Project 3: Report

Introduction

In this project, I mainly implemented the Primary/Backup server and the ViewServer which was supplemented by the client, KVStore, and AMOApplication from the previous projects. The goal of this project was to keep a primary and a backup for the KVStore to add fault tolerance to improve on the SimpleServer/SimpleClient design. The following are the new notable code and features added in this project:

- ViewServer is the server that communicates with all the clients and servers. It chooses the primary and backup servers when necessary (e.g. when it determines that a primary or backup is dead) and it also replies to client and server requests for the most updated View. The ViewSever is assumed to never fail for this project.
- PBServer is the server class that can become a primary, backup, or idle server depending on the View that it thinks it is in. As the primary, it is in charge of handling client requests and ensuring that the backup is also synchronized with the primary. As the backup, it is responsible for handling the commands sent from the primary to operate on the KVStore and keep consistent with the primary in case it needs to be the primary. The idle servers are backup for the backup servers in case the ViewServer thinks the backup is dead.
- PBClient is similar to the SimpleClient implementation, however I added the send GetView() function at initialization and in other parts of the code such as when it gets an InvalidView from the server or on the client timer. It also has a function to handle the ViewReply from the ViewServer
- Two new timers have been added: BackupRequestTimer and SyncRequestTimer. These are timers set between the primary and backup to ensure that the KVStores in both servers are in sync.

Flow of Control & Code Design

At initialization, the servers ping the ViewServer and the ViewServer will determine whether to assign the server to be the primary, backup, or idle depending on the state of the ViewServer. If there isn't a primary, the server will be assigned as the primary. If there is a primary but no backup, the server will become the backup server. And if the ViewServer state assigned the primary and backup servers and they are both alive, the server will remain as idle. The restriction in the ViewServer is that the View cannot change unless the primary acknowledges that it is the primary. In the code, I added a primaryLastSeenViewNum which keeps track of the last seen view number of the primary to determine whether the primary server acknowledged that it is the primary server. The ViewServer also triggers a ping check every PING_CHECK_MILLIS to keep track of the servers that are alive and removes any from the serverToPingCheckTimerMap if any of them did not ping the ViewServer within the specific time frame. During this process, if the ViewServer thinks that a server is dead and it is either the primary or backup server, it will try to change the view (iff primary server acknowledges that it is primary). It will also check if a server is dead on the ping request. Additionally, the ViewServer acts as the "global View" and so clients and servers will ping the ViewServer for the most recent view (handled in handlePing and handleGetView).

The PBClient is extremely similar to the SimpleClient behavior so I will mostly discuss the notable new changes. The PBClient will get the view from the ViewServer at initialization and send commands to the primary server that was assigned in the view from the ViewServer. If it gets a InvalidView reply from the server, then it means that the server was not the primary server (at least that server thinks so) and so the client will get the view again from the ViewServer. Once the command timer runs out, it sends it again, hopefully to the correct primary server this time. During this step, it also requests for the view from ViewServer since it may be the case that the primary server is dead. The timer ensures that the command keeps sending until it gets the desired reply from the server.

The PBServer behaves differently based on whether the server identifies as primary, backup or idle. The similarity is that they all ping the ViewServer every PING_MILLIS to notify the ViewServer that is it alive. If the backup or idle servers receive a request from the client, they will simply reply with an InvalidView. When the primary receives a request from the client, it checks if there is a backup in its view and if there isn't one, the server simply executes the AMOCommand ensuring the at-most-once rule and sends back the result to the client. If there exists a backup, it will add the command to the end of the backupOrderInPrimaryQueue to ensure that the order of execution is maintained. Then if the backup has done the synchronization of KVStore (will discuss this in the next paragraph), the server sends the command at the front of the queue to the backup and sets a BackupRequestTimer in case the message gets dropped, backup is dead, or actual backup is a different server. Once the request is received by the backup server, the backup server handles the request by executing the command ensuring AMO and sends back a BackupReply to the primary server (here I did not put a timer for the reply since if it gets dropped, the BackupRequestTimer will send again and that is fine since we use AMO). Once the primary server receives the reply with BackupSuccess result, it now does the execution of the command on its own KVStore and sends back the result to the client. The primary server at this point checks to see if there are any other commands in the queue and if there is, it request the backup server to execute those commands one at a time.

Now I will discuss how the promotion is maintained for the primary and backup server in PBServer. Whenever the primary server gets the ViewReply from the ViewServer, it checks to see if the backup server has changed. If it did change, that means that the backup server must synchronize with the primary server to keep a consistent KVStore. The primary server requests a SyncRequest containing the whole KVStore application to the backup and sets a SyncRequestTimer for it. If the backup received the SyncRequest, it sets a flag syncComplete to true indicating that it has the consistent KVStore and sends back a SyncReply with a SyncSuccess message. The primary server receives the reply and also sets the syncComplete to indicate that it also agrees that the synchronization is finished. Then it sends any commands in the queue for the backup to execute.

Design Decisions

In terms of the design decisions, I chose to use a queue to maintain ordering of the incoming commands from the client. At first, I considered not having any data structure to maintain the ordering and simply executing the commands and sending them to the backup server. However, this implementation does not necessarily maintain ordering because messages may be lost on the way to the backup and resending those messages will potentially make the executions unordered in the backup server. Because of this, I revise my design to add a queue to the primary server so that it can keep an ordering of the commands received and execute them one at a time.

In my original design, I planned on making the clients and servers ask the ViewServer for the most updated view every time there was some sort of error. However, by looking at the test cases and the structure of the project itself, I realized that this was not a good design since it relies too heavily on the ViewServer and taking advantage of the fact that it does not fail. In a real world distributed system, we cannot guarantee that some ViewServer will never fail. So, I chose to only get the view from the ViewServer in certain scenarios such as when the PingTimer is triggered or when the server's reply explicitly states an Invalid View in the client. When view is not consistent, I simply let the Ping Timer get the send a Ping to the ViewServer.

For the timer values, I originally thought 100 milliseconds on both the BackupRequestTimer and SyncRequestTimer is sufficient but I later chose to decrease the timer for SyncRequestTimer since it is more crucial compared to the BackupRequestTimer. My implementation relies on the SyncRequest to make sure that the KVStore in primary is copied to the backup when backup gets promoted. If the primary server dies and it was sending in the middle of sending a SyncRequest and the SyncRequest never reached the backup, essentially all the data is lost. This is why I revised the design to decrease the SyncRequestTimer so that, although not perfect, there's a higher chance that the backup receives the KVStore application.

Upon reflection, if I were to develop a project like this next time, I would first go through all the functions carefully to fully understand what is required. This time, I only designed a rough overview of how the project will be implemented and did not look at the code until I finished my rough design. This led to me changing some parts of my initial design due to the code restrictions. For example, I was planning on disabling a timer when some condition was met, however, that did not seem to be an option in code.

Missing Components

The search tests 19 and 20 from part 2 did not pass. These are the following errors from those two tests:

```
State violates "Clients got expected results"
Error info: client2 got KVStore.AppendResult(value=y), expected
KVStore.AppendResult(value=xy)
State violates "Sequence of appends to the same key is linearizable"
Error info: KVStore.AppendResult(value=y) is inconsistent with
KVStore.AppendResult(value=w)
```

From these error messages, I believe I am missing certain cases for linearizability. It seems that the values in the KVStore are not consistent with the expected result. Perhaps I am missing some case when the primary syncs with the backup, the primary executes messages on its own and results in an unsynchronized KVStore in primary and backup.

References

I referenced the course repository for the instructions:

https://github.students.cs.ubc.ca/CPSC416-2022W-T2/project3-primarybackup