Advanced Computer Systems (ACS), 2015/2016

Final Exam

This is your final 5-day take home exam for Advanced Computer Systems, block 2, 2015/2016. This exam is due via Digital Exam 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 Digital Exam (https://eksamen.ku.dk/) in electronic format, except in the unlikely event that Digital Exam is unavailable during the exam period. If this unlikely event occurs, we will publish the exam in Absalon 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. 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 Digital Exam is unavailable.

Learning Goals of ACS

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

(LG1) Describe the design of transactional and distributed systems, including techniques for modularity, performance, and fault tolerance.
(LG2) Explain how to employ strong modularity through a client-service abstraction as a paradigm to structure computer systems, while hiding complexity of implementation from clients.
(LG3) Explain techniques for large-scale data processing.
(LG4) Implement systems that include mechanisms for modularity, atomicity, and fault tolerance.
(LG5) Structure and conduct experiments to evaluate a system's performance.
(LG6) Discuss design alternatives for a modular computer system, identifying desired system properties as well as describing mechanisms for improving performance while arguing for their correctness.
(LG7) Analyze protocols for concurrency control and recovery, as well as for distribution and replication.
(LG8) Apply principles of large-scale data processing to analyze concrete information-processing problems.

Exercises

Question 1: Data Processing (LG3, LG8)

An online marketing firm, called `tripleclick.com`, provides analytics over clickstreams to their customers. A basic step in multiple analytics pipelines of `tripleclick.com` is to match each click event to the K other events closest in time. The staff of `tripleclick.com` calls this step the *K-closest event query*. The main data table over which the K-closest event query is performed is the click table with schema `clicks(eventid, time, event_details)`. The output of the K-closest event query is a set of pairs of `eventid`'s such that each left event is paired with its K closest as right events. For example, consider the events:

```
eventid          time       event_details
--------         -----      --------------
1                42         …
2                43         …
3                84         …
4                44         …
```

With K=2, then the K-closest event query returns:

```
left_eventid     right_eventid
-------------    --------------
1                2
1                4
2                1
2                4
3                2
3                4
4                1
4                2
```

You should assume that there are no indices available on the table – `tripleclick.com`'s scripts simply collect the clicks from multiple webservers into a single plain file, while ensuring that timestamps are comparable across events. You should assume that the consolidated click table does not fit in main memory, but that main memory is larger than the square root of the size of the click table. In addition, you can assume that K is a small number compared with the data sizes involved, so keeping a staging area with a number of events equal to K (or K multiplied by a small constant) is essentially free in terms of main memory. If you need to make any further assumptions in your answers, please state them clearly as part of your solution.

1. State an external-memory algorithm to answer the K-closest event query above. Argue for the algorithm's correctness and efficiency.
2. State the I/O cost of the algorithm you designed in part 1 above in terms of the number of pages *P* in table `clicks`. Explain why the algorithm has the cost stated.

NOTE 1: To state an algorithm, you can reference existing sort-based or hash-based external memory algorithms. You should not state all the steps of these existing algorithms from scratch again, but you should clearly state the steps that you need to change in the algorithms you reference, and also how you change these steps. To describe how you change a step, refer to the step and list the sub-steps that need to be executed to achieve your goal.

NOTE 2: Instead of just using several existing external memory algorithms in sequence as black boxes, you should design a single algorithm that addresses the whole query holistically. That is why in NOTE 1 we expect that you will need to show changes to steps of existing algorithms.

Question 2: Concurrency Control (LG7)

For each of the following items, you are asked to exhibit a *schedule* fulfilling the requirements stated, as well as to *justify why* the schedule fulfills the requirements:

(a) Show a schedule that is view-serializable, but not conflict-serializable. Explain.
(b) Show a schedule that is conflict-serializable, but could not have been generated by a 2PL scheduler. Explain.
(c) Show a schedule that could be generated by a 2PL scheduler, but not by a S2PL scheduler. Explain.
(d) Show a schedule that could be generated by a S2PL scheduler, but not by a CS2PL scheduler. Explain.

Programming Task

In this programming task, you will develop an *epoch-based interpreter* abstraction in a simplified auction market scenario. Through the multiple questions below, you will describe your design (**LG1**), expose the abstraction as a service (**LG2**), design and implement this service (**LG4, LG6, LG7**), and evaluate your implementation with an experiment (**LG5**).

