Bridgebuff: exportando e explorando estatísticas das partidas de Bridge Defense

Trabalho Prático 3 - Redes de Computadores

Bruno Esteves Campoi {2020054250}[1]
Mirna Mendonça e Silva {2021421940}[1]
Vinicius Silva Gomes {2021421869}[1]

[1] Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte, MG – Brasil

{bec, mirnamendonca, vinicius.gomes}@dcc.ufmg.br

1. Introdução

Nesse TP foi implementado um par cliente-servidor que se comunica usando RPCs através de uma interface REST. O objetivo desse par é exportar estatísticas obtidas um conjunto de partidas do Bridge Defense, como a média de canhões afundados pelos jogadores que mais afundaram navios e o posicionamento dos canhões dos jogadores com os menores números de navios escapados.

Dessa forma, dois programas principais foram desenvolvidos: uma API REST, que realiza uma conexão passiva e responde as requisições feitas pelos clientes, e um programa cliente, que é capaz de produzir arquivos CSV com as estatísticas obtidas para cada um dos comandos disponíveis para ele.

2. Funcionalidades extras disponibilizadas

Além das funcionalidades básicas para o servidor e cliente, que foram definidas na especificação, duas outras *features* foram implementadas pelo grupo: a documentação da API e uma interface *Web*.

Os *endpoints* definidos no trabalho foram documentados através do Swagger na rota `/api/docs`, de modo que seja possível verificar os detalhes de cada rota definida. Dentre essas informações, destacam-se os parâmetros que as rotas esperam e qual o *range* de valores válidos para cada um deles, qual o formato das respostas, quais são os erros que podem ser retornados em cada rota, etc.

Por fim, além do cliente na linha de comando, um cliente *Web* também foi desenvolvido. Esse cliente agrega as informações de cada *ranking* disponível numa tabela paginada e, ao clicar em alguma das linhas, o usuário é redirecionado para uma página separada com todas as informações daquela partida. Os detalhes de implementação dessa interface e das outras aplicações serão discutidos a seguir.

3. Detalhes da implementação

3.1. Desenvolvimento da API

O primeiro passo executado pelo servidor quando é iniciado é ler o JSONL com as partidas e processá-lo em uma lista de dicionários do Python. Os atributos `sunk_ships`, `valid_shots` e `shot_received` foram preenchidos com 0 nos registros em que eles não estavam definidos. Esses dados estarão na classe `Scores`, que fornecerá para os *endpoints* as funções de recuperar uma partida dessa lista a partir do ID e recuperar a lista ordenada de acordo com a definição de cada *endpoint*.

Para a rota que busca as informações de uma partida a partir do ID, primeiramente, o ID é recuperado da rota e a partida referente a ele é procurada na lista de partidas. Caso um jogo exista, ele é retornado em formato JSON, e caso não, o servidor responde um JSON com status 404 e com uma mensagem informando que um jogo com aquele ID não existe.

Para as rotas dos *rankings*, os *query params* `limit` e `start` são recuperados e validados. Caso não sejam informados ou estejam fora dos limites definidos, o servidor responde um JSON com status 400 e uma mensagem identificando o erro observado no *parsing*. Em seguida, ambas as rotas recuperam

a lista de partidas ordenadas de acordo com o critério desejado e os dados são passados para a função `paginate`. Essa função, será responsável por filtrar a lista de IDs de acordo com os valores de `limit` e `start` e, então, determinar se as páginas anterior e próxima existem. Caso sim, as URLs `prev` e `next` serão montadas e a resposta será retornada para o usuário.

Por fim, a API foi documentada através do Swagger. O grupo definiu um JSON com toda a especificação da API, assim como a documentação do Swagger prevê, e utilizou uma biblioteca externa, no caso a `flask-swagger-ui`, para interpretar a especificação no formato JSON e servi-la no formato de uma interface *Web*, de acordo com as *tags* definidas no JSON. Toda a documentação das rotas, parâmetros esperados e range de valores possíveis para eles, possíveis respostas, etc; são acessíveis através da rota `/api/docs`. Existem outras abordagens para escrever uma documentação que escalam melhor para aplicações mais complexas, como o uso de *decorators*. No entanto, dado o tamanho da aplicação a ser desenvolvida, o grupo optou por escrever a documentação completa no formato JSON.

3.2. Cliente de linha de comando

O cliente de linha de comando foi escrito em Python utilizando apenas módulos *built-in* da linguagem. O funcionamento principal dessa aplicação é realizar o *parsing* dos parâmetros recebidos e então chamar a função que executa o comando recebido.

