Universidade de Brasília - UnB
Faculdade UnB Gama - FGA
Engenharia de Software

BROMS: Uma nova apresentação visual e integração com BOCA e Codeforces

Autores: Leonardo de Araujo Medeiros
Lieverton Santos Silva

Orientador: Professor Dr. Edson Alves da Costa Júnior

Brasília, DF
2022

Leonardo de Araujo Medeiros
Lieverton Santos Silva

BROMS: Uma nova apresentação visual e integração com BOCA e Codeforces

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Professor Dr. Edson Alves da Costa Júnior

Brasília, DF
2022

Leonardo de Araujo Medeiros
Lieverton Santos Silva
BROMS: Uma nova apresentação visual e integração com BOCA e Codeforces/
Leonardo de Araujo Medeiros
Lieverton Santos Silva . – Brasília, DF, 2022-
79 p. : il. (algumas color.) ; 30 cm.

Orientador: Professor Dr. Edson Alves da Costa Júnior

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2022.

1. Palavra-chave01. 2. Palavra-chave02. I. Professor Dr. Edson Alves da Costa Júnior. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. BROMS: Uma nova apresentação visual e integração com BOCA e Codeforces

CDU 02:141:005.6

Leonardo de Araujo Medeiros
Lieverton Santos Silva

BROMS: Uma nova apresentação visual e integração com BOCA e Codeforces

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Trabalho aprovado. Brasília, DF, 30 de setembro de 2022:

Professor Dr. Edson Alves da Costa Júnior
Orientador

Professor Dr. Bruno Cesar Ribas
Convidado 1

Professor Dr. Daniel Saad Nogueira Nunes
Convidado 2

Brasília, DF
2022

Resumo

O BROMS é um projeto em desenvolvimento de um placar para competições de programação, idealizado para funcionar integrado ao BOCA. Contudo, em sua versão inicial não atendia a este objetivo e carecia de melhorias. Com o intuito de dar prosseguimento ao desenvolvimento do BROMS, este trabalho realizou a integração do projeto aos sistemas Codeforces e BOCA. A partir dessa integração é possível construir placares ao vivo dos eventos realizados em ambas plataformas. Este trabalho contempla também o desenvolvimento de novas funções relacionadas a apresentação visual do placar, bem como melhorias realizadas quanto a qualidade e manutenibilidade do código.

Palavras-chaves: placar. integração. API. requisições web.

Abstract

BROMS is a project under development for a scoreboard to be used in programming competitions, designed to work integrated with BOCA. However, in its initial version it did not meet this objective and needed improvements. In order to continue the development of BROMS, this work carried out the integration of the project to the Codeforces and BOCA systems. From this integration it is possible to build live scores of events held on both platforms. This work also includes the development of new functions related to the visual presentation of the scoreboard, as well as improvements made regarding the quality and maintainability of the code.

Key-words: scoreboard. integration. API. web requests.

Lista de ilustrações

Figura 1 – Maratona Rustrimeitor 25
Figura 2 – Trecho do arquivo *contest* fornecido pelo BOCA 27
Figura 3 – Trecho do arquivo *runs* fornecido pelo BOCA 28
Figura 4 – BROMS, visual antigo. 30

Figura 5 – BROMS, demonstração do novo layout. 35
Figura 6 – BROMS, Forma de Visualização: primeiros a acertar. 42
Figura 7 – BROMS, Forma de Visualização: informações extra. 42
Figura 8 – Trecho do arquivo `contest` que foi alterado para simulação. 43
Figura 9 – Comunicação com o Codeforces 44
Figura 10 – Comunicação com o Boca - *API mock* 49
Figura 11 – Comunicação com o Boca - *API mock* 50

Figura 12 – Linha do tempo do desenvolvimento do BROMS 52

Figura 13 – Requisições feitas à API do Codeforces 60
Figura 14 – Tempo de resposta das requisições feitas à API do Codeforces 61
Figura 15 – Tamanho das requisições feitas à API do Codeforces 61
Figura 16 – Tamanho e tempo de resposta das requisições feitas à API do Codeforces 62
Figura 17 – Placar da prova de APC 63
Figura 18 – Resultado final da prova de APC 64
Figura 19 – Erro de time duplicado 65
Figura 20 – *Top* 5 da prova de TEP no placar 65
Figura 21 – *Top* 5 da prova de TEP no Codeforces 65
Figura 22 – Erro de pontuação negativa 66

