Principles of computer system design exam

Mariuta Nicolae 1/22/2015

Contents

Question 1: Data Processing

For being able to make the algorithms to answer to the required query I have to make aggregation in the *friends(uid1,uid2)* and count the number of friends that each *uid1* has and in the next stage merge the results for counting with data in table *users(uid,network).* The aggregation will be done using both: sort-based and has-based algorithms.

Sort-based

Before deciding of how to implement the algorithm I have made the assumption that the relationships from table *friends(uid1,uid2)* are unidirectional, meaning that *uid2* is friend of *uid1* but do not know if *uid2* also considers that *uid1* is his friend unless we also have the entry *friends(uid2,uid1)* in the table. By making this assumption I make easier to count the number of friends for each person by only counting the number of apparitions for his uid in the first column in the table.

For this strategy, in the first phase, I have decided to use a mixture between Sort-Merge Join and the aggregation by using the Multi-Way External Merge Sort.

In the first stage I sort the list of *users* by the attribute *uid* by applying the usual merge-sort algorithm. After this the list of friends is sorted by using the Multi-Way External Merge Sort, meaning that blocks of the memory size are read from *friends* and inside of each block the entries are sorted by *uid1* and also the entries with same *uid1* are counted. Since uid2 does not matter anymore, the pairs (*uid1*,*uid2*) are turned into pairs (*uid1*,*count*).

In the next stage the sorted results are merged by attribute *uid* and the result for query are obtained.

IO cost of the algorithm:

The costs for the algorithm are:

$$Sort_U + Sort_F + (Pages_U + Pages_U)$$

This happens because after sorting F the size of the list will be same as U and perhaps even smaller if there are users that do not have friends at all. So in the first phase we have the costs for merge sorting list U and list F and at these are added the costs for merging the array U with the sorted array F which has same size as U after the merging with counting is done.

Hash-based

For the hash-based strategy I have thought about a variant of the Grace Hash Join with some changes.

In the first phase both the *users* and *friends* table are partitioned and for these chunks with size square root of largest table divided by two are loaded into memory and they are split into partitions and the hash key that is used for partitions is the *uid*. After a block is partitioned into buckets and they are loaded into the second disk, another block with the same size is loaded into memory for speeding up the process.

The elements in buckets are grouped so in each bucket with elements from *friends* table with same uid or some specific ids (in case it is needed to don't exceed a specific number of partitions) are loaded. And the buckets with *users* are created so each bucket contains *ids* from a specific interval (e.g. interval 1-10, 10-20, 20-30). I also have to take care that each bucket will not exceed the size limit of square root of the largest table divided by 2.

In the second phase each one bucket of *users* and one bucket of *friends* is loaded into memory and the entries from *friends* bucket with the same *key* are counted and their number is added for each *user.*

The main difference from the classic Grace Hash Join algorithm is that I count the number of entries with same key (key is same with uid1) and the output is list of users with their network and number of friends.

IO cost of the algorithm:

The total cost for this algorithm is 3*U + 2*F. This is because both U and F are read from the initial disk, then split into buckets and written on the second disk. After this, U and F are both read from the second disk and after that only users, each with the number of friends are written to the third storage disk because the friends are no longer needed for the result.

Question 2: Distributed Transactions

1. Local wait-for graphs

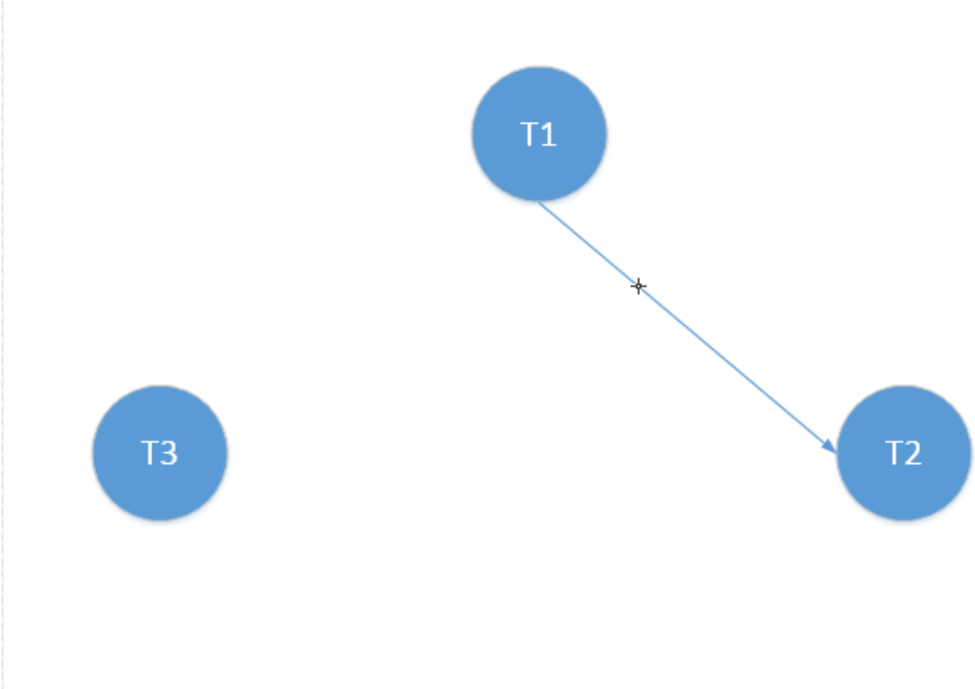

Figure 1 Local wait-for graph for node 1

In node 1 there is precedence between transaction T1 and transaction T2 because T1 reads value from X and T2 writes information in X and to avoid any error then T2 has to wait until T1 finished the reading. There is no deadlock in this node because the precedence graph is acyclic

