Principles of Computer System Design (PCSD), 2013/2014

Final Exam

This is your final 5-day take home exam for Principles of Computer System Design, block 2, 2013/2014. This exam is due via Absalon on January 21, 23:59. 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 http://www.diku.dk/~bonii/pcsd2013 and accept submissions through the email bonii@diku.dk. It is your responsibility to make sure in time that the upload of your files succeeds. Except for the exercise portion of the exam, where submission of a scan of your handwritten solution is encouraged, 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

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

(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.

Exercises

NOTE: For the exercises below, we encourage you to solve the exercises by hand with pen-and-paper and hand in a high-resolution scan of your solution as part of your report PDF. A solution composed with a text editor or LaTeX will also be accepted.

Question 1: Proximity (LG7, LG8)

Consider the problem of merging two big data sets of N_1 and N_2 records, respectively. Each record is l words long and the key used in record comparisons is k words long ($k \leq l$). For an integer parameter d, consider d-way merging algorithm when performing the merging task.

a) Analyse the CPU cost of this algorithm in terms of word operations.

b) Analyse the I/O cost of this algorithm. Use B to denote the size of the memory blocks (pages) transferred between the disk and main memory and M to denote the size of main memory; both measured in words.

c) Analyse the address-translation costs of this algorithm. Use W to denote the size of the address-translation cache measured in words and P to denote the branching factor of the nodes in the page table.

Question 2: Parallelism (LG7, LG8)

Assume that we perform a hash-join operation for two big relations R and S on a database engine that has p processors. Each processor has its own memory and disk, i.e., we operate with a shared-nothing system. The relations have N_R and N_S records, respectively. The join algorithm uses two hash functions h_1 and h_2. The first hash function h_1 is used to partition the records of R and S into k buckets $R_1, R_2, ..., R_k$ and $S_1, S_2, ..., S_k$. You can assume that the main memory of every processor can occupy $O(k)$ pages. The second hash function h_2 is used to map the records to a processor that performs the local joins of buckets R_i and S_i, for $i \in \{1, 2, ..., k\}$.

a) Give explicitly what are the domains and ranges of the two hash functions.

b) Describe briefly (preferably with a picture, not pseudo-code) how this parallel hash-join algorithm works.

c) Under the assumption of uniform hashing, how many I/O's do each processor perform?

d) Under the same assumption, how many messages do each processor send and receive?

Programming Task

In this programming task, you will develop an *integration broker* abstraction in a simplified order processing scenario. 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 assignments in this course, you should implement this programming task in Java, compatible with JDK 6. 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 constrast to a complex code handout, in this exam 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 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`).

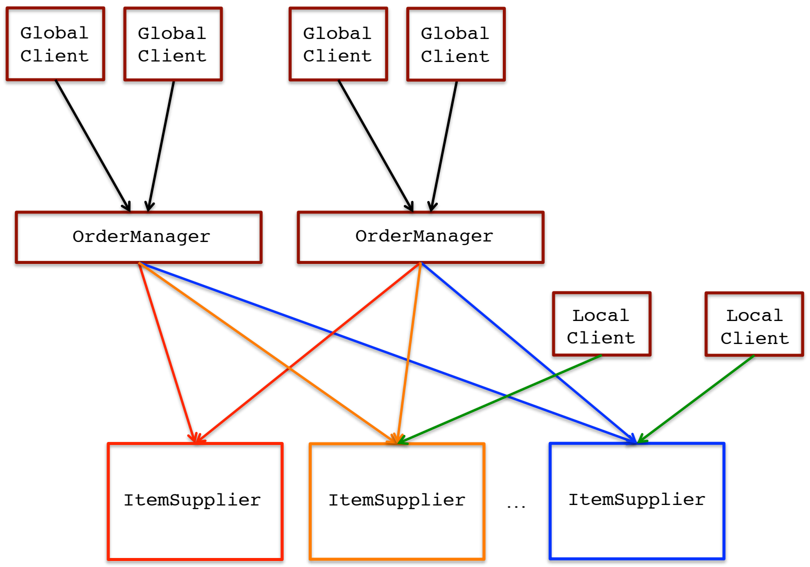
The `OrderManager` and `ItemSupplier` Abstractions

For concreteness, we focus on an example from global supply chains: A global manufacturer, such as an airplane builder, acquires items from multiple suppliers around the world, and employs these items in their own manufacturing process. For the global manufacturer, it is important to control orders for multiple items across suppliers; for the local suppliers; it is important to correctly record and query the orders received from the global manufacturer.

Since different suppliers employ multiple order processing systems, the global manufacturer needs an integration middleware, often referred to as an *integration broker*, to coordinate order workflows across the order processing systems of the suppliers. The integration broker is typically implemented over a *durable* communication abstraction. Workflows are submitted to this broker component and executed *asynchronously*. While workflows are not atomic as a whole, order registration steps with *individual suppliers* are *atomic*.

We model the broker abstraction in this assignment by an `OrderManager` interface, and expose order processing systems of item suppliers through a common `ItemSupplier` interface. For simplicity, we make the integration broker unidirectional, i.e., an order manager accepts order workflow requests from global clients, and executes them against item suppliers, but no requests flow from item suppliers back to order managers or to other item suppliers. Given this restriction, we expose `ItemSupplier` instances as services, accessible to `OrderManager` instances or local clients via RPC; `OrderManager` instances themselves are also accessible via RPC to global clients.

The general organization of components in the system is outlined in the following figure.



