Assignment 2

Big Data Systems

Johan Pedersen - `rlh177`, Gilli Fjallstein - `gwn787`

January 21, 2022

Question 1: `TransactionalActor` must enable reentrancy. Please explain what reentrancy is? And why does `TransactionalActor` need to enable it?

While grains always execute in a singlethreaded model, enabling reentrancy allows for interleaving requests. This means that while one request is waiting, another request can start to execute. Then when this second request is waiting the first one can continue or finish up. In the case of `TransactionalActor` we use it to allow for a grain to wait for other actions on the same grain.

For example if we receive a batch for which the previous batch has not been received, we can wait to receive and execute this previous batch. In this case, if the transactional actor would not be able to process any other calls while it was waiting for other batches to complete, it would lead lead to a deadlock when we must process batches in order.

Explain how you implement deterministic concurrency control? How do you make each actor build and maintain its local schedule?

With deterministic concurrency control we have a lightweight alternative to the more expensive 2PL method. We tried to implement it in `TransactionalActor` using two separate schedules. Firstly we would keep track of batches with their ID and prevID, then if we dealt with a batch for which the previous batch had not been computed, we would wait until that finished first. Then we would also maintain a schedule for transactions within each batch. We would make sure that the transactions would wait until the previous had finished before executing. To assure that actor only committed the batch transaction once when it was all finished, we would do it when the last transaction in the batch had finished with a simple if-statement. Thus we would enforce and guarantee a deterministic concurrency control by exploiting the reentrancy mechanic of actors in Orleans.

The method we would use to 'wait' for other transactions or batches is by simple dictionaries that map from a batch or transaction ID to an Orleans task. Then we made sure to set the tasks as complete when they had been executed.

Currently the data structure used to keep track of which batches an actor has processed does not get 'cleaned up' properly. As such it grows endlessly during runtime, which

means this is only useable for a limitided amount of data

Furthermore our current implementation does not support speculative execution, which should be possible to add with a simple if statement that implements this. But our previous attempts to make this stable have failed.

Lastly the program has a race condition where it will sometimes deadlock. This happens when two threads are waiting for their previous task to finish at `await localSchedule[context.bid].Item1[lasttid].Item1.Task;` Here one thread may wait for a task, that in turn waits for that same task to finish. Thus the program does not succeed at processing every run.

Explain if no concurrency control protocol is applied, what will go wrong? Particularly, will each `AccountActor`'s balance stay non-negative? Give an example.

Without a concurrency control the transactions will not follow a serial schedule which can introduce several problem. One is we would allow the actors to possibly read uncommitted data (dirty read). However since we do not have transactions autonomously aborting this is not a big issue for our case.

What instead causes trouble is executing the transactions concurrently with interleaving operations. Without a serial schedule for the executing we will see WW conflicts where two transactions both read the balance of an account actor and sees that subtracting the amount needed for the transaction will not make the balance negative. As this happens concurrently they are both allowed to execute their transaction, and since we don't over-write the balance but instead subtract, we can get in the situation where the account actor will have a negative account balance. This would not happen with a serial schedule as the second transaction would read the committed balance from the first transaction and see that it would not be allowed to execute and follow through with depositing the amount of 0.