Advanced Computer Systems (ACS), 2016/2017

Final Exam

This is your final 5-day take home exam for Advanced Computer Systems, block 2, 2016/2017. This exam is due via Digital Exam on January 19, 2017, 23:59. The exam will be evaluated on the 7-point grading scale with external grading, as announced in the course description.

Hand-ins for this exam must be individual. Cooperation on or discussion of the contents of the exam with other students is strictly forbidden. The solution you provide should reflect your knowledge of the material alone. The exam is open-book and you are allowed to make use of the book and other reading material of the course. If you use on-line sources for any of your solutions, they must be cited appropriately. While we require the individual work policy outlined above, we will allow students to ask *clarification* questions on the formulation of the exam during the exam period. A clarification question *should not include any solution ideas* to questions in the exam, but only pertain to the meaning of the problem formulations and associated assumptions in the exam text itself. These questions will only be accepted through the forums on Absalon, and will be answered by the TAs or your lecturer. The goal of the latter policy is to give all students fair access to the same information during the exam period.

A well-formed solution to this exam should include a PDF file with answers to all scenario questions as well as questions posed in the programming part of the exam. In addition, you must submit your code along with your written solution. Evaluation of the exam will take both into consideration.

Do not get hung up in a single question. It is best to make an effort on every question and write down all your solutions than to get a perfect solution for only one or two of the questions. Nevertheless, keep in mind that a concise and well-written paragraph in your solution is always better than a long paragraph.

Note that your solution has to be submitted via Digital Exam (https://eksamen.ku.dk/) in electronic format. In the unlikely event that Digital Exam is unavailable during the exam period, we will publish the exam in Absalon and expect any issues to be reported to the email bonii@diku.dk. It is your responsibility to make sure in time that the upload of your files succeeds. We strongly suggest composing your solution using a text editor or LaTeX and creating a PDF file for submission. Paper submissions will not be accepted, and submissions will only be accepted in Digital Exam.

Learning Goals of ACS

We attempt to touch on many of the learning goals of ACS. Recall that the learning goals of ACS are:

(LG1) Describe the design of transactional and distributed systems, including techniques for modularity, performance, and fault tolerance.
(LG2) Explain how to employ strong modularity through a client-service abstraction as a paradigm to structure computer systems, while hiding complexity of implementation from clients.
(LG3) Explain techniques for large-scale data processing.
(LG4) Implement systems that include mechanisms for modularity, atomicity, and fault tolerance.
(LG5) Structure and conduct experiments to evaluate a system's performance.
(LG6) Discuss design alternatives for a modular computer system, identifying desired system properties as well as describing mechanisms for improving performance while arguing for their correctness.
(LG7) Analyze protocols for concurrency control and recovery, as well as for distribution and replication.
(LG8) Apply principles of large-scale data processing to analyze concrete information-processing problems.

Scenarios

Question 1: Data Processing (LG3, LG8)

A telecommunications company offers a service in which users can exchange short messages with each other. These messages are recorded in a relation `msg(userid, time, msg_txt)`. You are a data analyst who got an extract of the `msg` relation and wishes to derive valuable insights from this data. However, the data is very large. You store the data for performance in the distributed main memory of a cluster of *N* nodes. The extract of the relation you got consists of *M* pages, and each node stores *M*/*N* pages of the relation (*M* >> *N*). These *M*/*N* pages almost saturate the main memory of an individual node; however, there are enough pages of main memory left for I/O with the node's disk subsystem and communication operations with all other nodes (2*N* pages). In addition, there is enough remaining memory for any small data structures.

The `msg` relation is given to you sorted by the `time` attribute, and thus each node stores a range of `time` values and the contents of each node's main memory is sorted by `time`. After discussing possible analyses with your team, you decide that you would like to repartition the `msg` relation instead by `userid`, but keep the `time` sorting within each partition. In other words, each node in the cluster should store in main memory a well-defined set of `userid`s; however, all records in the memory of a single node should still be sorted by `time`, and not by `userid`.