Lista de tabelas

Tabela 1 – Cálculo da penalidade pela regra do ICPC 24
Tabela 2 – Cálculo da penalidade pela regra do ICPC no Codeforces 24

Tabela 3 – Parâmetros utilizados no experimento 1 56
Tabela 4 – Parâmetros utilizados no experimento 2 56
Tabela 5 – Parâmetros utilizados no experimento 3 56

Lista de Quadros

1 Detalhes do objeto `contest` 31
2 Detalhes do objeto `team` 31
3 Detalhes do objeto `run` 32

4 Atividades por sprint 54

5 Detalhes do objeto `contest` 75
6 Detalhes do objeto `problem` 76
7 Detalhes do objeto `ranklistRow` 76
8 Detalhes do objeto `party` 77
9 Detalhes do objeto `member` 77
10 Detalhes do objeto `problemResult` 77

11 Detalhes do objeto `submission` 79

Lista de códigos fonte

Figura 1 – Código antigo, duplicações na classe `Row` 36
Figura 2 – Trecho principal da implementação da classe `Parallelogram` 37
Figura 3 – Trecho principal da implementação da classe `Text` 38
Figura 4 – Trecho da nova implementação da classe `Row` 39
Figura 5 – Atualização da tela feita com base no tempo 40
Figura 6 – Trecho principal da implementação da classe `EventManager` 40
Figura 7 – Trecho de código da classe `Camera` implementando o método `onEvent` . 41
Figura 8 – Interface `getContest()` 44
Figura 9 – Método `getMockContest()` 45
Figura 10 – Método `getCFContest()` 45
Figura 11 – Método getApiUrl() 46
Figura 12 – Métodos Conversores 47
Figura 13 – Executando o servidor mínimo Python 51
Figura 14 – Executando a *API mock* 52

Lista de abreviaturas e siglas

APC	Algorítimos e Programação de Computadores
API	*Application Programming Interface*
BROMS	*Brazilian Online Marathon Scoreboard*
CF	Codeforces
FGA	Faculdade UnB Gama
HTTP	*Hypertext Transfer Protocol*
ICPC	*International Collegiate Programming Contest*
JSON	*JavaScript Object Notation*
OBI	Olimpíada Brasileira de Informática
UnB	Universidade de Brasília
URL	*Uniform Resource Locator*
XP	*Extreme Programming*

Sumário

Introdução 21

1 REFERENCIAL TEÓRICO 23
1.1 Programação Competitiva 23
1.2 Juízes Eletrônicos 25
1.3 BOCA 26
1.4 Codeforces 27
1.4.1 API do Codeforces 28
1.4.1.1 Método `contest.standings` 29
1.4.1.2 Método `contest.status` 30
1.5 BROMS 30
1.5.1 *API Mock* 32

2 BROMS: NOVAS IMPLEMENTAÇÕES 35
2.1 Organização do código 35
2.2 Arquitetura 38
2.3 Animações 39
2.4 Visualizações 41
2.4.1 Primeiros a acertar 42
2.4.2 Visualização completa 42
2.4.3 Visualização por agrupamento 42
2.5 Comunicação com a API do Codeforces 43
2.6 Comunicação com o BOCA 48

3 METODOLOGIA 51
3.1 Ferramentas de Desenvolvimento e Ambiente 51
3.2 Metodologia de Desenvolvimento 52
3.3 Ensaios de Integração com a API do Codeforces 55
3.3.1 Parâmetros Utilizados 56
3.3.2 Procedimento 56
3.3.3 Possíveis Erros 57
3.4 Ensaio de Integração com o BOCA 57
3.4.1 Procedimento 57

