Assignment 5

This assignment is due via Absalon on January 6, 23:59. While individual hand-ins will be accepted, we strongly recommend that this assignment be solved in groups of two students. Groups may at a maximum include three students.

A well-formed solution to this assignment should include a PDF file with answers to all exercises as well as questions posed in the programming part of the assignment. In addition, you must submit your code along with your written solution. Evaluation of the assignment will take both into consideration.

Note that all homework assignments have to be submitted via Absalon in electronic format. It is your responsibility to make sure in time that the upload of your files succeeds. While it is allowed to submit scanned PDF files of your homework assignments, we strongly suggest composing the assignment using a text editor or LaTeX and creating a PDF file for submission. Email or paper submissions will not be accepted.

Learning Goals of PCSD

This assignment mimics an exam assignment, however, please note this assignment workload has been slightly reduced with respect to the actual workload of the PCSD exam, since we have less hours budgeted during normal course execution than during the exam situation.

We attempt to touch on many of the learning goals of PCSD. Recall that the learning goals of PCSD were:

(LG1) Describe the design of transactional and distributed systems.
(LG2) Explain how to enforce modularity through a client-service abstraction.
(LG3) Discuss design alternatives for a computer system, identifying system properties as well as mechanisms for improving performance.
(LG4) Analyze protocols for concurrency control and recovery, as well as for distribution and replication.
(LG5) Implement systems that include mechanisms for modularity, atomicity, and fault tolerance.
(LG6) Structure and conduct experiments to evaluate a system's performance.

(LG7) Explain techniques for large-scale data processing.
(LG8) Apply principles of large-scale data processing to concrete problems.

Given the material covered so far in the course, we will focus our attention in learning goals **LG1-LG6** in this assignment.

NOTE on Exercises: *In order to limit the workload in this assignment, we did not include any theoretical exercises.* This should in no way be taken to mean that such exercises will not be required of you during the actual exam. You can expect to have exercises in the exam, as in the other regular assignments.

Programming Task

In this programming task, you will develop a distributed *interpreter* abstraction based on data partitioning. Through the multiple questions below, you will describe your design (**LG1**), expose the abstraction as a service (**LG2**), design and implement this service (**LG3-LG5**), and evaluate your implementation with an experiment (**LG6**).

As with other assignments in this course, you should implement this programming task in Java. As an *RPC mechanism*, you are only allowed to use Jetty and XStream, and we expect you to abstract the use of these libraries behind clean proxy classes as in the assignments of the course. You are allowed to reuse communication code from the assignments when building your proxy classes.

In contrast to a complex code handout, in this assignment you will be given a simple interface to adhere to, described in detail below. We expect the implementation that you provide to this interface to adhere to the description given, and to employ architectural elements and concepts you have learned during the other qualification assignments for the course. We also expect you to respect the usual restrictions given in the qualification assignments with respect to libraries allowed in your implementation (i.e., Jetty, XStream, JUnit, and the Java built-in libraries, especially `java.util.concurrent`).

The `AccountManager` Abstraction

For concreteness, we will instantiate our abstraction with a simple example from finance. Consider a bank which needs to perform risk analysis against a set of accounts. The accounts are partitioned among multiple branches; in other words, each branch contains multiple accounts, but each account belongs to exactly one branch. An account is identified by *both* the branch identifier and an account identifier. The bank needs to manage each account (e.g., process debits and credits to balances) and to calculate the exposure of each branch to lending risk. These risks are reassessed often to allow branches to make further lending and borrowing decisions during their daily operation. Operations to manage accounts or calculate exposure may arrive at any time, so we cannot make any assumptions about whether operations can be shifted in time (e.g., processing updates only at night is not allowed by our bank).

