Advanced Computer Systems (ACS), 2017/2018

Final Exam

This is your final 5-day take home exam for Advanced Computer Systems, block 2, 2017/2018. This exam is due via Digital Exam on January 22, 2018, 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 y.wang@di.ku.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 car rental company BestCars maintains an operational database storing information on their customers, cars, as well as rental records.

The management of BestCars would like to analyze the interest of their customers in terms of the size of cars. In particular, they assign a bunch of data analysis tasks to you, the best data scientist and developer in BestCars. One of the tasks is as follows: "Produce a list of ids of the users who have only rent big cars, but not small cars."

Since the analysis is very heavy, to avoid any interference with the operational system, the necessary data for your task has been pre-extracted outside of the database and stored separately. For the aforementioned task, you are provided with two heap files storing two tables whose schemas are as follows: (1) *Cars(car_id, make, model, size_category)*, and (2) *Rentals(user_id, car_id, time)*. You can assume the values of *size_category* can be either "small" or "big". You are provided with one server with memory larger than the square root of any table, but insufficient to hold any one of them completely. This applies to any temporary tables that are generated by your algorithm. Therefore, you have to resort to external memory algorithms to solve the problem. If you need to make any further assumptions in your answers, please state them clearly as part of your solution. Please write down your solutions to the following questions.

1. State an external memory algorithm with the minimum cost in terms of I/O to answer the query above. Argue for the algorithm's correctness and efficiency.
2. State the I/O cost of the algorithm you designed. You can make your own notation of the size of each attribute in any table.

NOTE 1: 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 2: 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 1 we expect that you will need to show changes to steps of existing algorithms, if necessary.

NOTE 3: Your solution will be graded partly according to the optimality of your algorithm and the quality of the justifications. That is to say, a solution with justified lower I/O cost will in general help you get a better grade.

Question 2: Replication (LG7)

Three processes, P1, P2, and P3, replicate a common object. The object is updated asynchronously by three different clients, and each client happens to dispatch its update to a different process out of P1, P2, and P3. After receiving each update, the respective processes incorporates the update in its local replica and then propagates the new value to the other processes. The precise exchange of messages is shown in the following figure:

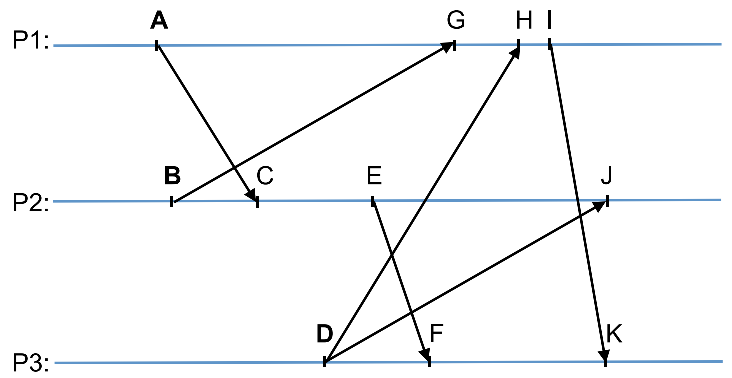

Figure 1: Exchange of messages among processes

Note that P1 got a client update at A, P2 at B, and P3 at D; the clients are omitted from the figure to avoid cluttering. The system uses vector clocks to keep track of the object version internally. You may assume that all local clock values start at zero at the beginning of the interaction shown. In addition, assume that *only updates from clients* cause an increment of a local clock, i.e., not every event causes an increment of the local clock except for original client updates. Therefore, (1,1,1) is the maximum vector clock possible in the given interaction. The latter policy ensures that, if update propagation reaches all processes, then we expect the object's version to converge as either overwriting or merging through a commutative associative function takes place.

Every time an update is received by a process *P*, the process chooses one out of the following actions to incorporate the update:

(A1) Keep *P*'s own value;
(A2) Overwrite *P*'s value with the latest value from the message;
(A3) Call an application-specific merge function to combine both values consistently.

For each point in the execution from A to K shown in Figure 1, give the value of the vector clock at the corresponding process before the update event is processed, the value of the vector clock received in the incoming message or sent in the outgoing message, the action taken by the process to incorporate the update, and the value of the vector clock after the update is processed. Use a table with the following format to report your answer:

Event	**Vector Clock at Process Before**	**Message Vector Clock**	**Action Taken by Process**	**Vector Clock at Process After**
A	…	…	…	…
B	…	…	…	…
…	…	…	…	…

In addition to the table, provide a paragraph with a brief justification for your reasoning for constructing the table, i.e., discuss when each action should be taken depending on the vector clock values at the given message and event.