4 RESULTADOS 59
4.1 Integração com a API do Codeforces 59

4.2 Integração com o BOCA 66

5 CONSIDERAÇÕES FINAIS 69

REFERÊNCIAS 71

ANEXOS 73

ANEXO A – OBJETOS QUE COMPÕEM O RETORNO DO MÉTODO CONTEST.STANDINGS DA API DO CODEFORCES 75

ANEXO B – OBJETO QUE COMPÕE O RETORNO DO MÉTODO CONTEST.STATUS DA API DO CODEFORCES 79

Introdução

O BROMS surgiu com o objetivo de ser um placar virtual para competições de programação hospedadas no BOCA, portável para a maioria dos computadores por meio de um navegador de internet, com uma instalação relativamente simples e com o mínimo de dependência possível (MAKAHA, 2021). O BROMS propõe funcionalidades que facilitam a transmissão dos eventos, como atualização do placar de forma automática e visualizações com informações.

Contudo, o BROMS em sua versão inicial apenas simula uma competição por intermédio de um servidor que lê dados de uma competição já finalizada de um arquivo estático provindo do BOCA, não atendendo ao seu objetivo principal, que seria acompanhar um evento em andamento. Outras funcionalidades do placar propostas por Makaha (2021) como as animações e novas visualizações, também não estão presentes nessa versão inicial.

O presente trabalho propõe dar continuidade no desenvolvimento do projeto implementando uma série de melhorias além de realizar a integração entre BROMS e os sistemas Codeforces e BOCA.

Quanto aos aspectos visuais, este trabalho objetiva adicionar funcionalidades voltadas para a transmissão dos eventos, com foco na apresentação e na usabilidade do apresentador, possibilitando o uso de atalhos de teclado para controlar a exibição das informações na tela.

Em relação à comunicação, o Codeforces foi escolhido por ser uma importante plataforma para comunidade de programação competitiva. De acordo com Mirzayanov (2011), em 2017 a plataforma contava com mais de 600 mil usuários registrados. O sistema conta com facilidades para realização de competições, incluindo um grande acervo de problemas, além de uma API voltada para desenvolvedores que possibilita a exportação dos dados de uma competição. A integração com o Codeforces objetiva servir como prova de conceito do funcionamento do BROMS em uma competição em tempo real, além de servir como base para a integração com o BOCA, que é o objetivo principal.

O BOCA foi desenvolvido para ser usado na Maratona de Programação da Sociedade Brasileira de Computação (SBC) (CAMPOS; FERREIRA, 2004). Por conseguinte, o BROMS tem como objetivo final ser utilizado na transmissão dos eventos SBC.

Objetivos

Este trabalho tem como objetivo dar continuidade no desenvolvimento do projeto BROMS. Nessa perspectiva será implementada a sincronização entre o BROMS e os placares de competições hospedadas no Codeforces e no BOCA.

Os objetivos específicos são:

- manter o menor número de dependências possível;
- desenvolver as animações visuais do placar;
- desenvolver as novas formas de visualização do placar;
- desenvolver o módulo responsável pela comunicação com a API do Codeforces;
- desenvolver uma API no BOCA para disponibilizar dados das competições;
- desenvolver o módulo responsável pela comunicação com a API do BOCA.

Estrutura do Trabalho

O presente trabalho é dividido em quatro capítulos. No Capítulo Referencial Teórico é apresentada a fundamentação teórica, contendo trabalhos relacionados às definições e conceitos presentes neste trabalho. No Capítulo Metodologia são apresentados os métodos aplicados na implementação do software e os processos realizados na coleta de dados para concepção dos resultados. No Capítulo Resultados são apresentados os resultados obtidos nos ensaios experimentais realizados. E, por fim, no Capítulo Considerações Finais são apresentadas as considerações finais dos autores do trabalho.

1 Referencial Teórico

O presente capítulo compreende a fundamentação teórica do trabalho. A Seção Programação Competitiva apresenta o conceito do referido termo e as ideias a ele associadas. Já a Seção Juízes Eletrônicos expõe detalhes de um juiz eletrônico. Em seguida a Seção BOCA apresenta o sistema BOCA, enquanto a Seção Codeforces aborda o Codeforces e sua API. Por fim, a Seção BROMS apresenta o BROMS, o objeto de estudo deste trabalho.

