University of St Andrews

CS4103 Coursework 1

Middleware

Author:
150008022

February 27, 2019

Goal

The goal of this practical was to implement a distributed application using communication middleware to collect data on user responses to the n-person prisoner's dilemma.

See the README.txt for instructions on how to start the applications.

Part I

Communication Set-Up

The application was set up using Spring Initializr [1], which provided a Spring Boot application template, with the web and test starter dependencies. A controller class to provide REST request mappings was implemented, and the test connection method was created in the ProsecutorService class. Unit tests were written for both, and Postman was used to test the request when the application server was running, and to fulfill the criteria for part one.

To provide a user-friendly client, create-react-app [2] was used to boostrap a react.js application. By using a proxy setting, CORS issues were avoided while developing locally. A simple test button was supplied that would perform an asynchronous call to the API test endpoint, and then print the response to the screen (Figures 1 and 2). When building distributed systems, it is advised to always design for failure, and so a clear "Failed to Connect" message is shown to the user if the request fails for whatever reason.

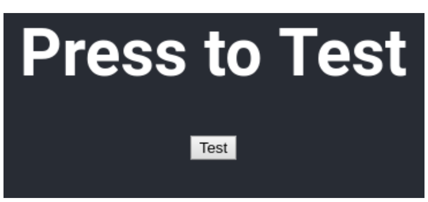

Figure 1: Before Press

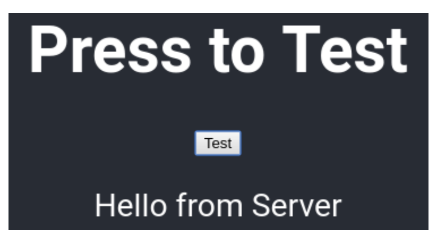

Figure 2: After Press

Part II
Single Client Game

First, the Persecutor service in the Spring Boot application was extended to include the *chooseOption* method, which would take the choice of prisoner one and return the number of years reduction based on a random choice from prisoner two. To reliably test all outcomes produced the desired result when using a random choice generator, the random choice was provided through a separate service which could then be mocked in tests.

Since only one game would exist at this stage, the endpoint `/prosecutor/games/1/prisoners/1` was chosen to be where the prisoner would send their choice in the JSON format. By PUTing their choice to this endpoint, the user could expect a return value of the number of years reduction.

Spring boot provided the (de)serialization between JSON and the Java object representation through the Jackson library.

Part III
Multiple Client Game - Server

As an extension, the system was made to handle multiple games. This was first done by extending the persecutor service API.

Given REST design relies on URLs mapping directly to resources, the first step was to design the URL structure in a way that would work for multiple games with multiple prisoners. A hierarchial structure was chosen, where games could be accessed by their ID, prisoners could be accessed by their ID and the game ID they belong to, and so on. Consistency in behaviour was essential, and so a HTTP GET to `/prosecutor/games` would return all games and all their prisoners, then `/prosecutor/games/1` would return game 1 and all its prisoners, `/prosecutor/games/1/prisoners` would only return the list of prisoners from game 1, and `/prosecutors/games/1/prisoners/1` would only return prisoner 1 from game 1. The full set of URLs can be found at `/swagger-ui.html` when the application is running, along with documentation which describes the JSON output structure and all the supported operations for each URL (Figure 3).