Question 3: ARIES Recovery (LG7)

In the recovery scenario described in the following, we ask you to explain the functioning and justification behind 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, four transactions, T1, T2, T3, and T4, enter the system and perform various operations. The detailed log follows:

```
LOG

LSN     PREV_LSN     XACT_ID     TYPE         PAGE_ID  UNDONEXTLSN
---     --------     -------     ----         -------  -----------
1       -            -           begin CKPT   -        -
2       -            -           end CKPT     -        -
3       NULL         T1          update       P3       -
4       3            T1          update       P1       -
5       NULL         T2          update       P1       -
6       NULL         T3          update       P3       -
7       5            T2          update       P2       -
8       NULL         T4          update       P2       -
9       4            T1          update       P4       -
10      8            T4          commit       -        -
11      9            T1          abort        -        -
12      7            T2          abort        -        -
13      12           T2          CLR          P2       A
14      10           T4          end          -        -
15      11           T1          CLR          P4       B
16      15           T1          CLR          P1       C
17      13           T2          CLR          P1       D
18      16           T1          CLR          P3       E
19      18           T1          end          -        -
-------- XXXXXXX ------  CRASH! ------- XXXXXXX --------
```

A crash occurs at the end of the log as shown. Considering again that the system employs the ARIES recovery algorithm, answer the following questions and justify each of your answers.

1. In the log, the *undonextLSN* values for several CLRs are represented by the letters **A**-**E**. For each letter, state the value of the corresponding *undonextLSN* and why it has the given value.
2. What are the states of the *transaction table* and of the *dirty page table* after the analysis phase of recovery? Explain why.
3. Show the additional contents of the log after the recovery procedure completes. Provide a brief explanation for why any new log records shown need to be added.
4. In the system shown, let us assume that the database is *memory-resident*, i.e., upon start-up all contents of nonvolatile storage are loaded into volatile storage (which is large enough to cache everything). Thus, no page replacements are ever necessary during normal operation. Furthermore, assume plenty of log buffer space is available and no background page writes are performed. Under these conditions, answer the following:
    a. Which log records could trigger writes to stable storage during normal operation in the scenario above? Why?
    b. Following the ARIES log record structure introduced in the course for *update* records, is there any information that would *not* need to be written to stable storage in such a scenario? If so, explain which information and why. Otherwise, justify why all information in these records needs to reach stable storage.

Programming Task

In this programming task, you will develop a simplified edge-to-cloud *distributed interpreter* abstraction in a scenario inspired by emerging self-checkout supermarkets. Through the multiple questions below, you will describe your design (**LG1**), expose abstractions as a services (**LG2**), design and implement these services (**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 3 to orchestrate experimental workload and measurements as well as JUnit testing skeleton code from the assignments.

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 of 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 Self-Checkout Supermarket Scenario

We focus on a scenario inspired by recent developments in self-checkout supermarkets.[1,2] In this near-future scenario, a machine learning component is deployed at each store of the fictitious `acertainsupermarket.com` firm to automatically identify items that are added or removed from each customer's cart. These events trigger function calls to an *edge service* that keeps track of the state of the carts for the multiple customers. Many edge service instances are run to account for geographical distribution and reduce latency on queries to the cart state. As such, an edge service instance can support the activities of a number of nearby supermarkets. However, a supermarket interacts with one and only one edge service instance.[3] The edge service instances, in turn, invoke function calls on a centralized *cloud service* that keeps track of item prices and stocks to support the supply chain operation of the supermarket. These calls to the cloud service comprise queries for price and other details for single items as well as updates to item stocks.

Two kinds of client components interact with the edge service. In order to allow the customer to see the current status of her shopping, *mobile clients* query edge service instances. Moreover, mobile clients trigger checkout operations when the customer is ready to leave the store. These client components are expected to run on a customer's mobile phone and on tablet devices attached to physical shopping carts. In addition to the mobile clients, the machine learning components also act as *store clients* that add or remove items from the shopping cart state kept by edge service instances.

[1] Amazon Go store concept website (July 2017). https://www.amazon.com/b?node=16008589011.
[2] Planet Money Episode 730: Self Checkout. Online podcast (May 2017). http://www.npr.org/sections/money/2016/10/19/498571623/episode-730-self-checkout.
[3] Note that we make this assumption for simplicity in this exam, but it could be extended by organizing customer interactions into sessions and allowing for new sessions to be started in other edge service instance in case of failures.

