PCSD
Final Exam

Arni Asgeirsson - lwf986

January 21, 2014

Contents

1 Introduction

The following is a report written as an answer to the final exam in PCSD DIKU 2014 January.

I assume that the reader has knowledge and understanding of the assignment text and the same principles and concepts as are expected of us.

In the first part of the report I will try and answer the posted theory exercises. The next part of the report will focus on the programming task and will have the same order as the questions asked in the assignment text.

Along with this report a zip file has also been handed in. This compressed file contains the Eclipse project which contains the source code and should be eligible to be directly imported into Eclipse and run.

As the assignment text implies then this report will not be a thorough description and assessment of each piece of my code, but rather instead a set of discussions on the design and decisions I have considered and made during the development of the supply chain system. I will therefore suggest the reader to look at the source code, comments and documentation if the reader desires a greater insight into my actual implementations.

2 Exercises

2.1 Question 1: Proximity

I assume that we are talking about merging two big data sets in a ordered fashion, e.g. like if we were to merge-sort two arrays.

I assume that one dataset contains N_1 elements and that the other dataset contains N_2 elements, where N_1 and N_2 may or may not be equal. I assume that the two data sets are unsorted before merging.

I must honestly say that I do not understand why there are *two* datasets, i.e. a fixed number, and how this correlates to a generic d-way merging algorithm with focus on the costs of this generic algorithm, and I therefore make the, possible dangerous, assumption that the two datasets are concatenated by appending one to the other before performing the d-way merging algorithm.

Based on the assumption above I will now hence forth let n denote the size of the two datasets combined.

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

I assume that the word operations we are interested in are the ones spent on comparing the elements to sort the dataset, and 'small' operations spent on finding the length of the dataset and maintaining index pointers when splitting the dataset are of no interest in this analyze. These described operations that are left out of the equation are also the ones used to divide the problem, and

therefore I will not consider the divide-phase of the divide-and-conquer algorithm.

I furthermore assume that one comparison is one word operation.

To sum it up; my goal is to try and analyze how many comparisons I must make in a d-way merge-sort algorithm.

As mentioned before I skip the divide-phase of my divide-and-conquer algorithm and therefore I will try and only focus on the conquer-phase i.e. the actual merging. Now if we look at the created recursion tree, which can be seen in **Figure** 1, we can see that it has the height of $log_d(n)$ and as we merge upwards in the tree we must perform $n*(d-1)$ comparisons per level in the tree, as it takes $(d-1)$ operations to compare 1 element to d others and for each level of the tree we must do this n times. Hence the CPU cost of this algorithm in terms of word operations, under the given assumptions, is $\bigcirc(log_d(n) * n * (d-1))$.