You may assume that the number of records per `userid` is small enough that there is no need to break down the same `userid` across different nodes. In addition, you may assume that there is no significant skew in the number of messages per `userid`, so that a partitioning by `userid` will again leave each node's main memory with roughly *M*/*N* pages of the `msg` relation. Finally, you may assume communication and I/O at a node is perfectly overlapped and parallel, given that the disk subsystem can also sustain a degree of parallelism of *N*. In other words, *N*-1 transfers of pages in main memory from different nodes to the *disk* of a given node *k* take the time of 1 I/O, but have a cost of *N*-1 I/Os to write the pages to the *disk* subsystem at *k*. Similarly, up to *N* page reads from the *disk* subsystem at *k* take 1 I/O time and incur *N* I/O cost. The network has enough bandwidth to support parallel all-to-all node transfers without any further delays. In effect, you may assume network transfer to have zero cost.

By answering the questions below, you will describe and analyze an algorithm involving parallelism and external memory to efficiently achieve this repartitioning.

1. State an algorithm to repartition the data of `msg` as specified above. Argue for the algorithm's correctness and efficiency. Clearly state which steps of the algorithm can be performed in parallel.
2. State the total I/O cost and the total I/O time of the algorithm you designed in part 1 above in terms of M and N. Explain why the algorithm has the costs stated.

NOTE 1: The total I/O cost corresponds to the total number of pages read or written to disk at all nodes, assuming network communication costs can be fully overlapped or are minimal by comparison. The total I/O time corresponds to the sequential I/O cost of the algorithm taking into account that several I/Os happening perfectly in parallel take the time of a single sequential I/O (c.f., parallel work and depth).

NOTE 2: To state an algorithm, you can reference existing sort-based or hash-based external memory algorithms. You should not state all the steps of these existing algorithms from scratch again, but you should clearly state the steps that you need to change in the algorithms you reference, and also how you change these steps. To describe how you change a step, refer to the step and list the sub-steps that need to be executed to achieve your goal.

NOTE 3: Instead of just using several existing external memory algorithms in sequence as black boxes, you should design a single algorithm that addresses the whole task holistically. That is why in NOTE 2 we expect that you will need to show changes to steps of existing algorithms, if necessary.

Question 2: ARIES Recovery (LG7)

Consider the recovery scenario described in the following, in which we use the ARIES recovery algorithm. At the beginning of time, there are no transactions active in the system and no dirty pages. A checkpoint is taken. After that, three transactions, T1, T2, and T3, enter the system and perform various operations. The system experiences a crash, and during recovery, the system experiences another crash. The detailed log follows:

```
LOG

LSN     PREV_LSN     TRAN_ID     TYPE         PAGE_ID
---     --------     -------     -----        --------
1       -            -           begin CKPT   -
2       -            -           end CKPT     -
3       NULL         T1          update       P5
4       NULL         T2          update       P2
5       NULL         T3          update       P7
6       5            T3          update       P5
7       3            T1          update       P7
8       4            T2          update       P3
9       7            T1          commit       -
10      6            T3          abort        -
11      9            T1          end          -
-------- XXXXXXX --- FIRST CRASH! --- XXXXXXX --------
12      8            T2          abort        -
13      12           T2          CLR          P3 (undonextLSN=4)
-------- XXXXXXX --- SECOND CRASH! -- XXXXXXX --------
```

The log ends right after the second crash, before the second recovery procedure is initiated. Considering again that the system employs the ARIES recovery algorithm, answer the following questions.

1. What is the state of the *transaction table* after the analysis phase in the recovery from the first crash? What is the state of the transaction table after the analysis phase in the recovery from the second crash? Explain how these states differ and why.
2. What is the state of the *dirty page table* after the analysis phase in the recovery from the first crash? What is the state of the dirty page table after the analysis phase in the recovery from the second crash? Explain how these states differ and why.
3. At what LSN does redo start in the recovery from the first crash? What about in the recovery from the second crash? Explain how these LSNs differ and why.
4. In the scenario, the redo phase in the recovery from the first crash completes. Can we say which updates will cause actual writes to data pages in the redo phase of the recovery from the second crash? Explain why or why not.
5. Show the additional contents of the log after the recovery procedure completes after the second crash. Provide a brief explanation for why any new log records shown need to be added.

Question 3: Replication (LG7)

A startup team has launched a successful web service that is experiencing a spike in requests, and is challenged by how to scale their single-threaded implementation over multiple machines. The requests can be modeled as `READ`s and `WRITE`s over a memory abstraction. The startup team observes that most requests are `READ`s, and decides to employ synchronous read-any, write-all replication to scale their service.

