PCSD 2013/14 Exam

Henrik Bendt (gwk553)

21. januar 2014

Question 1

a)

We follow the two-phase, multiway merge-sort of chapter 10 of the compendium.

We assume the following:

- The two data set are not sorted.
- Copying or moving a word costs 1 word operation, so copying or moving a record costs l word operations.
- Swapping two words costs 2 word operations.
- Comparing two keys costs k word operations, as a key is k words long and for simplicity we assume that the whole key must always be compared.

We start by splitting the two big dataset into d lists of records. If we move all records when splitting the records, to say, new memory blocks, this will cost $a = l \cdot N$ operations, where $N = N_1 + N_2$. We then get lists roughly of size $b = \frac{l \cdot N}{d}$ words.

We sort each recod with for example Quicksort. Lets say this is an average case for the Quicksort algorithm, so we must make $b\ log\ b$ comparisons, if we assume that all words of a record must be compared when comparing records, where each comparison costs k word operations, so we get a cost of $(b\ log\ b) \cdot k$. We must also make swapping and in worst case we swap for each comparison, so lets go with that for simplicity. We thus get a sorting cost for each list of $e = ((b\ log\ b) \cdot (k + l \cdot 2))$, as each swap costs 2 operations.

Thus we have a total cost for this being $b \cdot e$ operations, but we are not done. Now comes the d-way merge.

The d-way merge searches linearly through the d lists of records to find the smallest key (we could use priority queues, but for simplicity we just do it linearly). This costs $(d-1) \cdot k$ operations, as we can just use a pointer to the current lowest key (i.e. no need to copy it). We move the record of the lowest key to the output, costing l operations. This costs $f = (d-1) \cdot k + l$. For simplicity, we assume all lists have records until the last step, so we always consider all lists when searching for smallest key. We keep doing the above until all elements are moved. This gives roughly a cost of $N \cdot f$ (ignoring the cheaper last step).

The total cost, with the above assumptions and simplifications, must thus be $b \cdot e + N \cdot f$ word operations (note that this is neither an upper or lower bound).

b)

We assume that one I/O cost equals moving a block to or from disk.

If $M \cdot l > N * 2$, we do not need to do any I/O operations, as all can stay in memory.

Else, we assume that main memory can hold at least $B \cdot d$ blocks, so $M > B \cdot d$ (needed to combine the lists). For simplicity, we also assume that main memory can hold one of the list of records from before, that is, $M >= b$.

So, for simplicity, we assume all data is found on disk and main memory is empty. We must read each list to main memory, to be sorted, and write them back again, giving an IO cost of roughly $g \cdot 2$, if $g = \frac{b}{B} \cdot d$. Then we must read a block from each list back to memory and as we do not know what to do with the result (it is not known if it should be stored), we simply assumes it *dissapears*. So this is the same as reading and every block of every list once. This gives a cost of g. Thus the total IO cost is roughly $3g$ blocks, which is like results found in chapter 12 of the compendium.

c)

I do not find any information about this in the compendium nor do I see this fit into any of the learning goals of this course (I do not see this fit into the description of "large scale data processing", this is handled by the OS and is for an OS course). Not only that, the slides does not contain information about how to calculate on this, only high level description of the cost, so there is no description of how to calculate this.

But here is my proposal for a part of a solution:
The cost of finding an address in the cache would be hit_c and a miss would constitute another cost, say $miss_c$. Assuming each operation on a record needs a lookup, this gives a minimum number of lookups of $N + (n\ log\ n) \cdot 2$, with $n = \frac{N}{d}$ the number of records per sublist, for the splitting and sorting and swapping in the sorting. Then all sorted sublists must be merged, which, with the same assumptions as in 1a), roughly costs $(d-1) \cdot N$ lookups. This gives a bound of $(N + (n\ log\ n) \cdot 2) + (d-1) \cdot N$ lookups. The splitting of the records will constitute only misses, as no record is looked up twice. Assuming the addresses of a sublist can fit into the cache, each sorting will constitute n misses and the rest will be hits. This gives a cost of $N \cdot miss_c + n \cdot miss_c + ((n\ log\ n) \cdot 2 - n) \cdot hit_c$, and the final merge will have one miss and $d-1$ hits for each linear search of the smallest element, plus an initial d misses when doing the first linear search, giving a cost of $(((d-1) \cdot N) - N - d) \cdot hit_c + (N + d) \cdot miss_c$.

