Principles of Computer System Design
Exam

Tudor Dragan (xlq880)

January 21, 2015

Question 1: Data Processing

1. Sort-based external memory algorithm

```
//TODO: Insert pseudocode here
```

Figure 1: Sort-based external memory algorithm

For this part of the exam we must implement an algorithm that first applies an aggregated function (in our case the *count* function) on the table *friends*, and then combine the results to write up the table that consists of (uid, networh, nrOfFriends). In order to achieve this, I used a modified sort-merge join algorithm.

I made the following assumptions:

1. We don't have any indices on the tables so the entries are not in sorted order in any of the two tables.
2. The friends table has only uni-directional relations, in the sense that Person1 can be friends with Person2 but it's not requested that Person2 has to be friends with Person1. (Person1 has 1 friend and Person2 has 0 friends.)
3. The table with the biggest number of records has N records.
4. We know that the main memory can hold $\sqrt{N}$ records. If we read B records at a time in memory, the number of runs will be N/B.
5. Number of passes in Phase 2 is P then: $B(B-1)^P = N$

First I build the table with the aggregate count function for the friends table by altering the way that we merge the pages on a pass. I will explain how it is done in the following paragraphs on a concrete example:

1. Every entry in the friends table has this form : (*uid*1, *uid*2). The input is split up in multiple blocks. Let us assume that we have a block that has these following (*uid*1, *uid*2) relations: (3,7) (1,4) (2,3) (2,5) (1,3) (1,2) (2,4) (3,1)
2. In order to calculate the number of friends for each user, I will sort records with the form (*uid*1, $friendCount$). because every record in the *friends* table essentially means 1 friend added, when reading in the first phase, the $friendCount$ is 1. So we will have initially tuples with the form (*uid*, 1) that need to be sorted by *uid*. We assume that the buffer page number is 4. So after the first phase we will have [(3,1) (1,1) (2,1) (2,1)] [(1,1) (1,1) (2,1) (3,1)].
3. The next pass will be [(1,1) (2,2) (3,1)] [(1,2) (2,1) (3,1)], Then we combine the values by adding up the number of friends by comparing the heads of the lists (the heads contain the smallest *uid*) and add up or write to disk the smallest one *uid* with the friend count.
4. Finally we will have [(1,3) (2,3) (3,2)].
5. This will continue until all the pages buffers are empty and have no more data to fill them up with.

After we create the number of friends table that has entries sorted by *uid*, we sort the *user* table We sort the users by uid because we would like to split both sorted inputs into blocks and combine the result to (uid, networh, nrOfFriends) output form. Because we have space in memory for $\sqrt{N}$ records, each buffer block should have a size of $\sqrt{N}/2$. We compare the heads of the lists, because they always have the smallest possible id and merge the values together. We continue filling up the buffers and stop when there are no more entries to compare.

2. Hash-based external memory algorithm

```
//TODO: Insert pseudocode here
```

Figure 2: Hash-based external memory algorithm

The hash-based algorithm is very similar to the *GRACE hash join algorithm* with a slight modification that increments a counter for displaying the number of friends for each user. We make the following assumptions:

1. There are B-1 buckets, with B main memory buffers. These buckets will hold $\sqrt{N}/2$ records, where N is the highest of U and F.
2. The hashtable in the second phase is smaller than B-1 pages.

We first partition both relations U (user table) and F (friend table) via a hash function into B-1 buckets. We read one partition at a time. We iterate through it and place the record in one of the buckets by the hashing the join attribute (in our case the *uid*). Then we can be sure that if tuples of U and F join, they will wind up in corresponding buckets U_i and F_i for some i. After this phase we read a bucket of U from disk and hash it using another function and construct the hash table in memory. Then we load in an buffer for S_i and iterate through it.

When we match the uid, we increment the $numberOfFriends$ counter in the hash table. When the buffer is empty, we write to disk the hash table with $(uid, networth, numberOfFriends)$ and load the next entries in the buffer.

3. I/O costs

Sort-merge algorithm

For the sort-merge algortihm the "expected" cost is:

$$Sort_U + Sort_F + (Pages_U + Pages_F) \tag{1}$$

Hash-based algorithm

Question 2: Distributed Transaction

1. Local wait-for-graphs

Conflict-serializability is defined by equivalence to a serial schedule (no overlapping transactions) with the same transactions, such that both schedules have the same sets of respective chronologically ordered pairs of conflicting operations (same precedence relations of respective conflicting operations).