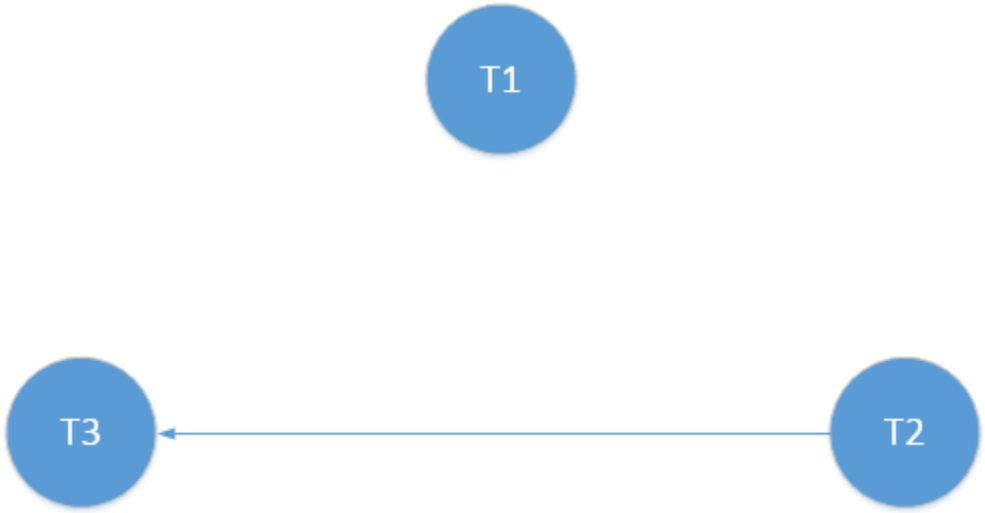

Figure 2 Local wait-for graph for node 2

In this node there is precedence between T2 and T3 because transaction T2 reads value from B and T3 has to write value into B. In this case there is also no deadlock because the graph is acyclic.

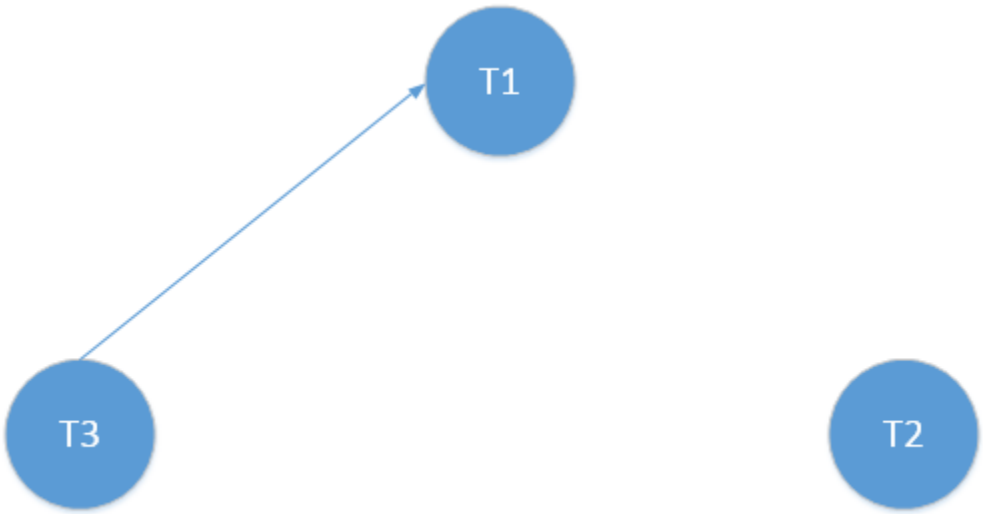

Figure 3 Local wait-for graph for node 3

In node 3 there is precedence between T3 and T1 because T3 reads C and T1 writes into C. Once again, there is no deadlock because graph is acyclic.

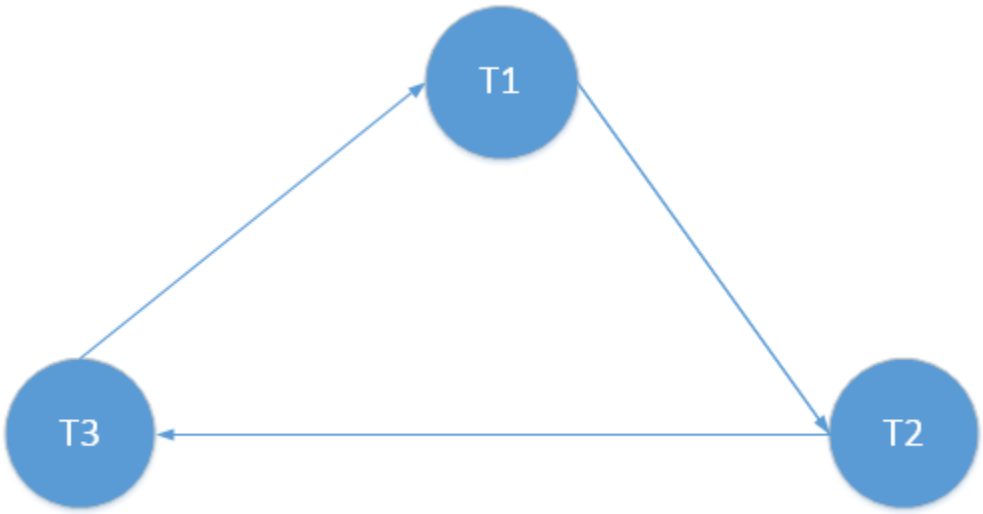

Figure 4 Merge of graphs from Node1, Node2 and Node 3

In case I merge the graphs then it appears to be deadlock since the graph is acyclic.