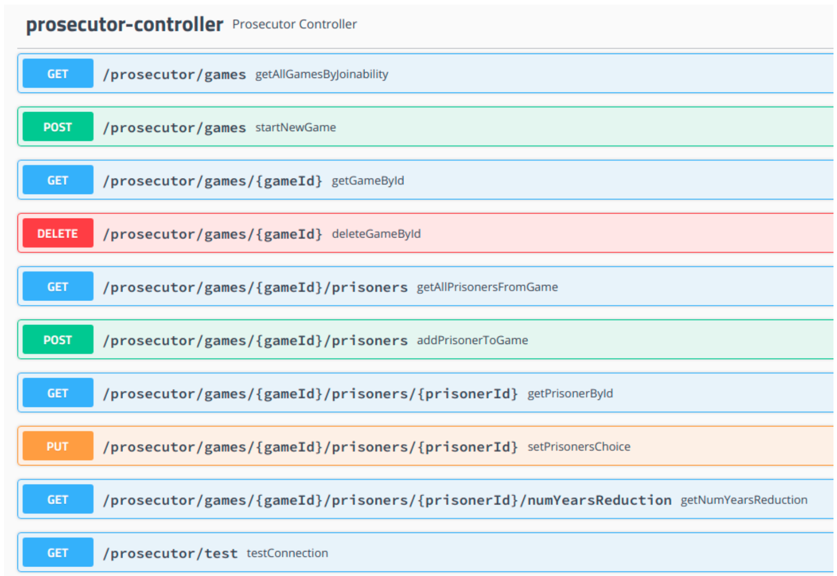

Figure 3: Swagger Documentation

In order to make the API friendlier to use, error messages were standardised such that any exception thrown by the service would provide a HTTP status code and a message for the user, which would then be sent in the response as JSON. This behaviour was tested for using *mockMvc* in *ProsecutorConrollerTest*. Validation was used for any method that took client input.

Though a production ready implementation would use a seperate database server, for this practical a simple H2 in-memory database was used. The repository abstraction provided by Spring Data would mean that with some simple configuration, an external database could be used instead.

The ProsecutorService was extended to provide all the functonality needed for each controller method, such as getting all games, getting a prisoner from a game, creating a new game, and so on. Unit tests were written for each method by mocking the repositories used. For the repository layer, only the custom queries were tested, as the framework supplied the rest (`save`,`findById`,`findAll`).

In order for the client to join a random game, an optional URL parameter was added to the GET request for all games that would allow the client to filter for 'joinable' games. Joinability in this context is any game that can accept at least another prisoner.

The games kept the same structure, where there would be two prisoners per game. Since REST prohibits the use of any kind of session, the client would have to use a polling mechanism to get their years reduction once providing their choice. Message based middleware avoids this issue. For example, a commonly used publish-subscribe messaging protocol middleware provided by Spring Websockets, called STOMP, would allow the client to subscribe to their prisoners status, and allow the server to push the number of years reduction to the client once they became available. REST on the other hand avoids all the overhead involved in maintaining subscriptions and message brokers. The structure of REST APIs also tends to encourage decoupling from any client application, which is useful if the service is ever to be expanded or used differently.

To ensure that a game would only ever have two prisoners at most, adding a new prisoner to a game had to be wrapped in a transaction. This would ensure that checking a game had less than two prisoners and adding a new prisoner would both happen atomically. If an external database was used, a better approach would be to use a database trigger on insert to check that this constraint was maintained, and handling the error appropriately in the API server.

Since the prisoner ID is unique for all prisoners across all games, it would be possible to establish a many-to-many relationship between games and prisoners. Since the heirarchial structure was easier to capture in a REST url design, this functionality was not provided. If it was to be used, a HATEOAS architecture would likely be preferred, where `/prisoners` would return a prisoner object with the URLs of all games they partake in, and `/games` would return a game object with the URLs of all the prisoners taking part in them. This would avoid cycles in the JSON response objects, and provides what is sometimes referred to as the "final level" of REST.

Given more time, integration tests that would test normal and malicious use patterns would also have been implemented.

Part IV
Multiple Client Game - Client

With the backend API tested and running, the client would now be able to start and join games. Since games used unique identifiers, players could start a game and share their game ID, to allow other players to join. This could be useful in an expiriment, as it could be used to track which participants took part in which games if they are supplied their game and prisoner IDs beforehand.

The game menu (Figure 4) provides four options: test connection, join game, start game, and view statistics. Test connection allows the client to check that the REST server is available before continuing, and to fulfill part one of the practical specification.

