PCSD Assignment 1

This assignment is due via Absalon on December 1, 23:59. This assignment is worth 10 points total, as announced in the course description. While individual hand-ins will be accepted, we strongly recommend that this assignment be solved in groups of two students. Groups may at a maximum include three students.

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

Note that all homework assignments have to be submitted via Absalon in electronic format. It is your responsibility to make sure in time that the upload of your files succeeds. While it is allowed to submit scanned PDF files of your homework assignments, we strongly suggest composing the assignment using a text editor or LaTeX and creating a PDF file for submission. Email or paper submissions will not be accepted.

Learning Goals

This assignment targets the following learning goals:

- Discuss the design of a top-level abstraction in terms of lower level abstractions and naming. Explain strategies employed for correctness, performance, and fault tolerance.
- Identify and explain techniques for improving performance in computer systems.
- Implement functionality in a client-service design providing for strong modularity with an RPC mechanism based on an HTTP transport.
- Reflect on characteristics of such a client-service design, including performance and fault-tolerance as well as how strong modularity allows for applying performance improvements in a way that is transparent to clients.

Exercises

Question 1: Fundamental Abstractions

Design a memory abstraction that exposes the whole aggregated memory of a cluster of machines as a single address space. You do not need to consider

redundancy to mask failures. However, you need to consider that individual machines can fail and have your abstraction handle this in a clean way (i.e., no segmentation fault of the whole system when a machine is down :-)).

The memory abstraction you should design is schematically shown in Figure 1.

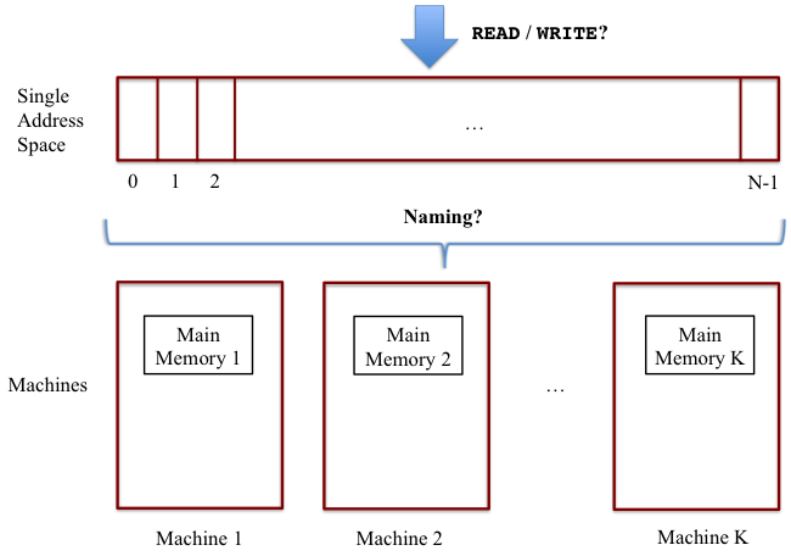

Figure 1: A top-level memory abstraction consolidating the lower-level memories of multiple machines in a cluster.

To phrase your solution, address the following:

1. Explain at a high level what technique you would use to map addresses at the single address space to address in each individual machine. Are there any centralized components in your design? How scalable is it as we increase the number of machines? What kinds of faults can the design tolerate? (2-3 paragraphs)

2. Show the pseudocode and API of the READ and WRITE functions of your memory abstraction. Note that your pseudocode should show both translation of addresses (naming) as well as communicate with individual machines to obtain the actual data. (for each function: API + 1-2 sentence description of API + pseudocode)

3. Are READ/WRITE operations against regular main memory atomic? Should the READ/WRITE operations against your memory abstraction be atomic? Discuss why or why not. If you believe they should be atomic,

argue how you would achieve that property when multiple clients are accessing your abstraction concurrently. If you believe that they should not be atomic, describe what abstraction you would need to provide to clients to implement atomicity when they need it. (1 paragraph)

4. One important aspect in the design of the naming scheme for this assignment is whether your name mapping strategy makes any assumptions on the number of machines in the cluster. For example, you may consider the number of machines K to be given and known a priori. Alternatively, you may consider that machines may enter and leave the system, and naming must adapt to these changes. Does the name mapping strategy you designed allow for dynamic joins and leaves of machines without making memory locations unavailable? If so, why? If not, what strategy could you use to achieve that property? (1 paragraph)