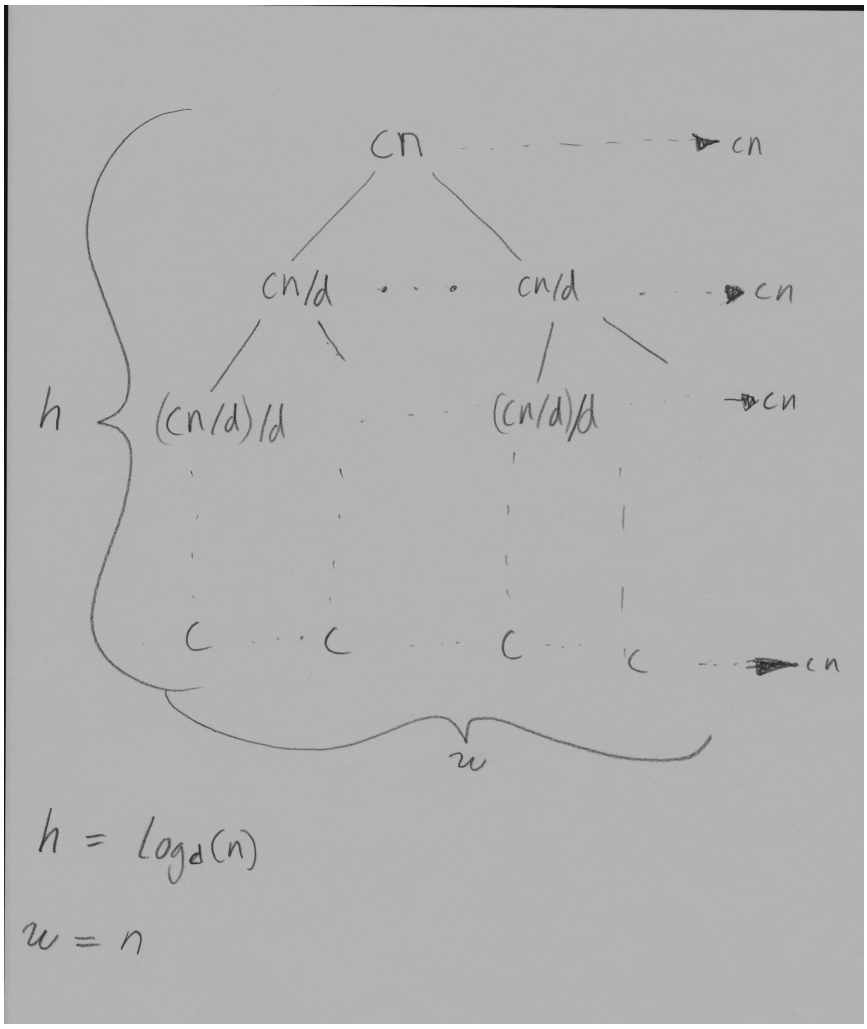

Figure 1: Recursion tree created when doing the algorithm.

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

First order of business is to try and take all the needed assumptions.

I assume that one I/O is either the act of loading an element into the limited sized main memory from the unlimited sized disk, if the element is not already in main memory, or writing the element back to disk, and we are trying to find how many of these I/O operations are being made during an execution of the described algorithm.

I assume that $l \leq B$. Furthermore I assume that when one element is being fetched from disk, it is not the single element that is brought back up, but the whole page it is contained in, hence I also assume that the machine uses paging (also what the assignment text implies). I do not assume that any extra I/O cost is associated with page replacement.

I assume that any I/O cost associated with doing the divide-phase is negligible in this analyze as well.

I further assume that everything is on disk and not in main memory before the algorithm starts.

I do not make any assumptions on what page replacement strategy is used, nor the actual implementation of the structure of the given dataset, but I do assume that when a page is fetched from disk, then magically the whole page is used and not needed again after it has been replaced.

I assume that the merged dataset is written somewhere else and not necessarily merged in-place.

If we from now on let t denote the total number of comparisons needed to perform the algorithm, then we need to make a total of t reads from the disk, although as multiple elements can be on one page we must divide t with the number of elements that can fit in one page, and then add n number of writes, as we must write each element once. This provides us with the cost of $\bigcirc(\frac{t}{B/l})$ and if we extend t we get $\bigcirc(\frac{log_d(n)*n*(d-1)}{B/l})$, which is the I/O cost associated with described algorithm with the given assumptions.

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

When analyzing the cost of address-translation during this algorithm there are indeed two different kinds of costs one might want to separate and analyze, these are the CPU cost and the I/O cost, these two will be described below in their respective subsections.

CPU cost If we consider that we need to make t reads and n writes, based on the I/O analyze above, we could say that we need to translate from virtual address space to physical address space $t+n$ times, and if for each time we need to translate an address we must walk through the page table, the cost must be the number of times we need to translate an address multiplied by the cost of performing a single translation by walking the page table.

By considering a logarithmic tree similar to the one shown above we can conclude that the cost of performing one translation must be the height of the generated tree, which is $log_p(n/B)$ where we get the full cost to be $\bigcirc((t + n) * log_p(n/B))$ and if we extend t we get the full address-translation CPU

cost of this algorithm, under the given assumptions, which can be written as $\bigcirc(((log_d(n) * n * (d-1)) + n) * log_p(n/B))$.

I/O cost I must honestly say that I do not have an answer to this, and after a lot of thinking and failed analyzing attempts, I decided to move on and spend my time on some of the other parts of the exam.

2.2 Question 2: Parallelism

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

As the first hash function h_1 must be able to accept R and S as input (and anything they contain) the domain of h_1 must be $\{R, S\}$ and as the output must be one of the k bucket indexes then the range of h_1 is $\{1, 2, ..., k\}$ assuming that the buckets are 1-indexed.

For the reason that h_2 must be able to accept any bucket index that h_1 is able to produce then the domain of h_2 must be the range of h_1 or put in another way: $\{1, 2, ..., k\}$. The assignment text specifies that h_2 must redistribute the different buckets on to the p different processes that are in the database, therefore the range of h_2 must be $\{1, 2, ..., p\}$ assuming that the processes are 1-indexed.

