Principles of Computer System Design (PCSD), Block 2, 2012/2013

Final Exam

This is your final 5-day take home exam for Principles of Computer System Design, block 2, 2012/2013. This exam is due via Absalon on January 22, 23:55pm. It is going to be evaluated on the 7-point grading scale with external grading, as announced in the course description.

Hand-ins for this exam must be individual. Cooperation on or discussion of the contents of the exam with other students is strictly forbidden. The solution you provide should reflect your knowledge of the material alone. The exam is open-book and you are allowed to make use of the book and other reading material of the course. If you use on-line sources for any of your solutions, they must be cited appropriately. While we require the individual work policy outlined above, we will allow students to ask *clarification* questions on the formulation of the exam during the exam period. These questions will only be accepted through the forums on Absalon, and will be answered by the TAs or your lecturer. The goal of the latter policy is to give all students fair access to the same information during the exam period.

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

Do not get hung up in a single question. It is best to make an effort on every question and write down all your solutions than to get a perfect solution for only one or two of the questions. Nevertheless, keep in mind that a concise and well-written paragraph in your solution is always better than a long paragraph.

Note that your solution has to be submitted via Absalon in electronic format, except in the unlikely event that Absalon is unavailable during the exam period. If this unlikely event occurs, we will publish the exam at the URL https://www.dropbox.com/s/4apo3ri62lh6qcq/pcsd-final-exam.pdf and accept submissions through the email pcsd2012exam@gmail.com. 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 solution, we strongly suggest composing your solution using a text editor or LaTeX and creating a PDF file for submission. Paper submissions will not be accepted, and email submissions will only be accepted if Absalon is unavailable.

Learning Goals of PCSD

Recall the learning goals of PCSD, as stated in SIS:

(LG1) Describe the design of a computer system.
(LG2) Explain how to enforce modularity through a client-service abstraction.
(LG3) Analyze protocols for concurrency control and recovery, as well as for distribution and replication.
(LG4) Implement systems that include mechanisms for modularity, atomicity, and fault tolerance.
(LG5) Apply principles of large-scale data processing to concrete problems.
(LG6) Structure and conduct experiments to evaluate a system's performance.

The multiple questions and tasks in this exam are targeted at the learning goals above.

Questions

Question 1 (Abstractions and Data Processing) (LG1, LG5). Recall that the MapReduce abstraction for data processing data consists of two user-defined functions[1]:

- `map(k1, v1) → list(k2, v2)`
- `reduce(k2, list(v2)) → list(v2)`

In this question, you will show how to implement a *single-threaded* MapReduce runtime that operates on large data in external memory. The runtime does not need to exploit parallelism; however, it should be able to operate on large, disk-resident data collections that *exceed the size of main memory*.

To phrase your solution, address the following:

(a) Show the pseudocode for the Map Phase of your runtime, which consists of applying `map` to all key-value pairs in a disk-resident input file, and saving the intermediate key-value pairs back to disk.

(b) Show the pseudocode for the Reduce Phase of your runtime, which consists of reading all Map Phase results, applying `reduce` appropriately, and saving the final output to disk. Note that each call to `reduce` gets a key emitted by *some* `map` application along with *all* values emitted by *any* `map` application for that key. Every key is given to a `reduce` call exactly once during the Reduce Phase.

(c) Does your implementation tolerate skew in the keys emitted from the Map Phase? If so, explain why; if not, explain how you could fix it.

When stating your pseudocode, make use where necessary of external memory algorithms, such as scan, external sorting, or hashing. Should you choose one of these algorithms, there is no need to re-state the complete algorithm in your solution, but you must explicitly state any changes you would make to the algorithm and how you would use the algorithm to implement your MapReduce runtime.

Note: You may assume that individual keys and values are small compared to main memory and that any list of values given as *input* to a `reduce` call fits comfortably in main memory as well. Explicitly state any other assumptions in your solution.

Question 2 (Serializability & Locking) (LG3). Consider the following two transaction schedules with time increasing from left to right. *C* indicates commit.

Schedule 1

T1:	R(X)				W(Y)	C		
T2:		W(X)	W(Y)				C	
T3:				W(Y)				C

[1] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing in Large Clusters. OSDI 2004. Available at: http://research.google.com/archive/mapreduce.html

Schedule 2

T1:	R(X)						W(Y)	C
T2:		W(Y)	W(Z)	C				
T3:					W(Y)	C		

Now answer the following questions:

(a) Draw the precedence graph for each schedule. Are the schedules conflict-serializable? Explain why or why not.
(b) Could the schedules have been generated by a scheduler using strict two-phase locking (strict 2PL)? If so, show it by injecting read/write lock operations in accordance to strict 2PL rules. If not, explain why.