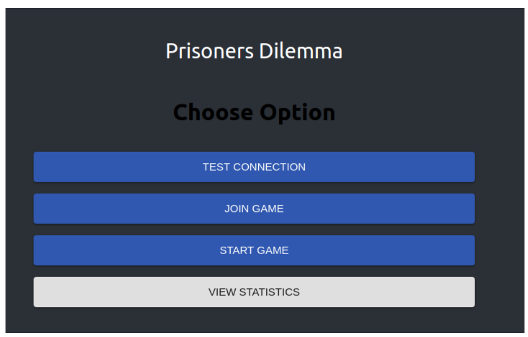

Figure 4: Start Menu

Join game allows the user to either join a random game, or join a specific game by providing a game id and/or a specific prisoner id. The random game feature uses the 'joinable' filter from earlier, and shows an appropriate error message if there are no joinable games.

If able to join a game, the user is then taken to the choice buttons screen (Figure 6) where they can see their prisoner and game id.

Start game allows the user to begin a new game, and immediately moves to the choice buttons screen. Once the player has made their decision, the client will begin polling the server for the number of years reduction (Figure 7).

In order to prepare for possible failure, prisoners who started a game and disconnected for whatever reason can return to the game to make their choice

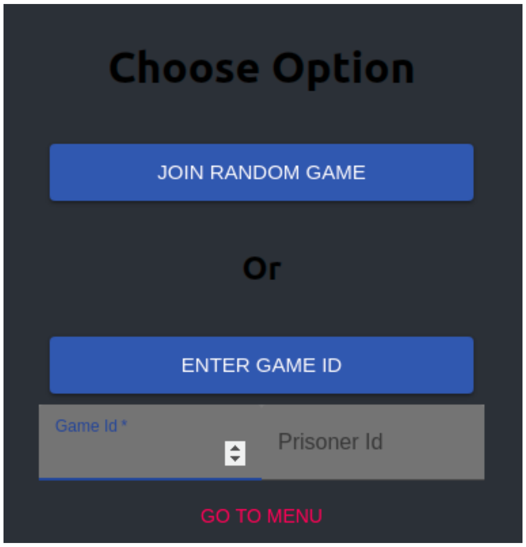

Figure 5: Join Game Menu

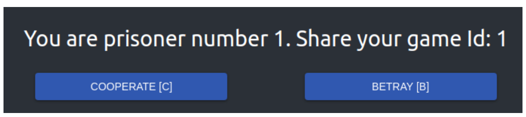

Figure 6: Choice Buttons

at a later time by rejoining with their game and prisoner Id. The same can be done if they experience a failure while waiting for the other prisoner, and in this case they will either be immediately shown the result of their choice, or the polling message screen.

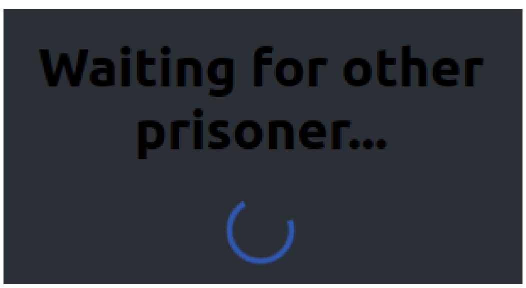

Figure 7: Polling Screen

To try and minimise server load, the polling mechanism uses random exponential back off with a cap to avoid ridiculous waiting times. The wait time begins at 2 seconds, and is multiplied by a random number between 1 and 2 after every failed try, and reset to 2 seconds after the time reacahes 7 seconds. This jitter is included to avoid stampedes when all clients experience failure at the same time and their polling times become synchronized, which could be catastraphic for the server. A graph of the network activity while polling is shown in Figure 8 which demonstrates the jitter with backoff.

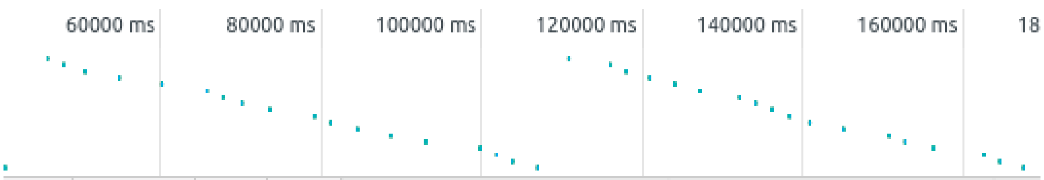

Figure 8: Exponential back-off with jitter

Finally, the view statistics option enables users to see the distribution of choices (Figure 9). CanvasJS [3] was used for drawing the graph. The graph data is generated by fetching all games from the endpoint `/prosecutor/games` and counting the occurrences of choices in each game. At a larger scale, this computation would likely be moved to the server to reduce the amount of computation and memory required from the client.