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

Assuming that the database has no central disk, and that the disk space of each process is also used to store the database content, meaning that the processes them selves are the database and therefore R and S are stored in the p processes and the joined relation T must be stored in the p processes.

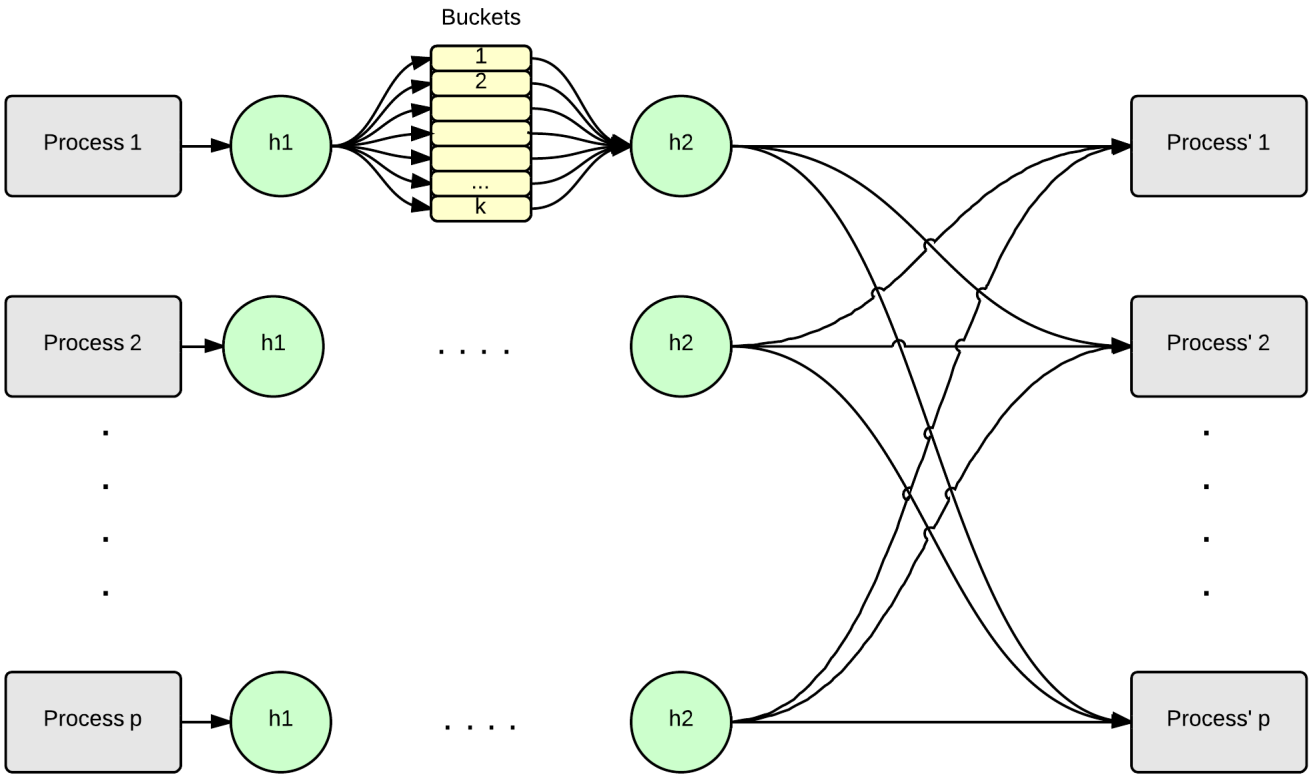

Figure 2: A diagram showing an overview of the parallel hash join algorithm.

In **Figure** 2 we can see that each process starts by bucketing its own partitions of R and S by using the hash function h_1. Then it uses h_2 to re-partition them to the computed destination processor. Then when a process has received all its destined partitions of R and S it starts to join them bucket-wise and store the result on its own disk.

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

Again I must honestly say that I do not have an answer to this, and after a lot of thinking and failed analyzing attempts, I decided to move on and spend my time on some of the other parts of the exam.