Ambos os comandos definidos operam sobre os *top 100* registros de cada um dos *rankings*. Para tanto, um *pipeline* comum foi definido para buscar os IDs desses primeiros registros e, em seguida, buscar as informações específicas de cada um deles. Para otimizar o número de requisições realizadas, a maior quantidade de partidas que a paginação suporta a cada requisição é solicitada (50), os IDs são salvos numa lista e as informações de cada jogo são buscadas através da rota `/api/game/{id}`.

Em seguida, os dados dos jogos são sumarizados e agregados num dicionário indexado pela informação relevante para cada comando: GAS'es ou a posição normalizada dos canhões. Esse dicionário é, então, ordenado de acordo com a especificação para cada comando e o CSV populado com as informações agregadas é escrito.

3.2.1. *Handler* para lidar com requisições HTTP

A classe `HttpClientHandler` foi escrita para abstrair a comunicação HTTP dentro do cliente, sendo ela a responsável por lidar com as requisições e com as respostas do servidor. A principal função dessa classe é a `make_get_request`, que recebe como parâmetro o caminho para o qual se deseja fazer uma requisição GET.

De início, a função estabelece uma conexão TCP com o servidor através das funções da biblioteca `socket` e envia uma requisição HTTP GET para ele, solicitando o recurso na URL informada como parâmetro. A requisição é formatada como uma String e enviada com a codificação UTF-8.

Após isso, a função passa a ler bytes do *buffer* de recepção da conexão TCP até que todo o cabeçalho da resposta HTTP seja processado, cujo fim é identificado quando uma linha em branco é lida. Com o cabeçalho todo processado, o valor de `Content-Length` é extraído e agora a função sabe quantos bytes a resposta enviada pelo servidor tem.

Com esse valor em mãos, a função lê um byte por vez do *buffer* de recepção da conexão TCP até que todos os bytes esperados na resposta sejam lidos. Esses bytes acumulados formam um JSON válido em formato de String, que será decodificado, em seguida, para um dicionário Python através da função `loads`, do módulo *built-in* `json`. Caso a decodificação seja feita com sucesso, a função retorna esse dicionário para a função que a chamou.

3.3. Interface *Web*

A interface *Web* foi projetada com a proposta de servir como uma visualização direta dos dados retornados pelos *endpoints* da API. Portanto, a ideia é que ela disponha os dados recebidos das rotas paginadas numa

tabela e permita que o usuário seja capaz escolher qual ordenação ele deseja visualizar: as partidas em ordem decrescente do número de navios afundados ou as partidas em ordem crescente do número de navios escapados. Dependendo da ordenação escolhida, a requisição feita pelo cliente será para uma rota diferente.

Para popular a tabela e apresentar uma visão um pouco mais detalhada de cada jogo, o cliente busca a página de IDs e depois busca as informações de cada uma das partidas a partir do ID, popula um *array* com essas informações e exibe na interface. Dessa forma, é possível exibir mais informações sobre cada jogo nas colunas da tabela. Todas as requisições são feitas de forma sequencial, de modo a preservar a ordem com que os dados são retornados na paginação.

Por fim, ao clicar no GAS ou no ID de um jogo, o cliente redireciona o usuário para uma página que apresenta um *overview* completo daquela partida. Isso é feito através de uma rota no *frond-end* que contém o ID do jogo que o usuário clicou e, então, o cliente busca no servidor as informações dele. Para a realização das requisições, foi utilizada a biblioteca `axios`, que descomplica o processo de se realizar requisições, receber e manipular respostas HTTP.

3.4. Escolha dos *frameworks*

Para o desenvolvimento do servidor, alguns aspectos foram avaliados antes da escolha da linguagem de programação e do *framework* utilizado. A primeira decisão feita pelo grupo foi escolher uma linguagem de programação que fosse familiar para todos os integrantes. Com isso em mente e seguindo a sugestão do professor, chegamos ao consenso de utilizar Python.

Dentre as alternativas disponíveis para o *framework*, como Flask, FastAPI, Web2py, Django, etc; optamos pelo Flask. Após analisar a documentação dessas ferramentas e como o servidor a ser desenvolvido não era complexo ou implementava *features* mais avançadas, como escritas num banco de dados, concluímos que o Flask era aquele que oferecia a API mais simples e versátil e, portanto, era a opção mais adequada. Ele possibilitou que escrevêssemos o servidor utilizando a organização mais conveniente para o grupo, além de fornecer funções que facilitavam a manipulação dos parâmetros e o retorno de respostas com status de sucesso ou de erro.