1.1 Programação Competitiva

A programação competitiva é a combinação de dois tópicos, o projeto de algoritmos, que consiste na resolução de problemas e no pensamento lógico, e a implementação de algoritmos, que requer boas habilidades programando (LAAKSONEN, 2018).

As maratonas de programação são eventos de programação competitiva. As regras das maratonas dependem do organizador. Por exemplo, na plataforma online Codeforces (MIRZAYANOV, 2011), a pontuação é realizada com base em valores diferentes para cada questão, proporcionais à dificuldade do problema e que decaem ao longo do tempo da competição. No caso do IEEEXtreme (IEEE, 2021), a pontuação de cada questão varia de acordo com o número de times que conseguiram resolvê-la, sendo as questões mais resolvidas consideradas fáceis e resultando em uma menor pontuação.

Este trabalho tem foco nas regras do *International Collegiate Programming Contest* (ICPC), responsável pela principal competição deste ramo, que consiste em um evento de múltiplas fases e que culmina em uma final mundial. Nestes eventos o vencedor é aquele que resolve a maior quantidade de problemas. Caso haja empate, o critério de desempate é quem possui o menor valor acumulado do tempo que o time levou para responder cada problema, adicionado de uma penalidade extra para cada submissão incorreta. A penalidade é aplicada apenas caso a equipe venha a acertar o problema tentado. Nos momentos finais deste tipo de evento é comum ocorrer a etapa *frozen*, em que há o congelamento do placar geral. Uma outra etapa parecida com a *frozen* é a etapa *blind* onde não serão atualizados tanto o placar quanto qualquer informação a respeito das submissões, contudo, o *blind* não é mais usado na maratona de programação ICPC. Com o encerramento do evento é revelado o resultado final.

Na Tabela 1 há um exemplo de como seria calculada a penalidade de um time ao longo de um evento fictício de apenas 3 problemas onde a penalidade por submissão errada é de 20 pontos.

Tabela 1 – Cálculo da penalidade pela regra do ICPC

Tempo	**Evento**	**Penalidade**			**Total**
		A	**B**	**C**	
0	Inicio do contest	0	0	0	0
25	Submissão correta para A	25			25
32	Submissão incorreta para B	25	(+1)		25
40	Submissão correta para C	25		40	65
45	Submissão correta para B	25	45(+1)	40	130

Fonte: Autores

Como mencionado anteriormente, o Codeforces possui suas próprias regras de pontuação, entretanto algumas outras regras podem ser usadas, inclusive do ICPC (MIRZAYANOV, 2011). Contudo, mesmo usando o sistema de pontuação do ICPC, a pontuação no Codeforces possui nuances no cálculo da penalidade. O Codeforces desconsidera no cálculo da penalidade as submissões que tenham apresentado erro de compilação ou que não passaram nos casos base de teste da questão.

A Tabela 2 mostra o cálculo no Codeforces do mesmo evento fictício anterior para o mesmo time, porém a submissão incorreta da questão B possui erro de compilação. Para esse caso há uma diferença de 20 pontos em comparação com o sistema de pontuação oficial do ICPC.

Tabela 2 – Cálculo da penalidade pela regra do ICPC no Codeforces

Tempo	**Evento**	**Penalidade**			**Total**
		A	**B**	**C**	
0	Inicio do contest	0	0	0	0
25	Submissão correta para A	25			25
32	Erro de compilação para B	25			25
40	Submissão correta para C	25		40	65
45	Submissão correta para B	25	45	40	110

