Tianze Shan

Assignment3.1

Optimistic Locking is a strategy where you read a record, take note of a version number (other methods to do this involve dates, timestamps or checksums/hashes) and check that the version hasn't changed before you write the record back. When you write the record back you filter the update on the version to make sure it's atomic. (i.e. hasn't been updated between when you check the version and write the record to the disk) and update the version in one hit.

Pessimistic Locking is when you lock the record for your exclusive use until you have finished with it. It has much better integrity than optimistic locking but requires you to be careful with your application design to avoid Deadlocks. To use pessimistic locking you need either a direct connection to the database (as would typically be the case in a two tier client server application) or an externally available transaction ID that can be used independently of the connection.

Assignment3.2

Conservative 2PL：
Conservative 2PL's transactions obtain all the locks they need before the transactions begin. This is to ensure that a transaction that already holds some locks will not block waiting for other locks.

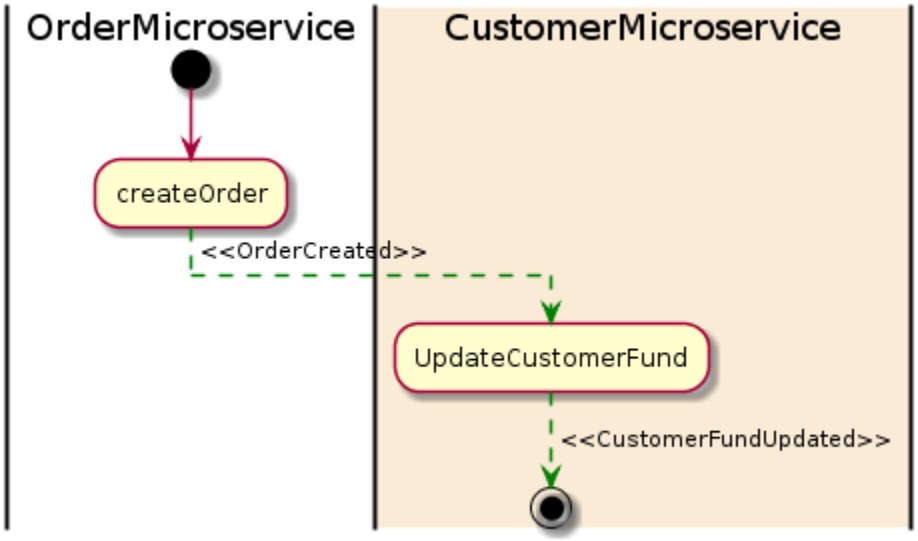
Wait-Die Scheme：
In this scheme, If a transaction requests a resource that is locked by another transaction, then the DBMS simply checks the timestamp of both transactions and allows the older transaction to wait until the resource is available for execution.

Wound Wait Scheme
In this scheme, if an older transaction requests for a resource held by a younger transaction, then an older transaction forces a younger transaction to kill the transaction and release the resource. The younger transaction is restarted with a minute delay but with the same timestamp. If the younger transaction is requesting a resource that is held by an older one, then the younger transaction is asked to wait till the older one releases it.