Question 3 (Recovery) (LG3). Consider the recovery scenario described in the following, in which we use the ARIES recovery algorithm. At the beginning of time, there are no transactions active in the system and no dirty pages. A checkpoint is taken. After that, three transactions, T1, T2, and T3, enter the system and perform various operations. The detailed log follows:

```
LOG
LSN  LAST_LSN  TRAN_ID  TYPE        PAGE_ID   UNDONEXTLSN
---  --------  -------  -----       --------  -----------
1    -         -        begin CKPT  -         -
2    -         -        end CKPT    -         -
3    NULL      T1       update      P1        -
4    NULL      T2       update      P1        -
5    4         T2       update      P7        -
6    NULL      T3       update      P3        -
7    5         T2       update      P3        -
8    6         T3       update      P5        -
9    8         T3       commit      -         -
10   7         T2       abort       -         -
11   10        T2       CLR         P3        5
12   9         T3       end         -         -
```

At this point in the execution, i.e., after LSN 12, the system crashes. Now answer the following questions:

(a) Apply the ARIES recovery algorithm to the scenario above. Show:
   1. the state of the transaction and dirty page tables after the analysis phase;
   2. the sets of winner and loser transactions;
   3. the values for the LSNs where the redo phase *starts* and where the undo phase *ends;*
   4. the set of log records that may cause pages to be rewritten during the redo phase;
   5. the set of log records undone during the undo phase;
   6. the contents of the log after the recovery procedure completes.

(b) In the example above, you have shown the values for the LSNs where the redo phase *starts* and where the undo phase *ends.* State the conditions that characterize these LSNs in general, independent of the scenario above. In other words, how far back into the log must ARIES scan during the redo and undo phases of recovery?

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**), implement this service (**LG3, LG4**), and evaluate your implementation with an experiment (**LG6**).

You should implement this programming task either in Java or C#. Make sure to use the latest version of Oracle's Java VM or of Microsoft .NET for this programming task. As an *RPC mechanism*, we will allow you to employ any one of the following to expose your implementation as a service: JAX-WS, RMI (Java), or WCF (C#). Recall that you are not required to use Azure for your tests. As long as you employ one of the RPC mechanisms above, you are free to use your local computational resources to solve this programming task. In the latter case, you are free to use the lightweight HTTP server in Java and C#, as opposed to deploying your code to Tomcat, IIS, or another full-blown application server.

Remember to clearly state in your solution all your choices with respect to RPC mechanism and configuration for tests. In addition, recall that you must submit your code as part of your solution. We recommend that you provide your code in a ZIP file with:

- A README file with brief remarks necessary to run your code, if any.
- A folder with all your source code files.
- A separate folder with all project files ready for running/inspection in the IDE you employed (e.g., Eclipse, NetBeans, VisualStudio) along with auto-generated files and initialization files.

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 is 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, 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.

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 account 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.

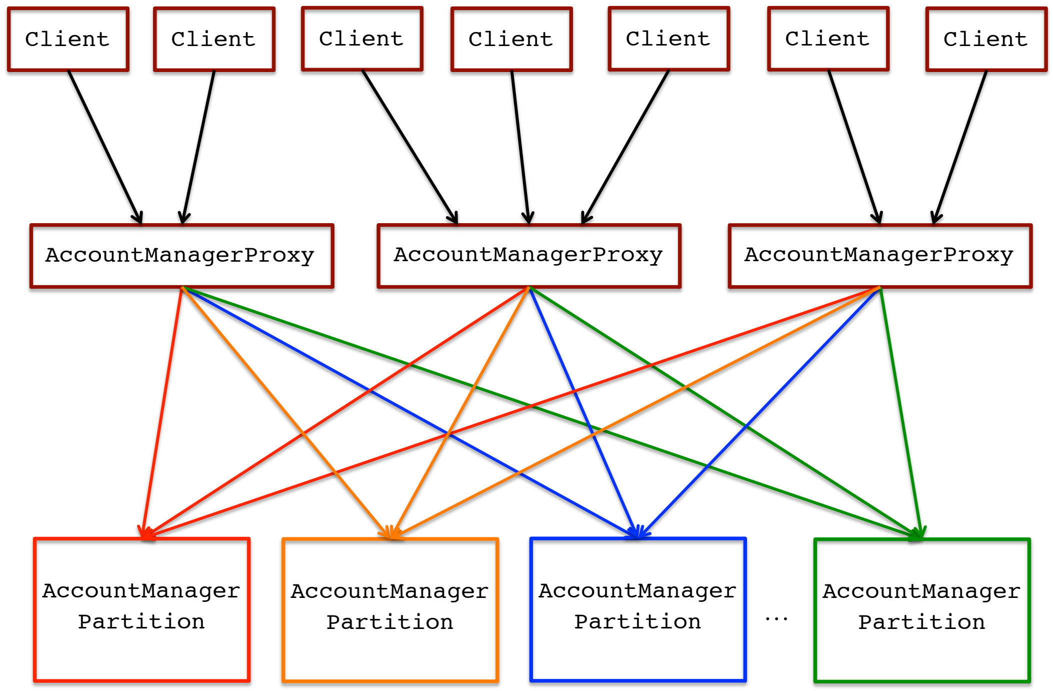
You implementation must conform to the above interface.

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, LG4): *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. If you make any assumptions about the distributions of branch or account identifiers, make sure to explicitly state your assumptions.*
- *Which RPC mechanism you have chosen, along with advantages and disadvantages of this mechanism.*

(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): *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 alternatives you considered (e.g.., locking, optimistic approach, or simple queueing of operations) would be better or worse than your choice. If possible, employ a back of the envelope calculation, or supporting numbers from your experiments.*

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)

Question 3 (LG4): *Describe how you have tested your service. In particular, mention the following aspects in your answer:*

- *How you tested that operations were indeed atomic.*
- *How you tested that the system indeed respects a fail-soft design.*

(2 paragraphs; 1 paragraph each aspect)

Experiments

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

Question 4 (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.*
- *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. Describe briefly your results. How does the observed throughput scale with the number of partitions? Explain why.*

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