Assignment 1

Big Data Systems

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

January 21, 2022

1. Questions and answers

1. Why does the Online Event Processing (OLEP) approach fit the application scenario presented in this assignment? In particular, what problems does OLEP address that makes it a fit for this application scenario?

Since every log can support many independent subscribers, it is easy to create new derived views or services based on an event log. The example the OLEP paper has is that it if an accounts spending limit has been reached an independent subscriber to the balance event log could trigger a push notification to the customer's mobile phone.

Recoverability is much easier than a traditional transaction system. Subscribers can be ignore incorrect event and start a process to recompute all views deriving from it. Furthermore events can be replayed in order to regain the state at the time of failure, this is has incredible value in a financial environment. This also plays into the benefit that the changes to state are much more readable than the traditional insert, update, delete, etc. transactions leading to easier debugging.

In a traditional distributed transaction scheme if any participating node is unavailable the whole transaction must abort, essentially allowing cascading failures. In contrast, in the OLEP approach each subscriber to a log is independent from each other in such a way that if one of them fails it does not effect the operation of the publisher or other subscribers, ensuring better containment of failure.

2. Do you believe the actor model is a natural fit for OLEP-based applications? Justify your answer.

Yes. The actor based model is all about modularity and fault isolation. The OLEP approach in contrast to transactions boosts these characteristics in a way that lets the actor models benefit from the modularity as it was designed to.

For example the OLEP approach assumes that each subscriber maintain it's own state which is exactly what grains are designed to do. Furthermore since the event log is always available any grain can recover in case of error incredibly efficiently to its previous or intended state without directly bottlenecking other actors.

3. What are the relationships and differences between BASE and OLEP?

There is a functional partitioning in BASE that is somewhat similar to OLEP. Each log in the OLEP approach is, hopefully, only subscribed to and updated by processes that need to. This ensures a more distributed load on both types of systems. But the BASE it is unclear how we will scale if the message queue suddenly becomes a bottleneck. On the other hand there will be no such cost in the OLEP approach because the event log will be able to handle new subscribers from a completely different server with only the bandwidth being the cost, and scaling them linearly.

Both models use a sort of persistent message queue but the use of them differ drastically. OLEP is much more loosely couples, not relying on a single handler to react on the messages. It also has the benefit of the messages acting as very descriptive logs at the same time. The BASE message queue is much more difficult to scale because it does not prescribe a thread model while OLEP is perfectly suited for a multi-threaded context.

4. Explain briefly how your application would differ if designed following the BASE model instead of the OLEP approach.

Using the base model we would have the freedom to choose the message broker that we wanted, but not use the log abstraction. The big difference here is that the log abstraction gives us a much needed FIFO guarantee. This guarantee is very important for the use case of bank transactions, and comes for free using a log. However it does not come with just a message broker, and thus a great deal of effort is needed to rewrite the application in a manner that allows us to relax this ordering constraint.

5. Suppose that instead of only having transfers from one account to another, the application requirements change to reflect the need to support account transfers from multiple sources. That is, a transaction may now involve multiple source accounts transferring money to a single account. Does your current OLEP application design safeguard atomicity in this case? If yes, justify why this is the case. If not, explain the necessary measures to adapt the design to the new application scenario.

Our OLEP application design does safeguard atomicity if we allow multiple source accounts in the transfers. In the design we know only the `withdraw` operation will throw exceptions, and thus we only need to safeguard this. Having multiple source accounts would involve doing multiple `withdraw` operations of which the first could succeed and the second could fail. Thus we implement a pseudo transaction systems where we take a list of withdrawals as input and check that the all the withdrawals need are allowed, before we "commit" and make the actual changes. We do this by adding up the total sum withdrawn from each `fromAccount` in a hashmap, and check that withdrawing the amount does not violate the non-negativity constraint. Doing it this way ensures atomicity for the account transfer no matter the amount of accounts involved in the transaction, but does incur a cost of looping over each transaction twice.

2. Implementation

The implementation relies on assigning each operation its own stream and having 2 separate workers subscribe to each stream. In the stream we send the ids of the account grains in which we wish to perform the operation, and the amount we wish to transfer. Inside the consumer we use the `GrainFactory` to get the correct account grain to perform the operation. All business logic, like account balance not going below zero is implemented inside of the `AccountGrain` itself.

We use the `SMSProvider` which guarantees FIFO behavior and as we choose to await each operation we ensure that one operation succeeds before continuing with the next.If any account involved in the withdrawal violates the non-negativity constrain, an exception will be thrown. This exception will be caught in the program loop, and thus secures that the `Deposit` part will not be executed afterwards.

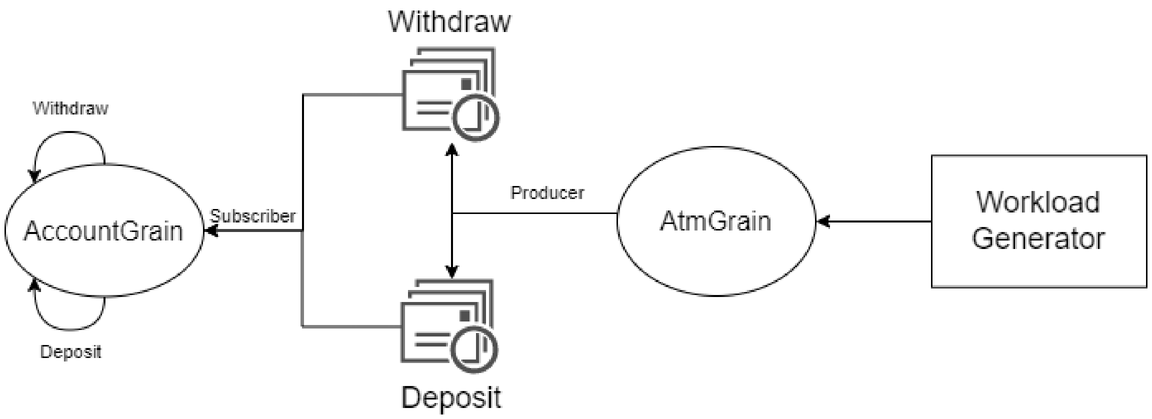

Figure 1: Simplified flow of our application