You are contacted by the startup to help their team in this task. Gladly, you have been working on the implementation of a multicast library that may be a great fit for this challenge. Your multicast primitive delivers messages to groups of processes, and has the following function prototypes:

- `void rw-multicast(group g, message m)` for sending a message *m* to a group of processes *g*.
- `rw-deliver(message m, client c)` for handling a received message *m* at a given process *p*. If a response needs to be given to the received message *m,* then this response can be sent one-directionally to the requesting client library *c* by invoking a `send` primitive on *c*.

The primitive supports overlapping groups, and delivers messages respecting *both* global FIFO ordering and global total ordering (as defined in page 242 of the compendium). In addition, you may assume that the infrastructure at the startup can be modeled as a synchronous system, and that the primitive is reliable.

You are asked to model synchronous read-any, write-all replication as expected by the startup team with your multicast primitive. To achieve that goal, answer the following questions:

1. Show the pseudocode for the function prototypes (a) `value READ(key k)`, and (b) `WRITE(key k, value v)` for the *client library* of a memory abstraction employing synchronous read-any, write-all replication implemented by making use of `rw-multicast`. Show also the corresponding code for the `rw-deliver` handler at the *server side* for each of (a) and (b).
   NOTE: Be explicit about which groups you need and which processes are in each group.
2. Argue for the correctness of the design you explain in part 1 above. Why does your pseudocode correctly implements synchronous read-any, write-all replication?

Programming Task

In this programming task, you will develop a simplified *transaction / data manager* pair of abstractions in an archetypical inventory management scenario. Through the multiple questions below, you will describe your design (**LG1**), expose the abstraction as a service (**LG2**), design and implement this service (**LG4, LG6, LG7**), and evaluate your implementation with an experiment (**LG5**).

As with assignments in this course, you should implement this programming task in Java, compatible with JDK 8. As an *RPC mechanism*, you are only allowed to use Jetty together with XStream and/or Kryo, and we expect you to abstract the use of these libraries behind clean proxy classes as in the assignments of the course. You are allowed to reuse communication code from the assignments when building your proxy classes. You are also allowed to use skeleton code from Assignment 4 to orchestrate experimental workload and measurements.

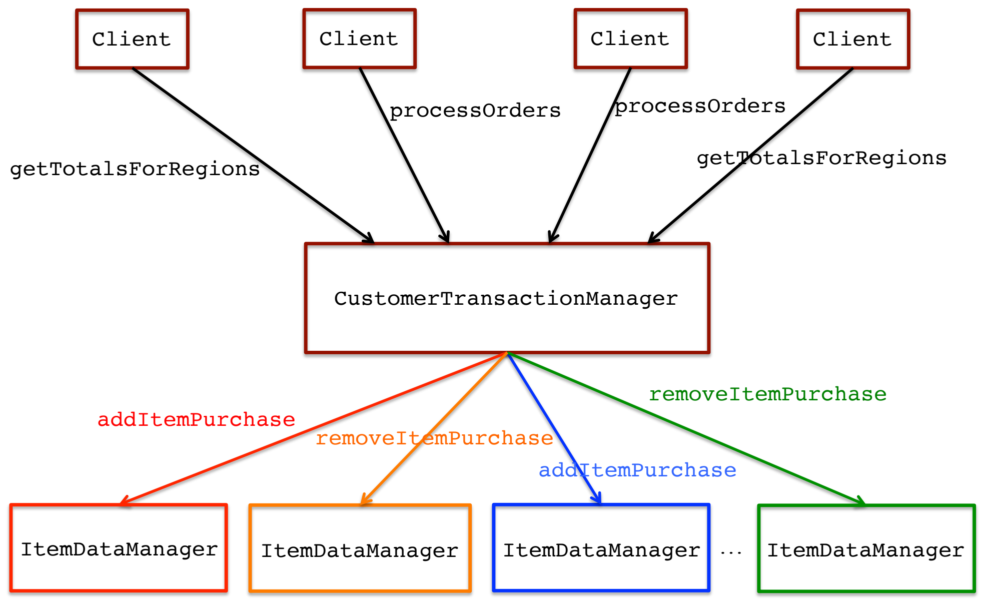
In contrast to a complex code handout, in this exam you will be given simple interfaces to adhere to, described in detail below. We expect the implementation that you provide to these interfaces to follow the description given, and to employ architectural elements and concepts you have learned during the course. We also expect you to respect the usual restrictions given in the qualification assignments with respect to libraries allowed in your implementation (i.e., Jetty, XStream, Kryo, JUnit, and the Java built-in libraries, especially `java.util.concurrent`).