2. Global wait-for graph

I have done the global wait-for graph with the 3 nodes and variables X, Y, A, B, C, D placed in each node. Outside the nodes I have the transaction T1, T2, T3 and the precedences.

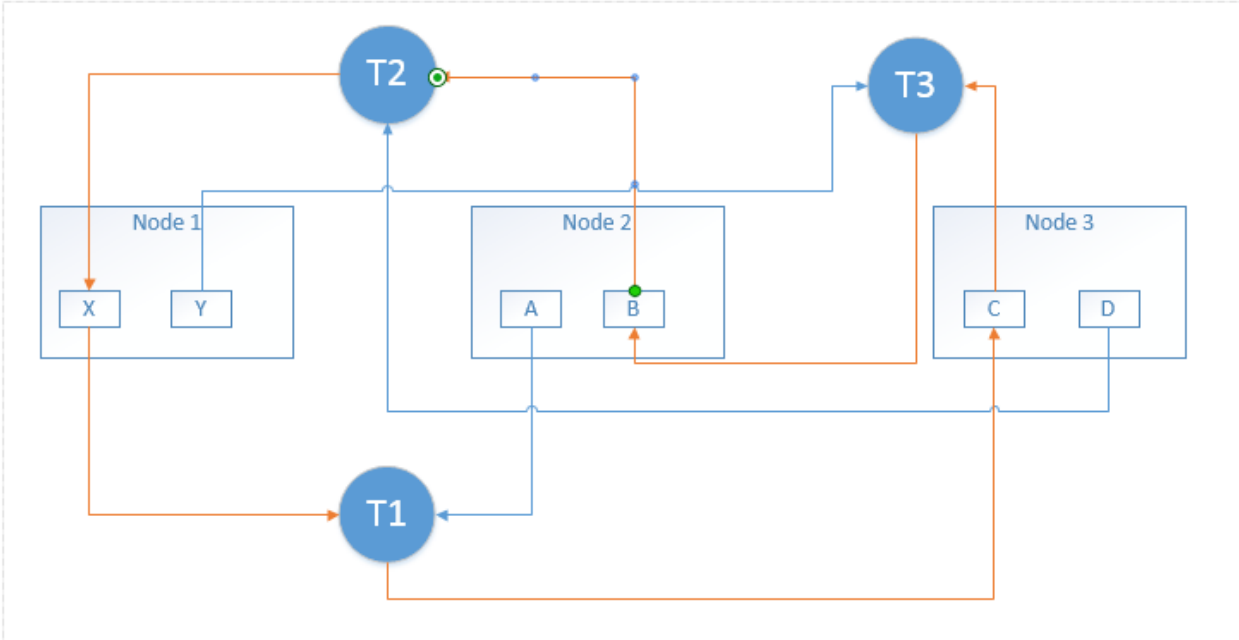

Figure 5 Global wait-for graph

I explain how I have drawn the arrow:

- transaction T1 has to wait for variables X and A which it reads but variable C has to wait for T1 to finish writing
- transaction T2 has to read from D in node 3 and B in node 2 and it variable X has to wait until the transcation finish writing
- transaction T3 has to read to read record C from Node 2 and Y from node 1 and variable B has to wait for T3 to finish.

With yellow orange arrows I have drawn cycle that I have found in graph and this cycle can lead to deadlock. In order to avoid the deadlock I have to remove X, B and C.

3. Is T1 allowed to commit?

T1 is allowed to commit into Node 2 and Node 3 because the other transactions in these nodes do not have to write on A or C while in Node 1 it is not allowed to commit because T2 has to write into X after T1 starts reading.

In order to commit, by considering the two-phase commit protocol T1 has to acquire write lock on A and C and release them after it finish writing to these locations. Information should be recorded into a log after the writing is finished. It should state the previous values written at A and C and the new values that are introduced. If the locking protocol applies shared lock on X it will then happen that T2 would not be allowed to write until T1 finish writing and then T1 could also be allowed to commit in Node 1.