A schedule is conflict-serializable if and only if it's precedence graph has no cycles. This is a graph of nodes and vertices, where the nodes are the transaction names and the vertices are attribute collisions. The local waits-for graphs on each node are:

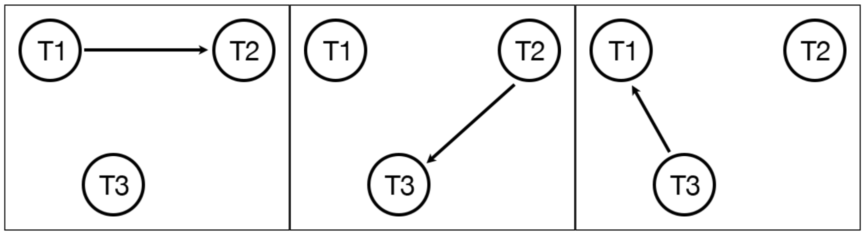

Figure 3: Local wait-for-graph for Node 1, Node 2 and Node 3

As we can see the graphs have no cycles, proving that there are no local deadlocks.

2. Global wait-for-graph

When constructing the global wait-for-graph we set up the pages that are being modified on each machine. If we have a cycle in the global graph, it means that we have conflicts and must abort the transactions. Below we have the global wait-for-graph:

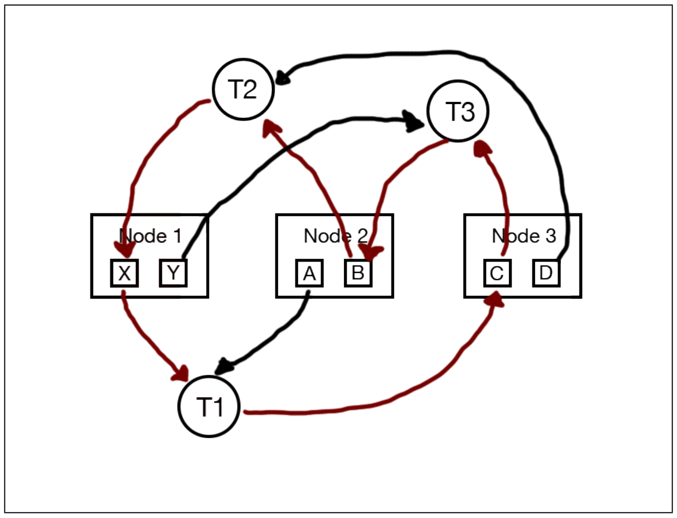

Figure 4: Global wait-for-graph

As we can see we have cycle between $X- > T1- > C- > T3- > B- > T2- > X$. To remove the deadlock within the system we must abort modifications that are being don on X, C and B.

Programming Task

Question 1

For the programming task I emphasized on building a strong, modular service that is based on RPCs. Im the following paragraphs I will discuss on the main components of the *acertain-farm.com* web-service. As we can see in Figure 5. the application is split up into 2 different servers: the Sensor Aggregator Server and the Field Status Server. Both of these servers are implemented by instantiating a multi-threaded server using the Jetty framework[1]. The methods available for both servers are split up into two interfaces: *SensorAggregator* and *Field Status*. These interfaces are implemented by *proxies* that serialize and send out the requests for the server to handle. The *Sensor Aggregator Server* implements the SensorAggregator interface in the FarmSensorAggregator class whose methods are invoked by the handler class SernsorAggregatorHTTPMessageHandler, which handles the request that were received by the Jetty HTTP server class FarmSernsorAggregatorServer class. A similar approach is don on the Field Status Server, but using another interface (the Field Status interface).

The RPCs are done by using a Jetty HTTPClient, which sends out HTTP request for the server to handle. This RPC mechanism is present throught the application and is used for communicating between the FarmFieldAccessPoints and FarmSensorAggregator, FarmMeasurementSender and FieldStatusServer, and FarmClient andFieldStatusServer. In the case of the Sensor Aggregator, the Field Access Point sends out a xml-serialized HTTP request through

[1]The Jetty 8 library (http://www.eclipse.org/jetty/) is used to provide the HTTP server and the HTTP client libraries.