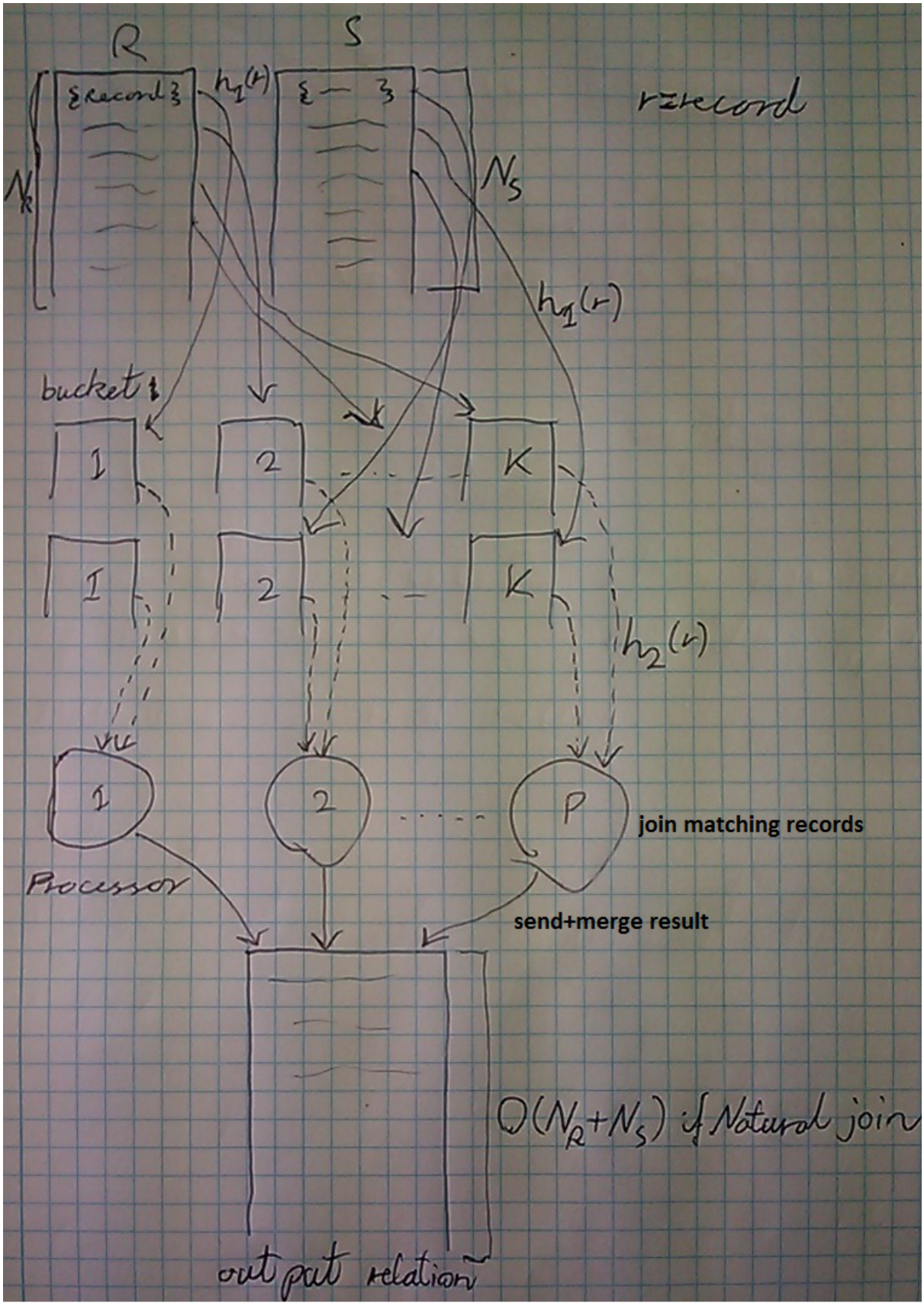
Question 2

a)