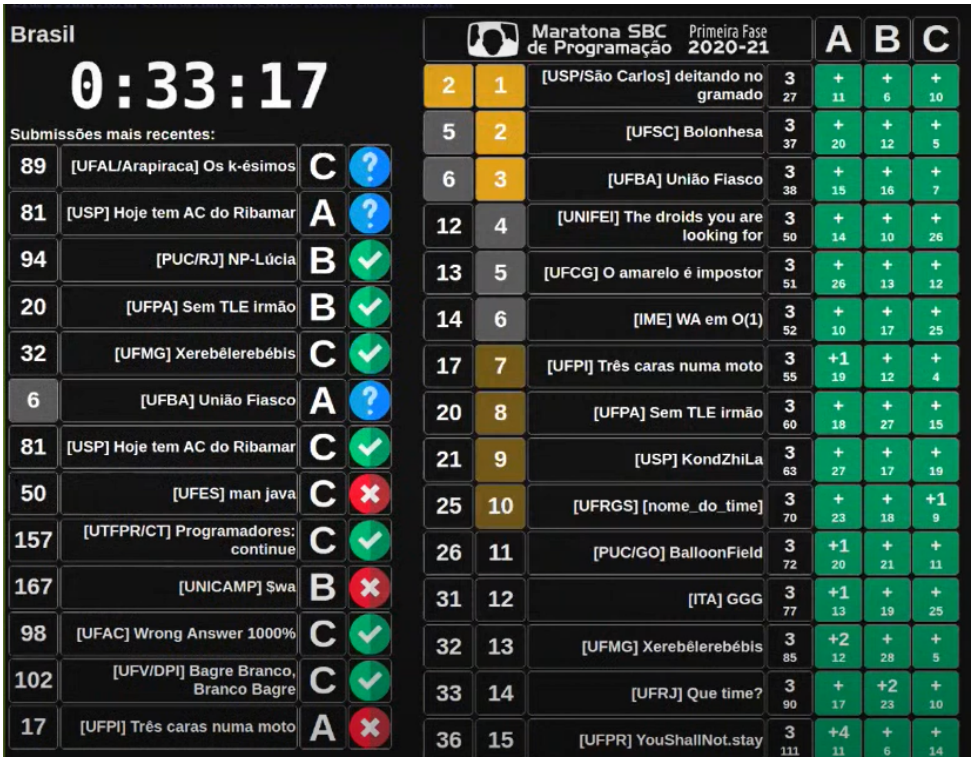
Fonte: Autores

No Brasil, os eventos mais populares são a Olimpíada Brasileira de Informática (OBI), voltadas ao público mais jovem e a Maratona SBC, organizada pela Sociedade Brasileira de Computação (SBC), que faz parte do ICPC. A Maratona SBC segue a regra de pontuação do ICPC, enquanto a OBI possui regulamento próprio. Um participante na OBI pode receber entre zero e 100 pontos, dependendo do número de testes para o qual a sua solução para a respectiva tarefa produz a resposta correta em cada questão (UNICAMP, 2022).

Os principais eventos, como a Maratona SBC e a final mundial do ICPC, são

geralmente apresentados por transmissões online, onde é feito o acompanhamento em tempo real do evento, com a colocação dos times em cada momento da competição. Nestas transmissões, geralmente é feito uso de um placar com estas informações. As edições mais recentes da Maratona SBC utilizaram o Maratona Rustrimeitor (RUSTRIMEITOR, 2021), exibido na Figura 1.

Figura 1 – Maratona Rustrimeitor



Fonte: Rustrimeitor (2021)

O software alvo deste trabalho é o BROMS, um placar para estas maratonas e será melhor definido na Seção BROMS.

1.2 Juízes Eletrônicos

Os juízes eletrônicos são sistemas que possibilitam a correção automática de soluções para problemas de programação. Para iniciar o processo de correção é necessário que o usuário submeta o código-fonte da solução proposta ao juiz eletrônico, levando em consideração as linguagens de programação aceitas pelo sistema. A submissão, também denotada como *run*, é enviada via rede ao juiz eletrônico. A correção é feita através de testes unitários que comparam as saídas do programa a determinadas entradas preestabelecidas. Em linguagens compiladas o sistema fará a compilação do programa antes da execução do mesmo. Uma solução é considerada correta se todos os casos de teste passarem dentro dos limites de tempo e memória definidos para o problema.

