T28

https://github.com/tecnico-distsys/T28-ForkExec

87653	87667	87681
Filipe Miguel Cerqueiro Henriques	João André de França Ferreira Galinho	Maria Catarina Macedo de Oliveira Duarte

Fault Tolerance model used – *Quorum Consensus*

The fault tolerance model used is called ***Quorum Consensus***. The idea behind it is that, by having **multiple replicas of the same server**, it becomes possible to **support the failure of a minority** of them, only **requiring that a majority of the servers are running** at any given time to **guarantee complete consistency and coherency**.

It achieves this by associating to every value stored in the server a *sequence number* and a *client id number*. We will call those numbers *tag*. Using it, a very specific asynchronous procedure to *read* and *write* the values from the server is used.

For the ***read* operation**, the front-end **calls the asynchronous read function** for the requested value on all servers. (To note that, since the calls are asynchronous, all of them are done before even starting to wait for any responses.) Each server, if working and available, responds with the value requested as well as the corresponding *tag* it has it currently associated with. **The front-end then waits for *q* responses**, where *q* is one plus the whole division of the number of servers by two, which gives the minimum number of servers to constitute a majority. From those responses, **it chooses the one with the higher *tag*** and returns its value. (*Tags* are compared by sequence number, and then by client id in case the sequence numbers are equal.)

For the ***write* operation**, the front-end **first does a *read* operation** (as described before) requesting the value to be written. Along with the value itself, the read operation also returns its current *tag*. The front-end then **calls the asynchronous write function** for the value to be written with then new value and a *tag* that consists of the ***sequence number* from the current *tag* returned from the read operation plus one** and its own *client id*. Finally, the front-end **waits for an acknowledgement from *q* servers** confirming they have successfully performed the operation, and then returns.

This makes it so that, at any given moment, **at least a majority of the servers have the most up to date version of any given data**. Because the read operation is always performed to at least a majority of the servers, it is guaranteed that at least one of the responses is the most up to date one, being identified by having the highest *tag* of all responses received.

However, this model does have one limitation. It **does not support concurrent operations** (be it *read* or *write*) **on the same value**. Therefore, the model requires the guarantee that no value is being accessed by more than one thread at any given time. This limitation **can be fixed by performing a *write back* operation right after any *read* operation**, at the cost of read response times and server load.

Implementation on ForkExec project

In this project the fault tolerance model is only applied to the **Points server**. This means that the Points module was the only one modified to support **asynchronous calls and *tag* storage** as well as to have **read and write operations to its values instead of specific logical operations**. Those operations were **delegated to the front-end**, which consists of the class **"*QCPointsHandler*"** on the Hub web server's domain, which now implements all the old operations using the *Quorum Consensus* model to read and write from the Points servers. The Hub domain class **was only changed to use the new class instead of the *PointsClient*** to call all the old operations.

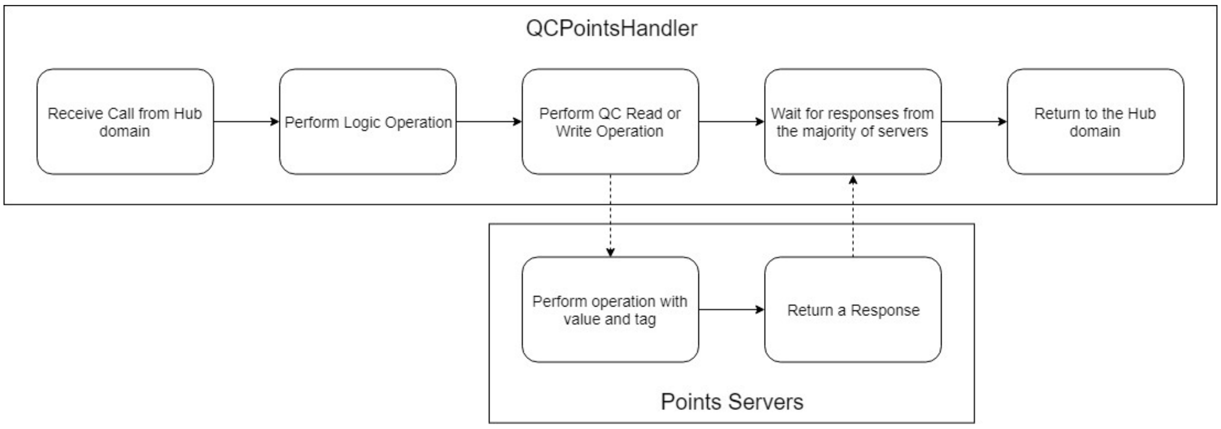
QCPointsHandler
Receive Call from Hub domain
Perform Logic Operation
Perform QC Read or Write Operation
Wait for responses from the majority of servers
Return to the Hub domain
Perform operation with value and tag
Return a Response
Points Servers

Image 1: Diagram of interaction between QCPointsHandler and the multiple Points Servers

To be noted that, although the diagram states that the *read* and *write* operations come after the logical operation, they can actually be both shuffled together. For example, the *spendPoints* and *addPoints* operations both perform a read operation to get the current value followed by the logical operation that determines the new value to be written, and finally performs the write operation. **This could be classified as just a write operation with a logical operation in the middle** because the final write operation does not need to read the value again.

The method used for the asynchronous calls is the **Polling method**. This method works by **actively and continuously checking a condition** that indicates if a response was already received.

In case **less than the majority of servers are traceable and able to be called**, the model **continuously searches for servers and calls all of them until a majority of them were able to be called**, and only then proceeds to wait for the responses.

Specific optimizations and simplifications

The specific implementation of the model in this project takes advantage of the following particularities of this use case:

- There is **only one** *HubClient* that performs operations on the Points servers;
- The *activateUser* operation **is done implicitly on other operations if it fails**, which means there is less need for a guarantee of its successful execution.

Because there is **only one** *HubClient*, **there is no need for the model to carry around a *client id*** in its *tag*, **because it would always be the same**. Consequently, in this implementation, **the *tag* only consists of the *sequence number***.

The behavior of all operations when dealing with a user that has not been activated changed from throwing an exception to implicitly creating it with the current starting amount. If the operation is to set the balance of the inactivated user, it is created with the specified amount instead. This means that the only problem that might arise from an unsuccessful activation, under very specific circumstances, is that the **expected exception indicating the email already exists might not be thrown in the consecutive activations of the same user**. It was therefore decided that, in this specific use case, **there was little need for the typical asynchronous calls** followed by a wait for the majority of the servers to respond. Consequently, a standard synchronous approach without guarantee of success (not waiting for responses) was preferred in order to **prioritize responsiveness**.

The fact that **the specific values accessed are known beforehand** was also taken advantage of for a more optimized approach to dealing with concurrent calls. When checking or changing the balance of a given user, the specific value that is going to be accessed on the server is already known. This means that there can be a **specific *lock* for each user that guarantees that there are no concurrent calls referring to the same user, but allows for concurrent calls of the same operation as long as it is referring to different users**. This **optimizes performance** by allowing a greater degree of parallelization.