Principles of Computer System Design (PCSD), 2014/2015

Final Exam

This is your final 5-day take home exam for Principles of Computer System Design, block 2, 2014/2015. This exam is due via Absalon on January 22, 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/pcsd2014 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 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

Question 1: Data Processing (LG7, LG8)

A global social network website for the wealthy, called `richierich.com`, stores information about users, their net worth, and their friendship connections. Given the worldwide trend in increase in numbers of millionaires, `richierich.com` asks their developers to implement data analyses that are scalable to large datasets. There are two main data tables at `richierich.com`: (a) a users table with the schema `users(uid, networth)`; and (b) a friend table with schema `friends(uid1, uid2)`.

You are asked to design external memory algorithms to answer the following query:

"*Produce a list of users with their uid, networth, and the number of friends they have.*"

You should assume that there are no indices available on the tables – `richierich.com` does not allow analytic queries in their production database, so you are given just plain files with the data, which you must analyze using your own server. Since `richierich.com` did not give you much money to buy your server, you should also assume that neither table fits in main memory, but that main memory is larger than the square root of the size of the larger table. If you need to make any further assumptions in your answers, please state them clearly as part of your solution.

1. State a sort-based external memory algorithm to answer the query above. Argue for the algorithm's correctness and efficiency.
2. State a hash-based external memory algorithm to answer the query above. Argue for the algorithm's correctness and efficiency.
3. State the I/O cost of the algorithms you designed in questions 1 and 2 above in terms of the number of pages U in table `users` and F in table `friends`. Explain why the algorithms have 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: Distributed Transactions (LG4)

The schedule for three distributed transactions (T1, T2, T3) spanning three nodes has been outlined below. Node 1 stores data elements X and Y, node 2 stores data elements A and B while node 3 stores data elements C and D. Strict two phase locking is used as the concurrency control mechanism.

NODE 1	**T1**	**R(X)**										
	T2			**R(X)**			**W(X)**					
	T3		**R(Y)**			**W(Y)**						
NODE 2	**T1**			**R(A)**			**W(A)**					
	T2		**R(B)**									
	T3							**R(B)**	**W(B)**			
NODE 3	**T1**									**R(C)**	**W(C)**	
	T2	**R(D)**				**W(D)**						
	T3			**R(C)**								

1. Construct the local waits-for graphs on each node. Is there a local deadlock on any of the nodes? Exhibit the graphs and explain why there are deadlocks or why there are no deadlocks.
2. Construct a global waits-for graph. Is there a global deadlock? Exhibit the graph and explain why there is a global deadlock or why there is no global deadlock.

   ** Recall that the waits-for graph is a directed graph in which nodes represent transactions, and edges represent that an object is held by a transaction, and another transaction is waiting for the object.
3. Is there any scenario starting from the situation above in which T1 is allowed to commit? Explain. If it is possible for T1 to commit, show the steps and messages needed by T1 to commit, assuming a two-phase commit protocol. Make sure to point out when information for a step should be recorded durably in the log of any of the nodes involved in the commit protocol.

Programming Task

In this programming task, you will develop a *snapshot query* abstraction in a simplified precision agriculture 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 7 or 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.

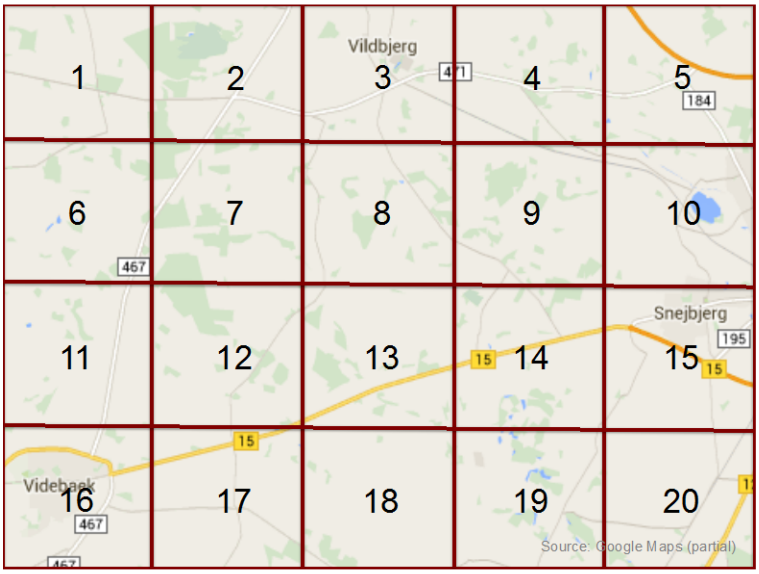
In contrast to a complex code handout, in this exam you will be given a simple interface to adhere to, described in detail below. We expect the implementation that you provide to this interface to adhere to the description given, and to employ architectural elements and concepts you have learned during 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 `SensorAggregator` and `FieldStatus` Abstractions