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

Assuming that the relations R and S are uniformly spread out onto each process before the join, and that each process does not send any messages before it has finished hashing R and S into the buckets, and that it at worst case will fill all k buckets. Furthermore I assume that the whole content of a single bucket can fit in one message. If also assumed that each process does not batch the messages, but instead sends them immediately after using h_2 to figure out which process must receive that specific bucket, and that the process does not need to send the joined relation back to someone or somewhere. I also assume a message send to your self is also considered a message send and received.

I assume that both R and S can be put inside the same k buckets and can send and received in the same messages, as one would be able to add some tag to these parts to denote which relation they initially belonged to.

Given the described assumptions then each process sends at most k messages.

As their are p processes and the k buckets are uniformly spread out to all the processes then each process receives $\frac{k}{p}$ messages per process which is $\frac{k}{p} * p = k$, i.e. each process receives at most k messages.

3 Programming Task

Development Environment I use JDK 6 and have used Eclipse Kepler to developing the exam.

I have used JUnit version 4.1.1 for my tests and Jetty version 8.1.13 and XStream version 1.4.6. For logging to an excel document I also added some Apache POI, dom4j and XmlBean JARs. The JARs are included in the hand in, but if the reader wants to download and add them themselves they can do that, although I do not guarantee that my project will work with any other versions of the ones provided.

3.1 Question 1

3.1.1 RPC semantics

When a client is trying to register an order workflow with an OrderManager through RPC then the semantics used is *at-most-once*, as the message is only send once and if the communication fails then it is not retried automatically. The reason for the chosen semantics is that it is okay if the workflow does not get registered opposed to *exactly-once* semantics, but it is not okay if the workflow gets registered twice, which could be a situation with the *at-least-once* semantics.

The same goes for when a client is trying to get an order workflow status, then the RPC also has the semantics of *at-most-once* since the call is not auto-

matically retried.

When a client is trying to execute a step in a ItemSupplier over RPC then the semantics used is once again *at-most-once*, the same reasoning, as when a client tries to register an order workflow at an OrderManager, also applies here.

The RPC semantics used when a client tries to get the list of orders per item in an ItemSupplier is also effectively implemented as *at-most-once* as the RPC call is not retried by force.

As the assignment text specifies, then the RPC semantics for executeStep when it is an OrderManger that tries to execute a step at an ItemSupplier, should be *exactly-once* as the OrderManager should keep trying until it gets a response from the ItemSupplier, but does not spam the message until it gets a response. Although I have decided that in my implementation I use the *at-most-once* semantics as this allows me to make testing of failures more simple and tangible. I have made a comment in the code (OrderManagerJob.java) where one can flip the semantics to be *exactly-once* instead of *at-most-once* and vice-versa.

3.1.2 Asynchronous OrderManager

My overall idea to make the workflow processing asynchronous at the OrderManager is to have a threadpool which the OrderManager schedules a workflow processing to, after it has durably stored the workflow to some datastructure/database and log. The scheduler then makes sure to create a new thread and workflow process asynchronous so the OrderManager can return to taking registrations.

Figure 3 shows in greater details my design of the OrderManager. The diagram shows how a request from a Global Client gets propagated through an OrderManager and properly processed. What is worth mentioning is that the asynchronously link happens when the OrderManagerImpl object schedules a new workflow process and the scheduler submits a new job to its work pool.