The interpreter abstraction for this example, called `AccountManager`, exposes as its API the following *operations*:

1) `credit(int branchId, int accountId, double amount)`: This operation credits the specified account at the given branch with the provided amount. The operation raises appropriate exceptions if the branch or the account are inexistent, or if the value given is negative.
2) `debit(int branchId, int accountId, double amount)`: This operation debits the provided amount from the specified account at the given branch. The operation raises appropriate exceptions if the branch or the account are inexistent, or if the value given is negative.
3) `transfer(int branchId, int accountIdOrig, int accountIdDest, int double amount)`: This operation transfers the provided amount from the origin account to the destination account at the given branch. The operation raises appropriate exceptions if the branch or either account are inexistent, or if the value given is negative. Note that transfer operations can only occur between accounts in the same branch, and no between accounts across branches.
4) `calculateExposure(int branchId)` → `double`: This operation calculates the sum of the balances of all overdrafted accounts in the given branch and returns its absolute value as a simple estimate of the total exposure for that branch. An overdrafted accounts is an account with a negative balance. If there are no overdrafted accounts, then the exposure is zero. The operation raises an appropriate exception if the branch is inexistent.

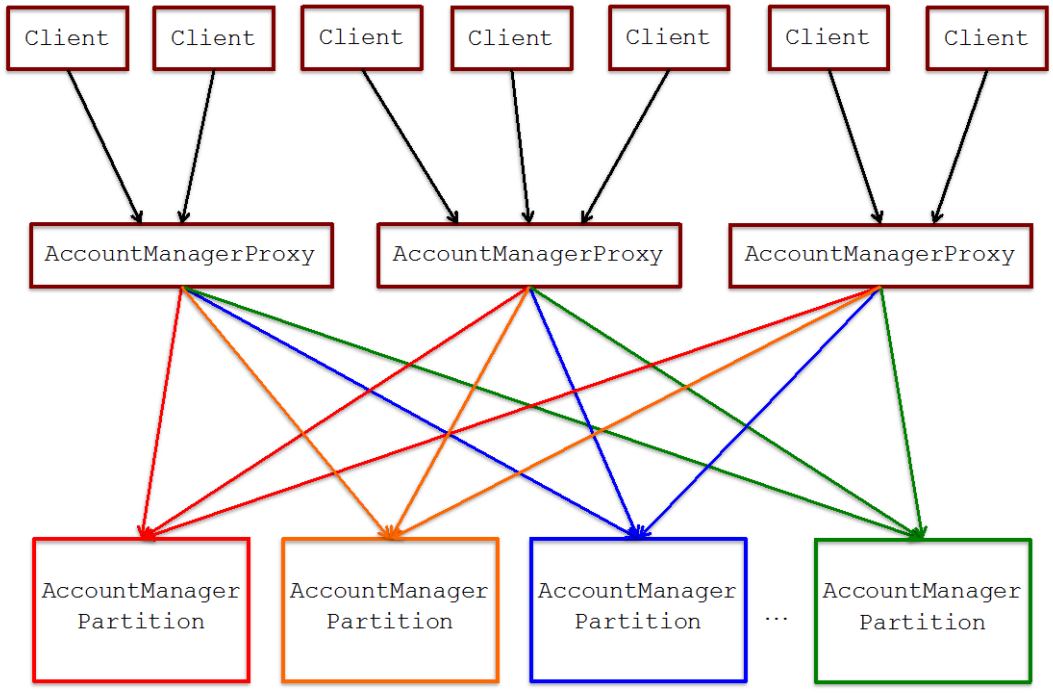
You implementation must conform to the above interface.

Proxies and Partitions

The system gives to users the illusion that operations are *atomic*, even though multiple operations may in fact be executed simultaneously. To do so, account data is partitioned by branch, and each operation is executed at a single partition uniquely determined by the branch the operation accesses. This decision implies that, given appropriate routing of operations to partitions, mechanisms for ensuring atomicity must only be provided at each individual partition in the system.

Note that as each operation touches a single branch, we must ensure that all data for a given branch is located in the same partition. However, nothing precludes data from more than one branch to be present in the same partition.

The general organization of components in the system is outlined in the figure below:



The system is organized into two main components: `AccountManagerProxy` and `AccountManagerPartition`. Proxies offer the `AccountManager` interface to clients and are responsible to route operations to partitions. Partitions execute operations atomically and return results to the calling proxies.

Clients always access the system by the *same proxy*, and you may assume that client and proxy *share fate*. If a proxy fails in the middle of processing a request from a client, then the client would need to be *restarted* with another proxy.