Question 2: Techniques for Performance

Regarding techniques to improve performance of a computer system, answer the following questions:

1. How does concurrency influence latency in a computer system? Is its influence always positive, always negative, or is there a trade-off? Explain. (2 paragraphs)

2. Explain the difference between dallying and batching. Provide one example of each. (2 paragraphs)

3. Is caching an example of a fast path optimization? Explain why or why not. (1 paragraph)

Programming Task

In this programming task, we provide you with a small system architected around an organization with clients and services, and implemented with a strong modularity mechanism based on RPCs. We ask you to understand the implementation of this system, extend its functionality following the same architecture, and then discuss and reflect on other possible extensions and performance optimizations that could be applied to the architecture.

The handout code released for this assignment will be used throughout Assignments 1-4. Each assignment explores *independent* implementation extensions starting from the same common code base. All code in programming tasks must be implemented in Java and be compatible with JDK 6.

A Certain Bookstore

You have just joined the team of `acertainbookstore.com`. The *bookstore* manages *books*, which are available for *clients* to query and buy. The bookstore offers

distinct functionality for external clients – the customers of the bookstore – and for internal clients – the store management personnel. These two client types are described as follows:

1. BookStore client - Clients which query about the books available in the book store and issue buy requests.

2. StockManager client - Clients which monitor the stocks of books in the bookstore and add books to the stock when necessary.

BookStore and StockManager clients can access the bookstore using client libraries to query/add/modify/buy books. The clients communicate with the server using remote procedure calls (RPCs) available through client libraries which are implemented using HTTP for communication [1].

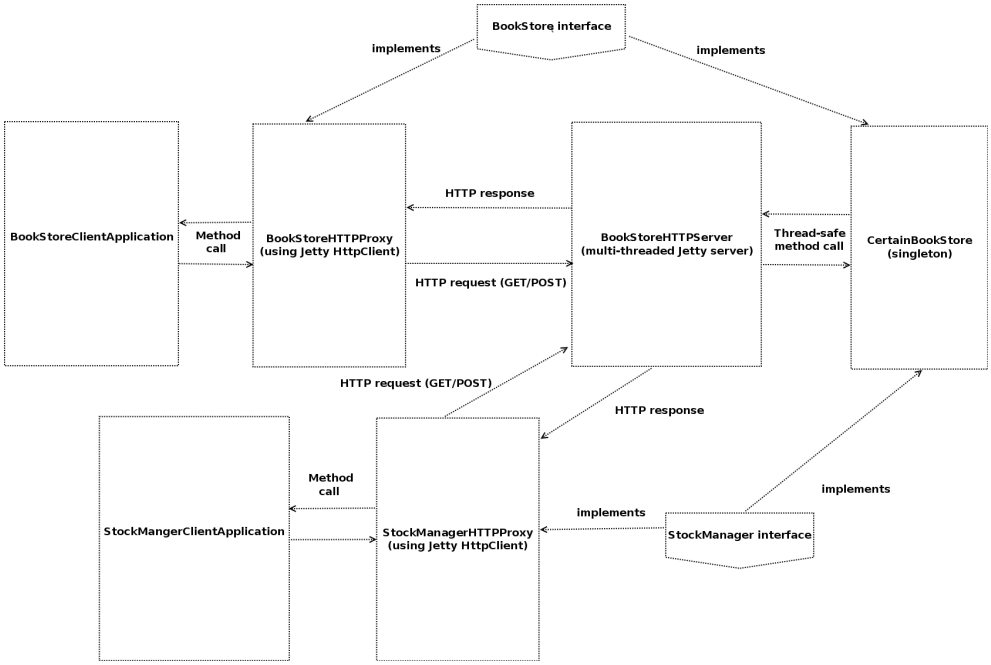

Figure 2: Architecture of bookstore application

The methods (RPC) available in the bookstore to BookStore clients and StockManager clients are defined in the *BookStore* and *StockManager* interfaces, respectively. The interfaces are implemented by the bookstore client library and

