CS262 - HTTP Sucks Chat

Victor Domene
victordomene@college.harvard.edu
Josh Meier
jmeier@college.harvard.edu
Gabriel Guimaraes
gabrielguimaraes@college.harvard.edu
Brian Arroyo
brianarroyo@college.harvard.edu

March 2nd, 2016

Contents

1 Overview

In this assignment, we had the task of building a chat application using two different protocols: REST and a hand-crafted protocol of our own. This paper will explain the details of implementation of RDTP (Real Data Transfer Protocol) and of REST, criticizing the two protocols. We have used Python (and many of its libraries), Flask (a simple HTTP server for routing) and MongoDB. For details on dependencies and some quick installation guidelines, refer to the repository linked in the end of this article.

2 Interface and Code Structure

The code is divided into several abstractions. First, the files `client.py` and `server.py` are used to start up servers and clients with the different protocols. The usage would be

```
python {server.py,client.py} {REST,RDTP}
```

Notice that a REST client will only work with the REST server. By default, these files start up the servers at `localhost:9999`, but that can be easily changed.

The `ChatDB` class in `chat_db.py` is responsable for all of the interactions with MongoDB. This allow us to have a nice separation, so that it would be quite easy to change from Mongo to any other type of database, by simply modifying this file. The `ChatServer` class in `chat_server.py` is an abstraction over the server, and it contains an instance of `ChatDB`, other than some common methods to all chat servers. Similarly, `ChatClient` in `chat_client` standardizes the chat clients, which includes error handling and printing results to the user.

Finally, at the heart of the project is `RESTServer` and `RESTClient`, as well as its `RDTP` counterparts. These inherit from `ChatServer` and `ChatClient` and implement the desired functionality for each of the protocols.

3 RDTP - Real Data Transfer Protocol

3.1 High-Level Implementation

The chat server we implemented in Real Data Transfer Protocol (RDTP) uses sockets to listen for client connections. The server will currently block on any currently open (TCP) sockets, waiting for any messages. When receiving a message, the server will first receive five items: a version number, a magic number, a status code, a length for the action name, and a length for the message. The next set of bytes correspond to the action; we spin loop trying to receive the number of bytes for the action string specified by the header and do the same for the message. Ignoring the status code, the server then sends the action and the message off to a dispatcher. Depending on the result of how the message is handled, the server will respond to the client with a status code. For any messages, we either try to deliver them immediately or queue them. Handling of that error code is done client-side.

When a user logs in, the socket used is recorded and a unique session token is generated and stored in a Mongo database. The session token is also sent back to the user, who will include it with any future requests and will be used by the server to verify that the user is who they claim to be. The user is also flagged as being logged in; this informs us of whether we should send a user a message meant for them immediately or queue it up for later. Anytime sending a message on a socket fails, we enqueue the message to be sent later.

While the client is expected to block on server responses, a timeout is set of 3 seconds to prevent the client from hanging.

3.2 RDTP Specification

RDTP has seven parts to the protocol, the first five of which are referred to as the header:

- 1 byte to present a magic number
- 1 byte to specify a version number
- 1 byte to indicate a status code
- 1 byte to establish the length of the action to be taken
- 1 byte to stipulate the length of the message
- The action to be taken, of length as expected by item 4
- The message, either as arguments to the action or the message to be delivered, of length as expected by item 5

The status code must be 0 to indicate a non-error, and non-zero to indicate an error (the client is expected to handle status codes itself).

4 REST

4.1 High-Level Implementation

For the implementation of `RESTServer`, we relied on `Flask`, a very simple HTTP server that allows us to define our routes very easily. Since it would be nothing but cumbersome to encode/decode HTTP requests, we decided that using a simple yet functional server would be enough for this part of the assignment.

Notice that this implementation does not support real-time conversation. In order to get messages, a user must call `fetch`. This is due to the nature of `REST`, and we will cover this in more detail later.

We also used a session token for our REST application. Sending username and password combinations over the web is not exactly secure, so we minimize this access by simply returning a session token on user login. We then pass in this token in the `Authorization` header every time we want to make a request to the server.

For implementing `RESTClient`, we relied on a Python library called `requests`. It allowed us to make the required HTTP requests in a much easier way (including the handling of authentication and session token bookkeeping).

Finally, notice that we are using a combination of HTTP and JSON for both requests and responses.

