Advanced Computer Systems

Assignment 1

Hans J. T. Stephensen
Department of Computer Science
University of Copenhagen
xkv467@alumni.ku.dk

Casper B. Hansen
Department of Computer Science
University of Copenhagen
fvx507@alumni.ku.dk

November 24, 2015

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Contents

1 Fundamental Abstractions

1.1 Question 1

Making no assumptions on the size of memory machines, given an address in our single address space we require a lookup table of some sort to pass on the request to the correct machine storing the data of that address. A simple mapping from an address in the single address space to an address in on of the k machines, would be storing the first $0 \dots m_1 - 1$ address in the first machine, m_i being the space of the i'th machine, and the next addresses $m_1, m_1 + m_2 - 1$ in the next machine. To do this we require a centralized machine with a lookup table to find the appropriate machine. For small k, a linear search would be fine. For larger k, one way is to use a binary search. This has the potential to increase the latency significantly, since each request would take logarithmic time.

If the central machine breaks, everything breaks. To alleviate this we could add redundancy by having multiple machines play the role of serving cached versions of the lookup table. This solution could also be implemented by having a set of proxies relaying requests to the appropriate machine. This type of redundancy breaks the atomicity of the write operation, since a write operation, followed by a read operation might be send to different proxies, and processed at a rate in which the `READ` reply arrives first.

If one of the k machines in not available (busy, down or broken), the central machine(s) could implement a timeout timer for when the reply from the machine should have arrived and reply to the client that the request failed with a timeout. Replies that arrived too late should be discarded.

1.2 Question 2

```
1  READ(ADDR) -> VALUE
2      let M_ADDR = proxy.lookup(ADDR)
3      read_request(M_ADDR, ADDR)
4      let response = wait_for_reply(M_ADDR, CONST_TIMEOUT)
5      case response of
6          exception e -> raise request_failure (to_string (e) )
7          value -> return value
8      end
9  end
10
11 WRITE(ADDR, VALUE)
12     let M_ADDR = proxy.lookup(ADDR)
13     write_request(M_ADDR, ADDR, VALUE)
14     let response = wait_for_reply(M_ADDR, CONST_TIMEOUT)
15     case reponse of
16         exception e -> raise request_failure (to_string (e) )
17         OK -> return
18     end
19 end
```

In each of the functions, we first lookup the IP of the machine storing the memory at the given address, send a reply and wait for the reply before returning. The `wait_for_reply` function could contain a loop looking for a reply in the reply buffer. This loop breaks if time `CONST_TIMEOUT` has passed. Similarly, we catch any exceptions that the machine might throw at us — thus ensuring that higher level abstractions can handle any failure that may occur.

1.3 Question 3

In regular main memory atomicity is required. Had it not been, any higher level abstractions that use main memory —such as the example discussed— could not be designed for atomicity, if required by such a design. In our design we believe that atomicity is key if we are to expect consistency in an environment with multiple clients. In this case, we could extend the implementation of our request API with locking mechanisms, that block reads from taking place whenever

a write request is received. In the event that we do not require strict consistency, we could allow this proxy method, where some policy loosely allow for acceptable irregularities within some constraint. Whilst if we do require such strict consistency, we must either use just one main machine that process requests, ensuring that the lookup table is always up-to-date, and reads cannot occur if the *write flag* is set on that particular machine.

1.4 Question 4

If a machine is unreachable we are covered since we raise a timeout exception, and since the data is unrecoverable in any case we can do little better. For efficiency, the central machine would keep track of unreachable machines, such as a flag in the lookup table. If new machines are added we can simply extend the single address space and lookup table with the new machine. The lookup table may become increasingly cluttered with unreachable machines. There's numerous ways to handle this (remapping addresses, moving data, etc.) each appropriate for different applications with different advantages and drawbacks.

2 Techniques for Performance

2.1 Question 1

Under the assumption that the workload is balanced evenly across processes, then latency is expected to decrease. Conversely, if workload is small, the overhead of doing the work concurrently outweighs the benefits.

