SYDE322 COURSE PROJECT PART 2

Manfred Cheung
20577603

Table of Contents

Revision of Detailed Design 2
New P2PComms module for student client 2
Modules for UI components 2
Server Data Storage 2
Simplification of server code 2
Simplification of Broker 2
Implementation Process 3
Deployment 3
Tools 3
React, Material-UI, React-Router, React-Table 3
PeerJS 3
Implemented Features 4
Server 4
Teacher Client 4
Student Client 4
Expectations for Future Enhancements 4
Software Testing 5
Back-End 5
Front-End 6
System 6

Revision of Detailed Design

Most of the implementation matches the software design document. However, there were a few minor deviations in the implementation from the detailed design previously described. The majority of these do not change the logic of the code concerned, however a few organizational changes were made when modules became too big and needed to be refactored. All the larger changes will be listed below.

New P2PComms module for student client

The detailed design originally called for the P2P code to be folded into the user interface code in the student client. It was anticipated that the code needed for P2P communication would be insignificant enough that it would not be worth splitting it into a separate module. When doing implementation, the amount of coded needed was greater than anticipated. Hence, the P2P code was split into a separate module and methods were imported into the user interface when necessary.

Modules for UI components

Both Teacher Client and Student Client systems in the detailed design document did not go into any detail on how the UI will be structured or implemented. This vagueness was intentional because the structure of the UI code will vary depending on the arrangement of the UI itself as well as the libraries and frameworks used. Neither of these were decided at the time the detailed design document was written.

It was decided that the UI will be implemented using the React framework. React highly encourages breaking down the user interface into nested components. I found that the best way of organizing these modules is to place each React component into its own module. Therefore, a much larger number of modules were generated than were specified in the teacher client class diagram or in the student client class diagram.

Server Data Storage

The Data Storage Bridge was supposed to store data locally in the detailed design document. However, during implementation, it was decided that it would be better to store data in the server rather then locally in the browser. Because a bridge is used, it is unnecessary to make large changes to the interface, however new methods were added to connect to the server and exchange information.

Simplification of server code

There were several changes made to the Express server code. This was done because there were a few changes to the functionality which the server is supposed to provide, and because the server was redesigned so that code written for application exercise 3 can be reused in this project.

The detailed design specified that there will be a Classroom Account class which ties P2P ids to individual accounts. During implementation, it was decided that this was unnecessary, so this feature was scrapped. Instead, the account is only necessary to retrieve save data as described earlier.

Simplification of Broker

The P2P broker no longer needs to tie the P2P ids to individual accounts as stated earlier. Therefore, the BrokerResponse component no longer needs to have access to the database.

Implementation Process

Deployment

As stated in the design document, this project is to be deployed on Heroku, which is a free hosting service. Heroku also provides a free PostgreSQL database so that will be used as well. The advantage to this, besides zero cost, is that the application exercise also used Heroku's PostgreSQL database, so it is possible to copy over the database handling code from the application exercise.

Tools

All the tools used such as libraries or unusual JavaScript APIs will be listed below. The name of the tool and its purpose will be listed, as well as some basic background on how the tool works and how it fits into the project.

React, Material-UI, React-Router, React-Table

React is the front-end framework used to build the UI. This framework provides a structure for creating front-end applications, improving readability and speeding up development. As well, React is built to accommodate single page applications (SPA), which hold several advantages including increased speed and decreased load on server, but are difficult to build from scratch.

Finally React also has a vibrant ecosystem of ready made tools and components. I have used three such components in this project; material-ui, react-router, and react-table.

Material-ui is a full UI component toolkit, which provides a library of the most commonly used UI components ready to use. It reduces the amount of front-end boilerplate code which I must write and provides an acceptable look to the application out of the box. This allows me to focus on the application specific logic, which is more important in a prototype.

React-router is an in-browser router which allows different pages to be loaded depending on the URL without creating resource requests to the server. Essentially this is what allows me to create a SPA without rewriting the routing code for it.

React-table is a library which provides a table component, which was needed in my project but is missing from material-ui. Again, I chose to use it because it would not be a good use of time to rewrite readily available front-end code when the prototype is intended to demonstrate feasibility.

PeerJS

PeerJS is a library which abstracts WebRTC, a technology which allows for P2P communication in the web. WebRTC is standardized by the W3C and is supported by most modern browsers. More specifically, this library simplifies and abstracts the negotiation of the P2P connection using the Interactive Connectivity Establishment (ICE) protocol, working with provided STUN and TURN servers to bypass NAT. There were several libraries available which make WebRTC more user-friendly, however PeerJS was chosen because it was the most complete feature-wise; it provided both client code and signalling server code. As well, the documentation was complete and there were several example projects available implementing PeerJS. The project itself does not seem to be actively supported now, however this was deemed an acceptable trade-off for the benefits of PeerJS, which are offered by no other library. Once a working prototype is complete, it should be possible to swap out PeerJS for a similar library if necessary.

Implemented Features

Server

On the server side, all the planned functionality has been implemented. This includes the signalling server which negotiates the P2P connection between teacher and student clients. As well, authorization, authentication, and encryption were implemented to control access to user saved data.

Teacher Client

In the teacher client, most of the planned functionality was implemented to a degree, however there were some notable omissions from the prototype. The P2PCommunications component is fully implemented, as is Data Storage, QuizProctor, QuizProctorView, QuizManager, QuizManagerView, Quiz, Question, and MultChoiceQuestion. This gives the teacher the capability to create, delete, and administer multiple choice quizzes to students.