4.2 Routes

The following REST routes are used in this application:

- `POST` to `/login` (for the `login` command)
- `POST` to `/logout` (for the `logout` command)

- POST to /groups (for create_group command)
- POST to /groups/<group_id> (for add_user_to_group command)
- GET to /users (with a wildcard as a parameter, for get_users command)
- GET to /groups (with a wildcard as a parameter, for get_groups command)
- POST to /users/<user_id>/messages (for send command)
- POST to /groups/<group_id>/messages (for send_group command)
- GET to /users/<user_id>/messages (for fetch command)
- DELETE to /users/<user_id> (for delete_account command)

These match the standard for REST.

4.3 Error Handling and Responses

We have tried to follow the JSON API standard for JSON response values. Therefore, in case of an error, we return with the appropriate HTTP Status Code (as given here) and with a suitable description. For instance, if a user tried to login sending an invalid JSON to /login, then we would return

```
{
  "errors": {
    "status_code": 400,
    "description": "Bad Request: The request cannot be fulfilled due to bad syntax"
  }
}
```

Similarly, any correct response would correspond to a JSON similar to the above, but with a header data instead. The RESTClient is responsable from converting these error codes and responses into appropriate return values to the common interface of ChatClient.

5 Critique of REST and RDTP

Among the many differences between REST and RDTP, we will focus on the following points.

5.1 Real-Time Conversation

While RDTP allows for ”real-time conversation” (i.e., users receiving messages immediately if they are online, without the need of calling fetch), REST clearly does not. The connection between server and client in the REST case lasts only for the duration of each request, and it is reopen every time a new request is made. Therefore, to allow real-time conversation, we would have to keep polling the server for new messages every few seconds or so. This is not exactly bad, but its performance is clearly much worse than RDTP's. On the latter protocol, connections between sockets are kept alive as long as the user is logged in, and messages can be instantaneously received.

Notice, however, that due to the command line nature of this application, messages may be received while you are typing. While usually on a chat application this is desired, since there will be a view for typing and a view for getting messages, this is not ideal in RDTP since we are using the command line. So, we may receive a message while we are typing. It does not affect the correctness in any way, but it is not very elegant. By not having real-time conversations, REST avoids this ”issue”.

5.2 Difficulty of Implementation

Of course, designing our own protocol by hand is harder than using a pre-defined set of rules. However, the implementation of RDTP server itself may also prove challenging, since we need to keep the server listening to many clients and responding to them appropriately. Not only that, but each RDTP client must also be asynchronous, since it must receive messages from the server and be able to send messages to it as well. This required quite a lot of playing around with the `select` system call and, in the case of the RDTP clients, a thread for listening to messages.

Notice that since we only have one socket per client, we must differentiate between messages from the server that are responses to our requests, and those that correspond to someone sending us a message. We do this by encoding the data with an "R" for request or an "M" for message, and letting a thread handle the responses with "M".

On the other hand, implementing the HTTP Server was not only a lot easier because of `Flask`, but it also does not keep the connections open. Therefore, building a server from scratch would not be as difficult.

5.3 Simplicity and Portability

REST is a widely used architecture, with a lot of resources to facilitate development. Furthermore, HTTP requests are things that can be understood by any browser. So, given the correct set of inputs, one could theoretically use the browser for using the REST application very easily. On the other hand, interpreting RDTP requires some specialization, which our `RDTPClient` provides.

5.4 Specificity

HTTP encompasses many details that are not truly necessary for a Chat application. Requests are inherently longer, while RDTP, specifically hand-crafted for this purpose, can be much shorter. However, notice that HTTP is much more flexible: there is no limit for the size of the responses or requests, as in RDTP.

Moreover, some of the HTTP status codes may have unclear semantics under our Application. It is slightly unclear what error to return if a `register` fails due to an already existing username, for instance.

5.5 Limitations

Our current implementation of RDTP uses only one byte for the length of the message. This limits us to simply very few characters on a message. To be safe, we are currently supporting only messages smaller than 128 characters. To fix this issue, it would be required to change the design decision of using only one byte for the length. Perhaps, 4 bytes would be a more appropriate length.

Finally, the RDTP server is much more prone to a failure. We did not do much exhaustive error handling for it, since we do most of it in the client, and then the server may crash in a situation where the REST server won't.

6 Code

All of our source code for this assignment is in the following GitHub Repository. It contains an instructive `README.md` file for dependencies and usage instructions.