For concreteness, we focus on an example from precision agriculture: A smart farm collects data from its fields by deploying potentially multiple sensors in each field. The sensors provide measurements of a number of variables, in particular temperature and humidity. These measurements are forwarded to access points and then to an aggregator, where measurements are averaged within fixed periods, say of five seconds. The updated averages are reflected in the field status, where a new consistent image of the state of the fields is created. This consistent image is queried by a number of clients in the farm, including for example tractors working the fields and farmers in the control room. We model the aggregation of sensor measurements through the `SensorAggregator` interface; the field update and query functionality are given through the `FieldStatus` interface.

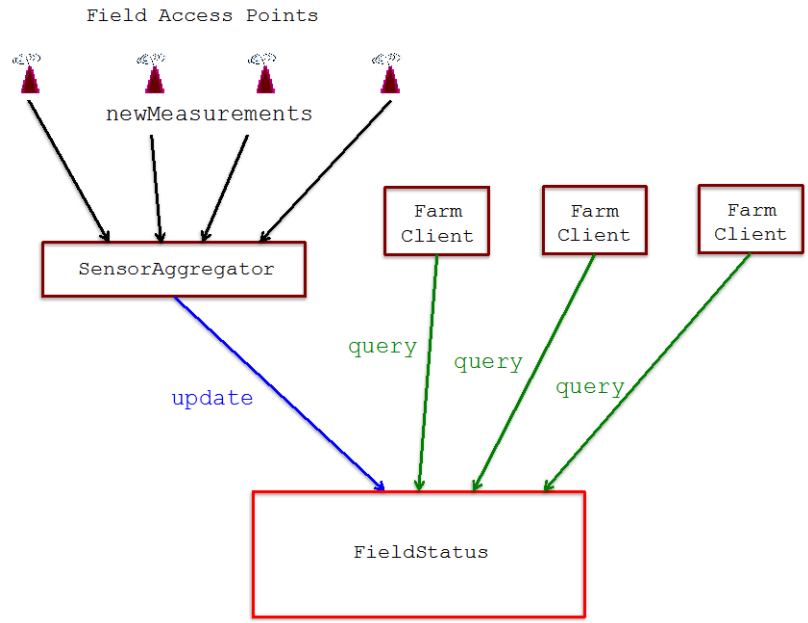
In more detail, the `SensorAggregator` interface is implemented by a server-side service that processes batches of measurements forwarded by access points in the farm. The `SensorAggregator` atomically timestamps the measurements and aggregates them both by time period and by field. Time periods are fixed for all measurements – e.g., all measurements within the same tumbling five-second window get aggregated into a single average – and the length of a time period is given as a configuration parameter to the `SensorAggregator`.[1] A field is a unit of space in the farm. Concretely, the cultivated space in the farm is gridded in 2D, and each cell in the grid is called a field. Each field is given a field ID sequentially in the grid as illustrated in the figure below.

[1] In a tumbling window, after all measurements are averaged, the data in the window is completely discarded, and a fresh window for the next time period is opened.



The `SensorAggregator` passes aggregated measurement events to a service implementing the `FieldStatus` interface. As pointed out above, these aggregated events are coalesced into batches according to time period. For example, if the time period is configured to five seconds, one aggregated event for each field that had updates is reported for that tumbling five-second window in a single batch. The `FieldStatus` service atomically applies the batches of aggregated events, and offers a query capability on the current status of fields for clients. The query functionality is also atomic, and thus queries offer always a snapshot of the fields in time.