The ResultManager and ResultManagerView components were not implemented. These components are necessary to allow the teacher to view and save the student results. As result, this functionality is not included in the prototype. As well, the OrderQuestion and FillBlankQuestion components were not implemented, hence the prototype only has the option to use multiple choice questions.

None of the unimplemented components should have any technical barriers to implementation, however the prototype needed to be complete in limited timeframe and the implementation of these extra features were deemed unnecessary to prove project feasibility.

Student Client

All planned functionality has been implemented in the student client. ConnectView, QuizView, and P2PReciever have all been implemented. Therefore, the student can connect to and login to the teacher client, as well as render quizzes. One thing to note is that for QuizView, although technically implemented, rendering code for only the quiz types which are supported by the teacher client in its current state are complete.

Expectations for Future Enhancements

There is still a lot of work to be done before this project is in a presentable state. For one, all the unimplemented functionality mentioned in the previous section must be completed. As well, the UI of the prototype is not acceptable for production, it needs to be drastically improved. The passwords sent from the student to the teacher are also not encrypted in any way. This must be fixed before it is fit for public use.

The project can still be greatly improved even when all planned development work is finished. I wish to implement a system which allows the project to be monetized, offsetting the expected hosting costs of this software. It is possible to implement a local file storage system which can be used instead of the database storage system currently used in the prototype. The bridge and the storage scheme allow this to be done with little to no change in existing code. Then two tiers of users can be created; where the paid tier has the option of using the database cloud storage while the free tier must use local file storage.

The prototype must use externally hosted STUN and TURN servers to negotiate the P2P connection as mentioned previously. While this has its advantages; the processing costs of maintaining both is offloaded to a free provider, it is worth at least hosting backup STUN and TURN servers locally for redundancy sake.

A system where a user can share questions or even entire quizzes with other users may also be beneficial. This would aid in developing an active user community by encouraging collaboration. As well, it would help encourage more users to adopt this project as their go-to software for virtual quizzing. The only issue with this is that a method would have to be devised to offset the higher costs which this feature would require, as result of increased database and bandwidth usage.

Finally, teachers may find it useful to have the capability to customize exports and imports of student accounts and results. This would provide the project a large advantage which services like Kahoot! may not have and streamlines the integration of the project into existing teaching plans.

Software Testing

Various testing strategies were used to detect and fix failures efficiently. Methodology used must be balanced with development speed to provide an acceptable result on time. One notable decision I made is that automated testing will not be used. This was mainly due to time constraints, I did not have the resources to integrate an automated testing solution into the project. As well, most of the code written is not conducive to automated testing, because it is either front-end React code or it is networking code.

Back-End

The following describes the testing procedure for back-end code. These are in general critical to the application and difficult to debug once integrated, thus it is important that these be tested thoroughly.

In general, each back-end component was built in stages and unit tested every step of the way. A set of test-cases are devised at each step to ensure that all requirements of the code created during that step are met. With mission-critical code, I general ensure that at least branch coverage is achieved during unit testing.

For example, when coding the teacher P2PComms component, the first method built is the initClassroom method. To test this method, the following test cases were used:

1. Does getPeer return a peer object?
2. Is the onConnection callback called on student connection?
3. Is the onData callback called when student sends data?
4. Does the data sent by the student arrive intact?
5. What if the student sends invalid data?
6. Does the validateLogin function handle login correctly? (This function is unit tested alone)
7. Does conn.send actually send data back to the correct student?

If the initClassroom method passed all tests, then development can then begin on the next method initStudent, which is called by initClassroom.

After the entire component is complete and unit tested, then the component is integrated into the project and integration tests are run.

Front-End

The following describes the testing procedure for front-end code, which mainly consist of React code which generate visual components. This number of modules which must be build in a short timeframe dictates that testing be done less thoroughly than is ideal.

In general, each front-end component was built and integrated into the project then subjected to integration tests. A set of test cases were devised to test the component and ensure that its requirements are being met. Boundary-value analysis is used to reduce the number of test cases required to test the component thoroughly.

For example, when coding the EditQuestion component, I finished coding the component entirely and then did integration testing. The following test cases were used:

1. Does loading a premade question cause errors?
2. Does loading a new blank question cause errors?
3. Does changing the title work?
4. Does changing the time allowed work?
5. Does changing the question work?
6. Does adding choices work?
7. Does deleting choices work?
8. Does picking an answer work?
9. Does deleting the question work?
10. Does saving the question work?

System

After all development is completed, system testing is performed on the entire project to ensure that all requirements are met, and no further bugs can be found.

Functional testing can be performed on the system relatively easily, but other testing methods are much more difficult. Performance testing is impossible to do without test automation; it is impossible for me alone to put enough load on the system to load it in any meaningful way. Same for stress and volume testing. Configuration testing is difficult as well. I can test connections between tabs locally, between my computer and one other computer which I have network access to, and between my computer and a computer in the computer lab. I do not have any other devices on which to perform configuration testing. Compatibility testing is unnecessary since the browser compatibilities for all the JavaScript features used are well known and published. Basic timing testing, which can be done through functional testing shows that all the response times for next to no load are below human perceptive ability. The only exception is the login, as the hashing algorithm is designed to have a runtime of roughly one second. Hence the response time will also be roughly a second. Security testing can be done but is beyond the scope of this prototype and will take too much time. Quality testing will require some time in a production environment to determine, which is again beyond our resources.