The `AccountManager` is based on a simple *fail-soft* design: If a partition is down, then operations on this partition will report an appropriate exception at the interface; however, operations on all other live partitions must succeed. For the purposes of this programming task, if all partitions fail, then the system becomes *unavailable*. In other words, you are not required to implement a recovery procedure for partitions.

Your implementation only needs to tolerate failures of partitions, and those failures must respect *fail-stop*: Network partitions *do not occur*, and failures can be *reliably detected*. We model the situation of partitions hosted in a single datacenter, and a service configured so that network timeouts can be taken to imply that the partition being contacted indeed failed. In other words, the situations that the partition was just overloaded and could not respond in time, or that the partition was not reachable due to a network outage are just assumed *not to occur* in our scenario.

Data and Initialization

All the data for branches and accounts is stored in *main memory* at each partition. You can generate an initial data distribution for branches, accounts, and balances according to a procedure of your own choice; only note that the distribution must make sense for the experiment you will design and carry out below.

You may assume in your implementation that the data is loaded once when each partition is initialized at its constructor. Similarly, the configuration of the components of the system is loaded by each proxy at its constructor. For simplicity, assume that the data for accounts and branches and the configuration of components are available as files in a shared filesystem readable by all instances (or alternatively, that these files are replicated in the same path in every instance).

High-Level Design Decisions and Modularity

First, you will document the main organization of modules and data in your system.

Question 1 (LG1, LG2, LG3, LG5): *Describe how you organized data distribution in your implementation of the* `AccountManager` *and your implementation of enforced modularity. In particular mention the following aspects in your answer:*

- *How you chose to partition branch data across nodes (e.g., round robin, hash, range) and justify why.*
- *How you handled failures of individual components.*

(2 paragraphs; 1 paragraph each aspect)

Atomicity and Fault-Tolerance

Now, you will argue for your implementation and testing choices regarding atomicity and fault-tolerance.

Question 2 (LG3, LG4, LG5): *An* `AccountManagerPartition` *instance executes operations originated at multiple* `AccountManagerProxy` *instances. Describe how you ensured atomicity of these operations. In particular, mention the following aspects in your answer:*

- *Which method did you use for ensuring serializability at each partition (e.g., locking, optimistic approach, or simple queueing of operations)? Describe your method at a high level.*
- *Argue for the correctness of your method; to do so, show how your method is logically equivalent to a trivial solution based on a single global lock.*

- *Argue for the performance of your method; to do so, explain your assumptions about the workload the service is exposed to and why you believe other altenatives you considered (e.g.., locking, optimistic approach, or simple queueing of operations) would be better or worse than your choice.*

NOTE: The method you design for atomicity does not need to be complex, as long as you argue convincingly for its correctness, its trade-off in complexity of implementation and performance, and how it fits in the system as a whole.
(4 paragraphs; 1 paragraph each for first two aspects, 2 paragraphs for third aspect)

NOTE on Testing: *In order to manage the workload in this assignment, we will not require you to describe the testing procedures you have employed for your code.* This should in no way be taken to mean that such a description will or will not be required of you during the actual exam.

Experiments

Finally, you will evaluate the scalability of your system experimentally.

Question 3 (LG6): *Design, describe the setup for, and execute an experiment that shows how the throughput of the service scales with the addition of more partitions. Remember to thoroughly document your setup. In particular, mention the following aspects in your answer:*

- *Setup: Document the hardware, data size and distribution, and workload characteristics you have used to make your measurements. In addition, describe your measurement procedure, e.g., procedure to generate workload calls, use of timers, and numbers of repetitions, along with short arguments for your decisions. Also make sure to document your hardware configuration and how you expect this configuration to affect the scalability results you obtain.*
- *Results: According to your setup, show a graph with your throughput measurements on the y-axis and the number of partitions on the x-axis. How does the observed throughput scale with the number of partitions? Describe the trends observed and any other effects. Explain why you observe these trends and how much that matches your expectations.*

(2 paragraphs + figure; 1 paragraph each aspect, figure required for Results)