Coursework CSC8016

Giacomo Bergami

26th of April, 2022

Use Case Scenario

We want to implement a bank system, whether the threads are either clients logging to the servers, or the bank server containing the information for each account. Each client interacts with the server through the `BankFacade` by accessing an `openTransaction` method, through which the each client can create multiple accesses to its bank account. This mimicks the possibility of multitasking operations via mobile phone, ATM, web banking. Each client can only pay or withdraw money, and gets the list of the total movements after commit-ting the operation. In a realistic scenarios, transactions might also abort; e.g.:

- The ATM receives the request of money, but the server fails and does not deduct the money from the on-line account.
- The server fails after deducting the from the on-line account, but no money is given at the ATM

Still, all of the operations done by the thread over the specific bank account will be confirmed and will become effective only when the transaction is going to be **commit**ted. After a commit or an abort, no further operation through that transaction is allowed, thus requiring the user to open another transaction within the same thread. More details on the operations' requirements are given in the final marking scheme.

As in any industrial setting where teams split up the duties, you are assigned an API that you need to implement. Such an API is provided both on Canvas and at `https://github.com/jackbergus/NCL_CSC8016`. This will then require to extend the `BankFacade` for implementing the definition of novel transactions, and the implementation of `TransactionCommands` for performing the client-server communication. The student is free to choose whichever is the best way to make these two entities communicate. E.g., the bank could be either modelled as a finer-grained monitor, but inside this monitor at least one thread per logged user should be running; also, such a bank could be also implemented as a consumer threads handling all of the clients' messages.

Assumptions

- In a realistic example, communications happen between processes via UDP messages. In this module, we don't require that. We can freely assume that each client (ATM, OnLine Banking) is mimicked by one single thread. We assume they directly exploit such an interface (no FrontEnd is required!)
- If the bank is implemented as a server, such a thread might receive the "client messages" through shared variables.

- The server should also keep track of the transactions that are performed for handling commit/abort correctly. You are not required to tolerate the server crash (this is more of a back-up task rather than a concurrent programming one), but you must tolerate the client ones (that is more related to concurrent programs' management)!
- We assume that the `BankFacade` class is initialized with the users having an account in their bank as well as the balance associated to that (constructor with `HashMap<String, Double>`). The system should not allow to open/close new bank accounts.
- The server should allow multiple users logging in running contemporarily on distinct bank account. In order to maximise seriality and concurrency requirements, the students might investigate *optimistic protocols* for transactions, but this is not strictly required.
- The actual implementation provided both in Canvas and on GitHub provides incorrect solutions for the following reasons:
  1. The execution of the process is **wrongly** always sequential, as one client always blocks the other clients from accessing the bank!
  2. Client's crashes are **wrongly** not tolerated, as those are always committing the operations to then main log without checking whether the action was effectively performed or not!
  3. Somehow, the computations are "logically" correct, that is pay, withdraw, commit, and abort implement the expected semantics. Still, this is not sufficient for passing the coursework with full marks.

Submission Requirements

1. `BankFacade` and `TransactionCommands` should be extended.
2. Submit the code as a zipped *Maven* project. with **no** *jar* and *classes*. The source code will be recompiled from scratch, and no jar/class is going to be run.
3. If you want to use an external Java library, please consider the following:
   - The Java library should be explicitly described as a `<dependency>` in the `pom.xml` file, and should only access the libraries from the default *Maven Central Repository*.
   - A library might provide single concurrency mechanisms primitives, but not ready-made solutions already composing those: semaphores, monitors, locks, just logs, thread barriers, thread pools, passing le baton mechanisms are allowed. Code reuse from the exercises and examples seen in class is permitted.
   - Systems completely solving the coursework for you are **strictly prohibited**: e.g., any kind of (data) management system having concurrency control (ensuring safe concurrent thread access to any data representation) and supporting concurrent transactions (implementing any kind of transaction protocol, either pessimistic or optimistic) **must be avoided**, as they both implement commit/aborts and thread-safe operations on data.
   - None of the (direct or indirect) dependencies of the coursework should rely on external servers or processes to run or to be installed.
   - The solution should **not** include external jar files.
   - If unsure whether the solution might be exploited, please ask before submitting.