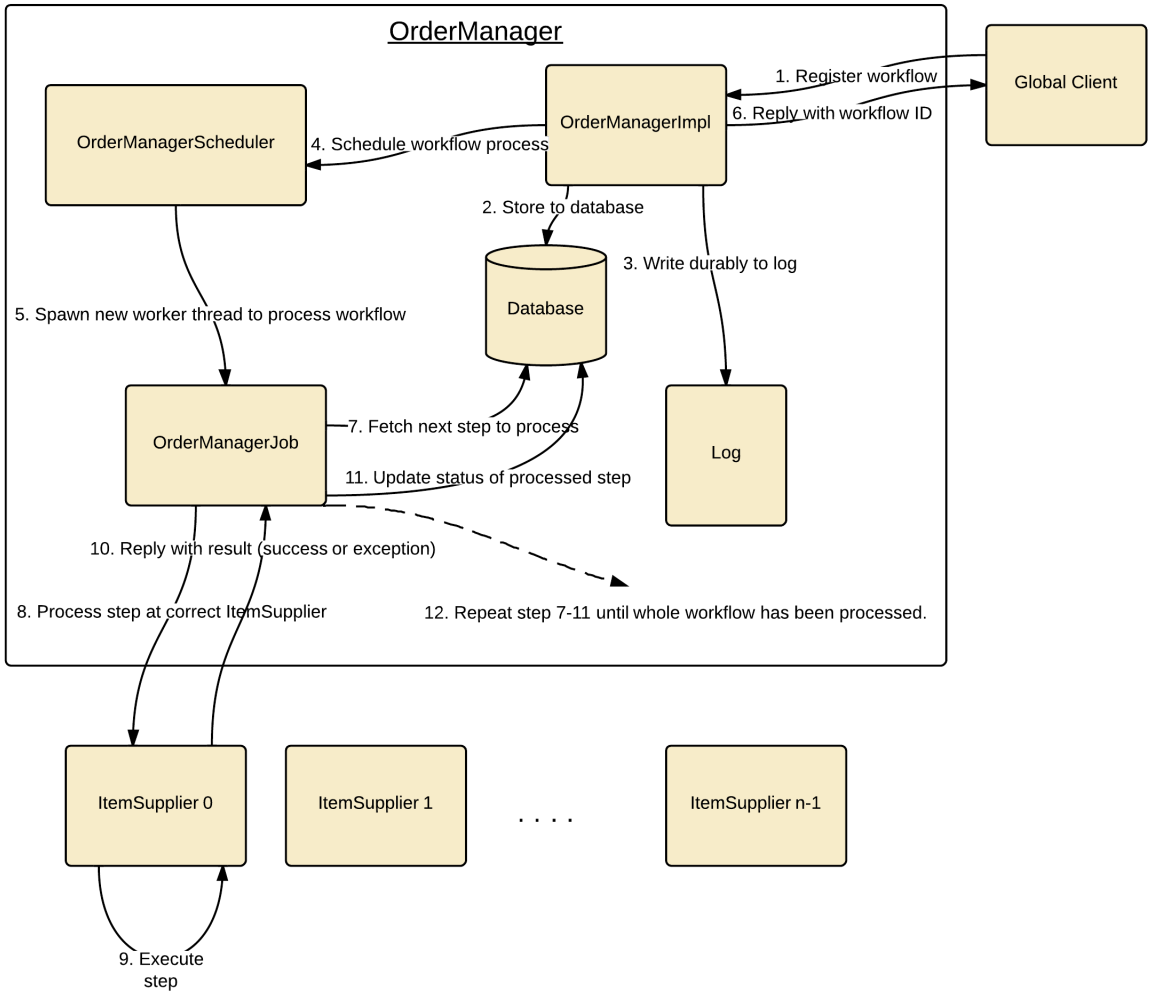

Figure 3: Diagram showing my design of my implementation of the OrderManager.

The database in the diagram is implemented as the two Map datastructures inside OrderManagerImpl. One contains all the registered workflows and the other all the current status on each step in each workflow. In my implementation the Map containing the workflows could easily be avoided by simply passing along the workflow, to be processed, to the spawned thread, instead of the workflow ID, but when making the design I decided to think of the system as it would be implemented in a more real-world implementation. Although to try and limit the overhead a little bit, the OrderManagerJob thread fetches the

whole workflow the first time, and not only a single step at the time.

It is worth noting that this design has focus on making the OrderManager asynchronously with an easy implementation and understanding and not so much on performance as currently in my implementation an OrderManagerJob thread only handles a single workflow and a workflow is only handled by a single OrderManagerJob thread which is a very bad design in some situations, especially if the *exactly-once* semantic is used as this would mean that the rest of the workflow would not be processed until a failed ItemSupplier was back online. Another situation could be if a single very big workflow was registered, but every step of the workflow was intended to a different ItemSupplier, then a lot of waiting time could be saved by scheduling several more threads and allow them to all work on the same workflow. Although as it seemed like that performance of the OrderManager was not of great focus in this assignment I decided not to dwell much in finding the ultimate solution, but instead make note on some of the other options and state that my solution is not optimal in many cases and this area has a great opportunity for optimization.

3.1.3 Failing components

I have subclassed OrderProcessingException and created a class called NetworkException which is thrown if an network exception occurs. The exception class is used to tell if a component has failed as we assume that network partitions do not occur, and a component do not timeout because it has become out of reach or is too slow to respond in time, and therefore can assume that if I receive a NetworkException then the component is deemed failed. If the OrderManagerJob thread catches such an exception then, as opposed to the statement in the assignment text, then the thread does not retry request to the ItemSupplier, but gives up the first time, this decision is taken to simplify implementation and testing. The thread then marks the step as FAILED and continues its work.