Segundo Skiena e Revilla (2003), os juízes eletrônicos, em geral, fornecem pouca informação sobre uma correção, não indicando em qual caso de teste o programa falhou, nem quaisquer dicas adicionais sobre o porquê ele falhou além dos prováveis veredictos retornados que são:

- ***Accepted* (AC):** O programa está correto e roda dentro dos limites de tempo e memória estabelecidos.
- ***Presentation Error* (PE):** As saídas do programa estão corretas, mas não são apresentadas no formato especificado.
- ***Wrong Answer* (WA):** O programa retornou uma resposta incorreta para um ou mais casos de teste.
- ***Compile Error* (CE):** Não foi possível compilar o programa.
- ***Runtime Error* (RE):** O programa falhou durante a execução devido a uma falha de segmentação, exceção de ponto flutuante ou problema semelhante. Comumente esse erro é causado por referências de ponteiro inválidas, acesso fora de um *range* ou divisão por zero.
- ***Time Limit Exceeded* (TLE):** O programa excedeu o limite de tempo estabelecido em pelo menos um dos casos de teste.
- ***Memory Limit Exceeded* (MLE):** O programa usou mais memória do que o limite estabelecido durante o processamento de ao menos um caso de teste.

Os sistemas BOCA e Codeforces são exemplos de sistemas que possuem módulos que agem como juízes eletrônicos. Esses sistemas são definidos respectivamente nas Seções BOCA e Codeforces.

1.3 BOCA

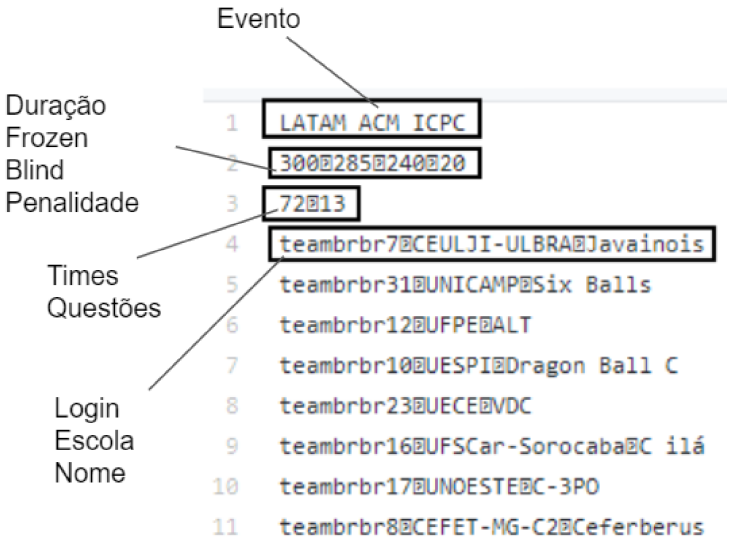
O termo BOCA é o acrônimo recursivo de BOCA Online Contest Administrator. Como o próprio nome se refere o BOCA[1] é um sistema de administração de competições de programação. O sistema tem grande relevância para a comunidade. O BOCA foi desenvolvido para ser usado na Maratona de Programação da Sociedade Brasileira de Computação (SBC) (CAMPOS; FERREIRA, 2004). Além de ser usado uma importante competição brasileira de programação universitária, o sistema também é profusamente usado no apoio a disciplinas em diversas instituições de ensino. O BOCA foi desenvolvido usando a linguagem de programação PHP: Hypertext Preprocessor (PHP). Para

[1] link do repositório oficial do boca: <https://github.com/cassiopc/boca>

armazenamento dos dados e controle de concorrência é usado o banco de dados relacional PostgreSQL (CAMPOS; FERREIRA, 2004). O sistema é um software livre e está licenciado sob a *GNU Public License 3* (GPL 3.0).

O BOCA pode disponibilizar dados do estado corrente de uma competição por meio de dois arquivos principais: *contest* e *runs* (MAKAHA, 2021). O arquivo *contest* contém informações da competição como o nome do evento, o tempo total da competição, a penalidade por erro, a quantidade de questões e as informações de cada time. A Figura 2 mostra um trecho de um arquivo *contest* e revela como os dados estão dispostos nos arquivos. De forma similar a Figura 3 detalha um trecho de um arquivo *runs* que contém uma lista com as submissões realizadas na competição.

