UNIVERSITÄT BASEL

Prof. Dr. Heiko Schuldt
Filip-M. Brinkmann, MSc
Ilir Fetai, MSc

UNI
BASEL

cs341: Distributed Information Systems HS 2014

Exercise 3 Deadline: 26.11.2014

Advisor: Filip-M. Brinkmann (filip.brinkmann@unibas.ch)

Modalities of the exercise: The results of the exercises have to be uploaded to the courses website [1] before the deadline and presented to the advisor by the group. For this reason, each group has to schedule an appointment in agreement with the advisor (by e-mail). The appointment must take place in the time between delivery and the following Monday, 1.12.2014. During this meeting, the work and the understanding of the technical background of the individual group members is evaluated in order to grade the exercise. Each group member has to attend this meeting!
Please note that a certain amount of the overall points are reserved for questions that you will be asked during the presentation. The questions will be related to the topic of the exercise and your specific solution.

Modalities of work: The exercise can be done in groups of at most 2 persons.

1 Motivation and Application

In this exercise, you will implement a distributed banking application with different systems and tools. The example application is offering bank transfers from a source account to a destination account. This *transfer* procedure consists of two basic operations. *Withdraw* will reduce the balance of a given account and *deposit* will increase the balance. In our distributed information system setting, a banking transfer may of course involve different database servers.

Question 1: Distributed Transactions with Oracle 12c (12 Points)

The goal of the first exercise is to implement our banking application in the *'TP-Lite'* variation with Oracle 12c by using appropriate stored procedures and database links. The techniques you need for this exercise are SQL and PL/SQL (the programming language for stored procedures in Oracle). A good Oracle tutorial covering these topics is available at:

[1] `http://courses.cs.unibas.ch`

`http://www.mathcs.emory.edu/~cheung/Courses/377/Others/tutorial.pdf`

The system environment for this exercise will consist of Oracle 10g database servers installed at three servers of the DBIS group. You will receive appropriate user accounts for the exercises and it is recommended to use these servers for the exercise. From outside of the University network the servers are only reachable using a VPN connection.

Each database server represents a bank with a given bank identifier code (*BIC*). Ideally, the BIC number should correspond to the number identifier of the corresponding server (e.g., p6). Each bank is able to perform local *withdraw* and *deposit* procedures. In order to keep some level of security, a bank is only allowed to call a remote *deposit* procedures but **not** remote *withdraw* procedures. For a money transfer, a local *withdraw* and a local or remote *deposit* are executed from within a *transfer* procedure.

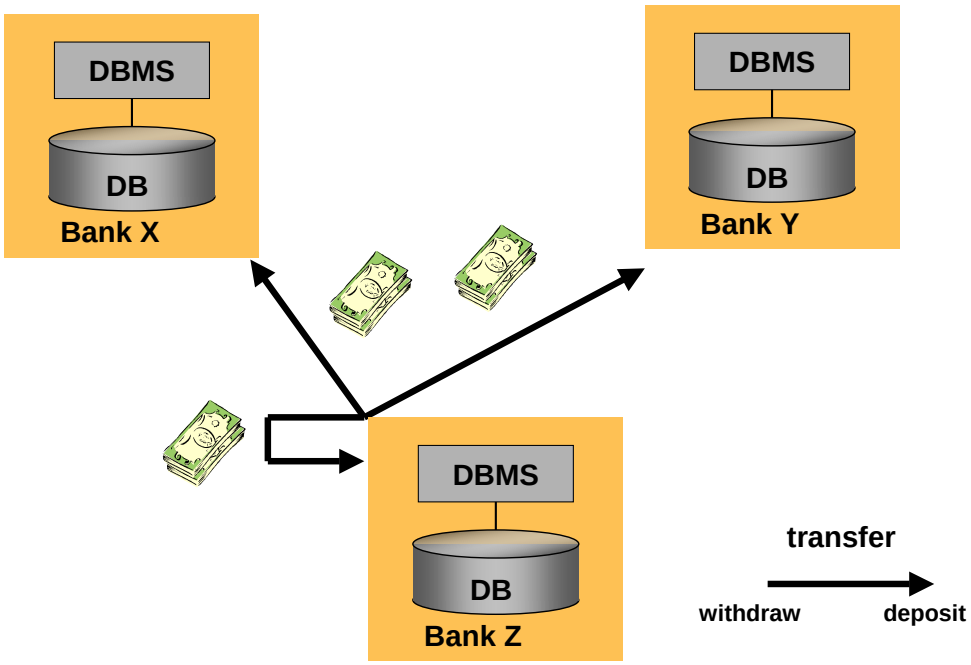

Figure 1: Money transfer possibilities of Bank Z

In order to execute the remote deposit procedure, a database link to the remote bank is necessary. If you have a database link established to the remote bank, you can access remote Oracle objects by adding an `@database_link_name` at the end of the object identifier. For example, `SELECT * FROM account@LinkToBank14` will deliver you all entries of the table `account` from the database which the link `@LinkToBank14` is connecting.

For creating the database link you need the following information:

- Database Link Name: *LinkToBank...* is an appropriate name you can choose
- Username: Name of a valid user login for the remote oracle server
- Password: Password of the remote login

- Remote Hostname: Possible values are: *p6.cs.unibas.ch* or *p7.cs.unibas.ch*
- Remote Host Port: *1521*
- Service Name (SID): *orclSERVERNAME*, where SERVERNAME is one of p6 or p7, depending to which server you connect; so e.g. *orclp6*.