3.2 Question 2

3.2.1 Ensuring serializability at each item supplier

I decided to use readwrite locks to ensure serializability at each item supplier. Each item supplier keeps a map of readwrite locks where one lock maps to one specific item ID. When an OrderStep is to be executed the item supplier creates the missing locks, if the OrderStep carries any item IDs which the item supplier has not seen yet. Then when actually executing the order then the item supplier locks all write locks associated with the item IDs in the given step in an ascending order, and once the step has been executed then the locks are released in the reverse order they where taken. Then when getOrdersPerItem is called then the same procedure is performed except now the item supplier locks and unlocks read locks instead of write locks.

3.2.2 Correctness of my method

I decided to try and make my design similar to strict two-phase locking (2PL) and will therefore argue for the correctness of my method by comparing it to that specific variant of 2PL. I assume that the reader is familiar with 2PL and its variants and will therefore not dwell in its definitions. My method conforms to strict 2PL since I take exclusive locks on objects that are modified and take shared locks on objects that are only read. Furthermore I only take a lock if a need it, and I wait as long as possible before acquiring the lock and when releasing the locks I make sure to release them all at once, i.e. I do not acquire any lock(s) after I have released one lock.

Although note that when executing a step I actually conform to the conservative strict 2PL protocol as I acquire all the locks before executing anything, aside from only take the lock right before I need to use it.

Since strict 2PL allows for deadlocks I decided to only acquire locks in an ascending order and only release locks in a descending order, to make sure that deadlocks cannot happen.

The same method is used both in ItemSupplierImpl and OrderManagerImpl even though the OrderManagerImpl is much simpler than ItemSupplierImpl and only takes one lock at the time.

3.2.3 Performance of my method

Early in my development phase I decided to use locks and try and implement something that was logically equivalent to strict 2PL as this protocol allows almost full parallelism of the locks and is a rather simply and straightforward approach.

As hinted in the above when designing and developing my method my focus has been on parallelism of the ItemSupplier and how to utilities the ItemSupplier the most in a concurrent way. Of course I could have achieved a lot more parallelism by not using *strict* 2PL but simple 2PL although this made it harder for me to assure that the executeStep function in the ItemSupplierImpl class is atomic due to my use of ConcurrentHashMap.

I assume that the most workload will occur in the executeStep method, and that each supplier will be more or less specialized in a small set of items or a specific type of items, which would imply that they would have a small set of high end items that are ordered a lot and then a larger subset of items that are not as frequently ordered.

One performance issue that might arise when an ItemSupplier is heavily used is that acquiring and releasing locks are CPU expensive operations, and performing many much operations might cause an overhead that is not worth it in the long run. Where as an optimistic approach or a queuing method would avoid a big part of this overhead. Although the optimistic approach might cause a greater overhead if a lot of threads are ordering the same high end items, since a lot of abort control and fixing would be needed, but would be a great approach if the workload of the ItemSupplier was more spread out and the ItemSupplier

was not as frequently used.

3.3 Question 3

I have tried to design my log files to allow for easy recovery and did not have a great focus on human-readability, except that a timestamp is included which could be avoided in a real-world deployment. In the shown log file snippets then the timestamps is replaced with a log line number.

I am pretending that the implemented clear() function is non-existent when going through my log files, as I only added this function to ease testing of the components.

3.3.1 OrderManager

After a failure in an OrderManager then the state that must be recovered is the registered workflows and all the updated step statuses.

The snippet shown below is an example of an OrderManager log file.

```
...
3 INITOM 1                      // INITOM orderManagerID
4 REGISTER 0 [2,(1,12),(4,33)]
      // REGISTER new_workflowID [supplierID,(itemID,itemQuantity),..] ..
5 UPDATE 0 0 SUCCESSFUL         // UPDATE workflowID stepIndex new_status
6 REGISTER 1 [2,(2,10)] [6,(1,1),(10,12),(2,3)]
7 UPDATE 1 1 FAILED
8 UPDATE 1 0 SUCCESSFUL
9 CRASH
...
```

Here we can see that the OrderManager starts by logging an initialization line with its ID. Then if a workflow is to be registered then it is logged to the file along with its assigned workflow ID and other needed information to recreate the workflow.