Programming task

Question 1

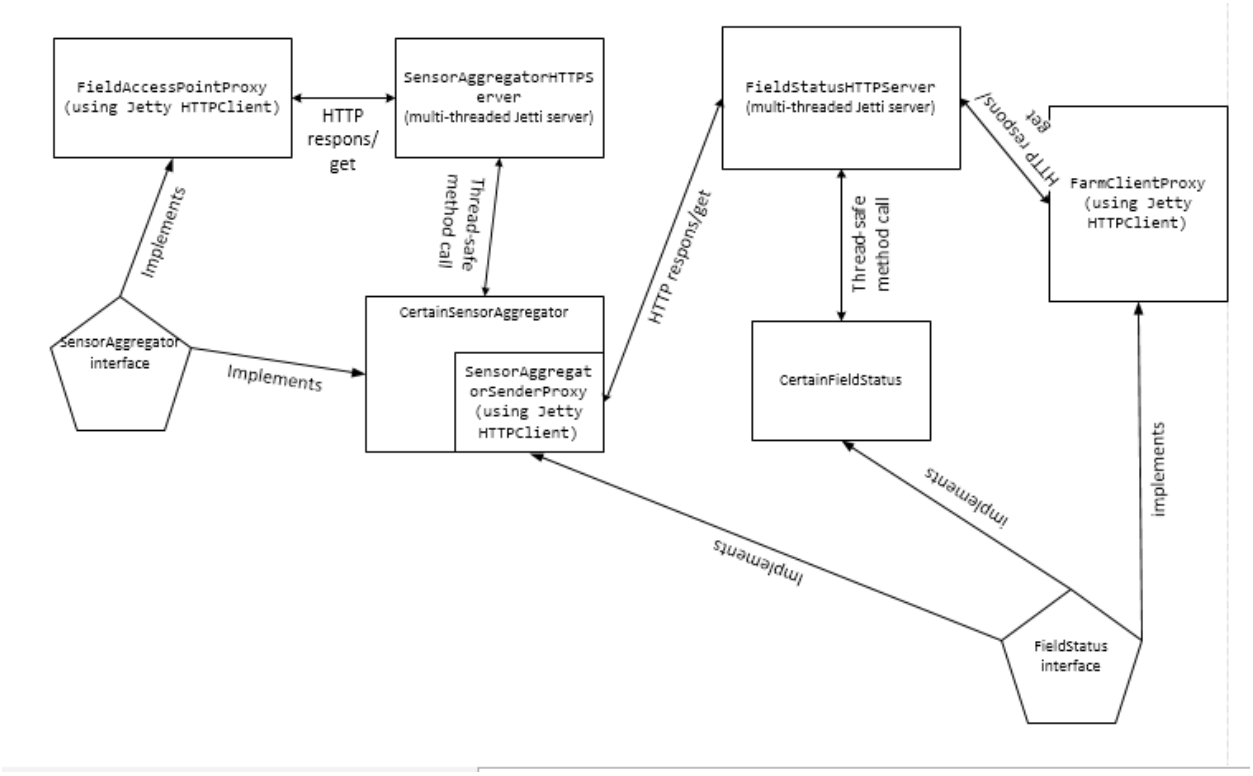

Figure 6 Diagram for acertainfarm

Before describing other aspects, I will state that I have used the programming language Java as required, I used Jetty as RPC mechanism and I have used JUnit for making the tests. For the most part I have used the code from the assignments to create the proxy classes, the servers, the message handlers and also as model for making the tests and check the workload.

As it is visible in the diagram that I have drawn for my implementation of *acertainfarm* an important aspect are the two severs which handle the operations in the farm:

- *SensorAggregatorHTTPServer* which handles the *newMeasurements* operations regarding the data from Field Access Points and ensures that data regarding temperature and humidity of fields are sent periodically to the FieldStatus as *update*.
- *FieldStatusHTTPServer* which has the role to receive the *update* from the SensorAggregator and also answers to the queries from the two clients.

I have chosen to handle operations at farm in two servers mostly because I thought it suits the best to the scheme in the assignment and also ensures atomicity of the application, making sure that the two types of interactions with the farm (measurements from sensors and requests from clients) are handled separately and each of them may continue to run if the other one fails.

On the *SensorAggregatorHTTPServer* I have connected the following components:

- *FieldAccesPointProxy* which is the client that implements the *SensorAggregator* Interface and has the role to send lists of *Measurements* to the *SensorAggregatorHTTPServer*. This proxy is basically the communication tool used by Field Access Points that send measurements to the SensorAggregator.
- *CertainSensorAggregator* is the class that implements the *SensorAggregator* Interface and is attached to the server and handles the measurements received from the Field Access Points. In this class I keep a map of type <*Integer,FieldMeasurements*> where the integer is the *FieldID* of each field that sent measurements and *FieldMeasurements* is a class that I have created which contains a list with the last measurements of temperatures and humidities and also has methods to calculate the average of these parameters. Like this, I store a map with all measurements, if the SensorAggregator receives measurements from a new field they are added to map, otherwise new measurements are added to the corresponding f*ieldMeasurements.* The method also has a parameter that holds the date of the first measurements and when new measurements arrive, it checks the difference between the current time and the one that is saved. If the desired timer passed, then measurements are sent to the FieldStatus server and the map is cleared and the date is updated.
- *SensorAggregatorSenderProxy* is a class implementing the *FieldStatus* interface and is kept internally by the *CertainSensorAggregator* in order to send updates to the FieldStatus once the timer has passed. This proxy has only the method *update()* implemented but there was no reason to also implement the query because this proxy only has the role of sending updates from SensorAggregator to FieldStatus and there is no reason that the SensorAggregator has to request information from FieldStatus. I could simply send to the *FieldStatusHTTPServer* the

update information but I have chosen to do a proxy which is kept internally by the SensorAggregator mostly for the purpose to make the application more modular and easier to test only the functionality of FieldStatus. I have thought to split the FieldStatus interface and only make an interface with update method but I have decided to follow more accurately the structure that was proposed in the handout code.