Figura 2 – Trecho do arquivo *contest* fornecido pelo BOCA



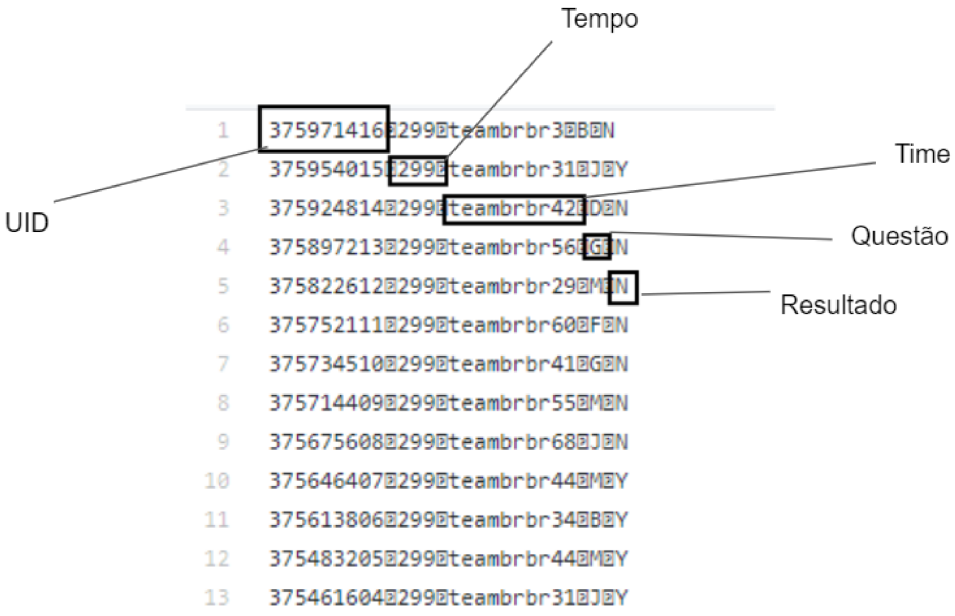
Fonte: Makaha (2021)

A forma de acessar os arquivos citados é descrita na Seção Comunicação com o BOCA do Capítulo Metodologia.

1.4 Codeforces

De acordo com Mirzayanov (2011), o Codeforces é uma rede social que reúne interessados em competições de programação. Competições são realizadas regularmente na plataforma, no qual o desempenho de cada participante é refletido em um *ranking* geral e as questões ficam disponíveis com o objetivo de serem usadas como forma de estudo e preparação. Além do mais, a plataforma permite aos participantes a oportunidade de organizar suas próprias competições para treinamento e aprendizado. Por fim, o Codeforces disponibiliza uma API para acesso a dados em formato JSON a desenvolvedores por meio de requisições do tipo `GET`.

Figura 3 – Trecho do arquivo *runs* fornecido pelo BOCA



Fonte: Makaha (2021)

1.4.1 API do Codeforces

Uma interface é um ponto de entrada ou saída bem definido em um sistema com o objetivo de favorecer uma interação. Não diferente, uma *Application Programming Interface* (API) é um meio de acesso facilitado, sem a exposição da complexidade do sistema um todo (PREIBISCH, 2018).

As APIs web são pontos de acesso a servidores, nos quais é permitida a comunicação com sistemas clientes por meio de requisições usando o protocolo *Hypertext Transfer Protocol* (HTTP). Em geral, os dados transmitidos se encontram no formato JavaScript Object Notation (JSON). Os principais métodos de requisição HTTP são:

- `GET`: busca um recurso,
- `POST`: adiciona um recurso,
- `PUT`: altera um recurso,
- `DELETE`: exclui um recurso.

Como dito anteriormente, o Codeforces possui uma API que disponibiliza alguns dados do sistema. O foco será dado nos métodos `contests.standings` e `contest.status`, que contêm, respectivamente, dados a respeito de competições e submissões.

O acesso aos dados pode ser feito a partir de uma requisição HTTP do tipo `GET` ao endereço <https://codeforces.com/api/{methodName}> com parâmetros específicos

do método. Entretanto, dessa forma, somente os dados públicos poderão ser acessados via API.