2.2 Question 2

Dallying builds up buffer of operations to run creating an artificial delay before performing them all, whilst batching executes immediately, but ordering the operations to run batches of them together. Thus for batching preservation of the order is not guaranteed.

An example of dallying is OpenGL, that builds up a command buffer, which upon flushing the buffer sends the buffer to the graphics processing unit for execution.

An example batching is the scan algorithm in a classic hard disks. The read/write head moves up and down reversing direction at the top and bottom, reading and writing to the disk only when it passes over the area.

2.3 Question 3

Caching is indeed a case of fast path optimization, as it attempts to eliminate the need reads and writes to lower level memory, and only travels deeper if it needs to.

3 Discussion on Architecture

3.1 Question 1

a)

We follow the *all-or-nothing* semantics by way of exceptions; whenever an exception occurs, nothing is changed. All checks occur beforehand, and only when everything has been validated we perform the requested action. With regards to testing this, our tests are oriented towards failing by producing such an exception for cases where it can fail.

For a description of the implementation, please refer to the *Programming Tasks* section.

b)

...

3.2 Question 2

a)

The architecture is strongly modular because it is based on message tags, which allows different stores to handle each request differently. There is a very clear separation between messages and the action being performed, which greatly increases the overall extensibility and ease of maintenance of the system. Another powerfull advantage of the modularity in this design is that it also allows us to implement features that aren't necessarily exposed to the client or business, before all components work properly, by way of simply not exposing the message tag. Hence we can test in a live environment, without risking the client or business to be affected by any anomalies as we do so.

b)

There is a clear separation between the client and the service, as neither communicates directly with eachother. Both rely on the server to receive and process each and every request performed by either.

c)

Yes it is, as there is no way of communicating directly between the client and the bookstore. All messages are passed through the locally instantiated server. The locality has no effect on the outcome.

3.3 Question 3

a)

3.4 Question 4

In the architecture, this can be seen from the way HTTP communication is handled. Clients send and receive messages to the bookstore using the Jetty 8 HTML client/server library, wrapped in the `SendAndRecv` method. Since the client does not attempt to resend messages, and since the bookstore servers plan for handling errors is simply to pass the on to the client, the architecture exhibits an *"At most once"* behavior.

3.5 Question 5

3.6 Question 6

3.7 Question 7

4 Programming Tasks

In order to implement the functionality, we added the message tags for the desired functions to `BookStoreMessageTags` — namely `GETBOOKSINDEMAND`, `RATEBOOKS` and `GETTOPRATEDBOOKS`, as shown in excerpt 1.

```
1     LISTBOOKS,
2     ADDCOPIES,
3     GETBOOKS,
```

Figure 1: Code excerpt of *../utils/BookStoreMessageTag.java*, lines 12–14

4.1 `rateBooks`

The implementation of `rateBooks` is rather straight forward. First we check that the argument isn't null. Then, for each of the ratings passed to us, we validate that the ISBN is valid and contained within the book map, and lastly that the rating is actually valid. If any of these fails, we throw an exception. If not, we proceed to perform the rating.

```
1     public synchronized void rateBooks(Set<BookRating> bookRating)
2             throws BookStoreException {
3         if (bookRating == null) {
4             throw new BookStoreException(BookStoreConstants.NULL_INPUT);
5         }
6
7         // validate
8         int ISBN, rating;
9         BookStoreBook book;
10        for (BookRating bookRatingToRate : bookRating) {
11            ISBN = bookRatingToRate.getISBN();
12            rating = bookRatingToRate.getRating();
13
14            // validate ISBN
15            if (BookStoreUtility.isInvalidISBN(ISBN))
16                throw new BookStoreException(BookStoreConstants.ISBN + ISBN
17                        + BookStoreConstants.INVALID);
18
19            // validate in store
20            if (!bookMap.containsKey(ISBN))
21                throw new BookStoreException(BookStoreConstants.ISBN + ISBN
22                        + BookStoreConstants.NOT_AVAILABLE);
23
24            // validate rating
25            if (BookStoreUtility.isInvalidRating(rating))
26                throw new BookStoreException(BookStoreConstants.RATING + rating
27                        + BookStoreConstants.INVALID);
28        }
29
30        // rate all books
31        for (BookRating bookRatingToRate : bookRating) {
32            book = bookMap.get(bookRatingToRate.getISBN());
33            rating = bookRatingToRate.getRating();
34            book.addRating(rating);
35        }
36
37        return;
38    }
```