Other than the edge service instances, a third kind of client interacts directly with the backend cloud service. These *admin clients* are used occasionally by the staff of the supermarket to query and update several items to replenish stocks.

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

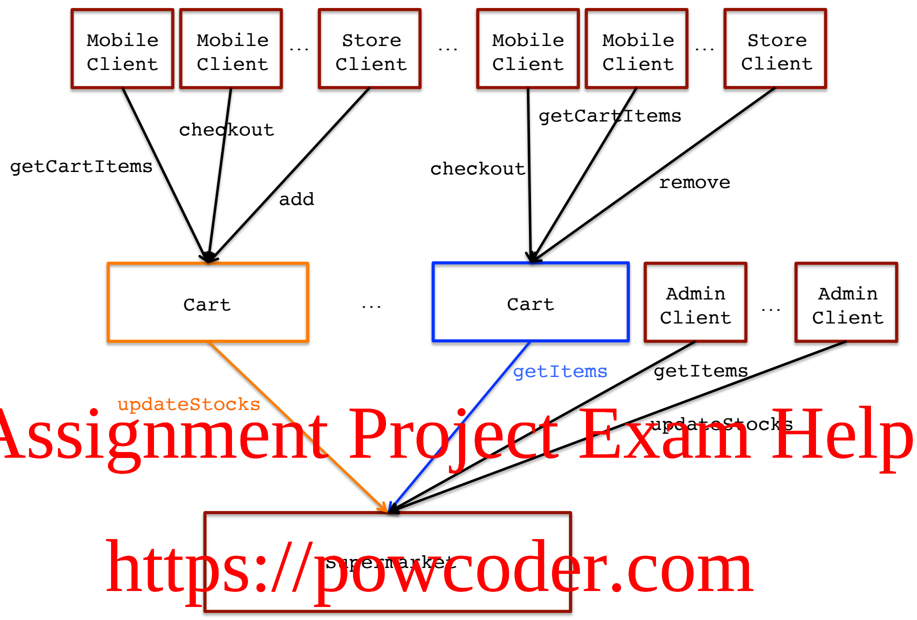

Figure 2: Organization of components in `acertainsupermarket.com`'s system architecture

The edge service layer exposes to clients the `Cart` interface, while the cloud service layer exposes to the edge service layer the `Supermarket` interface. For better use of computational resources, both service layers are multi-threaded, and we wish to make the implementations of the operations in both `Cart` and `Supermarket` behave atomically. Client programs invoke functions on `Cart` and `Supermarket` instances via RPC, and so do `Cart` instances on the `Supermarket` instance.

Your implementation should be focused on the `Cart` and `Supermarket` 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 `Cart` *or the* `Supermarket` *interfaces.*

The `Cart` Abstraction

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

1) `add(int cartId, int itemId)`: This function adds a unit of the item with given item ID to the cart identified by the provided cart ID. The operation must be atomic with respect to all other operations in the same `Cart` instance. The operation validates the cart ID to ensure that it is

nonnegative and among the IDs managed by this `Cart` instance. You may assume for the purposes of this exam that the collection of cart IDs handled by a given `Cart` instance is fixed and known a priori. Cart IDs are unique within a `Cart` instance, but are not required to be globally unique. Once validation of the cart ID succeeds, if the item is already among the items in the customer cart, then the quantity of the item is incremented by one. Otherwise, the operation looks up the item by calling `getItems` on the `Supermarket` service. This returns all the details of the item or raises an appropriate exception if the item ID is not valid. If the item has not yet been associated to the cart, then the item name and price are saved in the Cart instance along with a quantity of one as a `CartItem`, so as to ensure that no subsequent price changes are observed by the customer. Note that validation failures should raise an appropriate exception. Moreover, if an exception is raised by the call to the `Supermarket` service, this exception should be propagated further to the client.

2) `remove(int cartId, int itemId)`: This operations removes one unit of the item with given item ID from the cart identified by the provided cart ID. The operation must be atomic with respect to all other operations in the same `Cart` instance. The operation validates the cart ID in the same manner as the `add` operation, raising appropriate exceptions if needed. Moreover, the operation also raises an exception if the item is not associated to the cart. To make this operation behave as an inverse of add, two cases are handled. If the operation would make the quantity of the item drop to zero in the cart, then the corresponding `CartItem` instance is simply removed. Otherwise, the quantity of the item is decremented by one.
3) `getCartItems(int cartId)` → `List<CartItem>`: This operation lists the items in the cart identified by the given cart ID. The operation must be atomic with respect to all other operations in the same `Cart` instance. The operation validates the cart ID in the same manner as the operations above, raising appropriate exceptions if needed.