The Inventory Management Scenario

For concreteness, we focus on an example of an inventory management system. The system is run by `acertaininventorymanager.com`, a fictitious firm that manages inventory as a service for stores that sell assorted goods. These stores, henceforth called *customers*, request delivery of goods kept by `acertaininventorymanager.com`. Customers belong to sales regions and interact with region managers at `acertaininventorymanager.com` to mediate their orders. These region managers use client programs that may operate disconnected, e.g., a mobile application, and eventually send a request for multiple customer orders to the inventory management system. Upon such a request, the inventory management system records the detailed items bought by customers along with the quantity ordered and unit price at the time of purchase for each item. In addition, the inventory management system updates the total monetary value spent by each customer in the platform. To better serve customers, `acertaininventorymanager.com`'s region managers also interact with the inventory management system to aggregate and analyze totals for their sale regions and plan future marketing actions.

Given that the number of detailed item purchase records is expected to far exceed the number of customers, the architects of `acertaininventorymanager.com` have decided to organize their system into two primary layers: an item data manager layer and a customer transaction manager layer. The item data manager layer keeps the detailed information for items purchased, and only provides single-element atomic operations. This layer of the inventory management system is distributed over a number of components and each such component exposes the `ItemDataManager` interface. The customer transaction manager layer provides higher-level atomic operations for processing orders from customers and aggregating sales region monetary totals. This layer is centralized into a single component, stores customer information, and exposes the `CustomerTransactionManager` interface. Note that both layers of the system are multi-threaded.

The overall organization of components in the `acertaininventorymanager.com`'s inventory management system architecture is summarized in the following figure.



In the figure, we can see one instance of `CustomerTransactionManager` and several instances of `ItemDataManager`. The client programs used by sales managers invoke functions on `CustomerTransactionManager` through RPC; the latter as a result invokes RPCs on `ItemDataManager` instances. As such, clients never interact directly with `ItemDataManager` instances.

Item purchase data is distributed across the multiple instances of `ItemDataManager` by a hash function *h* mapping item IDs to `ItemDataManager` instance numbers. For simplicity, you may assume in your implementation that the number of `ItemDataManager` instances is fixed, that instance numbers are assigned in the domain [*0, …, N-1*], and that a simple hash function, namely *h*(*itemID*) = *itemID* mod *N*, is sufficient to ensure good load balancing across `ItemDataManager` instances.

Your implementation should be focused on the `CustomerTransactionManager` and the `ItemDataManager` services. For testing and experimentation purposes, you will need to create programs that simulate the calls made to these services by client programs.

To limit the workload in this exam, all services are assumed not to provide durability. In other words, all data should be stored in data structures in main memory, and you do not need to implement a mechanism for durability for the methods of either the `CustomerTransactionManager` *or the* `ItemDataManager` *interfaces.*

The `CustomerTransactionManager` Abstraction

The `CustomerTransactionManager` service exposes as its API the following *operations*:

1) `processOrders(Set<ItemPurchase> itemPurchases)`: The operation processes orders for items from a number of customers under the conditions agreed with a sales manager. The operation must be atomic with respect to all other operations in the

`CustomerTransactionManager`. The sale conditions are recorded in individual item purchase records, which detail the order ID, the customer ID, the item ID, the quantity ordered, and the unit price agreed for the item. The operation validates the given item purchase records in a number of ways. First, the order ID is asserted to be a positive integer; the service otherwise assumes that order numbers are generated appropriately by clients. Second, the customer ID of each item purchase must refer to an existing `Customer` instance in the `CustomerTransactionManager` service and also be a positive integer. Third, the quantity and unit price for all item purchases must also be positive integers. This operation throws appropriate exceptions if any of these conditions are violated. Otherwise, the operation adds the item purchase records to the `ItemDataManager` instances corresponding to the item IDs given and then finally updates the total value bought for each affected customer, while ensuring all-or-nothing semantics at the application level. In other words, if a failure occurs in an `ItemDataManager`, any added purchase records are removed from other `ItemDataManager` instances, and `processOrders` throws an appropriate exception without applying any updates. NOTE: Given that durability is not provided by any of the services, you should *not* implement a commit protocol to coordinate the interaction of the `CustomerTransactionManager` and `ItemDataManager` instances.