If we assume h_1 partitions as in chapter 12 (15.5.1) in the compendium, but work with records instead of tuples, we have
Domain: $\{r \mid r$ is a record of a relation$\}$
Range: $\{b_i \mid b_i$ is the i'th bucket, $i \in \{1, 2, ..., k\}\}$

h_2
Domain: $\{r_i \mid r_i$ is a record of a bucket i, $i \in \{1, 2, ..., k\}\}$
Range: $\{P_i \mid P_i$ is a processor performing join on buckets i, $i \in \{1, 2, ..., k\}\}$

b)



Figur 1: A picture of how the parallel hash-join algorithm works. All records partitioned to buckets, which again are distributed to processors joining them by taking all records of R_i joining on a record of S_i, assuming natural join. This is possible because records of R_i will only match records of S_i. The output is a hash-joined relation with its records being all joined records of R and S.

c)

With a uniform hashing, each bucket gets an equal amount of records, and assuming an also uniform distribution to processors p, each processor must make an even amount of I/O's.

We assume partitioning, i.e. applying both hashes $h1$ and $h2$ and sending the record to the right processor, can be done by each processor while also recieving and storing partitioned records from other processors. We also assume a uniform distribution of records to each processor disk before parallel hash-joining them. Then each processor reads and writes as much as send, which is the same amount as recieved, that is $p_{IO} = \frac{bps}{p}$ records with bps being the bucket pair size $bps = \frac{N_R+N_S}{k}$ records. Each processor musth then again read all of these records when joining them, so we get that each process do a number of I/O's proportional to the number of records, that is $3 \cdot p_{IO}$. The actual number of I/O's depends on the number of records b to fit in a block, so we get $\frac{3 \cdot p_{IO}}{b}$

d)

With the same assumptions as above, and also assuming a uniform distribution of actual joins (i.e. records that joins), each processor will do the same amount of messaging.

Each processor will send and recieve p_{IO} records when partitioning (applying both hashes) and will then send some number j joined records to output. Thus a processor will send an amount proportional to $p_{IO} + j$ and recieve an amount proportional to p_{IO}. The actual amount of messages depends on the number m of records possible to be send per message. So we get $\frac{2 \cdot p_{IO}+j}{m}$.

Programming Task

0.1 Assumptions

I assume that you cannot change item supplier ids or item ids after initialization, neither delete any item. I also assume that you cannot reduce the quantity of an item in any way.

I assume you cannot change address of any component after initialization.

I assume that an order manager loads the ids and addresses of item suppliers, but does not have any knowledge of the items of the individual supplier - so when validity checking input, though it checks for known supplier id, it only validity check item ids and quantities – if the number is positive. I also assume you cannot delete any known item supplier of an order manager.

I assume the logger should always be enabled and that the placement of the log should always be in the same folder as the Java-file.