The following SQL code creates a database link:

```
CREATE DATABASE LINK "LinkToBank"
   CONNECT TO "username" IDENTIFIED BY "password"
   USING '(DESCRIPTION =
     (ADDRESS_LIST =
       (ADDRESS =
         (PROTOCOL = TCP)
         (HOST = xxx.cs.unibas.ch)
         (PORT = 1521)))
     (CONNECT_DATA =(SID = xxxOracle_SID_here_xxx)))';
```

On the course website you will have the following files available:

- `database_schema.sql`: is the schema of the bank application and
- `banking_procs.sql`: as a starting point for your PL/SQL procedures.

a) Implement the procedures `withdraw` and `deposit` in the package `banking_procs`. The procedure `transfer` in the package `banking_procs` needs to be implemented such that it transfers money from a local bank account to a remote or local bank account. Keep in mind to check for proper exception handling in order to have exceptions if error situations occur (e.g., transferring money from a non existing account). Finally, every exception has to lead to a rollback of the complete distributed transaction. Do not forget to set the explicit *commit* and *rollback* statements at the proper locations!

b) Develop a protocol for a system test in order to check for failure situations that may arise (e.g., not enough money, wrong account numbers, etc.). For this you have to define some failure situations first (Pulling the plug of a PC **shall be not tested** during this exercise!). Deliver this testing protocol as an SQL-file. You should test your application at least with one remote bank! Check if atomicity of the distributed transaction is guaranteed.

Question 2: Distributed Transactions with Java and JDBC (12 Points)

The system environment for this exercise consists again of the three Oracle 12 servers and you will use the same database schema as in the previous exercise (`database_schema.sql`). For the purpose of this exercise, you are not using stored procedures and database links, but you implement a Java application which is connecting to two bank servers in order to offer a money transfer from one account to another. For this exercise, you do not need to implement a user interface for your Java application. Since the focus is on the distributed transaction, you can have the transfer details hardcoded in your Java code.

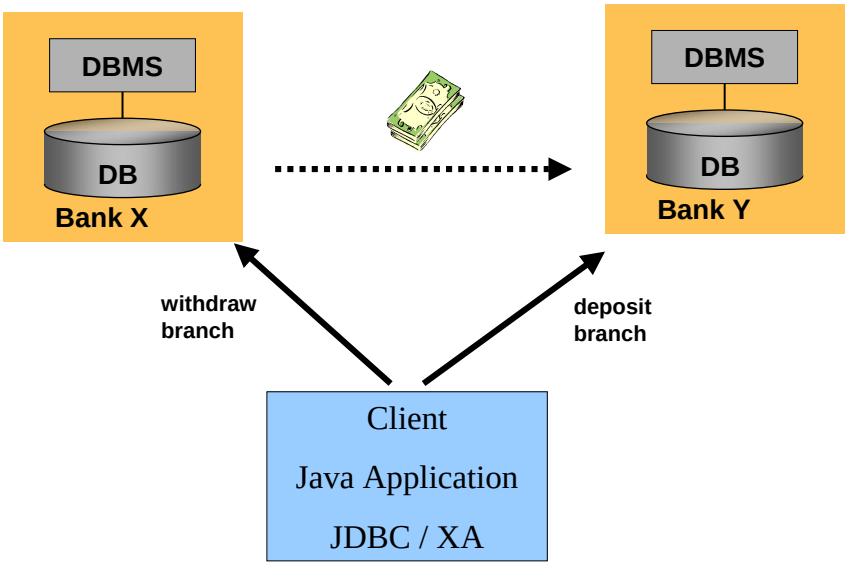

Figure 2: Money transfer of the Java application

The Java application will consist of two different transaction branches. One transaction branch is the *withdraw* operation at the first bank. The other transaction branch is the *deposit* operation at the second bank. In order to guarantee global atomicity, you will apply the 2PC protocol to your application. In short, you will *'prepare to commit'* each successfully executed branch. When *'prepare to commit'* of each branch was successful you are allowed to *commit* all branches otherwise you have to *rollback*. It is important that your application checks if a branch was executed successfully, (e.g., no withdrawal from a non-existing account)!

For this exercise we provide you with an Eclipse project which you can download from the course website. It contains a class file `XATest.java` which you can use as a start for your solution. In the *resources* folder, you find an SQL file which allows you for creating the database schema on both database servers.

a) Extend the given Java application in order to offer a money transfer between two banks. In particular, you have to use the XA extensions of JDBC to guarantee global atomicity of the distributed transactions. In this exercise, you must not use stored procedures and database links. Keep in mind to check for proper exception handling in order to have exceptions if error situations occur (e.g., transferring money from a non-existing account). Finally, every exception has to lead to a rollback of the complete distributed transaction. Do not forget to call the *commit* and *rollback* methods at the proper locations!

   Test your Java application on various failure situations (you can use the failure situations defined for the previous exercise) that may arise (e.g., not enough money, wrong account numbers, etc.). In particular, check if atomicity of the distributed transaction is guaranteed.

(10 Points)

b) In the lecture, you have seen two variants of 2PC: *Presumed Abort 2PC* and *Transfer of Coordination 2PC*. (It is not possible for you to implement these variants in the code, since XA does not support them.) Does it make sense to apply any of the variants to the current scenario? If it does, annotate your code at the proper locations, explaining what exactly would happen at this stage of the process. If the variant cannot be applied to the scenario, explain exactly why.

(2 Points)

Question 3: 2PC and 3PC failure handling (6 Points)

In what follows, we will analyse different failure scenarios of distributed transactions and their handling by both 2PC and 3PC.

a) Let us assume that all available agents are in the **ready** state. The coordinator crashes before sending the decision to any of the available agents, i.e. the available agents are **uncertain**. Does 2PC block in this case or can the available agents take a decision on their own? If a decision is possible, which one (commit or abort) should be taken so that atomicity is not violated? Explain your answer.

(3 Points)

b) What is the behaviour of 3PC in the mentioned failure scenario?

(3 Points)