Exercise 4

Below are some notes regarding the implementation of the project.

Question 1: BankServer Implementation

getBalance() method has been implemented, following the suggested temporary queue approach.

One problem might occur if a remote bank (for some reason) puts a request in our local queue, and later figures out that the transaction has to be aborted. So the remote bank also puts an "abort" response in our local queue. Now, since SQS doesn't guarantee FiFo, the local bank might first process the abort response, but after that still apply the request (which was sent before the abort, i.e. the changes it makes should be rolled back). This way however, the request is still processed, and atomicity is violated.

This problem is at least somewhat mitigated by the 2-queue-solution in our case, since requests are always processed before responses. Also, some application logic would help: simply mark each request/response with the transaction ID. That way, when an abort of ID xyz comes in, simply ignore requests with that same ID.

Question 2: Publish / Subscribe

SNS has been used together with SQS. Queues act as subscribers to the topics. One topic per bank has been used, with attributes to differentiate between request and reply messages. The *BankServer* class has been extended to create *SNSBankServer*. Additionally, a unit test for testing remote operations has been created.

SNS allows greater scalability (1:n message sending without knowing recipients in advance). SQS ensures messages are delivered and kept safe even if the host server is down.

getBalance() method was implemented by saving the reply to the transaction table, and later querying that transaction.

Bonus exercise

As we did not do the bonus exercise, we've marked persistence unit tests with *@Ignore* jUnit flag.