2) `getTotalsForRegions(Set<Integer> regionIds)` → `List<RegionTotal>`: This operation calculates the total value bought for the customers of each of the regions given, and returns a list of these aggregate values per region. The operation must be atomic with respect to all other operations in the `CustomerTransactionManager`. The operation is read-only; it does not update any customer data. The operation includes the following validations: 1) The region IDs given are positive integers; and 2) Each region ID matches at least one customer. This operation throws appropriate exceptions if any of these conditions are violated.

As a constraint, the method you use for ensuring before-or-after atomicity at the `CustomerTransactionManager` must allow for concurrent read-only operations, i.e., two `getTotalsForRegions` operations executed in different threads must not block each other.

The `ItemDataManager` Abstraction

The `ItemDataManager` service exposes as its API the following *operations*:

1) `addItemPurchase(ItemPurchase itemPurchase)`: The operation adds an item purchase record to the `ItemDataManager` instance. The operation is atomic with respect to all other operations in the *same `ItemDataManager` instance*. Given that durability is not provided, the item purchase only needs to be recorded in a data structure in main memory. The operation assumes the input to have been validated externally, i.e., at the `CustomerTransactionManager`, and thus does not perform any further validation.
2) `removeItemPurchase(int orderId, int customerId, int itemId)`: The operation removes the item purchase identified by the given combination of order ID, customer ID, and item ID from the `ItemDataManager` instance. The operation is atomic with respect to all other operations in the *same `ItemDataManager` instance*. Similarly to `addItemPurchase`, the operation only needs to remove the item purchase from a data structure in main memory. The operation assumes the input to have been validated externally, i.e., at the `CustomerTransactionManager`. However, if an item purchase is not found for any reason with the parameters given, the operation raises an appropriate exception.

To reduce the workload in this exam, you do not need to perform any unit testing of your implementation of the `ItemDataManager` interface; rather, your unit tests should focus on correctness of your *implementation of the `CustomerTransactionManager`. Still, your implementation of the `CustomerTransactionManager` should interact correctly with `ItemDataManager` instances, and you should provide an implementation of the `ItemDataManager` interface along with arguments for correctness (see Questions 2 and 4 below).*

Failure handling

Your implementation only needs to tolerate failures that respect the *fail-stop* model: Network partitions *do not occur*, and failures can be *reliably detected*. We model the situation of components hosted in a single data center, and a service configured so that network timeouts can be taken to imply that the component being contacted indeed failed. In other words, the situations that the component was just overloaded and could not respond in time, or that the component was not reachable due to a network outage are just assumed *not to occur* in our scenario.

Since durability is not required in this exam, failures imply loss of the entire data stored in main memory at the affected component. However, individual components in your implementation must be isolated, and other components in the system should continue operating normally in case of such a failure. This strategy ensures that the `CustomerTransactionManager` follows a fail-soft design with respect to `ItemDataManager` instances.

Data and Initialization

As mentioned above, all the data for the `CustomerTransactionManager` and `ItemDataManager` services is stored in main memory. You may generate and load initial data for the services at class constructors according to a procedure of your own choice; only note that the distribution of values generated must make sense for the experiment you will design and carry out below.

You may assume that any configuration information is loaded once each component is initialized also at its constructor, e.g., naming information for the `CustomerTransactionManager` service needed to access the `ItemDataManager` instances. For simplicity, you can encode configuration information as constants in a separate configuration class.

High-Level Design Decisions, Modularity, Fault-Tolerance

First, you will document the main organization of modules and data in your system.

Question 1 (LG1, LG2, LG4, LG6, LG7): *Describe your overall implementation of the `CustomerTransactionManager` and the `ItemDataManager` interfaces. In addition to an overview of the organization of your code, include the following aspects in your answer:*

- *What RPC semantics are implemented between clients and the `CustomerTransactionManager`? What about between the `CustomerTransactionManager` and the `ItemDataManager`? Explain.*
- *How are failures of `ItemDataManager` instances contained at the `CustomerTransactionManager`? Why does your failure containment implementation turn out to guarantee a fail-soft design for the `CustomerTransactionManager` with respect to `ItemDataManager` instances?*
- *You have implemented the services under the assumption that durability is not provided as a property. Consider what would happen if the staff of `acertaininventorymanager.com` decided to migrate the implementation of both the `CustomerTransactionManager` and the `ItemDataManager` to utilize a two-level memory with durability provided by a recovery*

protocol such as ARIES at each component. Would there be a need then to implement a commit protocol among the several components? Explain why or why not.
(1 paragraph for overall code organization + 3 paragraphs, one for each of three points above)

Before-or-After Atomicity

Now, you will argue for your implementation choices regarding before-or-after atomicity. Our focus for this exam will be in the `CustomerTransactionManager` service, though we start for completeness by the `ItemDataManager` service.