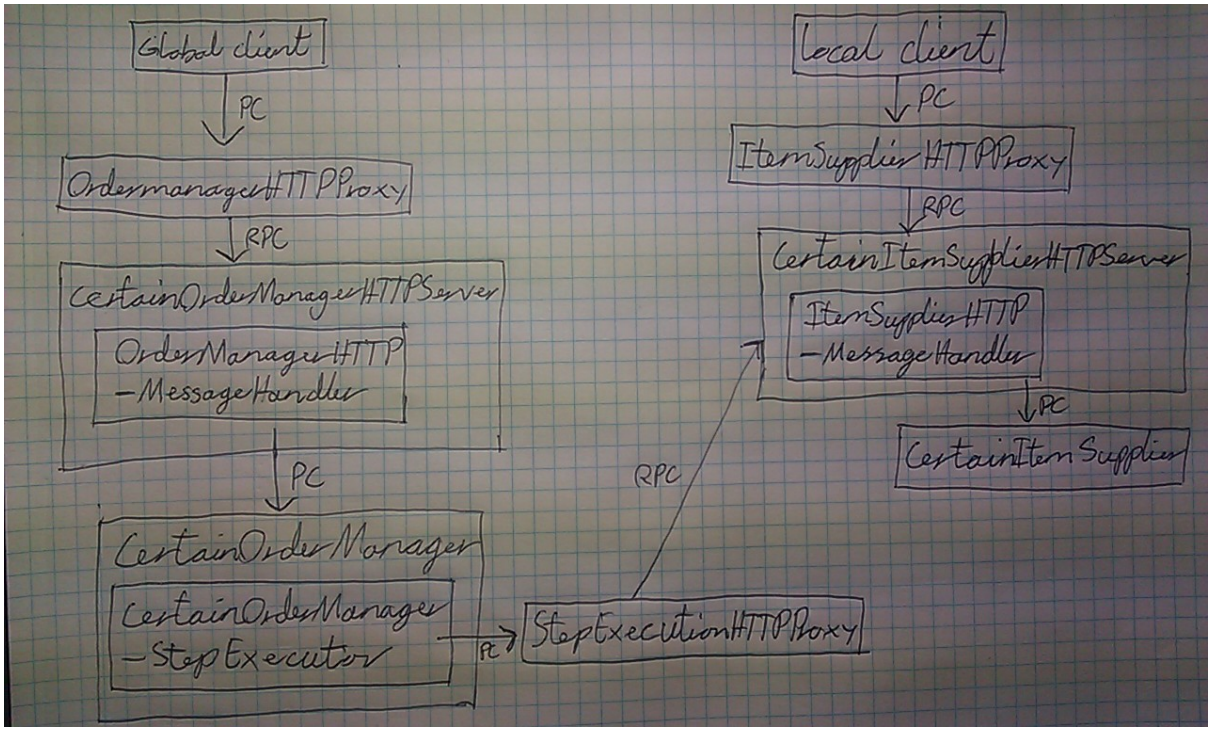
0.2 Organization of Modules

Here is an overview of the folders of the project

- `com.acertainsupplychain` contains the given objects and interfaces along with their implementations and the classes handling execution of workflows of the `OrderManager`.
- `com.acertainsupplychain.proxy` contains the proxies and proxy constants.
- `com.acertainsupplychain.server` contains the HTTP servers and handlers and server utilities.
- `com.acertainsupplychain.test` contains the JUnit test classes and the `TestServer` class. Also contains a configuration file for one of the tests.
- `com.acertainsupplychain.utils` contains utility classes and methods.
- `com.acertainsupplychain.workloads` contains the workload experiment classes and the experiment setup.

At root of */src/* lies an example of a server configuration file. Generally, a configuration file must be of the layout:
"managers=[managerId] [managerAddress];[managerId] [managerAddress]...
suppliers=[supplierId] [supplierAddress] [itemId,itemId,..];[supplierId].."
An example:
"managers=1 localhost\:8080;2 localhost\:8081
suppliers=3 localhost\:8082 1,2,3,4,5,6;4 localhost\:8083 2,3,4"

All log files (see subsection Logging(0.4)), along with results of experiments (called *benchmark.remote.dat*) will be placed at root.



Figur 2: Architecture of the supply chain application.

The singleton design of the bookstore of the assignment is not used in this implementation. Thus multiple order managers and item suppliers can run on the same JVM.

There is a seperate class for multiple local servers (on the same JVM), namely `com.acertainsupplychain.test.TestServer`, used for testing and the experiment.

I should have made immutable versions of OrderStep and ItemQuantity, as you should use immutable objects when sending them through other classes as a well as responses to RPC, to ensure they do not change (by accident) underway. I did however not do this, but I should have.

0.3 Changes to interface

Functionality has been added to `OrderStep`; see subsection Logging(0.4).

0.4 Logging

I have made logging for each component forced to filesystem as described by the assignment. Each component writes to a log defined by their unique id and component type at the root of the project, named *componentType.id.log*.

The problem with logging and recovery of an OrderManager after a failure was that it was needed to know whether an execution of an order step, which was being handled while the OrderManager crashed, was successful or not – that is, if it changed the state of the ItemSupplier and was commited. The crash could occour right after recieving the result of a succesful execution but right before being able to update (or log) the recieval of such result. Thus this change of state of the ItemSupplier would be lost and inconsistency between the OrderManagers and the ItemSupplier would be in effect. This should not be acceptable, at least not if we strive for BASE-attributes (eventually consistent).

As it is with the handed out interface, the `OrderStep` does not give any chance of knowing exactly what order step from what workflow from what order manager was executed. This is probably because it is of no interest to the item supplier, but if we need to keep consistency between item suppliers and order managers in case of crashes, I must expand `OrderStep` to possibly include an order manager id a workflow id (which can be null, so local clients can still use same interface).