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



Sensors are statically assigned to an access point in each field. So every region of a field is associated to a nearby access point. As a consequence, adjacent fields may report measurements to the same access point. In other words, an access point may report measurements originating at different fields.

The field access point relays batches of measurements of the form `<sensorID, fieldID, currentTemperature, currentHumidity>` to the `SensorAggregator`. The `SensorAggregator` assigns a time period to the measurements received according to its current clock value and the starting time of the current time period, and aggregates the measurements into events of the form `<fieldID, avgTemperature, avgHumidity>`. The `SensorAggregator` then updates the `FieldStatus` service with batches of these events. The `FieldStatus` service also offers a query function to farm clients. *All communication between components is realized via RPCs*.

The **`SensorAggregator`** service exposes as its API the following *operation*:

1) `newMeasurements(List<Measurement> measurements)`: This operation takes a batch of sensor measurements of the form `<sensorID, fieldID, currentTemperature, currentHumidity>` and *atomically* updates the average temperature and humidity per field kept by the `SensorAggregator` for the current time period. Since time is strictly increasing in the `SensorAggregator`'s clock, and we expect measurements to be fed continuously to the `SensorAggregator`, this operation also checks for the end of a time period before processing the current batch of measurements. If a time period is concluded, then a batch of events of the form `<fieldID, avgTemperature, avgHumidity>` is created and a corresponding call to the `FieldStatus`'s `update` operation is performed. The batch should only include events for fields that produced at least one measurement in the time period. This simple strategy obviates the keeping of timers inside the `SensorAggregator`.
   An implementation that guarantees atomicity with a single global lock is acceptable for this function; however, your solution will be valued higher if your implementation includes a *fast path* that allows for concurrent execution of updates to batches including disjoint sets of fields *within* a given time period. In other words, we expect that time period boundaries be infrequent, and thus most of the processing be performed in the fast path, making such an optimization very effective in practice.
   This operation throws appropriate exceptions if any values are out of bounds. SensorIDs are expected to be strictly positive, while fieldIDs range from 1 to N, where N is given as a configuration parameter to the system. It is expected that temperatures will range between -50 and 50 degrees Celsius, humidity from 0% to 100%, and that all values are integer.

The **`FieldStatus`** service simply keeps a logical mapping of fieldID to temperature and humidity, and exposes as its API the following *operations*:

1) `update(timePeriod, List<Event> events)`: This operation takes a batch of field events of the form `<fieldID, avgTemperature, avgHumidity>` and *atomically* updates the temperature and humidity for the given fields in the `FieldStatus` service. In order to offer *durability* of updates, this operation also writes the batch of events to a log.
   Durability is achieved at the `FieldStatus` component 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. You will need, however, to design a format for your log, ensure*

correctness when multiple threads cause writes to the log concurrently, and make log writes durable as described above.
The operation throws appropriate exceptions if any of the events are malformed (i.e., fieldIDs, time periods, temperatures, or humidity out of bounds), or on any error condition that precludes durable recording of the request. In addition, the method additionally checks that time periods are the same for all events, and are strictly increasing across different calls to the update method.

2) `List<FieldState> query(List<Integer> fieldIDs)`: This operation takes a list of fieldIDs and returns a list of tuples of the form `<fieldID, temperature, humidity>` reflecting the latest status of the fields requested. As with the update operation, this operation is also *atomic* with respect to other operations in the `FieldStatus` service. It is a requirement that the `FieldStatus` service can process multiple query operations concurrently such that query operations do not block each other. This operation throws appropriate exceptions if any fieldID is invalid.

Your implementation should be focused on the `SensorAggregator` and `FieldStatus` services. For testing and experimentation purposes, you will need to create programs that simulate the calls made to these services by field access points and by farm clients. However, *you do not need to formally structure nor model interfaces for field access points and farm clients in your solution*.

