Big Data Systems Assignment 2

This assignment is due on Dec 20, 23:59. While individual hand-ins will be accepted, we strongly recommend that this assignment be solved in groups of maximum two students.

A well-formed solution to this assignment should include a PDF file with answers to all questions posed in the programming part of the assignment. In addition, you must submit your code along with your written solution. Evaluation of the assignment will take both into consideration. Note that all homework assignments have to be submitted via Absalon in electronic format. It is your responsibility to make sure in time that the upload of your files succeeds. While it is allowed to submit scanned PDF files of your homework assignments, we strongly suggest composing the assignment using a text editor or LaTeX and creating a PDF file for submission. Email or paper submissions will not be accepted.

Learning Goals

1. Understand the building blocks of deterministic databases, especially deterministic concurrency control.
2. Understand the system design of Snapper [1], including the batch generation algorithm, actor local schedule management and batch commit protocol.
3. Get more training with actor abstraction, asynchronous programming and concurrent programming.
4. Implement deterministic concurrency control for multi-actor transactions.

[1] Snapper is a transaction library for actor systems. The system design of Snapper is introduced in one of the lectures. And the slides can be found here: *Absalon → Files → Slides → 7. Deterministic Database.pdf*.

Programming Task

In this assignment, we provide you with the **simplified** implementation of Snapper. In the following part of this document, whenever "Snapper" is mentioned, it refers to the simplified version.

【Code organization】The key components of Snapper – `Coordinator` and `TransactionalActor` – are wrapped in the `Concurrency` folder. User-defined actors such as `AccountActor` must extend `TransactionalActor` and invoke actor calls via the `Execute` API to get transactional guarantees provided by Snapper.

【Server and client】When the server (`Silo`) is started, `TransactionalActor` and `AccountActor` are automatically loaded into the grain factory. You can start the server by simply building and running the project `Silo`. The client code is wrapped in the `Client` folder. Currently, `STEP 3 – 7` can not work, because part of the `TransactionalActor` has not been implemented yet. When you finish your programming task, you should be able to run `STEP 3 – 7`.

【Your task】You should implement the deterministic concurrency control for multi-actor transactions. More specifically, you should **enforce each actor to execute transactional calls in the predetermined order.** (Notice: the places where you might need to add some code are annotated with comment "`ATTENTION`", remember to check all of them. Besides, you are allowed to add code to anywhere else that has no “`ATTENTION`”, but you should reason about it in the report.)

【Assumptions】

(1) Assume the whole system has only one coordinator.
(2) Assume no failure will happen and the transaction logic will never abort a transaction, thus there is no case a transaction will abort. The whole assignment is about generating global order, doing deterministic concurrency control and applying a commit protocol to enforce atomicity.
(3) Assume each actor does speculative execution.
(4) Assume no transaction will access the same actor more than once.

Report

1. `TransactionalActor` must enable reentrancy. Please explain what reentrancy is? And why does `TransactionalActor` need to enable it? (Hint: https://dotnet.github.io/orleans/docs/grains/reentrancy.html#reentrancy)

2. Explain how you implement deterministic concurrency control? How do you make each actor build and maintain its local schedule? (Hint: are transactional calls that belong to different transactions arrive an actor in the order of `bid` and `tid`? If not, how do you make sure they are executed in order?)

3. Explain if no concurrency control protocol is applied, what will go wrong? Particularly, will each `AccountActor`’s balance stay non-negative? Give an example. (Hint: how does an `AccountActor` do a `Transfer` transaction? If there is no concurrency control, how transactions will interleave with each other?)

Transaction Workflow

A client submits a transaction by calling the `SubmitTransaction` API on the first actor that should be accessed by the transaction (`(1)`). This actor then issues the `NewTransaction` request to the coordinator (`(2)`). In return, the actor gets a `TransactionContext` instance, which includes the `bid` and `tid` assigned by the coordinator (`(3)`).

The coordinator periodically generates a batch which is triggered by the `timer`. All the transactions received so far are wrapped together into one batch. The coordinator then generates a sub-batch for each actor that will be accessed by the transactions within this batch. And sub-batches are sent to corresponding actors asynchronously (`(4)`).

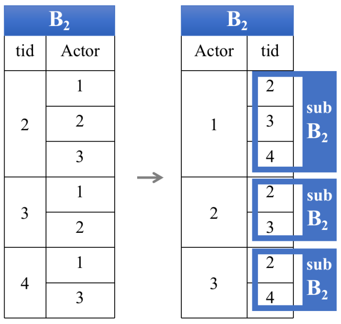

Figure 1: Sub-batch Generation

When an actor finishes executing a sub-batch, it will send a `BatchComplete` message to the coordinator (`(6)`). When the coordinator has received ACKs from all participating actors of a batch, it will try to commit the batch. When the batch is committed by the coordinator, a `BatchCommit` message is sent to every related actor via `ReceiveBatch` API (`(7)`).

Notice 1: Recall that message delivery time and order are non-deterministic in Orleans, i.e., messages can arrive at a destination actor in a different order than the sending order. Thus each actor may not receive sub-batches in the order of `bid`.

Notice 2: An actor cannot start executing the transactional call until: (1) The corresponding sub-batch / schedule has arrived at the actor. (2) According to the predetermined order, it is its turn to execute.

Edge	Message Type
(1)	SubmitTransaction
(2)	NewTransaction
(3)	TransactionContext
(4)	SubBatch
(5)	TxnData
(6)	BatchComplete
(7)	BatchCommit
(8)	TransactionResult

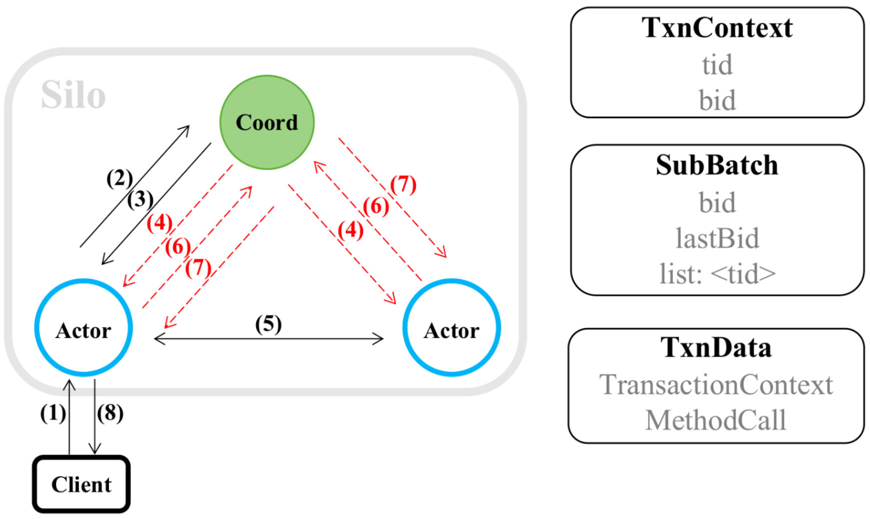

Figure 2: Transaction Workflow