Do note that if a workflow of an order manager, say, contains only equivalent order steps, it is not guaranteed that these order steps are updated exactly in the right order as before the crash. This is because we cannot distinguish between individual order steps. I have assumed this is an irrelevant minor detail, but this could be handled by expanding the data structure of the workflows to contain individual ids for each order step.

Each record of the log for the order manager contains the workflow id, supplier id, list of item ids, list of item quantities and status of order step. Each record defines an update to the workflows of the order manager.

0.5 Testing and functionality

I have included all tests in the subfolder *com.acertainsupplychain.test*. Everything should work correctly, but I find some thread-could-not-be-stopped exceptions (stacktraces) being printed to console from a Jetty client. I was unable to find the cause of this, but it seems to only happen with lots of trafic(it is found when doing the experiment).

Question 1

We must have effectively implemented the ACID-methodology between clients and components because:

- **A**tomicity: all RPCs are atomic.
- **C**onsistency: all RPC are consistent (changes the component from one state to another).
- **I**solation: all RPCs have before-or-after atomicity.
- **D**urability: logging ensures that the results of a done (commited) RPC is persistent on the client.

We must have effectively implemented the BASE-methodology between the OrderManager and the ItemSupplier:

- **B**asically-**A**vailable: Only failed components become unavailable, not any other components (especially not the whole system).
- **S**oft-state: Components can fail (fail-stop) without affecting availability of other compontents but components may be out-of-date, e.g. an execution of a step might not have returned and updated OrderManager correctly before a crash of either one.
- **E**ventually consistent: Asynchronous calls from the OrderManager to the ItemSupplier ensures that the OrderManager will eventually be consistent with the result of the executed steps of a workflow, which eventually are updated accordingly in the OrderManager.

For the order manager to process workflows asynchronous, all order executions are sent as jobs to a seperate thread (pool) which handles this. Each thread makes the synchronous RPC-call to the item supplier of the given order step, and stores the result in a list (containing all unhandled results). A seperate thread handles this list of results; if the call was successful, updates the workflow of the given

order step with the result of the execution, and if unsuccessful, meaning a fail-stop, the job is resubmitted. After the thread handling the results of the latest requests is done, it checks if any new results has arrived or waits until some has. Thus this is a permanent background thread.

Failure of the order manager is trivial, as this only affects external global clients, and thus cannot propagate. Failure of an item supplier affects only the order managers (who knows of the item supplier), and can thus only propagate to them. This propagation is handled by simply *ignoring* the failure of an item supplier and simply try to execute the order step to it again, as specified in the assignment. The only problem with this approach, as the item supplier has no recovery, is that it slows down performance of the order manager, which has only a limited amount of threads to execute step orders. Execution of order steps to other item suppliers will get higher latency and lower throughput, and in case of a continues increase of order steps against the dead item supplier, throughput and latency of execution of order steps will keep respectivly decreasing and increasing.

Question 2

Atomicity of ItemSupplier operations is ensured by read/write-locking. As it was not mentioned what kind of atomicity was in mind, I have made both all-or-nothing, by validity checking input, and before-or-after atomicity, by locking before an operation and unlocking after the whole operation is done. When locking, the whole set of items is locked. This ensures correctness at the cost of performance, as there can only be one writer (though multiple readers at once) at a time. This works just like a conservative strict two phase locking, as because of the simple operations, all locks are acquired at the same time before the operation and released at the same time after the operation. Note that the locking is fair, i.e. "When the currently held lock is released either the longest-waiting single writer thread will be assigned the write lock, or if there is a group of reader threads waiting longer than all waiting writer threads, that group will be assigned the read lock."
(`http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/locks/ReentrantReadWriteLock.html\#readLock\%28\%29`)

The used method is should be correct, as it works like a global exclusive/shared lock for the individual item supplier and it is implemented as a two phase locking-mechanism.

This is not an optimal solution performancewise, if you assume a workload with a lot of writing (executing steps). It performs better than with queuing, as the later means sequential reads, but a much better performance would be found with a two phase lock on each item. This would ensure much more concurrency and throughput, as many more could access the item pool at the same time (both readers and writers), but all at the risk of deadlocking. This could be resolved by having an operation time stamp, where the oldest operation can take an intention lock from a younger operation. When having obtained all intention locks, take the locks and then make all the writes, releasing the lock for each write. A third option could be optimistic concurrency control, but as the number of items per order step is increases, throughput would decrease because of many roll backs. Also this is a complex implementation. Note that validity checking is done without locking, as this only checks if item ids exists – items cannot be added or removed in this implementation – which gives some increase in performance compared to locking from the start of the method (with for example synchronize).