Figure 2: Code excerpt of *../business/CertainBookStore.java*, lines 332–370

Now that the functionality is implemented, we can add the message handling code.

Firstly, we parse the serialized XML from the received POST request. Then we attempt to perform the rating using the set of book ratings. If this fails we catch the generated exception. And lastly, if all went well, we respond accordingly.

```
            case RATEBOOKS:
                xml = BookStoreUtility.extractPOSTDataFromRequest(request);
                Set<BookRating> ratingSet = (Set<BookRating>) BookStoreUtility
                        .deserializeXMLStringToObject(xml);

                bookStoreResponse = new BookStoreResponse();
                try {
                    myBookStore.rateBooks(ratingSet);
                }
                catch (BookStoreException ex) {
                    bookStoreResponse.setException(ex);
                }
                listBooksxmlString = BookStoreUtility
                        .serializeObjectToXMLString(bookStoreResponse);
                response.getWriter().println(listBooksxmlString);
                break;
```

Figure 3: Code excerpt of *../server/BookStoreHTTPMessageHandler.java*, lines 251–266

Now the server is able to process book ratings, so all we have to do is to expose the client-side to this API. To do so, we simply serialize the book ratings and send the POST request with it and wait for a response from the server.

```
    public void rateBooks(Set<BookRating> bookRating) throws BookStoreException {
        ContentExchange exchange = new ContentExchange();
        String urlString = serverAddress + "/" + BookStoreMessageTag.RATEBOOKS;

        String listRatingsxmlString = BookStoreUtility
                .serializeObjectToXMLString(bookRating);
        exchange.setMethod("POST");
        exchange.setURL(urlString);
        Buffer requestContent = new ByteArrayBuffer(listRatingsxmlString);
        exchange.setRequestContent(requestContent);

        BookStoreUtility.SendAndRecv(this.client, exchange);
    }
```

Figure 4: Code excerpt of *../client/BookStoreHTTPProxy.java*, lines 132–144

4.2 `getTopRatedBooks`

To implement this, we first check that we are being asked for a valid number of books to retrieve. If not, we produce an exception. Then we make a `Collection` containing the values of the book map, such that we can sort the values based on the output of a `Comparator` that compares the `getAverageRating` method. Once sorted, we merely need to cut the upper entries of the resultant list to get the requested number of toprated books. We then make a set of the ISBN numbers and use the `getBooks` functionality to produce and return the books.

```
1     public synchronized List<Book> getTopRatedBooks(int numBooks)
2             throws BookStoreException {
3         if (numBooks < 0) {
4             throw new BookStoreException("numBooks = " + numBooks
5                     + ", but it must be positive");
6         }
7
8         // get the list of books
9         List<StockBook> listBooks = new ArrayList<StockBook>();
10        Collection<BookStoreBook> bookMapValues = bookMap.values();
11        for (BookStoreBook book : bookMapValues) {
12            listBooks.add(book.immutableStockBook());
13        }
14
15        // sort the list
16        Collections.sort(listBooks, new Comparator<StockBook>() {
17            public int compare(StockBook a, StockBook b) {
18                return a.getAverageRating() >= b.getAverageRating() ? 1 : -1;
19            }
20        });
21
22        listBooks = listBooks.subList(0, numBooks);
23
24        Set<Integer> listISBNs = new HashSet<Integer>();
25        for (StockBook book : listBooks) {
26            listISBNs.add(book.getISBN());
27        }
28
29        return getBooks(listISBNs);
30    }
```