Question 2 (LG1, LG4, LG6, LG7): *Each instance of the* `ItemDataManager` *interface executes concurrent addition and removal of item purchases. Describe how you ensured before-or-after atomicity of these operations. In particular, mention the following aspects in your answer:*

- *Which method did you use for ensuring serializability at each* `ItemDataManager` *instance? Describe your method at a high level.*
- *Argue for the correctness of your method; to do so, show how your method is logically equivalent to a trivial solution based on a single global lock or to a well-known locking protocol, e.g., a variant of two-phase locking.*
- *Argue for the performance of your method; to do so, explain your assumptions about the workload the service is exposed to and why you believe other alternatives you considered, e.g., different granularities of locking, would be better or worse than your choice.*

NOTE: The method you design for atomicity does not need to be complex, as long as you argue convincingly for its correctness, its trade-off in complexity of implementation and performance, and how it fits in the system as a whole.
(3 paragraphs, one for each point)

Question 3 (LG1, LG4, LG6, LG7): *The* `CustomerTransactionManager` *service executes operations originated at multiple clients. Describe how you ensured before-or-after atomicity of these operations. In particular, mention the following aspects in your answer:*

- *Which method did you use for ensuring serializability at the* `CustomerTransactionManager` *service (e.g., locking vs. optimistic approach)? Describe your method at a high level.*
- *Argue for the correctness of your method; to do so, show how your method is logically equivalent to a trivial solution based on a single global lock or to a well-known locking protocol, e.g., a variant of two-phase locking.*
- *Argue for whether or not you need to consider the issue of reads on predicates vs. multi-granularity locking in your implementation, and explain why.*
- *Argue for the performance of your method; to do so, explain your assumptions about the workload the service is exposed to and why you believe other alternatives you considered (e.g., locking vs. optimistic approach) would be better or worse than your choice.*

NOTE: The method you design for atomicity does not need to be complex, as long as you argue convincingly for its correctness, its trade-off in complexity of implementation and performance, and how it fits in the system as a whole. However, note that an implementation that provides more concurrency will be valued higher than one providing less.
(4 paragraphs, one for each point)

Testing

You will then describe your testing strategy, with a focus exclusively on the `CustomerTransactionManager` as mentioned above.

Question 4 (LG4, LG6): *Describe your high-level strategy to test your implementation of the* `CustomerTransactionManager`*. In particular, mention the following aspects in your answer:*

- *How did you test all-or-nothing atomicity of operations at the* `CustomerTransactionManager`*?*
- *How did you test before-or-after atomicity of operations at the* `CustomerTransactionManager`*? Recall that you must document how your tests verify that anomalies do not occur (e.g., dirty reads or dirty writes).*

(2-3 paragraphs; 1 paragraph for first aspect, 1-2 paragraphs for second aspect)

Experiments

Finally, you will evaluate the scalability of one component of your system experimentally.

Question 5 (LG5): *Design, describe the setup for, and execute an experiment that shows how well your implementation of the* `CustomerTransactionManager` *service behaves as concurrency in the number of clients is increased. You will need to argue for a basic workload mix involving calls from the clients accessing the service, including distribution of operations and data touched. Depending on the workload, you may wish to also scale the corresponding data sizes with increasing number of clients. Given a mix of operations from clients, you should report how the throughput of the service scales as more clients are added. Remember to thoroughly document your setup. In particular, mention the following aspects in your answer:*

- *Setup: Document the hardware, data size and distribution, and workload characteristics you have used to make your measurements. In addition, describe your measurement procedure, e.g., procedure to generate workload calls, use of timers, and numbers of repetitions, along with short arguments for your decisions. Also make sure to document your hardware configuration and how you expect this configuration to affect the scalability results you obtain.*
- *Results: According to your setup, show a graph with your throughput measurements for the* `CustomerTransactionManager` *service on the y-axis and the numbers of clients on the x-axis, while maintaining the workload mix you argued for in your setup and any other parameters fixed. How does the observed throughput scale with the number of clients? Describe the trends observed and any other effects. Explain why you observe these trends and how much that matches your expectations.*

(3 paragraphs + figure; 2 paragraph for workload design and experimental setup + figure required for results + 1 paragraph for discussion of results)