As I assume there will be a lot of writes, i.e. order steps, compared to reads, the fair read/write lock is better than just a read/write lock, but way worse than

Question 3

For the ItemSupplier, the accumulation of quantities must be readded to each item. This is done by traversing the whole log from the beginning to the end and do all actions (only successful executions are logged).

For the OrderManager, the workflows must be readded and the status of each order step of each workflow must be updated. This is also done by traversing the log from the beginning to the end and update the data structure accordingly. However, this is not enough. There can be some order steps with status *registered*. We cannot know if these have actually been executed, as the crash could have occoured while updating the status of a successful executed (and commited) order step. Thus we must ask all

item suppliers of these order steps for their log, traverse these logs and look at all logs regarding our OrderMangager id and for each of these logs, check if the corresponding workflow is updated correctly with regard to the order steps with status *registered*. Do note that I have made changes to `OrderStep`, so an item supplier can log both the id of a possible order manager and the current workflow.

An improvement to the above could be to add checkpoints at some points, so the whole log need not be traversed at each crash. This checkpoint could be a description of the current data in the data structures of the OrderManager and ItemSupplier, so only this datastructure from the last checkpoint needs to be reconstructed, as well as updated with the logs after that checkpoint.

Question 4

To see asynchronicity of execution of workflows, simply test, after an added workflow, that multiple retrieves of that workflow's status shows a specific order step status as *registered*, but at some point this changes (to either *successfull* or *failed*). In case of multiple runs, this must be done asynchronously, as the function (adding workflows) has already returned.

To test atomicity of operations of the ItemSupplier, I made 5 threds to update the same item quantity and 2 threads to get orders of that item (not using this to anything, simply to interfere with the other 5 threads). Then, if the quantity of the item added up to the sum of updates, the operation must be atomic (have tested for both successful and failure).

I have tested the interface for error conditions and responses by simply applying invalid input and expect exceptions to be returned.

I have tested for failure of multiple components. This is done by having two threads using individual `OrderManager`s (through proxies) and also having some `ItemSupplier`s active. Then shut down one of the `OrderManager`s and see that the other components keep living. Then I wanted to shut down one of the `ItemSupplier`s, but then I found that Jetty does not throw an exception, when a connection is refused, but simply prints the stack trace. As I do not know any other way to fail-stop a component, I cannot show that shutting down an `ItemSupplier` (in use) will not propagate to any other component. But it is clear how I would have tested it from the above explaination.

Question 5

We shall test how a single `ItemSupplier` behaves as concurrency, that is the number of clients, local as well as globals, increases.

I would assume a higher number of local clients making a medium, but small in size, number of orders, and a small number of global clients, making fewer, but greater workflow-orders. As a global client probably have large workflow-orders, this results in a high number of greatly sized orders. In the end this will be a 30-70 distribution of calls between local and global clients, respectively. As we experiment on the item supplier, each global client has one order manager, so increasing number of global clients linearly increases the number of order managers as well. As order sizes, i.e. the number of items, does matter in performance (latency and throughput) of the supplier, these sizes should be of constant size for each of the local and global clients to get valid results. Thus we set that a local client only orders two items at a time and a global client orders 10 items at a time. We set the item supplier to have 20 items. As it seems likely that each workflow only contacts an item supplier once, each workflow only has 1 big order step. Thus we end up mesaureing on a workload mix of small and large orders, with a mix of 3-7 calls (RPC), increasing in number linearly as we increases the total number of the mix. To emulate worst case scenario, the successratio of calls is $>= 99\%$, as this means that at most calls, all items are updated which gives the highest workload on the item supplier.

