Advanced Computer Systems

Programming Assignment 1

University of Copenhagen

Julian Sonne Westh Wulff <bkx655@alumni.ku.dk>

Toke Emil Heldbo Reines <bkx591@alumni.ku.dk>

December 2, 2021

1 Implementation and tests

`rateBooks`

We implemented `rateBooks` with respect to the all-or-nothing semantics by first having the method validating the input is not null, and if the input is null it throws an exception. If the input is not null the method then loops through all the book ratings and validates each of them with respect to the API given in the assignment. Which for `rateBooks` is validating that the rating is in the range $[0, 5]$ and that the ISBN is valid. If any of the book ratings in the input fails any of the above validations the method throws an exception, thus adhering to the all-or-nothing semantics. Lastly, if all book ratings is successfully validated, then all the book ratings a looped through and the corresponding book in the `bookMap` is updated accordingly.

We test that our implementation of the `rateBooks` method addresses the all-or-nothing semantics with 3 separate tests. One test checks that a set of more than one valid book rating is successful, by checking that the rated books received their expected ratings. The two other tests separately check that a set of some valid book ratings and some invalid does not actually change any of the ratings for both the valid and invalid ratings. Here the invalid book ratings are made using a rating outside the range $[0, 5]$ for one of the tests and with invalid ISBNs for the other. Besides these 3 tests, we also created one additional test to check that the average rating was calculated correctly.

`getTopRatedBooks`

We implemented `getTopRatedBooks` with respect to the all-or-nothing semantics, similar to the `rateBooks` by throwing an error on invalid input, thus returning nothing on an error or else all. We tested `getTopRatedBooks` with 4 separate tests. 3 for testing the correctness of the behavior of the method given different inputs. Thus one test uses a negative number, one with a small positive

number and one with a number larger than the number of books in the store. Whilst the last test simply test for the correct behavior when the method is called in between a new rating that changes the order of top-rated books.

`getBooksInDemand`

We tested `getBooksInDemand` with 2 different test. One that checks whether the method returns the expected books in demand, and one that tests that no books are returned when no books are in demand.

Generally

To test whether the service behaves according to the interface regardless of using RPCs or local calls, we made sure to run each of the test classes twice with the `localTest` variable configured to true in one run and false in the other run.

Besides the test described above, we also extended the existing tests to uncover untested aspects of the functionality. We extended the existing tests with the following:

- `BookStoreTest`
  - `testGetEditorPicks`: Tests that editor picks can be retrieved.
  - `testGetNegativeKEditorPicks`: Tests that editor picks cannot be retrieved for a negative K.
  - `testGetLargeKEditorPicks`: Test all editor picks is retrieved for K larger than the number of editor picks.
- **StockManagerTest**
  - `testGetBooks`: Checks the method returns the current state of all books in the system.
  - `testEditorPickInvalidISBN`: Tests that editors picks respects the all-or-nothing semantics on invalid ISBNs.
  - `testRemoveBooksAllOrNothing`: Tests that `removeBooks` respect the all-or-nothing semantic.
  - `testGetBooksByISBNAllOrNothing`: Tests that `getBooksByISN` respects the all-or-nothing semantics.

2 Strong modularity architecture

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

The architecture is strongly modular because the read/write operations are isolated in the CertainBookStore module. Any client library could be built on top of it - in this case, we have a stock manager and a regular consumer.

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

The architecture provides isolation and protection with respect to what kind of read and writes operations are allowed for the different types of clients, through the usage of interfaces.

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

Modularity is no longer enforced through the usage of HTTPProxies. The architecture is still strongly modular since all logic is contained in the bookstore, but any client can spin up their own instance of the bookstore.

3 Naming Service

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

Yes. The naming service maps a function call to the called object, such that the caller, the client proxy, can communicate with the right object, the server.

4 RPC semantics

The RPC semantic used in the implementation is the at-most-once RPC semantic. This means that making a combined order of valid books in stock and one or more invalid books, results in an invalid order, and no supply-counter is therefore decremented. The same goes for all batch-write operations (addBooks, buyBooks, updateEditorPicks...) - all input actions (objects) are validated before anything is written/modified.

5 HTTP & web proxy servers

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

Yes

(b) If so, explain why this is safe and describe in between which components these proxy servers should be deployed. If not, why not?

As the web proxy servers are only used in the architecture as a middle layer that handles communication between the server and clients used for scalability purposes without caching the is no risk of the server responding with dirty data.

6 Bottlenecks

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

Yes

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

Bottlenecks in the architecture could be present in either the application/proxy layer or the server layer which would depend on what kind of resources is available to the different layers. Though it seems most reasonable to suspect that the server layer would be the bottleneck as this layer does the majority of the work

7 Robustness

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

The book store would require atomicity at the service level. In this case, the crash is on the server where the CertainBookStore class is being run. Calling the crashed server directly would result in a response from the client's HTTP Client, probably something like a "404 not found". Calling a running proxy could result in a different response, as the proxy would call the crashed server, get the "404 not found" and then maybe modify the response. The proxy could also have a built-in retry logic, resulting in higher latency for the client before receiving a "404 not found" response.

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

Caching could mask the failure for the client only when the client's requests are GET (read) requests. Any POST would fail.

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

The bookstore service offers an at-most-once RPC service, and web caching would not change that fact. There would be race conditions between reads and writes (`getBooks` / `addBooks`). Calls to `getBooks()` might give back false positives about their stock, and then subsequently `buyBooks()` calls, of those books, might fail.