On the *FieldStatusHTTPServer* I have created the following components:

- *FarmClientProxy* which is the client implementing the *FieldStatus* interface and has the purpose to send *query* to the FieldStatus and also receive results for the queries. This class also has only one of the two methods from the interface implemented (the query()) method since the only operation that the client can do is to query .
- *CertainFieldStatus* is the internal class of FieldStatus that also implements *FieldStatus* and it handles the methods *update()* and *query()*. This class also holds a mapping between the FieldIds and values of temperature and humidity that have to be sent to clients that request information about the fields. This class also takes care that all the updates received from the SensorAggregator are written to the log; into a file.

The RPC semantics that are used in the implementation is Exactly-once in both communications: between field access points and *SensorAggregator* and also between *SensorAggregator* and *FieldStatus* because each of the parts that are sending messages is waiting for a confirmation from the server and in case of error it is simply shown as exception but there is nothing done by the proxys(clients) or by the server in order to try to send the message again. If error happens it is simply printed but the system keeps working just as if nothing has happened. This type of semantics if fell fit for the application that I created for several reasons: in case of that the communication between field access points and *SensorAggregator* fails there is no reason that the access point to try to send the message again since it will however send another measurement and the *CertainSensorAggregator* only counts the measurements that are successfully received and average them. If one measurement in the interval is missing, nothing bad can happen. And the same between *SensorAggregator* and *FieldStatus*: in case this communication has problems, the FieldStatus simply keeps working by using the last information that was received in the answers to the queries. And the operations themselves which are done in the application are not that critical to errors at sending one message like a bank for example. And by doing like this it is also ensured that any of the components still keep working if another fails.

Regarding the failures of both services I have implemented the fail-stop model. A timeout is set in case we have an error, so the client will abort in order for other clients not to be blocked. These two mechanisms are enough to contain failure propagation because, as I described previously, in case of failure I ensure that both components are still working and there is not that a critical problem with the FieldStatus will runs by offering to clients older measurements of temperature and humidity. The problem with this strategy is that a slow-running system will appear to be stopped but it is easier to implement.

Question 2

I have tried an implementation of fast path by assigning locks in the *newMeasurements()* method in a way to leave writelocks only on parts of the code where writing are actually being done. I have used the *ReetrantReadWriteLock* class from Java library which implements reentrancy.

In order to make the fast path work I have used locking. I have mostly tried to put write lock on parts of the code where new measurements are written to the map that holds them. My method *newMeasurements()* from *CertainSensorAggregator* has three parts: in the first part of the code It checks if timer has expired and updates have to be sent. In this part of the code I have put *readLock* but I have also inserted a write lock at part of the code where the date that is saved in the class is changed and also in the part of the class where it is deleted the data that was stored in the map. In the second part of the code the measurements are checked, in a loop, for errors that should cause *AttributeOutOfBoundsException* and in this part of the code I have placed a *readLock* since data is not modified there. In the third part of the function there is another for loop that iterates through all measurements that are received and the map where they are stored is updated. I have placed *writeLock* on this loop because this is part of the code where writings are actually done and like this I have ensured that if a Field Access Point can send measurements from disjoint fields can write both at the same time.

I think that my method is correct since it implements a variant of String Two Phase Locking because it includes places of locks (read and write) as they are needed during the execution of transaction. I have placed *readLock* on part of the code that checks how much time elapsed and on part of the code where the received measurements are checked for errors but I have placed write lock on part of the code where the date and map of measurements are stored. By this I ensured greater concurrency because there is not a general single write lock for all function which would have also locked parts of the code where writings are not done.

I think that my method allows that batches with disjoint set of fields can execute concurrently because at least what I tried is to ensure that each new measurement only locks the slots in the map of FieldMeasurements that it has to modify. If another sensor is sending measurements for other fields then it should be allow modifying the map. I have also considered using the *ConcurrentHashMap* from Java but that one seems to implement itself the functionality that I wanted to make and thought it was not correct to use it.

Predicate locking is actually what I have considered for my implementation because that is what I wanted to do: lock only the indexes in the data that are modified. Perhaps an another method to implement the fast path could be to store another list with fieldIds that are currently modified and a new call of this method should check that list and if the field Ids that it wants to modify are not there then the writing should be allowed.

Question 3

In order to ensure serializability I have used the locking mechanism. At the *SensorAggregator* I have implemented the fast path with a variant of Strict 2PL as I described previously. At the *FieldStatus* I have implemented a simpler lock mechanism: write lock over the entire method of *update()* which has to write data and read lock on *query()* .

I described previously why the locking mechanism at *newMeasurements()* is ensured but at the *FieldStatus* I have done things a lot simpler by implementing a variant of Conservative Strict Two Phase Locking by using exclusive locks on *update()* where data is modified and share lock on *query()* where data is read. I think this approach is correct because I ensure that there are not concurrent readings and writings at the both time. Concurrent reading are allowed because these cannot lead to errors but items are locked if there is any writing at the data.

The performance for this approach if very good especially because there it only one *SensorAggregator* that sends updates at certain periods of time which are not very short, since it is about temperature and humidity that are quite slow evolving parameters and if there is an update, for example, at every 5 seconds as I tested, then the time when the data gets exclusive locks is very short, in comparison to client queries that happen to be multiple at a same time. Adding to these the fast path that I have created between field access points and *SensorAggregator* then the performance of design of application ensures the necessities of the farm. In case of optimistic approach the performance would have been lower because the clients that want to read while an update is made then they their transaction would not be committed and there it could be delay in answer for the query. And there it was no need to consider queuing of operations since there is only one client that may send write requests (updates from the SensorAggregator).