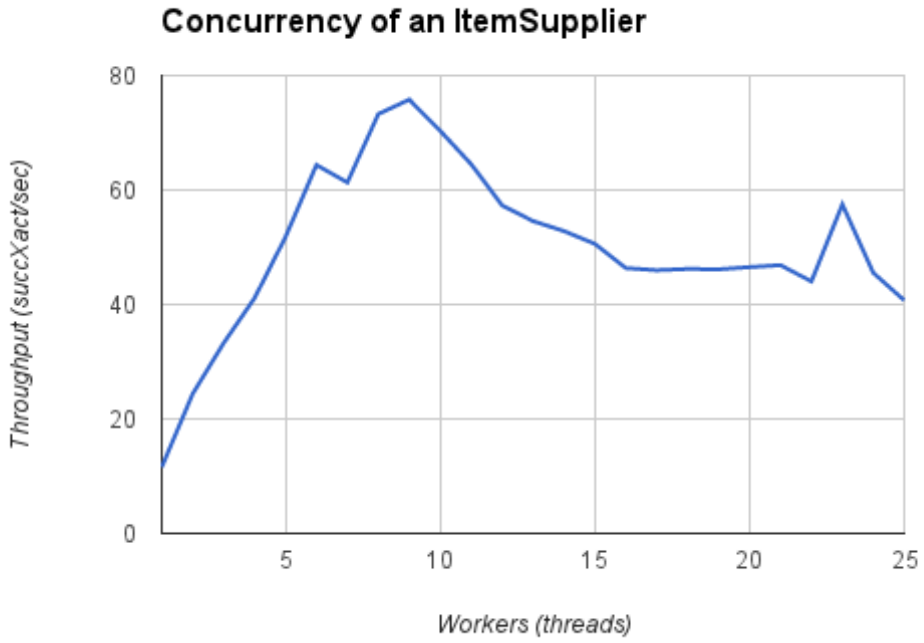
We do all runs on the same computer. This makes consistent results, which works fine for testing how an item supplier handles concurrency, but do note that it is not a realistic situation. The experiment is run on a Lenovo X230 with SSD.

- Processor: Intel® Core™ i5-3320M (2.60 GHz, 3MB L3, 1600MHz FSB) (dual core)
- Memory: 4 GB PC3-12800 DDR3 SDRAM 1600 MHz

- Storage: 250 gb SSD

The main limit of these specs is the CPU, which might limit the scalability to a level, but the main limit of scalability is the number of threads running on the item supplier.

The experiment consists of a number of steps, where each step is a number of worker threads running the specified mix on the item supplier. Each worker makes a number of calls distributed by the specified mix. It starts by making some warm-up runs and then do the runs to benchmark. We use warmup runs for each run to reserve the resources from the opererating system, so we get a more even run time for the actual runs. Warm up runs are for each worker, and so, with multiple workers, each worker can to some extend do warmup runs while another worker is benchmarking. This unfortunate as it can affect the last real runs of the latest worker, but as all workers do the same number of warmup runs, it should be minimal and not affect the results. When doing a global client execution, the client waits for the result being updated in the workflow before doing another call.



Figur 3: The behavior of throughput as concurent use of an ItemSupplier increases. Each worker (thread) runs 200 times, trying to fulfill the mix 30-70 for local-global clients.

To note: The successratio is kept at 100 percent and the client ratio is kept between .68 and .72 (for the global client) for all number of workers.

We see some unexpected behavior at 7 and 23 threads. The successratio and client ratio is fulfilled at these points, so it is hard to say what causes this. It might be an operation system thing, though I did not touch the computer while running the experiment, or the JVM garbage collector. At 23, it might be because of some threads failing without the program noticing (Jetty have issues with throwing errors in some cases, many a time they simply print stacktrace and keeps going). Because the experiment prints a lot to the console log(about what is happening in the experiment), this is not present in the end, because the console can only have so many lines, so maybe I should not have printed to the console log at all (to see such errors). Alas, I did not have time to do this. Otherwise we see that the throughput equalize at about 46 successful transaction per second when reaching a certain number of concurrent calls. It is expected for the supplier to reach some stable throughput at some point, though this should keep steadily going down (this is not clear from the results though). We also see the optimal number of concurrent threads being about 9. It was expected to have an early optimal number of threads and then, as threads increases, the throughput should go down fast until a somewhat stable level is reached, by where it should keep going down at a steady pace. This is because workers interfere with eachother

when accessing shared resources. If I had used a more effecient locking-mechanism, the optimal number of threads might have been higher, but the overall behavior should have been the same.