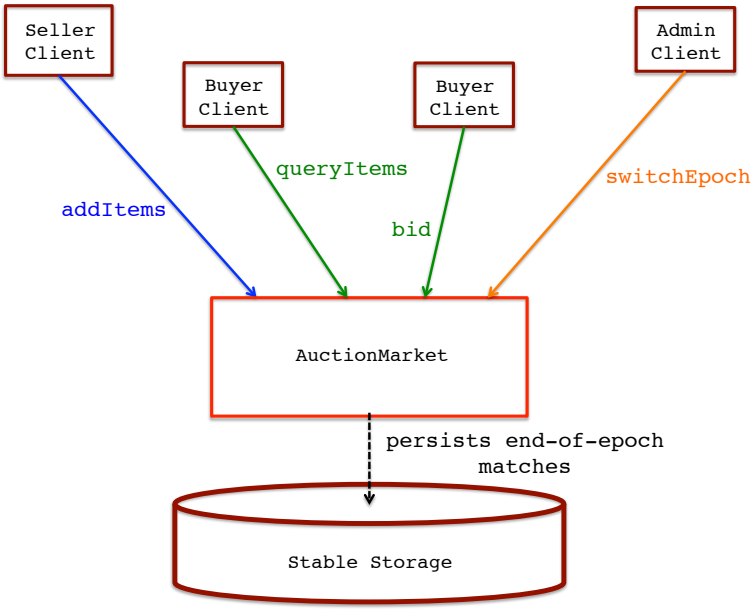
As with assignments in this course, you should implement this programming task in Java, compatible with JDK 8. As an *RPC mechanism*, you are only allowed to use Jetty and XStream, and we expect you to abstract the use of these libraries behind clean proxy classes as in the assignments of the course. You are allowed to reuse communication code from the assignments when building your proxy classes.

In contrast to a complex code handout, in this 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 `AuctionMarket` Abstraction

For concreteness, we focus on an example of an auction market. The processing of the market is organized into *epochs*. During an epoch, the market collects items available from sellers and bids for these items from buyers. At the end of the epoch, the market calculates the best matching for the collected items and bids, makes the matching durable, removes all of its state (including unmatched items), and starts a new epoch.

We model the market through the `AuctionMarket` interface, which provides access to all operations of the market. Access control to market operations by clients is orthogonal to the design of the auction market and can be ignored in your solution. However, all operations in the market must be made atomic. The general organization of components in the system is outlined in the following figure.



During an epoch, *the whole state of the* `AuctionMarket` *service is kept in main memory, and accessed concurrently via RPC by a variety of clients*. In a typical interaction with the auction market, a buyer client employs the `queryItems` operations to list the items that were added by seller clients through `addItems` operations. The buyer client then updates the bids of a given buyer organization for a subset of the items using the `bid` operation. Bids of a given buyer organization may be updated concurrently by multiple buyer clients. The market keeps bids blind, i.e., a buyer organization does not have access to the bids of other buyer organizations (nor its own through the auction market service, though organizations could keep track of their own bids in a separate service of their own).[1]

The end of an epoch is marked by a call to the `switchEpoch` operation. Epochs typically have long duration, such as an hour or even a whole day. At the end of an epoch, each item is matched with its best bid (or the empty bid if no bids were performed for that item), and the whole matching is checkpointed to stable storage. *For simplicity, you may assume that flushed writes to the filesystem are enough to force the records of the matching and make them durable.* After the matching is made durable, the whole in-memory state of the `AuctionMarket` service is removed and the `switchEpoch` operation completes, marking the start of a new epoch. *To limit the workload in this exam, you do not need to implement procedures for dealing with partially written matching results. You will need, however, to design a format for the matching results written to stable storage, ensure correctness of execution of* `switchEpoch` *calls with respect to other operations called by multiple threads concurrently, and make the checkpoint of a matching durable as described above.*

The **`AuctionMarket`** service exposes as its API the following *operations*:

1) `addItems(Set<Item> items)`: The operation adds the given set of items to the items available for trading at the auction market in the current epoch. The operation must be atomic with respect to all other operations in the `AuctionMarket`. Each item added includes an item ID, an item description, and a seller organization ID. All IDs of items and organizations are restricted to positive integers. This operation throws appropriate exceptions if any of the items already exists in the auction market or if any IDs are out of bounds.
2) `List<Item> queryItems()`: The operation lists all the items that are available for trading at the auction market in the current epoch. The operation must be atomic with respect to all other operations in the `AuctionMarket`.
3) `bid(Set<Bid> bids)`: The operation records or updates the given bids in the auction market for the current epoch. A bid is composed of an item ID, a buyer organization ID, and a bid amount. For each combination of item ID and buyer organization ID, there can be exactly one active bid, corresponding to the latest bid amount recorded (or updated) by this operation. The operation must be atomic with respect to all other operations in the `AuctionMarket`. This operation throws appropriate exceptions if any IDs are out of bounds, refer to inexistent items, or include bids with negative amounts.
4) `switchEpoch()`: The operation first calculates the best bid for each item added in the current epoch. The best bid is the one of highest amount for the item, i.e., the highest bidder wins. Second, the operation makes the item information (item ID, item description, and seller organization ID) together with the information on the corresponding best bid (buyer organization ID and bid amount) durable for all items (by flushed writes to the filesystem as pointed out above), and finally removes all items and bids from the auction market. The operation must be atomic with respect to all other operations in the `AuctionMarket`. This is the only operation