I have only considered the issue of predicate locking at the fast path between Field Access Points and SensorAggregator where I had to ensure that there are locks only on items that are modified.

Question 4

I have chosen to write data to log like this:

$$< Date, new\ data, old\ data >$$

I have chosen to keep it simpler and considered it is enough to have stored the time at which updates have been done together with the new data that was received and the old values of fields that were stored. Of course, the log writings could also include an ID of update and pageID but I considered these are enough since the update data is received from a single *SensorAggregator* and the frequency of

updates is not very high. The information that I have written to log should be enough for doing both: REDO and UNDO because I have chosen to store both old data and the new updates so the information required for the recovery is found in the file. I have chosen now to also use checkpoints because I store information from each update inside of a line of text.

In case of recovery the *FieldStatus* server should just read the last line in the log, store the old information in the map that holds the status of fields and apply the last update. With simple parsing of the text that is written to the log it should be no problem to obtain all data that is needed for the system to resume working. The recovery procedure should not miss any useful information because all the old information and the updates are stored and the implantation of the recovery procedure can choose how to make the undo and from which moment to make the recovery. The only problem could be that in case of long timer at *SnesorAggregator* for how often to send updates it may happen that clients would receive too old information about the fields in the time gap between the recovery and the next update that is received.

In case of recovery there should not be needed to have any interaction between the *SensorAggregator* and *FieldStatus* servers because the *FieldStatus* will resume operations by sending to clients values from the last updates before failure and when it receives updates from SensorAggregator then the Field Status will just update the values that are stored and use them for further queries. It may be a problem if the time gap between two updates sent from SensorAggregator is set to be very long (for example, a day) and in that situation it may be better if the *FieldStatus* would request from the *SensorAggregator* a new update with the values of temperature and humidity from the fields. But there it may be a small issue with that the *SensorAggregator* may not hold enough measurements at that time from the field sensors and will not have enough information to send a new update. In that situation it may be good if the *SensorAggregator* also stores the average temperature and humidity values in a log and send the last value in case the *FieldStatus* requires new update after a failure.

The situation may become difficult in case a new SensorAggregator is added to scheme. In that situation it will be needed to write in the log also the ID of the aggregator that is sending the updates.

Question 5

For testing the implementation I have done two types of tests. In the first type of tests I have *FieldStatusTest* and *SensorAggregatorTest* and for these I have created basic tests to check if the functions newMeasurements, *update* and *query* actually work in the classes *CertainSensorAggregator* and *CertainFieldStatus*. I have tested with both: sending correct values and also wrong values that should trigger *AttributeOutOfBoundsException* and they actually worked. I could probably do more of these tests to check the error conditions but I have chosen to focus more on how the components actually work when the servers are running.

In the second part of testing I have created the class *ConcurrentFarmTests* where I have checked how the servers work at communication to each other and separately. For this I have created 3 thread classes that run the functionality of the 3 proxies in the system separately and send requests to the servers: *FarmClientEmulator*, *SensorSenderEmulator* and *FieldSensorEmulator*. using these I have written two tests:

- *testFieldStatus()* there where I have only checked the atomicity of *FieldStatus* by starting the *SensorAggregatorSenderProxy* proxy that sends updates to *FieldStatus* and query requests from the *FarmClientProxy* and I have the checked if the answers received by clients are correct. In this test I have just used the *run()* method of the threads because I did not want concurrency on the operations. I wanted to make sure that the update sender has finished its job before the client sends a query. Like this I simulated somehow what happens if the SensorAggregator is failing: the *FieldStatus* continues to send information with the last updates to the clients. I have also checked what happens if I run the threats using method start() so they work concurrently. In that situation the *SensorAggregatorSenderProxy* does not manage to send the update before the client sends the request, but in this situation it just happens that the client receives an empty list if there is no data in the server of *FieldStatus*. I should have probably done a separate test for that situation but I did not have enough time to make more similar tests. With this test I have proven that the operations of *FieldStatus* are atomic and it can still run if it happens that the other server in the system has a failure.
- In order to check how the time-based aggregation of events in *SensorAggregator* are working I have created the test *testFullSystem* where I have all the components in the system working: sensor that is sending a number of updates at every 500 milliseconds and the client that is sending a query. I have checked the results received by the client to ensure that they are correct. Once again, I have waited until the sensor finish sending measurements mostly to ensure that there are updates that client is receiving but this time I have used the *join*() method of *Thread*. And the aggregation actually works properly: If I make the sensor thread to send 15 updates (that is 15*500=7500milliseconds) then an update is sent to the *FieldStatus* server and the client is receiving the expected results. But if I choose to only send 5 measurements(2500milliseconds) then the client receives an empty list because the *SensorAggregator* does not send any update.

By doing these tests I have shown that the core functionality of *CertainSensorAggregator* and *CertainFieldStatus* is working properly and also shown that the system works even if *SensorAggregator* is failing. Perhaps I should have done more tests regarding how the updates are sent by the aggregator by this should have implied to make more public functions in *CertainSensorAggregator* and I have chosen not to change the structure that was proposed by the interfaces and I also did not have enough time to make more tests.

