Slick Communications

CS5500 – Managing Software Development Report

Team – 101

Northeastern University
Fall 2018

Akshat Shukla (shukla.ak@husky.neu.edu)
Meghna Venkatesha (venkatesha.m@husky.neu.edu)
Mathew Lazarcheck (lazarcheck.m@husky.neu.edu)
Jason Teng (teng.ja@husky.neu.edu)

I. Project Goals

This project is part of CS5500 — Managing Software Development and entails inheriting a legacy code base to make a basic chat server. The overarching goal of the project was to build upon the legacy codebase that had core functionalities implemented using concepts and strategies learned in the course in the first half of the semester.

The workflow of the project spanned across 6 weeks consisting of 3 bi-weekly sprints that followed the Agile development model including: bi-weekly reviews, extensive documentation, continuous integration, testing and deployment to AWS. The project was designed in a manner that rewarded creative ideas to implement features that add value to the system. Java was used to write the Prattle server, as well as the convenience Chatter code. Prattle is the server application that the project's emphasis was placed on.

The legacy codebase provided basic functionality of a client-server communication application with limited features including a Prattle server that handled multiple clients that could send broadcast messages to all other user currently logged in. This implementation did not have persistence. Each sprint built upon the previous, moving towards creating an application that was ready to be delivered to a client. Sprint expectations were divided into functionalities and environment goals. Each expectation type including various differing weighted. Functionalities described the new use cases that could add value to the project. Environment goals were tasks for the team to create to maintain an environment for the development process and management of the application. Stretches for a sprint defined the use cases that were optional to complete but added additional functionality and/or improved the environment.

The team followed the Agile process, but instead of daily scrum meetings we utilized daily Slack-ups. Tasks were created and assigned among the team using JIRA and we communicated on Slack. GitHub was used as Version Control; the team used feature branches in a manner to continuously integrate using Jenkins and SonarQube. Smart commits were used to monitor development process for JIRA tasks. Slack notifications were implemented for Jenkins and GitHub to monitor builds and commits. The project was used Apache Maven as a build automation tool and we used Junit 5 for unit testing.

Sprint 1 involved gaining familiarity with the legacy code base and creating unit tests to pass the SonarQube quality gates. Stretches included smart commits, Jenkins and GitHub slack notifications, user and group CRUD functions, directing messages to individuals and group messages. Sprint 2 focused on CRUD functionality for users and groups, group messages and private messages. Stretch goals included message persistence, encrypted login/register and MIME types. Sprint 3 expanded to create more robust features including wiretaps for subpoenas, queueing messages for offline users, to deliver them in chronological order and to create a dynamic logger. Stretch goals included system testing including stress and performance tests, parental control for inappropriate messages, adding message meta-data to include IP addresses of the parties involved and message search functionality.

II. Result Overview

The team was able to all non-stretch expectations and some of the stretch goals for each sprint. To describe the functionalities implemented over the three sprints, 4 client use cases will be used: “A”, “B”, “C” and “D”.

The inherited system was capable of broadcasting messages to any connected client to the server i.e. if the client “A” sends a message, it will be sent to the server which sends it to all connected clients, which are “B”, “C” and “D”. Following sprint 1, this existing system with legacy code was unit tested using Junit 5 approximately 87% coverage.

In sprint 2, new features were implemented including: user creation, user search, username updating and user deletion. Functionality was added for users to create groups and for any user in group to add existing users, remove users, update the group name and delete the group. Messages in a group are only visible to the users who belong to the group. Users can login or register using a username and password; the password was encrypted using Base64 encoding in the server while storing. Also, private messaging was implemented for users to send messages to one other user i.e. user “A” could send a message to “B” and “C” and “D” would not receive it. Message persistence was also incorporated into the server. Storing messages was a key feature to accomplish future goals in the sprint 3 expectations. We stored private messages in the same manner as group messages by creating groups with a group name that is the hash of the sender and receivers name sorted alphabetically. Following sprint 2, the codebase expanded rapidly, but we maintained test coverage of about 86.7%.

In sprint 3, a message queue was implemented to send messages in chronological order to a user when they logged in or joined a group chat that they were not logged in for and/or not in a group chat. We added functionality to store messages with senders and receivers IP address and added database object methods to search for a message by multiple attributes such as sender, receiver or time. In addition, we added wiretap functionality for subpoena compliance. This worked by having an admin user create an account for an agency to use to receive a copy of messages to and from a user or group. Logging was improved, by adding functionality for the admin to turn logging on and off. We added a parental control feature that flags messages with vulgarities in our database. These features were added to the chat server without hindering the existing features. Sprint 3 concluded our task of expanding a a chat server application into a more reliable, secure and robust messaging application.

Continuous integration ensured that our chat server was property tested before being merged to master. The continuous integration process of refactoring, documenting, testing and frequent committing made the development of the chat server smooth. At the final sprint review, the application stood at 86.4% coverage on master branch, where the missing coverage occurred mainly due to some higher-level classes being untestable without drastic measures, due to the way the pre-existing code base was written. Our unit tests used an actual client application to interact with the chat server when needed. This strategy lends credence to the usability and functionality covered by the chat server application.

