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

Client-Centric Consistency Written Report

Introduction

The advent, growth, and relevance of the internet have increased the importance of distributed systems [1]. Distributed systems are a necessity for social media networks and gaming involving multiple players. Such systems usually involve users in different geographical locations. Replication allows clients/users from different geographical locations to be able to access shared data types with low latency [2]. Replication should be done transparently so that clients should not notice differences in the replica of data. Keeping all sites synchronized after each update is the ideal model. This model is also known as strong consistency or linearizability [2]. The disadvantage of strong consistency includes noticeable delay in propagation across the network, leading to low availability. On the other hand, high availability can be promoted by allowing not enforcing synchronization of updates. However high availability without synchronization for a specific process (client) cannot be tolerated in some situations, like in a banking system.

An important operation in a distributed system is monotonic writes [1]. In monotonic writes consistency, a process write on item x has to be completed (this may include being propagated to other nodes) before the same process makes any other write on x [3]. Note that this successiveness of write is specific to a process. In other words, another process' write order is not important here. While monotonic-write consistency is client-centric, it resembles data-centric FIFO (first in first out) consistency in that write operations have to be done in the correct order everywhere. The difference is that our focus is on a specific process (client) in the case of client-centric consistency.

Another client-centric consistency model is read-your-writes. In this situation, quoting from Tanenbaum and Steen, 2001, the "effect of a write operation by a process on data item x will always be seen by a successive read operation on x by the same process" [3]. This can invariably be seen as satisfaction of the monotonic writes may satisfy read-your-write consistency. The absence of read-your-writes consistency can be seen sometimes when updating Web documents [3]. Immediately viewing the update may not happen because the update has not been propagated. This behavior cannot be tolerated in certain distributed systems like that of a bank. In the bank system monotonic writes leading to propagation of writes across replicas and a resultant read-your-writes consistency is a must. For example, if process one execute three writes

monotonically, e.g. x1;x2;x3, the effect of x1 will always be seen by subsequent read operations (e.g. R(x2) and R(x3)) by the same process. A simpler example is if I have a bank balance of $200 and I (a process) made a deposit in the first write $200, then I withdrew in another write $370. Provided the first write is executed before the second, I will be able to read the effect of the first write (deposit) after my second write. So that (1) I will have enough balance to withdraw $370 and (2) I will be able to have a query that will return $30.

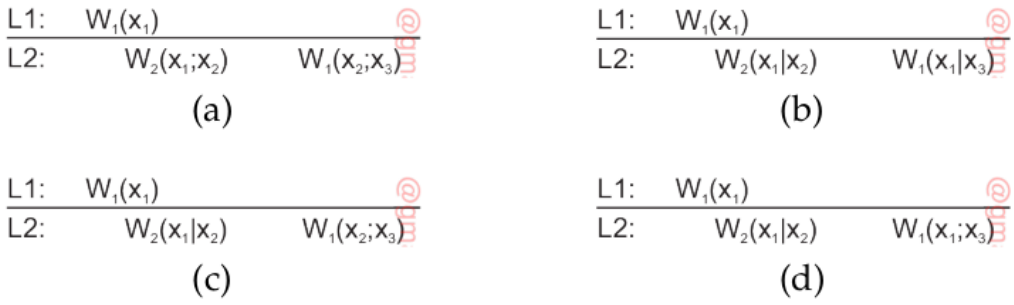


Figure 1: The write operations performed at two different local copies of the same data store. (a) A monotonic-write consistent data store. (b) A data store that does not provide monotonic-write consistency. (c) Again, no consistency as WS (x1 | x2) and thus also WS (x1 | x3). (d) Consistent as WS (x1 ; x3) although x1 has apparently overwritten x2. [Adopted from A. Tanenbaum and M. V. Steen, 2001]

Figure 2: (a) A data store that provides read-your-writes consistency. (b) A data store that does not. [Adopted from A. Tanenbaum and M. V. Steen, 2001]

Problem Statement

This project is an extension of the first project(gRPC). In the previous projects, the focus was the data, maintaining the data consistency regardless of which process is involved. The focus of this project is to implement enforcement of the client-centric consistency, specifically monotonic writes and read-your-writes consistencies. The focus here is a specific customer/process (client-centric) instead of different customers communicating with specific branches. Regardless of which branch the customer assesses to carry out transaction, the order of the write transaction should be monotonic. That is write order for the process should be followed and propagated across each branch before the subsequent write is executed in the right order. Secondly, the effect of previous writes should be seen by the same process in subsequent reads (read-your-writes).