Question 6

For the workload design I have chosen to use an approach very similar to the one that was used at *acertainbookstore* in the assignments:

- a class *FarmSetGenerator* with the purpose to generate sets of lists for the three operations that are done in the Farm: one function for generation a list of random *Measurements*, one function to generate a random list of Integers corresponding to the FieldIds and one function to generate a random list of *Event* for the updates. I have chosen that in my workload setup to possibly also try to only test the workload on the *FieldStatus* alone to see how it works atomically but because I did not have enough time, I have actually only tried the workload on the full system with both servers running so there is actually no call for the method to generate the random list of Events that are sent by the *SensorAggregatorSenderProxy* to the *FieldStatus* server. I have also considered irrelevant to see how the FieldStatus reacts where there are also a lot of updates sent by SensorAggregator since temperature and humidity are some slow varying parameters and I expect that in a real application like this, the updates sent by aggregator do not happen very often(even the 5 seconds timer that I have set for this is quite short for such system). All the numbers that are generated are created so they fit into the limits allowed for temperature, humidity and the number of fields that are available.
- a *WorkloadFieldSensor* to simulate the functionality of field sensors which is a Thread class and I have created it to be able to start a lot of measurements to the SensorAggregator server which is also sending updates to FieldStatus and at the same time clients send requests to the FieldStatus.
- class WorkloadConfiguration that holds parameters used in the workload setup: number of measurements that have to be sent by *WorkloadFieldSensor*(I set this number high because I want to make sure that *SensorAggregator* works continuously during the), number of field sensor which actually defines the number of *WorkloadFieldSensor* threads that are started during the workload, number of updates (this was considered for the case I test *FieldStatus* server alone as described previously), number of queries that are to be made by clients and the number of fields about which information is requested and finally, the number of runs and number of actual runs. There are also setters and getters for these parameters
- The class *WorkerRunResult* that keeps data about the results of the workload: elapsed time, number of successful runs and number of actual runs.
- The class *Worker* that starts the threads of *WorkloadFieldSensor* and runs the defined number of queries that are made by the server.
- The class *CertainWorkload* where I instantiated the proxies for communication, started the *Worker* and collected the results in the *reportMetric* method; in the reportMetric method I have also calculated the goodput, the throughput and the latency

The system on which I have runned the workload test is the following laptop:

System Information

Current Date/Time:	Thursday, January 22, 2015, 1:08:07 PM
Computer Name:	NICOLAE-PC
Operating System:	Windows 7 Ultimate 64-bit (6.1, Build 7600)
Language:	English (Regional Setting: English)
System Manufacturer:	TOSHIBA
System Model:	SATELLITE L850-13P
BIOS:	InsydeH2O Version 03.72.011.30
Processor:	Intel(R) Core(TM) i5-2450M CPU @ 2.50GHz (4 CPUs), ~2.5GHz
Memory:	4096MB RAM
Page file:	4062MB used, 4047MB available
DirectX Version:	DirectX 11