III. Development Process

We used the AGILE methodology for our project development management. The project was spanned across 3 bi-weekly sprints with a sprint review at the end of each with an assigned teaching assistant. For each sprint, the team would meet in person to determine the most beneficial manner to divide tasks and determine which sprints to do. We created and assigned tasks to complete in JIRA. Each task was assigned and distributed to utilize each members individual skillsets, emphasizing completing the required functionality before the targeted stretch goals. Sprint two and three sprints began with an assessment of the previous sprints work to determine how to increase quality of the application in the coming sprint. The structure of the project was decided during this meeting at the start of the sprint to maintain a mutual understanding of how new and pre-existing project components interact and work together.

Each member worked on separate feature branches on GitHub, which branched off the master branch or the most stable branch. Feature branches is an essential part of the continuous integration process. Feature branches reduce merge conflicts and gives each member independence to work on their own code for the assigned feature. The development environment was setup with Jenkins, Slack and GitHub linked together, which helped team members keep track of changes being made at any moment by any collaborator due to the slack notifications. Each member followed the smart-commit protocol where each commit made was associated with a certain ticket that helped manage the sprint goals.

The merge requirements with master branch were difficult to achieve with the threaded approach to the chat server application that was inherited with the pre-existing code. The team made sure all testable new code was tested and did not pollute the branch being worked on. In some scenarios with heavy workload and lesser active team members, the test cases had to be written after implementing features, which is a bad practice, not adherent to small and reliable commits and frustrating. This is certainly one thing the team could have done better. Other than that, the team managed to work actively, incorporating each feature along with robust test coverage.

Along the span of the sprints, there were numerous moments where all the team members were not available due to extraneous circumstances, but we ensured that we had regular Slack-ups. Slack was the most often used communication channel for our team, where we tagged and assigned each other issues and concerns, as well as kept each other updated on our progress and availabilities. Slack is an effective communication medium for software development.

Although Slack was used to discuss the workflow, major design decisions were made in person. One such decision was during sprint 2 where we needed to incorporate CRUD operations with users, groups and add message persistence to the chat server. Our initial approach was to use Java Database Connector (JDBC) with a MySQL database for the features, having laid down most of the database access object classes already. The team decided to switch to MongoDB after discussions about the high-level design of the schema, structure and the performance of MongoDB versus MySQL. This decision helped the team integrate the project classes easily and provide scalability. In person meet-ups along with regular slack-ups were the key to effective communication in our group.

One constraint of JIRA is that is only allows a single assignee for a task, we generally worked together on tasks. We also pair programmed at numerous points during the project timeframe; this was extraordinarily useful to debug faster and deliver quality code in lesser time than it would normally take. In addition, we found bugs before they were even bugs Documentation of the code was written as methods and classes were added.

Since, it was a team project and everyone's coding style and coding environment preferences are different, we faced some minor issues with code refactoring and indentations, but they were quick to deal with before each sprint review. With regular slack-ups, the team members knew exactly what they needed to do, without hindering each other. After each task completion, new tasks were assigned based on an analysis of those in the backlog.

IV. Project Retrospective

None of the team members had previously worked in a realistic agile development of an application. This was a valuable learning experience since we worked on a real-world problem (communication systems are widely used) as a team that had high-level design decision power and used a common technology in software development. Our team primarily worked away from the University (large external monitors are a wonderful productivity booster), so we relied on Slack for communication.

We learnt a great deal about the challenges of working with a non-local team mate for the last two sprints. At first it was hard communicating the tasks and explaining decisions over slack as not all members could be involved in the personal face-to-face meetup. But gradually, there were compromises for the greater good, where somehow in the end, every single team member contributed their share of work in the success of the project in spite of communication delays. The team learnt that no task is easy, whether it is testing of components, integrating them, developing them and that each of them require more time than probably expected. Moreover, it was a challenge for the team to build upon an existing codebase that was implemented in an untestable and strange manner. The pre-existing code was well-documented, which made it easier to adapt. The team did not have a member that organized everyone early on, which built pressure as the deadlines approached. We were lacking that individual, since we were 4 individuals who tend to wait until close to the deadline.

We faced a lot of issues having to build our project on Jenkins numerous times. We think next semester there should be more Jenkins instances to spread across the groups. Jenkins would our two versions of our tests concurrently on one test mongo database when a pull request was made. This caused concurrency issues in the database and failed the builds because the tests would fail. This issue was ultimately solved by the team by separating the tests by a 50% chance of timeout, a less than ideal solution. An alternative and a better solution would have been to use Mockito to mock the database instance and run tests which do not conflict with each other.

In addition, we would recommend having a code walk for the inherited code. Not everyone has socket programming experience, especially in Java. We recently learned that the inherited code would have allowed significantly more concurrent users on the server if it did not spawn a thread

for each client and was implemented in a non-blocking manner. In addition, the threads created difficulties in writing tests for the chat server. This would have been resolved if we changed the implementation of the inherited code in the first sprint, but we were not aware of this at that time.