ACS Programming Assignment 1

This assignment is due via Absalon on November 27, 23:59. While individual hand-ins will be accepted, we strongly recommend that this assignment be solved in groups of maximum two students. NOTE: The KU IDs of ALL group members MUST be stated on a separate `group.txt` file to ensure all group members get feedback and get the assignment accounted for in Absalon.

A well-formed solution to this assignment should include a PDF file with answers to all 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:

- 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.

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 Programming Assignments 1-3 and Exercise 4. Each assignment explores *independent* implementation

extensions starting from the same common code base. In particular, you do not need a working solution to Programming Assignment 1 in order to prepare solutions to Programming Assignments 2 or 3. All code in programming tasks must be implemented in Java.

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. The Jetty 9.4.11 library is used to provide the HTTP server and the HTTP client libraries. XStream 1.4.10 and Kryo 4.0.2 are used to serialize/deserialize objects in XML or binary format, respectively depending on a configurable parameter in `BookStoreConstants.java`. JUnit 4.12 is used as the unit test library. To make things easier, the necessary jars have been provided in the lib directory of the code handout. Make sure to add the jars in the lib directory in the classpath in case you do not use the supplied build.xml (ant) or .classpath file. You need to download and install a JDK though (JDK 10 is the recommended version). Please note that the handout code has been tested with Java 10 and the libraries mentioned. Be careful if you use an older/newer JDK and run into compatibility issues.

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 the store manager client library using the *BookStoreHTTPProxy* and *StockManagerHTTPProxy* classes respectively. The server implements both interfaces in the *CertainBookStore* 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 1.

Interfaces

The *BookStore* interface exposes the following functionality:

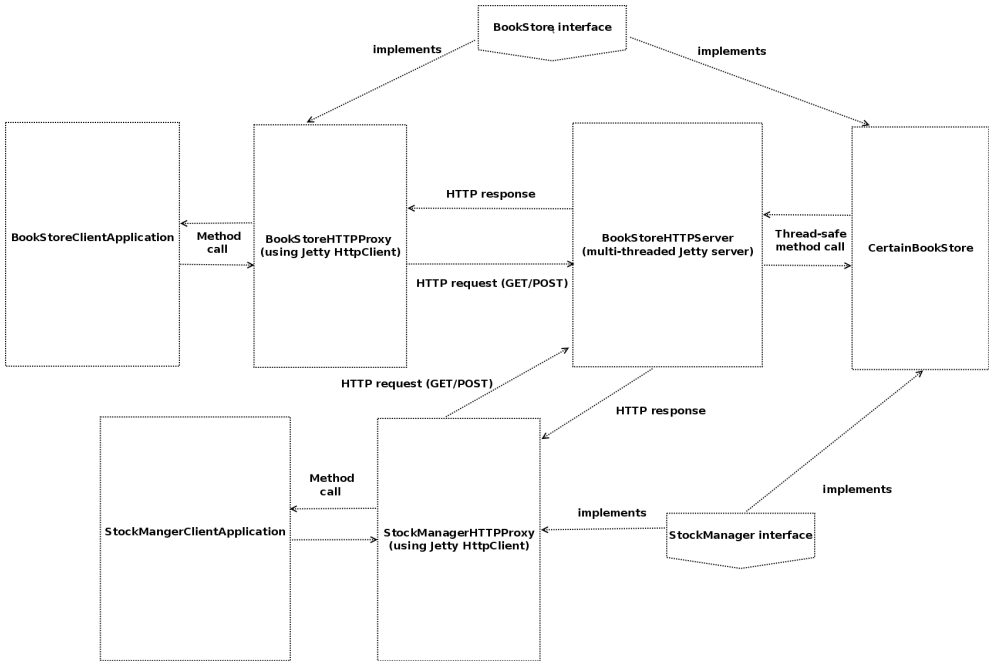

Figure 1: Architecture of bookstore application

- `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 does not store the particular instance of the buy request that was not satisfied, but provides a coarse indicator to the books that must be replenished by the store manager. 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.