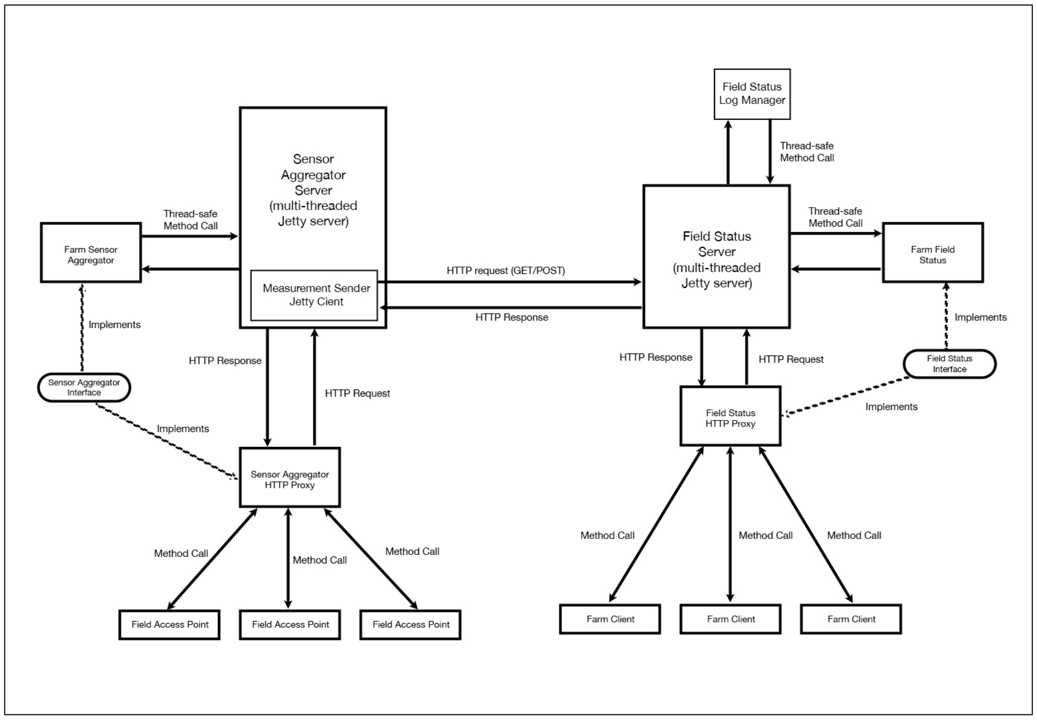

Figure 5: Architecture of the farm's monitoring application

the FarmSensor AggregatorHTTPProxy class. The request is sent and handled by the server. When the server receives the requests, id deserializes it in the handler and passes it to the FarmSensorAggregator instance through a thread-safe method call. Then the request is processed by the FarmSensorAggregator class. between the Sensor Aggregator Server and Field Status Server, the communication is done directly by using an HTTP Client instantiated on the server (in our case, it is incapsulated in the FarmSensorAggregatorSender) that sends out POST Requests directly to the Field Status Server (the request doesn't pass through the HTTP Proxy).

Because I use two different servers, each one is independent from another. In a more concrete case, if the Field Status Server is not responding the Sensor Aggregator Server continues polling and processing Field Access Points Measurements, but the requests that are being sent to the Field Status Server will respond with a TIMEOUT by the FarmSensorAggregatorSender class.

On other side, in the case of the Sensor Aggregator Server failure, the Field Status Server can continue handling requests from Farm Clients with the last known measurements that it received from the other server. As we can see both failures are contained within each service and one does not influence the behavior of the other. The Field Status Server continuously listens for further updates from the Sensor Aggregator Server.

Question 2

For the implementation of the locking protocol used in the FarmSensorAggregator class I used the *ReentrantReadWriteLock* class available in the *java.util.concurrent.locks* package. A ReentrantLock is owned by the thread last successfully locking, but not yet unlocking it. A thread invoking lock will return, successfully acquiring the lock, when the lock is not owned by another thread. The method will return immediately if the current thread already owns the lock. This

Lock type	**read-lock**	**write-lock**
read-lock		**X**
write-lock	**X**	**X**

Table 1: Lock compatibility table

approach guarantees atomicity within transactions but does not allow a lot of concurrent access to the map that is modified. I implemented the strict two-phase locking (S2PL) by requiring a write lock when modifying the data at the start of the method. The exam question states that it is recommended that we do a fast path solution. First I tried using a Multiple Granularity Lock Protocol because we need to modify only a key-value pair in the map and we do not want to keep a lock on the entire HashMap, so that other threads can modify different pairs concurrently. I tried using the ConcurrentHashMap[2] class that is part of the java.util.concurrent package, but eventually I gave up on the implementation and went with the safe approach. I had problems when sending the map to the Field Status Server through the Sender class so I decided it was best to stick to the solution that works, rather than going with one that didn't.

Question 3

[2]A hash table supporting full concurrency of retrievals and adjustable expected concurrency for updates. This class obeys the same functional specification as Hashtable, and includes versions of methods corresponding to each method of Hashtable. However, even though all operations are thread-safe, retrieval operations do not entail locking, and there is not any support for locking the entire table in a way that prevents all access. This class is fully interoperable with Hashtable in programs that rely on its thread safety but not on its synchronization details.