Figure 7 System on which experiments were runned

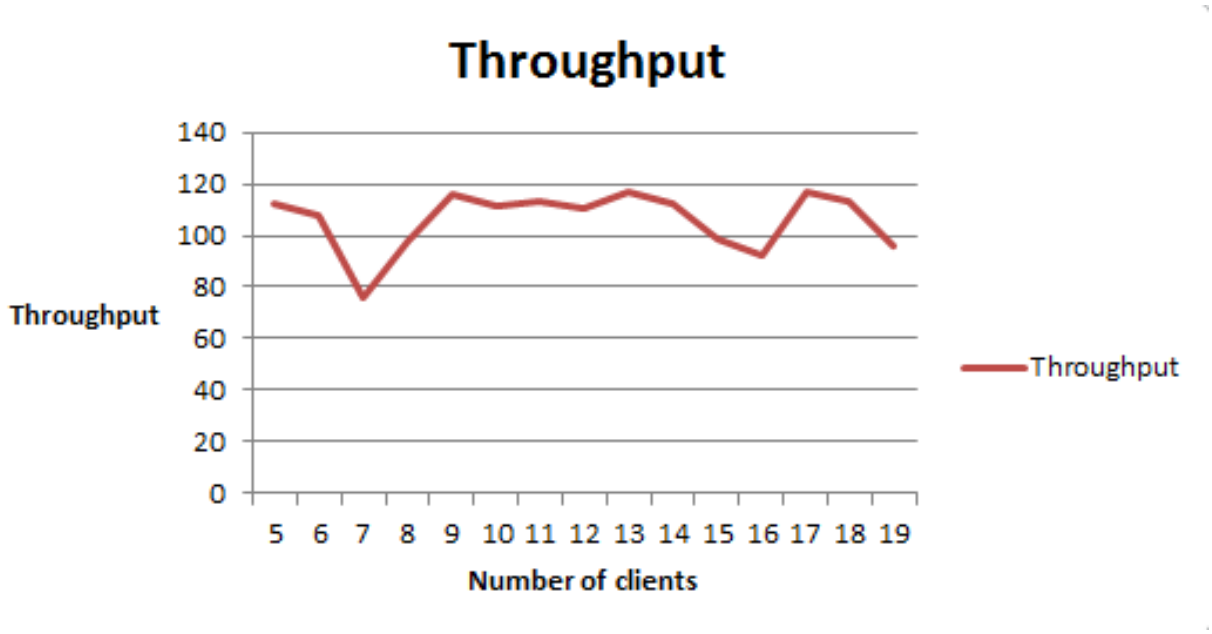

Figure 8 Throughput of the designed workload test system

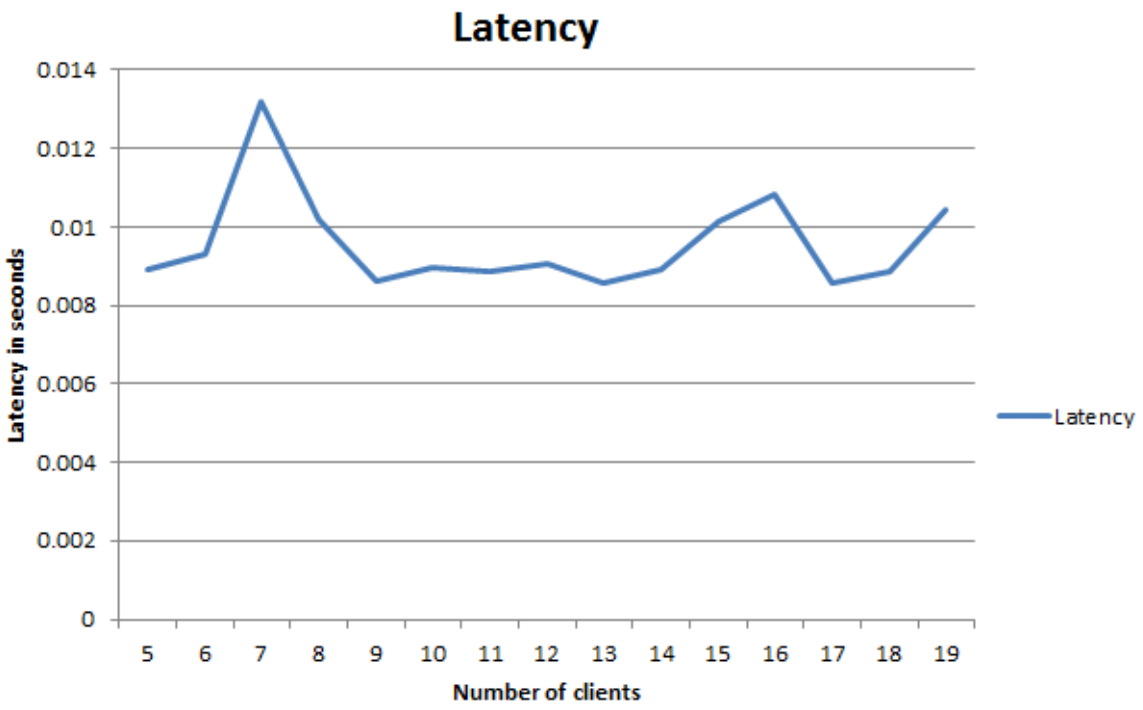

Figure 9 Graphic showing the evolution of latency according to number of client requests

There it seems to be a strong increase of throughput between 9 and 10 clients and it is quite constant until it reaches 14 client requests done at a time. It is the expected bell shape of the graph with one point at which the throughput has a strong growth and after that it decreases. 10 is not a large number of clients but perhaps values are different on a system with better hardware. I have also plotted the evolution of latency that is quite high for a small number of clients and then it has a constant decrease. This happens that because the system has a higher utilization of resources when it has to deal with a larger number of clients. Because I have ensured serializability by using the lock system then it happens that the wait time for response to a large number of clients to be quite constant.

One thing to notice is that even if I included *SystemAggregator* in the system that is sending updates to the server, it does not have a huge impact on the performance of *FieldStatus* server because updates are sent every 5 seconds. Perhaps it was a good idea to also use the Proxy sender of updates separately to send a lot more updates to the *FieldStatus* perhaps in the same manner I simulated the sensors from the fields, by using a several threads to do this repeatedly. But I thought it is not necessary for this application which is designed to have only one *SensorAggregator* and as I also stated previously, it does not have to send very frequent updates since it has to take care of measurements for temperature and humidity. However, since I have implemented the fast path for the *SensorAggregator* if I had time I wanted to somehow test the *SensorAggregator* separately with a high number of *FieldAccessPointsProxy* sending measurements continuously to see how it handles concurrent interaction with a large number of Field Access Points.

Another thing to mention is that I have run the workload in both: using *synchronized* for the methods *newMeasurements*, *update* and *query* and after introducing the locking protocols that I have described and the throughput was a little faster but I have not made a graphic for that one also.

Conclusions

I have made an implementation that grants atomicity of the both servers in the system: both *SensorAggregator* and *FieldStatus* can work independently and continue function if the other component fails. The modularity is ensured because conceptually, by using the proxies, client and services may work in different computers. The communication is restricted to messages only by having client request/service response schema implemented.

I have ensure serializability by introducing read/write locks by implemented variants of Two Phase Protocol which I have proven why I thought they work very well in this application.

The communication between the two servers is assured by the sender proxy that is located in the *SensorAggregator* and by having this implementation, that proxy's function could also be called from within tests and I have not done it actually but it could have also been called from the workload to test the performance of *FieldStatus* alone.

I have done tests to check how different functions in the design work separately and also together as whole system and have also checked the scalability by creating workload environment.

I think that my implementation fulfills very well the requirements of the application that was proposed and can be easily extended by added more *SensorAggregators* and other elements in the system.