clock_events. If the clock_events variable in the branch is behind the id. I will halt the current read process to wait for the others to propagate, else it's going to continue reading.