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 the `SensorAggregator` does not disrupt `FieldStatus` queries performed by farm clients;[2] conversely, failure of the `FieldStatus` service does not disrupt calls to `newMeasurements` at the `SensorAggregator`, except for the fact that any `update` calls by the `SensorAggregator` will fail to be executed. However, the `SensorAggregator` must keep running, and ready to send new `update` calls whenever the `FieldStatus` is recovered. Note that updates that were ready during the downtime of the `FieldStatus` service are simply lost.

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 `SensorAggregator` and the `FieldStatus` services is stored in *main memory* at the corresponding components. You can generate an initial data distribution for field temperatures and humidity 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.

[2] You may assume that any measurements that should be reported to the `SensorAggregator` while it is down are simply discarded.

You may assume in your implementation that the initial data on fields as well as any configuration information is loaded once each component is initialized at its constructor. The configuration includes the number of fields N, and you may assume a dense numbering of fields from 1 to N as in discussed earlier. 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* `SensorAggregator` *and* `FieldStatus`*, including the following aspects in your answer:*

- *What RPC semantics are implemented between field access points and* `SensorAggregator`*? What RPC semantics are implemented between* `SensorAggregator` *and* `FieldStatus`*? Explain.*
- *How did you handle failures of the* `FieldStatus` *service at the* `SensorAggregator`*? How did you handle failures of the* `SensorAggregator` *at the* `FieldStatus` *service? Explain why the two mechanisms you chose are enough to contain failure propagation.*

(1 paragraph for overall code organization; 1 paragraph for first point; 2 paragraphs for second point)

Atomicity and Fault-Tolerance

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

Question 2 (LG4, LG5): *In the implementation of the* `SensorAggregator` *service, you had a choice to employ a solution with a single global lock. However, your solution will be given higher value if it includes a fast path for concurrency of operations within a given time period. In this question, we ask you to:*

- *State whether you have implemented the fast path or not. If so, answer the following points within this question.*
- *Which method did you use for ensuring serializability of operations given your fast path implementation at the* `SensorAggregator` *service (e.g., locking, or optimistic approach)? 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 how your method ensures that operations acting on batches with disjoint sets of fields can execute concurrently with only minimal blocking within a time period.*
- *Argue for whether or not you need to consider the issue of predicate locking in your implementation, and explain why.*

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 the requirements set for the fast path.
(One simple statement for first point; 4 paragraphs, one for each point, if fast path was implemented)

Question 3 (LG3, LG4, LG5): *The* `FieldStatus` *service executes operations originated at the* `SensorAggregator` *and at multiple farm 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 the `FieldStatus` 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)

Question 4 (LG4, LG5): *In your implementation, you were not required to implement a procedure for log-based recovery of the `FieldStatus` service. In this question, we ask you to document what should be done by such a recovery procedure and why.*

- *Considering the semantics of the operations, and that all data is resident in main memory, should the log contain information for redo of update events, undo of update events, or both? Explain why.*
- *How would the `FieldStatus` service utilize the log you have implemented to recover its state up to the latest update operation fully processed? Explain why this recovery procedure does not miss any update acknowledged to the `SensorAggregator`, and why this recovery procedure correctly applies the updates to the `FieldStatus` data structures.*
- *Given the mechanism you have employed at the `SensorAggregator` to tolerate failures of the `FieldStatus` service, explain if any interaction between the two components is needed to conclude the recovery procedure, and why or why not.*

(3 paragraphs; 1 paragraph for each point)

Testing

Question 5 (LG4, LG5): *Describe your high-level strategy to test your implementation. In particular, mention the following aspects in your answer:*

- *How you tested the time-based aggregation of events in the `SensorAggregator`.*
- *How you tested that operations were indeed atomic at the `FieldStatus` service.*
- *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 6 (LG6): *Design, describe the setup for, and execute an experiment that shows how well your implementation of the `FieldStatus` service behaves as concurrency is increased. You will need to argue for a basic workload mix involving calls from the `SensorAggregator` as well as farm clients. Given a mix, you should report how the query throughput of the service scales as more farm 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* `FieldStatus` *service on the y-axis and the numbers of farm 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 farm 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)