Por fim, a escolha do *framework* para o desenvolvimento da interface *Web* se deu principalmente a partir da familiaridade dos integrantes do grupo com cada ferramenta e a facilidade de se fazer o que propusemos em cada uma delas. Após discussão, chegamos a conclusão que o React ofereceria todas as APIs necessárias para desenvolver a interface e fazer as atualizações na DOM de forma descomplicada, enquanto mantinha o código simples e fácil de manter. De modo geral, outros *frameworks Web*, como o Vue.js, Svelte ou Angular; também seria competentes para a tarefa. Nesse sentido, um fator decisivo para a escolha do React em detrimento dos outros foi a sua alta adesão por parte da comunidade. Bibliotecas bem documentadas e muitas discussões ativas resolvendo problemas comuns facilitaram bastante o processo de *debug* durante o desenvolvimento.

4. Dificuldades encontradas

Diversas dificuldades foram enfrentadas ao longo do desenvolvimento das aplicações. Dentre elas, as mais notáveis foram a escrita da paginação generalista, a escrita da documentação utilizando o Swagger e a escrita do *handler* que lida com requisições e recebe respostas HTTP.

A escrita da paginação, em especial a definição das URLs `prev` e `next`, foi desafiadora no início devido a dificuldade de se ajustar corretamente os novos valores para `limit` e `start`. Após diversos testes que exercitavam diferentes casos para esses valores, foi possível determinar o cálculo correto para os parâmetros e esse desafio foi superado.

A escrita da documentação com o Swagger também foi relativamente complicada por causa das diferentes formas possíveis de escrever a especificação da API utilizando esse padrão. Essas formas foram investigadas e exemplos de documentações classificados como bons pelo grupo foram usados de base para que o grupo fosse capaz de escrever a especificação da API do trabalho.

Por fim, o último desafio foi o desenho do *handler* que trocava mensagens com o servidor utilizando o protocolo HTTP. O envio das requisições foi resolvido com certa facilidade, no entanto, a implementação

inicial do processamento das respostas lia mais bytes do que deveriam ser lidos. Foi então que percebemos que o conteúdo da resposta HTTP sempre vem depois de uma linha em branco, assim, era possível identificar onde o cabeçalho terminava e o corpo da resposta começava. Com o cabeçalho acumulado, o campo `Content-Length` podia ser processado e, então, era possível ler o número de bytes corretos da resposta sem que houvesse o risco da leitura excedê-los e acabar lendo o conteúdo de outra resposta.

5. Como executar cada aplicação

5.1. API

Para executar a API REST desenvolvida, o primeiro passo é acessar o diretório `server` e executar os comandos a seguir:

```
python3 -m venv .venv
.  .venv/bin/activate
pip3 install -r requirements.txt
```

Esses comandos são responsáveis por criar o *virtual environment* e instalar as versões apropriadas de todas as bibliotecas necessárias para que o servidor possa ser executado. Uma vez que tudo tenha sido instalado corretamente, o comando

```
python3 server.py
```

pode ser executado para iniciar a API. Por padrão, o Flask inicia o servidor no endereço de *loopback* IPv4, utilizando a porta 5000. Essas informações foram mantidas como padrão e, portanto, o acesso do servidor deve acontecer no endereço `http://127.0.0.1:5000`.

5.2. Cliente de linha de comando

Para executar as funcionalidades do cliente, basta inicializar o servidor assim como descrito na seção anterior, acessar o diretório `client` e rodar o comando

```
python3 client.py <IP> <PORT> <ANALYSIS> <OUTPUT>
```

5.3. Interface *Web*

Para executar a interface *Web*, é necessário ter o Node.js e algum gerenciador de pacotes JavaScript instalado. Para a instalação de ambos, o grupo recomenda a documentação oficial do Node.js. Basta seguir os passos descritos no guia para instalar a versão LTS do Node e o gerenciador de pacotes NPM.

Com o *runtime* e o gerenciador de pacotes instalados, basta acessar o diretório `web` e executar os comandos a seguir para baixar os pacotes utilizados no desenvolvimento da interface e então executá-la:

```
npm install
npm run dev
```

Por padrão, a interface estará disponível no endereço `http://localhost:5173`.

6. Solução de problemas

De modo geral, ao seguir os guias de *setup* de ambiente e execução de cada aplicação, todas eles devem ser capazes de rodar apropriadamente. O grupo apenas enfatiza o cuidado na hora de acessar as pastas certas para executar os comandos, o uso de versões mais recentes do Python (3.10.12+) e do Node (20.15+) e o cuidado de sempre executar o servidor antes de consumir os clientes implementados.

Por fim, o *setup* de ambiente virtual que foi descrito para o servidor considera que os comandos estejam sendo executados no macOS ou Linux. Caso o programa esteja sendo executado no Windows, sugerimos que a documentação do Flask sobre essa parte seja consultada. No caso de quaisquer outros problemas, os membros do grupo estão a disposição para auxiliar na resolução deles.