Goal

The goal of this project is to enable the implementer/programmer/student to implement a client-centric distributed system using gRPC API, in which monotonic writes policy and read-your-writes policies are enforced. The implementation should fulfill the above problem statement. The specific goals of the project are to ensure that the student is able to:
a. Understand the basics concept of client-centric consistency with a focus on monotonic writes and read-your-writes consistencies.
b. Implement functions/methods that enforce the two consistencies.

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 wishes to run the code will have the appropriate libraries.

Implementation Processes

I implemented this project, using write sets to enforce the two client-centric consistencies. This enforcement will be handy, in a situation of concurrent multiprocessing of requests. It is interesting to note that although my aim was to achieve two customer-centric consistencies, a single implementation achieved both aims. I was quite fascinated by this that I discussed this with the instructor during a virtual event. The ability of a model of consistency to also result in achieving another model of consistency also agrees with the observation by Tanenbaum and Steen, 2001; that a data-centric FIFO model resembles and could achieve client-centric monotonic read [3].

Requests will not be executed until the previous write in the right order has been completely executed (monotonic writes consistency). Also, the effect of a write will always be seen by subsequent read operations by the same process (read-your-write consistency). The project was implemented in a single folder which I named HannahAjoge_ClientCentric_Project_Code. The folder has all the expected gRPC files (proto file, pb2 and pb2_grpc), client (Customer.py), server (Branch.py), an input file (input.json) for the read-your-write example and its output file (output.json). In addition, there is a folder (First_InputOutPut_Files) that contains the input and output files of the first run (i.e. the monotonic writes example). Below are more details on the implementation:

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 definitions to describe the service that the server provides. First, I described the syntax to use 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 a hint from the class in the given Branch.py

file) which defines the interaction of the servers with the clients. 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. 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 files:*** The provided test input was copied and pasted into a file and saved as "input.json" for each execution. This json file contains a list of Customers and Branch processes. Please note that for the read-your-write example, there was a typographical error (a coma after the query request). This typo was removed before execution as this was inconsistent with the code reading of a json file. The input json files are appended below in the appendix.

D. **Implementing the client (Customer):** I updated the createStub and the executeEvents methods which are both for creating the Customer stub and sending out events to the branches. To ensure that my requests from the Customer are multi-processed, I created a function called setupRequest outside the Customer class. The setupRequest function accepts a list and passes the list elements to Customer class. Then the createStub and the executeEvents methods of the Customer class are executed. Summarily, the following is achieved in the createStub and executeEvents methods respectively:

a) Within createStub, gRPC channel that is specific for customer-branch communication is opened. The specific port to be communicated with is deduced from the "dest" a unique identifier of the branch in the input file customer process. The calculation involves adding 50047 to the "dest" 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 inside executeEvents. The executeEvent method send a string with the information of (1) a unique identifier of the customer, (2) an event/transaction of deposit/withdrawal/query, and (3) a write set. This write set is critical for enforcing client-centric consistencies. The write set is generated by simply assigning consecutive integers starting from 0 to each subevent in the expected order of execution for that process. For example in the example input for monotonic write, the first event which is a deposit is assigned the write set of 0, the next (a withdrawal) is assigned a write set of 1 and the final (a query) is assigned a write set of 2.

Finally, I ensured that on running the Customer.py file, the "input.json" file is read and process for sending requests to the appropriate server. Then response with the last write set is stored in a file ("output.json"). In addition to the requirement, I ensured that after each request has been responded to and written to the output file. The client should shut down the appropriate servers.

E. **Implementing the servers (Branches):** The aim here is to enforce two client-centric consistency models, the monotonic writes and read-your-writes. The Branches accept a message which carries information on the event (type of transaction to be made), the customer unique ID, and a write set from the Customer. It is very important to note that the customer only sends events to the right branch. for example in the monotonic writes example, the customer sends the deposit event to only branch 1 (not to both) and sends the withdrawal event to branch 2 only. The branch receiving message from the customer compare the message's write set with its present write set. If it is not the same, it will sleep for 0.1 seconds and then check again; this is implemented with a while loop. Each branch's write set is initially set to 0. This way the branch who should execute the request that must be executed first is able to do so. This first transaction executing branch, updates its balance and its write set and also propagates the amount for updating other branches balance and write set. Once the write set and balance for each branch are updated, the branch to execute the next transaction in the process order of transactions, will now be able to break out from the while loop and execute its transaction. This process is repeated until all events are handled by the branches in a monotonic write way. This implementation with write set was sufficient to also fulfill the read-your-write consistency and to generate the right output for read-your-write too. I discussed with the instructor during the live event and he confirmed that once the output is correct there is no problem. My implementation was carried out using four methods of the Branch class and another function as following:

a) The eventRequestExecute method handles how a message from the Customer is handled before propagation. The method accepts the request from Customer and generates what has to be added to the existing balance. The variable known as self.money carries this amount that will be added to the balance. If the request is a deposit, then self.money is assigned the money amount to be deposited. If the request is a withdrawal, self.money is assigned a negative value of the amount to be withdrawn. Else self.money is assigned 0 (for ease of implementation, query is regarded also as a deposit of $0). Because write sets will change in the lifetime of the branch, for each transaction, the branch assigns a message's write set to a variable known as self.workingWrite set. self.workingWrite set has to be appended to the message returned back to the customer so that the customer can know that this reply is for the message with the same write set.