4. Attached to the source code, please provide a short report motivating the compliance of the source code to each point and sub-point of the marking scheme. The report should also state what is the name of the class of the class extending `BankFacade` and its package.
5. At the zip's root, please write a *classinfo.txt* file containing the package and class information to the class extending `BankFacade`. E.g.,

   uk.ncl.CSC8016.jackbergus.coursework.wrongimplementation.WrongBankWithLocks

Marking Scheme

The marking scheme is capped at **100%**.

- Single-Thread Correctness [**+50%**]

  +10%: I cannot open a transaction if the user does not appear in the initialization map.

  +10%: I can always open a transaction if the user, on the other hand, appears on the initialization map.

  - The returned `totalAmount` reflect the amount that was committed.

  +15%: After committing a transaction, the results provides the total changes into the account.

  - The returned commit information should contain at least one *Operation* of *OperationType*.
  - After paying money into the account, the final total amount is the sum of the previous amount of money and the amount being paid.
  - I cannot re-commit or abort a closed transaction.
  - The successful operations should contain all of the operations before the commit, as well as the commit.

  +15%: No overdraft is allowed.

  - I can always withdraw 0.0 money from my account.
  - I can never withdraw an amount of money which is greater than the amount at my disposal.
  - The commit should list all of the operations, thus including the attempt to overdraft.
  - The operation causing the overdraft shall not consider as an unsuccessful operation, rather than an *ignored* one.

- Multi-Threaded Correctness [**+40%**]

  +10%: A single user can open concurrent transactions.

  - As other concurrent transactions are not committed yet, each thread can only see the committed statues from their account.
  - Any user should be allowed to concurrently log in in its account (e.g., through different possible devices).
  - Absence of **write-write** conflicts.

  - Under the assumption that user's threads never abort, the final amount of the money in the bank account correspond to the overall total of pays and withdraws.

+10%: Multiple users can open concurrent transactions. *[Same requirements as above, plus the following:]*

  - Different users should never run serially, as they operate on different accounts (The code unrealistically assumes that neither pay other accounts nor receive money from them).

+10%: No dirty reads allowed.

  - A thread shall never see in the total account balance after the commit the amount of money being paid/withdrawn before other threads' commits.
  - A thread shall never be able to withdraw money paid by other non-committing threads.
  - Ignored operations should consider the unsuccessful withdraws from the account.

+10%: Handing aborted transactions:

  - Operations from aborted transactions should be completely ignored/reversed.
  - In particular, aborted threads should leave the bank in a consistent state, so further transactions run as expected.

- Advanced Features
  - [**+1%**] The bank is emulated realistically as a separated thread.
  - [**+1%**] The code exploits Java's concurrent collections.
  - [**+1%**] The program allows to visually determine the correctness of the operations performed by the threads (e.g., terminal prints or graphical user interfaces).
  - [**+1%**] The student correctly exploits semaphores.
  - [**+2%**] A thread perceives the updated account balance as soon as any of the remaining concurrent thread is committed.
  - [**+2%**] The student exploited the optimistic transaction principle, and the operations were logged in a journal (either in primary memory or in secondary memory).
  - [**+2%**] Usage of monitors or multithreaded producers and consumers (semaphores might be also exploited).
  - [**+3%**] Thread pools are used to handle multiple requests from multiple users.
  - [**+3%**] Any Java library imported via `pom.xml` 'not violating the 3$^{\text{rd}}$ Submission Requirement.
  - [**+4%**] In addition to the previous point, bank thread crashes are also tolerated, e.g., all of the transactions are assumed to be aborted, and the bank journal is stored on disk.