- `getTopRatedBooks`: allows a client to query for the top-K books according to average rating, for a given positive integer K. The average rating is computed by dividing the aggregate rating by the number of ratings. 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. Editor picks are books explicitly marked by editors as recommended. The books returned as result are selected uniformly at random from the books that are marked as editor picks. 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. It also resets the sales miss indicator for the book whose copies have been added. For simplicity, we are not storing the history of sales misses to check if a previous buy request can be satisfied after the copies for a particular book have been replenished.

- `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 which incurred a sales miss. 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.

- `getBooksByISBN`: allows a client to query the current state of several books, identified by ISBN. The method performs basic validation of its inputs, and respects all-or-nothing semantics.

- `removeAllBooks`: allows a client to reset the current state of the bookstore to an empty one. The method respects all-or-nothing semantics, and does not have any input parameters.

- `removeBooks`: allows a client to delete certain books from the bookstore collection identified by their ISBN. The method performs basic validation of its inputs, and respects all-or-nothing semantics.

The view of the book collection returned by the BookStore interface is not the same as the one returned by the StockManager interface.

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 1. 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 (in the integration test spirit) 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* class 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

- In the `com.acertainbookstore.business.CertainBookStore` class, implement the stubbed out method `rateBooks`. 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.
- In the `com.acertainbookstore.business.CertainBookStore` class, implement the stubbed out method `getTopRatedBooks`. 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.
- The `com.acertainbookstore.server.BookStoreHTTPMessageHandler` as well as the `com.acertainbookstore.client.BookStoreHTTPProxy` classes encapsulate HTTP communication code. Make the methods above available to the service via RPC by implementing the corresponding code in these classes. For inspiration, study the code of other methods already implemented there.

Implement `getBooksInDemand` functionality

- In `com.acertainbookstore.business.CertainBookStore`, implement the stubbed out method `getBooksInDemand`. In order to honor protection, the function should return instances of `com.acertainbookstore.business.ImmutableStockBook`. For inspiration, study the `getBooks` method.
- The `com.acertainbookstore.server.BookStoreHTTPMessageHandler` as well as `com.acertainbookstore.client.StockManagerHTTPProxy` classes encapsulate HTTP communication code. Make the method above available to the service via RPC by implementing the corresponding code in these classes. For inspiration, study the code of other methods already implemented there.

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. Provide a short description of your implementation and tests, in particular focusing on:
   (a) How does your implementation and tests address all-or-nothing semantics?
   (b) How did you test whether the service behaves according to the interface regardless of use of RPCs or local calls?
2. We have stated above that the architecture achieves strong modularity. Explain this in the context of the following questions.

(a) In which sense is the architecture strongly modular?

(b) What kind of isolation and protection does the architecture provide between the two types of clients and the bookstore service?

(c) How is enforced modularity affected when we run clients and services locally in the same JVM, as possible through our test cases?

3. (a) Is there a naming service in the architecture? If so, what is its functionality?

   (b) Describe the naming mechanism that allows clients to discover and communicate with services.

4. 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 your answer.

5. Services employing HTTP as a communication mechanism often deploy web proxy servers for scalability in the number of simultaneous client connections.

   (a) Is it safe to use web proxy servers with the architecture of Figure 1?

   (b) 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.

6. Given the discussion in the question above, consider now the following questions:

   (a) Is/are there any scalability bottleneck/s in this architecture with respect to the number of clients?

   (b) If so, where is/are the bottleneck/s? If not, why can we infinitely scale the number of clients accessing this service?

7. Suppose the server-side of the architecture fails by a crash of the server machine where the *CertainBookStore* class is being run. In this context, explain the following.

   (a) Would clients experience failures differently if web proxies were used in the architecture?

   (b) Could caching at the web proxies be employed as a way to mask failures from clients?

   (c) How would the use of web caching affect the semantics offered by the bookstore service?