[1]The Jetty library (http://www.eclipse.org/jetty/) is used to provide the HTTP server and the HTTP client libraries. We also make use of the XStream library for serialization. The code provided works with Jetty version 8. Make sure you add the Jetty 8 and XStream 1.4.5 jar files in your buildpath (http://download.eclipse.org/jetty/stable-8/dist/ and https://nexus.codehaus.org/content/repositories/releases/com/thoughtworks/xstream/xstream/1.4.5/xstream-1.4.5.jar).

the store manager client library using the *BookStoreHTTPProxy* and *StockManagerHTTPProxy* classes respectively. The server implements both interfaces in the *CertainBookStore* class. The *CertainBookStore* class is a singleton[2] class whose methods are invoked by the handler class *BookStoreHTTPMessageHandler*, which multiplexes HTTP requests received by the Jetty HTTP server class *BookStoreHTTPServer*. The architecture of the application is outlined in Figure 2.

Interfaces

The *BookStore* interface exposes the following functionality:

- `buyBooks`: allows a client to buy several books identified by ISBN, given in a list. For each entry in the list, the stock of the book is decremented by the number of copies bought, assuming enough stock is available. The method performs basic validation of its inputs: If any of the books is not in the bookstore catalog or not in stock, then an exception is thrown. The method respects all-or-nothing semantics: either all books are acquired, or none. Note that all-or-nothing semantics are defined with respect to application-level logic errors for each method (checked exceptions), and not for unexpected errors coming from the runtime system (runtime exceptions). If a book cannot be acquired due to lack of stock, it is marked as having a sales miss. This information is used below to determine which books are in demand.

- `rateBooks`: allows a client to rate several books, identified by ISBN, given in a rating list. Ratings let users express their opinion about a book by an integer score between zero and five. The bookstore only keeps an aggregate of all ratings given by users per book, represented as a sum of ratings and a number of ratings given to the book. In other words, this method sums the rating given for a book to the sum of ratings for that book, and increments the number of ratings for the book. The method performs basic validation of its inputs, namely ISBNs and ratings, and respects all-or-nothing semantics.

- `getBooks`: allows a client to query the information for several books, identified by ISBN. This method is expected to be used by a client that obtained ISBNs for specific books by other query functionality. For all query methods in this interface, not all book information maintained by the bookstore is visible to clients, since some information is to be manipulated and visible only to store managers. In addition, clients should not be able to update books returned as query results. Other than providing protection against clients, the method performs basic validation of its inputs, namely ISBNs, and respects all-or-nothing semantics.

[2]Only a single object can be created for this class. This was chosen to simplify the programming task, but you are free to change the implementation if you want. If you change the implementation, make sure it is thread-safe.

- `getTopRatedBooks`: allows a client to query for the top-K books according to rating, for a given positive integer K. As with other query methods, the method provides basic protection against clients with respect to information access and updates, performs basic validation of its inputs, namely the parameter K, and respects all-or-nothing semantics.

- `getEditorPicks`: allows a client to query for K books marked as editor picks, for a given positive integer K, selected at random. Editor picks are books explicitly marked by editors as recommended. The bookstore does not provide differentiated ranking to markings from different editors, or from the same editor. As with other query methods, the method provides basic protection against clients with respect to information access and updates, performs basic validation of its inputs, namely the parameter K, and respects all-or-nothing semantics.

The *StockManager* interface exposes the following functionality:

- `addBooks`: allows a client to add new books to the collection of books offered by the bookstore. As with the methods in the *BookStore* interface, this method performs basic validation of its inputs, and respects all-or-nothing semantics.

- `addCopies`: allows a client to increase the number of copies in stock for several books, identified by ISBN. This method performs basic validation of its inputs, and respects all-or-nothing semantics.

- `getBooks`: allows a client to query the current state of all books managed by the bookstore. The method respects all-or-nothing semantics, and does not have any input parameters.

- `getBooksInDemand`: allows a client to query all books for which there were missed sales due to lack of stock, as flagged by `buyBooks`. In other words, the books in demand are the books for which there were any sales misses. The method respects all-or-nothing semantics, and does not have any input parameters.

- `updateEditorPicks`: allows a client to either flag or unflag several books, identified by ISBN, as being editor picks. This method performs basic validation of its inputs, and respects all-or-nothing semantics.