b) The ClockUpdate method handles the events in a Branch that receives propagated message. That is, ClockUpdate method's executions result in a propagation of writes and write sets across all branches. In the branch to branch communication, The propagated message is the amount carried by variable self.money from the propagating branch. This variable carries the amount which could be positive (deposit) or negative (withdrawal). Self.money is added to the balance of the receiving branch. Also the write set received is increased by 1 and becomes the new write set for the relevant branch.

c) The propagateRequest method handles the propagation of self.money (amount to be deposited or withdrawn) to other branches. The propagating Branch send a message, awaits for the other Branches to receive the messages and acknowledge receipt. Then it updates its own branch's balance and write set.

d) The MsgDelivery method puts it all together. It ensures that request from the client is appropriately executed. This method also enforces client centric consistencies using the write set, while loop and time.sleep. If the write set and the incoming message write set from the customer is not the same, the branch waits for 0.1 seconds and check again. Once the branch write set becomes the same as the message write set, the while loop is broken and the execution will now proceed. In essence, the first write set to be executed and propagated is 0. The next event in the write order of the same process will wait until the increment is made to 1 and then a message with write set 1 can be executed. This continues until all executions are completed.

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 requests meant for the branch; so that the right responses are sent back to the client(s). The specific port is calculated to be specific for a specific branch 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. This spawns multiple servers. In the given example cases this resulted in two servers listening on ports 50048 and 50049. On receiving a request from the appropriate client, the server processes the request and respond as expected.

I did not implement methods to specifically work on either deposit, withdrawal or queries. Instead, I ensured that based on the content of the message a Branch receive from a Customer or another Branch, the Branch chooses how to implement the specific subevent.

Results

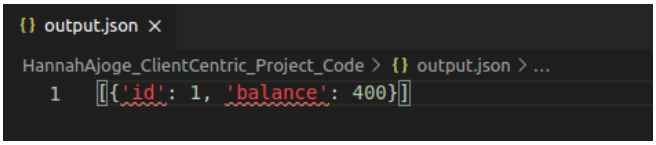
On executing the implementation process explained in the preceding section, an output file (output.json) with the expected output is generated. I wish to succinctly explain the overall results and the justification as follows:

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 using a write set that has to be the same for the incoming message and the branch's write set; I was able to enforce both the monotonic writes and the read-your-writes. It was fascinating to see that an implementation could fulfill both consistency models. This system ensured that different branches are able to handle requests from the same process (from a single customer) and achieve both client-centric model of consistencies.

b) My implementation's algorithm was more geared generally to maintaining the needed client-centric consistencies and not to dwell on different methods for different transaction types.

c) 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 connections with bank branches once a client has successfully completed their transactions, the client stub terminates the server after writing the expected result to the output.json file. Figure 3 shows the output of the two examples.

(a)



(b)

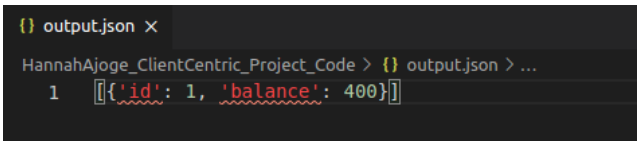


Figure 3: (a) Output file (output.json) of the monotonic writes example. (b) Output file (output.json) of the read-your-writes example

References

1. Feldmann, Michael, Christina Kolb, and Christian Scheideler. “Self-Stabilizing Overlays for High-Dimensional Monotonic Searchability.” Stabilization, Safety, and Security of Distributed Systems. Cham: Springer International Publishing, 2018. 16–31. Web.

2. Girault, Alain et al. “Monotonic Prefix Consistency in Distributed Systems.” Formal Techniques for Distributed Objects, Components, and Systems. Cham: Springer International Publishing, 2018. 41–57. Web.

3. Tanenbaum, A. and M. V. Steen. “Distributed systems: Principles and Paradigms.” (2001).

Appendix

Figure A1: Input file (input.json) of the monotonic writes example.

Figure A2: Input file (input.json) of the read-your-writes example