4) `checkout(int cartId)`: This operation performs the checkout of the current contents of the cart identified by the given cart ID. The operation must be atomic with respect to all other operations in the same `Cart` instance. The operation validates the cart ID in the same manner as the operations above, raising appropriate exceptions if needed. Afterwards, the operation invokes `updateStocks` in the `Supermarket` service to reduce the stocks for all items in the cart by their quantity. If the update is successful, then the function purges all items from the cart, leaving it empty. If an exception is raised by the call to the `Supermarket` service, however, then this exception should be propagated to the client.

As a constraint, the method you use for ensuring before-or-after atomicity at `Cart` instances must allow for concurrent read-only operations, i.e., two `getCartItems` operations executed in different threads must not block each other. In addition, solutions that allow for some concurrency of `getCartItems` with other operations in the interface will be valued higher than ones that do not.

The `Supermarket` Abstraction

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

1) `updateStocks(List<ItemDeltas> itemDeltas)`: This operation adds to or removes from the stocks of multiple items the amounts specified as differences to item quantities (`itemDeltas`). The operation must be atomic with respect to all other operations in the `Supermarket` service. Each item ID provided is validated by this operation to ensure that it is nonnegative and among the IDs managed by the `Supermarket` service. Appropriate exceptions must be raised upon validation failures.
2) `getItems(Set<Integer> itemIds)` → `List<Item>`: This operation returns the details of the items identified by the provided item IDs. The operation must be atomic with

respect to all other operations in the `Supermarket` service. Each item ID is validated as in the `updateStocks` operation, raising appropriate exceptions if needed.

As a constraint, the method you use for ensuring before-or-after atomicity at `Supermarket` instances must allow for concurrent read-only operations, i.e., two `getItems` operations executed in different threads must not block each other. In addition, solutions that allow for some concurrency among `updateStocks` and `getItems` operations will be valued higher than ones that do not.

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 `Cart` instances follow a fail-soft design with respect to the `Supermarket` instance. In other words, if the `Supermarket` instance fails, `Cart` instances should still be able to respond to queries on the state of shopping carts, but will raise exceptions when some operations, namely, item additions or checkout, are invoked.

Data and Initialization

As mentioned above, all the data for the `Cart` and `Supermarket` 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 needed by `Cart` service instances to access the `Supermarket` instance. 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 `Cart` and the `Supermarket` 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 `Cart` or `Supermarket` services? What about between the `Cart` and the `Supermarket`? Explain.
- How are failures of the `Supermarket` instance contained at the `Cart` instances? Why does your failure containment implementation turn out to guarantee a fail-soft design for the `Cart` instances with respect to the `Supermarket` instance?

- How did you achieve all-or-nothing atomicity at the `Cart` service, even in the cases where exceptions are propagated from the `Supermarket` service? How did you achieve all-or-nothing atomicity at the `Supermarket` service? Explain.

(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 for both the `Cart` and `Supermarket` services. *NOTE: For both questions below, the method you design for before-or-after 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. In general, implementations that provide for more concurrency will be valued higher than ones providing less, as long as it can be reasonably argued that the system can take advantage of the added concurrency.*

Question 2 (LG1, LG4, LG6, LG7): Each instance of `Cart` needs to provide for before-or-after atomicity on the operations of the interface. 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 `Cart` 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 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.

(4 paragraphs, one for each point)

Question 3 (LG1, LG4, LG6, LG7): The `Supermarket` 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 `Supermarket` 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.

(4 paragraphs, one for each point)

Testing

You will then describe your testing strategy for the two services, with a focus on before-or-after atomicity. *NOTE: While testing for all-or-nothing atomicity may help you eliminate errors from your implementation, affecting the evaluation of Question 1, these tests will not per se be evaluated for the exam. We will consider test code and arguments for the property of before-or-after atomicity only.*

Question 4 (LG4, LG6): Describe your high-level strategy to test your implementation of both the `Cart` and the `Supermarket` services. In particular, how did you test before-or-after atomicity of operations at both the `Cart` and the `Supermarket` services? Recall that you must document how your tests verify that anomalies do not occur (e.g., dirty reads or dirty writes).
(1-2 paragraphs)

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 `Supermarket` 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 `Supermarket` 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 paragraphs for workload design and experimental setup + figure required for results + 1 paragraph for discussion of results)