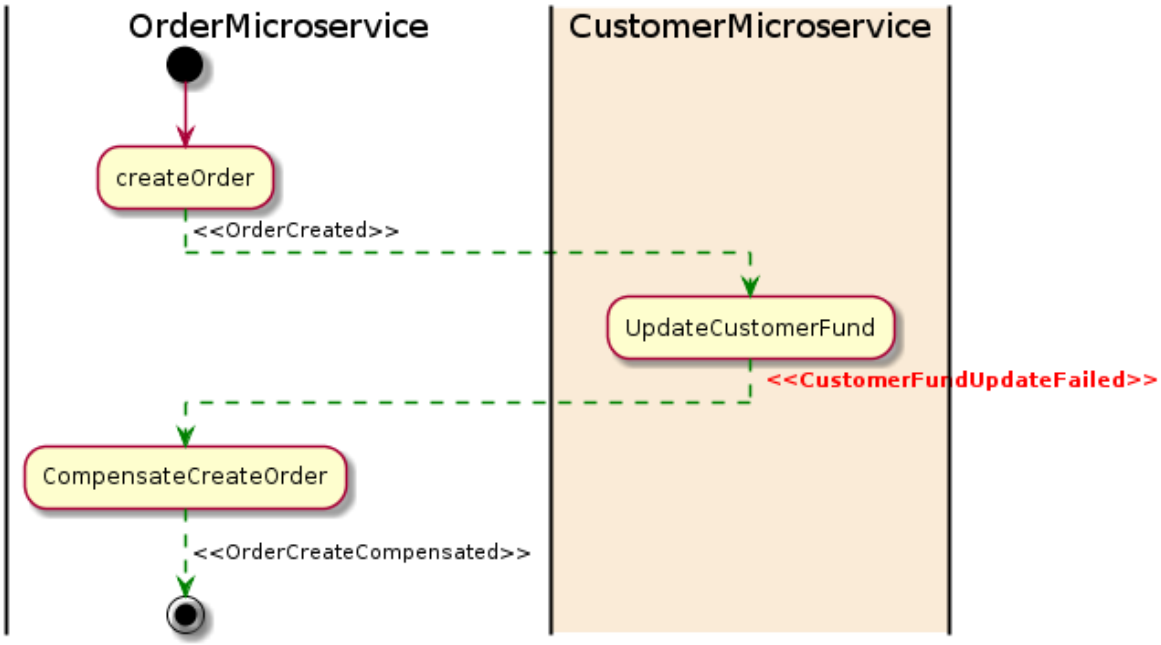
Assignment3.3

The Saga pattern is another widely used pattern for distributed transactions. It is different from 2pc, which is synchronous. The Saga pattern is asynchronous and reactive. In a Saga pattern, the distributed transaction is fulfilled by asynchronous local transactions on all related microservices. The microservices communicate with each other through an event bus.

In the example above, the OrderMicroservice receives a request to place an order. It first starts a local transaction to create an order and then emits an OrderCreated event. The CustomerMicroservice listens for this event and updates a customer fund once the event is received. If a deduction is successfully made from a fund, a CustomerFundUpdated event will then be emitted, which in this example means the end of the transaction.



If any microservice fails to complete its local transaction, the other microservices will run compensation transactions to rollback the changes. Here is a diagram of the Saga pattern for a compensation transaction:



In the above example, the UpdateCustomerFund failed for some reason and it then emitted a CustomerFundUpdateFailed event. The OrderMicroservice listens for the event and start its compensation transaction to revert the order that was created.

	Advantage	Disadvantage
Saga	2pc is a very strong consistency protocol. First, the prepare and commit phases guarantee that the transaction is atomic. The transaction will end with either all microservices returning successfully or all microservices have nothing changed. Secondly, 2pc allows read-write isolation. This means the changes on a field are not visible until the coordinator commits the changes.	While 2pc has solved the problem, it is not really recommended for many microservice-based systems because 2pc is synchronous (blocking). The protocol will need to lock the object that will be changed before the transaction completes. In the example above, if a customer places an order, the "fund" field will be locked for the customer. This prevents the customer from applying new orders. This makes sense because if a "prepared" object changed after it claims it is "prepared," then the commit phase could possibly not work.
2PC	One big advantage of the Saga pattern is its support for long-lived transactions. Because each microservice focuses only on its own local atomic transaction, other microservices are not blocked if a microservice is running for a long time. This also allows transactions to continue waiting for user input. Also, because all local transactions are happening in parallel, there is no lock on any object.	The Saga pattern is difficult to debug, especially when many microservices are involved. Also, the event messages could become difficult to maintain if the system gets complex. Another disadvantage of the Saga pattern is it does not have read isolation. For example, the customer could see the order being created, but in the next second, the order is removed due to a compensation transaction.