[1] This scenario is loosely inspired by markets employed by food banks in the US, as described in the podcast "The Pickle Problem", available at http://www.npr.org/sections/money/2015/11/25/457408717/episode-665-the-pickle-problem.

that makes in-memory state at the `AuctionMarket` service durable. When control returns to the caller, a new epoch for the auction market has started.

Your implementation should be focused on the `AuctionMarket` service. For testing and experimentation purposes, you will need to create programs that simulate the calls made to this service by seller, buyer, and admin clients. However, *you do not need to formally structure nor model interfaces for these clients in your solution*.

Failure handling

Even though recovery of the `AuctionMarket` from partially written matching results during an epoch switch 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 the `AuctionMarket` does not propagate to clients and vice-versa by employing an appropriate RPC mechanism. Stable storage is assumed not to fail; nevertheless, handlers of I/O exceptions raised while writing to stable storage should conservatively crash the whole `AuctionMarket` service, simulating a failure of the service itself.

Your implementation only needs to tolerate failures that 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 the `AuctionMarket` service is stored in *main memory* during an epoch. Epochs always start from an empty state. To populate the state, you can generate item additions via calls to `addItems` and bids via calls to `bid` according to a procedure of your own choice; only note that the timing of calls and distribution of values must make sense for the experiment you will design and carry out below.

You may assume that any configuration information is loaded once each component is initialized at its constructor (e.g., naming information for the `AuctionMarket` service needed by clients). For simplicity, you can encode configuration information as constants in a separate configuration class.

High-Level Design Decisions, Modularity, Fault-Tolerance

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

Question 1 (LG1, LG2, LG4, LG6): *Describe your overall implementation of the* `AuctionMarket`*, including the following aspects in your answer:*

- *What RPC semantics are implemented between clients and* `AuctionMarket`*? Explain.*
- *How does your implementation ensure that the operations of the* `AuctionMarket` *behave with all-or-nothing atomicity with respect to other operations? How is that guaranteed even in the case of failures? Explain.*
- *Is checkpointing of the final matching state at the* `switchEpoch` *operation sufficient for durability of all the operations at the* `AuctionMarket` *service? If so, argue how checkpointing provides durability for auctions run by the* `AuctionMarket` *service; if not,*

explain why and describe what strategy you would employ to provide durability for all operations.

(1 paragraph for overall code organization +3 paragraphs, one for each of three points above)

Before-or-After Atomicity

Now, you will argue for your implementation choices regarding before-or-after atomicity.

Question 2 (LG1, LG4, LG6, LG7): *The* `AuctionMarket` *service executes operations originated at multiple clients. Describe how you ensured before-or-after atomicity of these operations. In particular, mention the following aspects in your answer:*

- *Which method did you use for ensuring serializability at the* `AuctionMarket` *service (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 whether or not you need to consider the issue of predicate locking in your implementation, and explain why.*
- *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.*

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, one for each point)

Testing

You will then describe your testing strategy.

Question 3 (LG4, LG6): *Describe your high-level strategy to test your implementation. In particular, mention the following aspects in your answer:*

- *How did you test all-or-nothing atomicity of operations at the AuctionMarket?*
- *How did you test before-or-after atomicity of operations at the AuctionMarket? Recall that you must document how your tests verify that anomalies do not occur (e.g., dirty reads or dirty writes).*

(2-3 paragraphs; 1 paragraph for first aspect, 1-2 paragraphs for second aspect)

Experiments

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

Question 4 (LG5): *Design, describe the setup for, and execute an experiment that shows how well your implementation of the* `AuctionMarket` *service behaves as concurrency in the number of buyer clients is increased. You will need to argue for a basic workload mix involving calls from the various clients accessing the service. You should present results of clients accessing the service via RPC and compare them with results in which clients access the service via direct method call. You can focus on a single*

epoch, and thus ignore admin clients. Given a mix of operations from seller and buyer clients, you should report how the query throughput of the service scales as more buyer clients 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 the* `AuctionMarket` *service on the y-axis and the numbers of buyer clients on the x-axis, while keeping the workload mix you argued for in your setup fixed. How does the observed throughput scale with the number of buyer clients? 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)