The **`OrderManager`** exposes as its API the following *operations*:

1) `int registerOrderWorkflow(List<OrderStep> steps)`: This operation registers an order execution workflow with an order manager. The order manager must record the workflow *durably* before returning a workflow ID to the client. The workflow is also kept in a data structure of the order manager, so as to answer queries on the status of the workflow (see below). The order workflow is executed *asynchronously* by the order manager. Each step of the workflow is executed against a single item supplier. The order manager throws appropriate exceptions if steps are malformed, or on any error condition that precludes durable recording of the request.
2) `List<StepStatus> getOrderWorkflowStatus(int orderWorkflowId)`: This operation returns the current status of each step in the given order workflow. The operation raises an appropriate exception if the workflow ID is inexistent.

In order to obtain reliable execution of workflow steps and also allow for queries from local clients, the **`ItemSupplier`** exposes as its API the following *operations*:

1) `executeStep(OrderStep step)`: This operation executes an order step with the item supplier. An order step consists of ordering given amounts from a set of items. After validating the requested step, the item supplier must record the step *durably*. In addition, the item supplier also executes the whole operation *atomically* with respect to other operations on the same item supplier. Each item supplier keeps a data structure with the items and their ordered amounts, which gets updated by execution of an order step. The item supplier throws appropriate exceptions if the step is not intended to it or is malformed, or on any error condition that precludes durable recording of the request.
2) `List<ItemQuantity> getOrdersPerItem(Set<Integer> itemIds)`: This operation returns the current cummulative ordered amount for each item in the input set. This operation is executed *atomically* with respect to other operations on the same item supplier. The operation raises an appropriate exception if any of the items cannot be found by their ID.

Durability is achieved at each component (i.e., order managers and item suppliers) by keeping a *log of operations* on disk. You may assume that flushed writes to the filesystem are enough to force log records. *To limit the workload in this exam, you do not need to implement procedures for checkpointing or for recovery from the log.*

Failure handling

Even though recovery of a component from its log is not required for this exam, failures of individual components must be isolated, and other components in the system should continue operating normally in case of such a failure. You should ensure that failure of an order manager does not disrupt item suppliers; conversely, failure of an item supplier implies that any ongoing order steps against that supplier will temporarily fail to be executed. However, the workflow must remain active with the order manager, and the steps should be retried (in principle, until the item supplier recovers, even though recovery is not necessary for this exam). Failure of one step should also not compromise execution of subsequent steps in the workflow.

Your implementation only needs to tolerate failures which respect the *fail-stop* model: 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 component being contacted indeed failed. In other words, the situations that the component was just overloaded and could not respond in time, or that the component was not reachable due to a network outage are just assumed *not to occur* in our scenario.

Data and Initialization

All the data for items and their ordered amounts, as well as data for order workflow steps and their status is stored in *main memory* at the corresponding components. You can generate an initial data distribution for items 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 each component is initialized at its constructor. Similarly, the configuration of the item suppliers in the system is loaded by each order manager at its constructor. For simplicity, assume that the data and the configuration of components are available as files in a shared filesystem readable by all components (or alternatively, that these files are replicated in the same path in local filesystems accessible by each component).

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 your overall implementation of the OrderManager and ItemSupplier, including the following aspects in your answer:*

- *Considering your use of logging, what RPC semantics are effectively implemented between clients and `OrderManager` or `ItemSupplier`? What RPC semantics are effectively implemented between `OrderManager` and `ItemSupplier`? Explain.*
- *How did you make workflow processing asynchronous at the `OrderManager`?*
- *How did you handle failures of individual components? In particular, how do you handle failure propagation?*

(3 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 ItemSupplier executes operations originated at multiple instances of* `OrderManager` *as well as local clients. 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 item supplier (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 or to a well-known locking protocol (e.g., a variant of two-phase locking).*
- *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)

Question 3 (LG4, LG5): *In your implementation, you were not required to implement a procedure for log-based recovery of individual components. Explain how you would recover from a failure of:*

- *an OrderManager*
- *an ItemSupplier*

For each of the two scenarios, explain any necessary interaction with other components to achieve recovery by replaying the log. Also explain how you would use the log during recovery, i.e., how is the information in the log used to rebuild data structures in main memory, and why is it sufficient.
(2 paragraphs; 1 paragraph for each scenario)

Testing

Question 4 (LG4, LG5): *Describe your high-level strategy to test your implementation. In particular, mention the following aspects in your answer:*

- *How you tested execution of workflows by the* `OrderManager`*, considering asynchronicity of execution.*
- *How you tested that operations were indeed atomic at the* `ItemSupplier`*.*
- *How you tested error conditions and failures of the multiple components.*

(3 paragraphs; 1 paragraph each aspect)

Experiments

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

Question 5 (LG6): *Design, describe the setup for, and execute an experiment that shows how well your implementation of a single ItemSupplier behaves as concurrency is increased. You will need to argue for a basic workload mix involving calls from order managers as well as local clients. Given a mix, you should report how the throughtput of the service scales as more local clients / order managers are added.*

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 for a single ItemSupplier on the y-axis and the number of local clients / order managers on the x-axis, maintaining the workload mix you argued for in your setup. How does the observed throughput scale with the number of local clients / order managers? Describe the trends observed and any other effects. Explain why you observe these trends and how much that matches your expectations.*

(3 paragraphs + figure; 2 paragraph for workload design and experimental setup + figure required for Results + 1 paragraph for discussion of results)