As I assume that someone will be monitoring the OrderManager instance, this someone will also be able to tell the rebooting OrderManager that it must now recover after reboot. For instance calling another constructor which will start recovering.

When an OrderManager recovers it must look through the log file from the highest log line, i.e. most recent log line, to the oldest line and find the first occurrence of an initialization line and read the OrderManager ID from there, then from that log line down to the most recent log line, the OrderManager must redo everything. Luckily there is nothing that needs to be undone when recovering an OrderManager and therefore when everything has been redone, then the OrderManager is back in business.

Although it must be stated that the only thing this log file is not capable of recovering is the situation where an ItemSupplier gets the executeStep request and completes it, but the OrderManager fails before receiving the reply. This

can be fixed if each step also has its own ID and the ItemSupplier can discard any already processed steps.

3.3.2 ItemSupplier

First order of business is to define what state the ItemSupplier must recover; which is the supplier ID and the summed orders of the orders that has been fully successfully executed within the ItemSupplier, where I assume that if an order is fully executed then it has also been written to log.

To be able to recover to the assumed state we need to log the provided supplier ID and each successfully executed order step. I therefore see no reason to log any failed steps, i.e. incoming steps that are not valid, or any calls to getOrdersPerItem as these do not alter any defined state in the ItemSupplier.

I am not saying that these situations and informations are not interesting in many other cases, they are just not needed for recovering the defined state.

```
...
14 INITSUP 3    // INIT supplierID
15 EXEC-START 0 // EXEC-START logID
16 WRT 0 0 10   // WRT logID itemID itemQuantity
17 EXEC-START 1
18 WRT 0 1 11
19 WRT 1 2 2
20 EXEC-DONE 0  // EXEC-DONT logID
21 WRT 1 4 1
22 CRASH
...
```

The snippet above shows a subset of an ItemSupplier log file, where as I have manually added the "21 CRASH" line and some comments to try and describe the form. The first shown line is logged when a ItemSupplier is initialized and is given a supplier ID and is not trying to recover from a failure. When a valid OrderStep is to be executed, then it is assigned a log ID which is used to determine which OrderStep a write belongs to and be able to log that an OrderStep has been fully successfully executed.

As above I assume that someone will be monitoring the ItemSupplier, this someone will also be able to tell the failed ItemSupplier that it must now recover after reboot. For instance calling another constructor which will start the recover phase. In such case the ItemSupplier would start reading the log from bottom up (i.e. start by the highest log line and then read towards the smallest) until it hits an initialization line which contains the supplier ID and marks the beginning of the lifetime of the ItemSupplier. From there on it would be able to move downwards again and redo every WRT line from the INIT line and downwards, and in the mean time as it goes down the list, the ItemSupplier keeps track of all the OrderSteps that has been started but was never able to finish before a failure, those "EXEC-START logID" line that does not have an

accompany "EXEC-DONE logID" line. When the bottom has been reached and all steps have been redone, then all the unfinished OrderSteps must be undone before the ItemSupplier has regained its defined state.

In the recovery phase nothing is needed to be added to the log file as the datastructure is kept in main-memory and therefore upon a failure, everything is lost, and the whole procedure must be redone from scratch, and any pass recovery attempts are ignorable.

3.4 Question

3.4.1 Test of OrderManager with asynchronic workflow processing

As the OrderManager uses asynchronic execution of registered steps then I had to define somehow to assert that the OrderManager and underlying ItemSuppliers are in the expected states at the expected times and situations. By the nature of asynchronicity this is hard to do. I therefore decided that I would only assert the state of a OrderManager or an underlying ItemSupplier either before a workflow is registered or after a registered workflow has finished execution of all its steps. As the RPC semantics, when executing a step between an OrderManager and an ItemSupplier, is *at-most-once*, this is possible.

With this setup at hand I can now easily test the OrderManager and the underlying ItemSuppliers by register a set of workflows and then assert the expected states both before and after registration.

To perform the waiting I have added the waitForJobsToFinish method to the OrderManager.java interface which waits for any ongoing processing threads to finish before returning. By then I can be sure that it is not 'random' which state the OrderManager or ItemSupplier's will be in when their states are being asserted.