Conclusion

This implementation demonstrates a simple distributed system with a thin-client and server architecture, using REST as middleware.

Transparency is achieved for the most part, as there is nothing to suggest to the user that the application is communicating with another machine. If the system was to be properly scaled, there would likely be several servers for serving the client webappp to requesting users that could all be accessed at the same domain (e.g. `www.prisonerdilemma.com`). The REST API server could also be duplicated behind a load balancer in order to provide redundancy and allow the service to handle more requests. For this to be done, the in-memory database would have to be swapped out for an external database server as discussed earlier.

Given more time, I would like to have refactored and tested the client-side

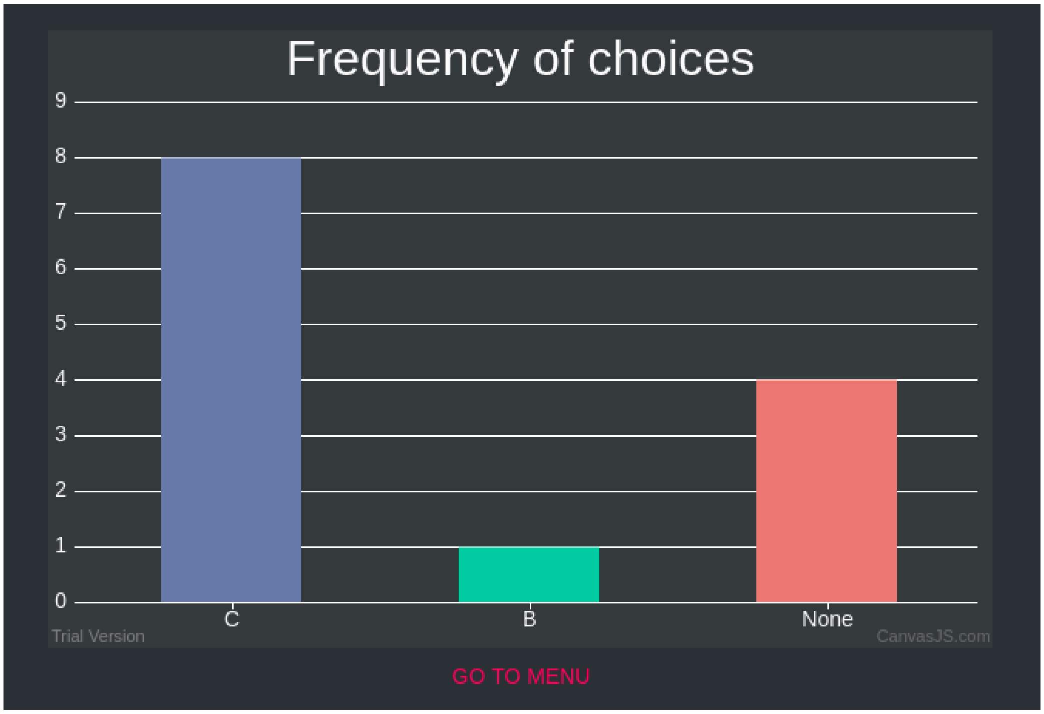

Figure 9: Statistics view with mock data.

code. Having only used React.js briefly in the past, I learned as I built, and so the layout and reusability of components is not optimised. The statistics view was the final component added, and it is the closest to 'good' code structure. By wrapping a 'dumb' stateless component (`Results`) that managed the rendering in a 'smart' stateful component (`ResultView`), responsibilities were better isolated. To ensure components were supplied with all the functions that they required, the proptypes definitions were also added to both of these classes.

The practical highlighted both the benefits and drawbacks of REST as middleware, was a great oppertunity to learn a new front end technology, and was fun to create.

References

[1] Pivotal Web Services. Spring initializr. `start.spring.io`.

[2] Facebook. create-react-app. `github.com/facebook/create-react-app`.

[3] fenopix. Canvasjs. `canvasjs.com`.