Figure 5: Code excerpt of *../business/CertainBookStore.java*, lines 280–309

The server now needs to recognize and handle the message. We do so by extracting and decoding the `BOOK_NUM_PARAM` parameter and pass it to the book store object, which performs the retrieveal, and then we serialize the returned object into XML and then write it to the response object.

```
1             case GETTOPRATEDBOOKS:
2                 numBooksString = URLDecoder
3                 .decode(request
4                         .getParameter(BookStoreConstants.BOOK_NUM_PARAM),
5                         "UTF-8");
6                 bookStoreResponse = new BookStoreResponse();
7                 try {
8                     numBooks = BookStoreUtility
9                             .convertStringToInt(numBooksString);
10                    bookStoreResponse.setList(myBookStore
11                            .getTopRatedBooks(numBooks));
12                } catch (BookStoreException ex) {
13                    bookStoreResponse.setException(ex);
14                }
15                listBooksxmlString = BookStoreUtility
16                        .serializeObjectToXMLString(bookStoreResponse);
17                response.getWriter().println(listBooksxmlString);
18                break;
```

Figure 6: Code excerpt of *../server/BookStoreHTTPMessageHandler.java*, lines 268–285

To finish things up, we implement the client-side, which initiates the action. In this, we encode the number of toprated books to be fetched and use it to produce the URL parameter `BOOK_NUM_PARAM`, and then sending the request and wait for a response.

```
1     public List<Book> getTopRatedBooks(int numBooks) throws BookStoreException {
2         ContentExchange exchange = new ContentExchange();
3         String urlEncodedNumBooks = null;
4
5         try {
6             urlEncodedNumBooks = URLEncoder.encode(Integer.toString(numBooks),
7                     "UTF-8");
8         } catch (UnsupportedEncodingException ex) {
9             throw new BookStoreException("unsupported␣encoding␣of␣numbooks", ex);
10        }
11
12        String urlString = serverAddress + "/"
13                + BookStoreMessageTag.GETTOPRATEDBOOKS + "?"
14                + BookStoreConstants.BOOK_NUM_PARAM + "=" + urlEncodedNumBooks;
15
16        exchange.setURL(urlString);
17
18        return (List<Book>) BookStoreUtility.SendAndRecv(this.client, exchange);
19    }
```

Figure 7: Code excerpt of *../client/BookStoreHTTPProxy.java*, lines 148–166

4.3 `getBooksInDemand`

The implementation of this method mirrors the `getBooks` method to a large degree. The difference boils down to checking if the book has had a sales miss using the `StockBook` method `getSaleMisses` and adding the book if this value is positive. As in the `getBooks` method, we convert the book to the safer immutable `ImmutableStockBook` type before returning. More interestingly, the `StockManagerHTTPProxy.java` now implements the interface method `getBooksInDemand`. This again mirrors the getBooks implementation, as the details were exactly the same. A unique message tag for this bookstore operation `GETBOOKSINDEMAND` was added to be handled in the `BooksStoreHTTPMessageHandle.java` file.

Much of the work is done for us here already. A case for the new massage tag was added and an implementation similar to the `GETBOOKS` case was carried out.

¿¿SOMETHING ABOUT THE EXCEPTION HERE¡¡

```
1     public synchronized List<StockBook> getBooksInDemand()
2             throws BookStoreException {
3
4         List<StockBook> listBooks = new ArrayList<StockBook>();
5         Collection<BookStoreBook> bookMapValues = bookMap.values();
6         for (BookStoreBook book : bookMapValues) {
7             if (book.getSaleMisses() > 0) {
8                 listBooks.add(book.immutableStockBook());
9             }
10        }
11
12        listBooks.get(0).getAverageRating();
13
14        return listBooks;
15
16        // What errors can happen
17        // throw new BookStoreException("Not implemented");
18    }
```

Figure 8: Code excerpt of *../business/CertainBookStore.java*, lines 312–329