3.4.2 Test of atomicity in executeStep

To test that the executeStep function in the ItemSupplierImpl.java class is indeed atomic I decided to make what I call a 'stress' test. I try and put pressure on the property to see if I can break it.

My setup is as follows; I create a single ItemSupplierImpl instance and then a number of Runner threads whom have one job and that is to execute steps to the item supplier instance. Before starting the Runner threads I create a Checker thread which repeatedly calls getOrdersPerItem and asserts the return value with an expected value, and if the return value is something unexpected at any point then the test fails. All of which happens within the same JVM-process. The Runners and the Checker is then started, and I wait until they are all done and see if the Checker at one point deemed the test to be a fail. If the test succeeded I try again with a greater number of threads and iterations, until it fails or I decide that I don't want to wait an hour anymore before the test finishes.

The downside of this test is that it does not guarantee that my implementation of executeStep in ItemSupplierImpl is indeed *all-or-nothing* and *before-and-after* atomic, but it does greatly hint that my implementation might be correct.

Furthermore as I try to always develop with the test-driven development mindset, I created this test before ensuring/implementing atomicity in the executeStep function, and the function failed horribly every time, after implementing my method of atomicity the test has yet to fail, still after countless of test runs.

3.4.3 Test of error conditions and failures of the multiple components

Testing error conditions My general strategy for testing that the proper exceptions are thrown is that I give the components that are under testing some initial state and assert that they are in this initial state. Then I assert that the error occurs as expected and finally I assert that the components are still in the same state, or what ever is the expected state after the tested error condition.

Testing failure When designing my failure handling test I had to decide on when to manually fail a component and still be able to assert some expected state.

I decided to use the strategy of placing the components in some initial state then manually fail one component type followed by making sure that the still functional components are still working as expected, then I manually fail one component of the other type and once again make sure that the rest of the components are still working.

This strategy allows me to make assumptions on the state of the individual components both before and after a component fails. Furthermore this strategy is easy to control and removes an amount of 'randomness' from the tests.

What this strategy is not capable of handling is when a component fails while being mists in communication with another component. Although in this case I argue with my use of the NetworkException takes care of this.

3.5 Question 5

As the time is now 6 minutes to deadline this chapter, very unfortunately will not be one of the good ones...

3.5.1 Setup

I refer to the Experiment.java class for a description of the test setup.

3.5.2 Results

As we can in **Figure** 4 that when scaling the number of clients the average latency keeps something that looks like a straight line, right until we hit a to

big of a thread overhead.

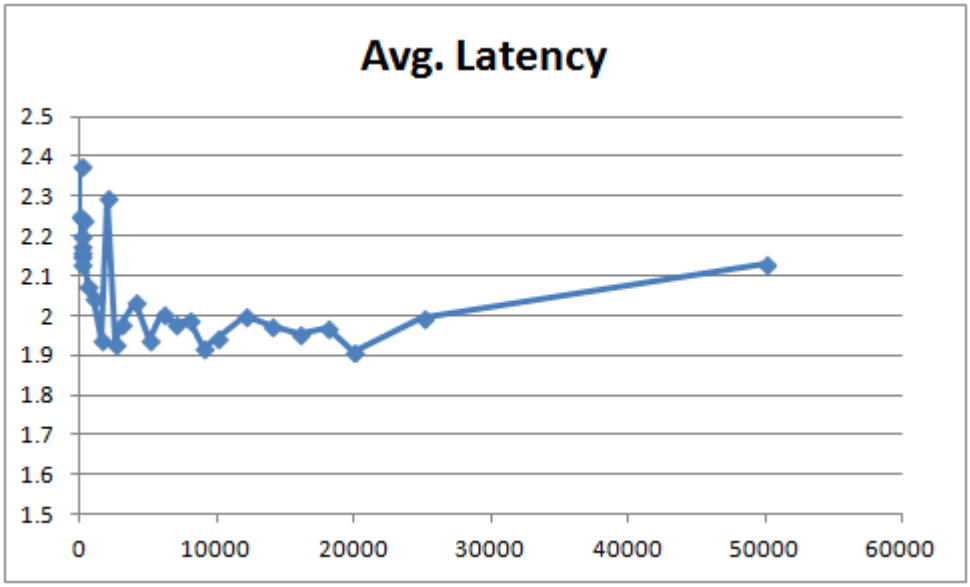

Figure 4: Avg. Latency