Implementation

Most of the methods in the interfaces above are already implemented in the *CertainBookStore* class, and they are exposed as RPCs according to the architecture of Figure 2. However, a few methods are missing, namely `rateBooks`, `getTopRatedBooks`, and `getBooksInDemand`. Your first task as part of the `acertainbookstore.com` team is to add this missing functionality to their service, while respecting the overall architecture.

Create additional tests

At `acertainbookstore.com`, test-driven development is very much encouraged. We have provided you with a set of *basic* JUnit tests in the package `com.acertainbookstore.client.tests`. Study these tests carefully and extend them with further test cases that cover untested aspects of the functionality of the service in general, but particularly of the functionality you are going to implement below. You should document any extra tests you add and explain their purpose as part of your solution text. The test cases given in the handout code only perform *basic* tests and are intended to be extended by you.

NOTE: While we set up our tests with HTTP proxy instances of our interfaces, the test code is oblivious to communication and RPCs. As a consequence, you may run local tests in the same JVM by changing test setup to use the *CertainBookStore* singleton instead of the HTTP proxy instances. These tests may be helpful to validate the core functionality implemented in the *CertainBookStore* class, but naturally running tests in this way will not exercise your RPC communication code. Make sure to also test this part of the code by using the appropriate HTTP proxy instances.

Implement `rateBooks` and `getTopRatedBooks` functionality

- Implement the stubbed out method `rateBooks` in the `com.acertainbookstore.business.CertainBookStore` class. Note that `rateBooks` must validate ISBNs and ratings, as well as validate that all books are in the bookstore collection. For inspiration, study the `buyBooks` method.

- Implement the stubbed out method `getTopRatedBooks` in the `com.acertainbookstore.business.CertainBookStore` class. Note that `getTopRatedBooks` must validate the `numBooks` parameter (referred to as K above). In addition, in order to honor protection, the function should return instances of `com.acertainbookstore.business.ImmutableBook`. For inspiration, study the `getEditorPicks` method.

- Make these methods available to the service via RPC by implementing the corresponding HTTP communication code in `com.acertainbookstore.server.BookStoreHTTPMessageHandler` and `com.acertainbookstore.client.BookStoreHTTPProxy`. For inspiration, study the code in these classes.

Implement `getBooksInDemand` functionality

- Implement the stubbed out method `getBooksInDemand` in the `com.acertainbookstore.business.CertainBookStore` class. In order to honor protection, the function should return instances of `com.acertainbookstore.business.ImmutableStockBook`. For inspiration, study the `getBooks` method.

- Make this method available to the service via RPC by implementing the corresponding HTTP communication code in `com.acertainbookstore.server.BookStoreHTTPMessageHandler` and `com.acertainbookstore.client.StockManagerHTTPProxy`. For inspiration, study the code in these classes.

Questions for Discussion on Architecture

In addition to the implementation above, reflect about and provide answers to the following questions in your solution text:

1. We have stated above that the architecture achieves strong modularity. In which sense is the architecture strongly modular? What kind of isolation and protection does the architecture provide between the two types of clients and the bookstore service? Is the same kind of isolation enforced when we run clients and services locally in the same JVM, as possible through our test cases? Explain.

2. Is there a naming service in the architecture? If so, where is it? Describe the naming mechanism that allows clients to discover and communicate with services.

3. We have studied three types of RPC semantics: at-least-once, at-most-once, and exactly-once semantics. What RPC semantics is implemented in the architecture? Justify.

4. Services employing HTTP as a communication mechanism often deploy web proxy servers for scalability in the number of simultaneous client connections. Is it safe to use web proxy servers with the architecture of Figure 2? If so, explain why this is safe and describe in between which components these proxy servers should be deployed. If not, why not?

   NOTE: Sometimes, web proxy servers are also used for caching. Assume for this specific question that *no* caching is employed, but the web proxies would only be used for scalability in the number of connections.

5. Given the discussion in the question above, consider now the following question: Is/are there any scalability bottleneck/s in this architecture with respect to the number of clients? If so, where is/are the bottleneck/s? If not, why can we infinitely scale the number of clients accessing this service?

6. Suppose the server-side of the architecture fails by a crash of server in which our *CertainBookStore* class is run. Would clients experience failures differently if web proxies were used in the architecture? Could caching at the web proxies be employed as a way to mask failures from clients? How would the use of web caching affect the semantics offered by the bookstore service? Explain.