Para acessar dados privados, como os dados de uma competição privada, é necessário realizar a autenticação. Deve ser gerada uma chave de API na página <https://codeforces.com/settings/api> que será usada na requisição *web*, sendo que a chave é composta por dois parâmetros: `key` e `secret`. Para construir a URL da requisição, além do endereço base <https://codeforces.com/api/{methodName}>, é preciso adicionar os seguintes parâmetros:

1. **`apiKey:`** deve ser igual a `key`;
2. **`time:`** hora atual no formato Unix;
3. **`apiSig:`** os seis primeiros caracteres devem ser randômicos, denotados como `rand`. O restante do parâmetro é uma representação hexadecimal do código `hash SHA-512` da seguinte `query string: <rand>/<methodName>?param1=value1&param2=value2 ...&paramN=valueN#<secret>` onde `(param_1, value_1), param_2, value_2), ..., (param_n, value_n)` estão todos os parâmetros da requisição – incluindo `apiKey`, `time`, mas excluindo `apiSig` – com valores correspondentes, classificados lexicograficamente primeiro por `param_i`, depois por `value_i`.

1.4.1.1 Método `contest.standings`

O método `contest.standings` retorna dados sobre a competição e a classificação. O retorno é composto por um objeto JSON formado por três outros objetos: `contest`, `problems` e `rows`. O `contest` contém os dados da competição em si enquanto os atributos `problems` e `rows` são listas de objetos de `problem` e `ranklistRow` respectivamente. O objeto `problem` contém os dados de uma questão da competição, já o `ranklistRow` contém os dados de um time participante da competição.

O retorno desse método é complexo e muitos dos atributos encontrados nos objetos não são aproveitados no contexto do BROMS. A partir disso os atributos usados no BROMS são definidos na Seção Comunicação com a API do Codeforces do Capítulo Metodologia. Contudo, o Anexo A apresenta os detalhes de cada objeto que compõem o retorno do método `contest.standings`.

O método `contest.standings` aceita seis parâmetros: `contestId`, `from`, `count`, `handles`, `room`, `showUnofficial` (MIRZAYANOV, 2011). O único campo obrigatório e que possui relevância para o BROMS é o `constestId`, que é o identificador para competição. Com o uso dos outros parâmetros é possível restringir a consulta a um grupo ou intervalo de participantes, mas tal recurso não será utilizado no desenvolvimento do BROMS.

1.4.1.2 Método `contest.status`

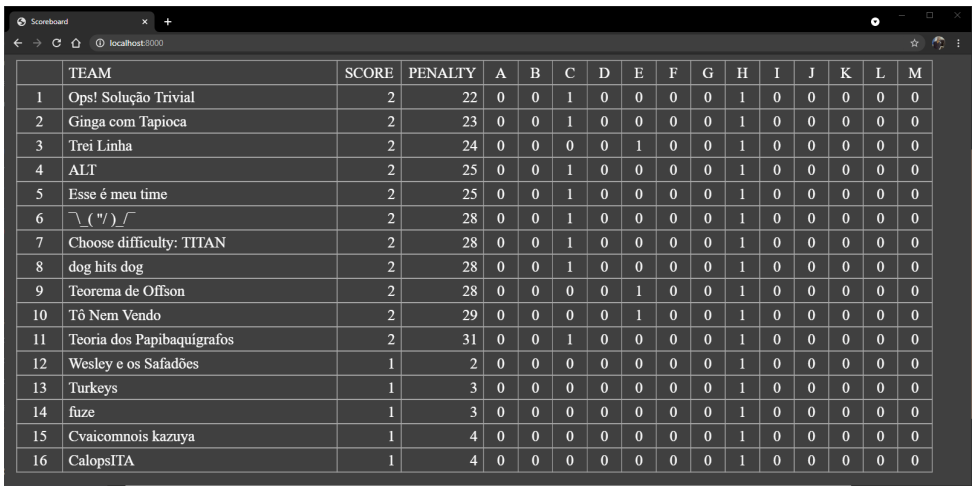
O método `contest.status` retorna uma lista em ordem decrescente de envio com dados sobre as submissões feitas na competição (MIRZAYANOV, 2011). Entre esses dados estão o identificador da submissão, o tempo da submissão, o problema respondido, o autor e o veredito. Mais detalhes sobre o objeto de retorno podem ser vistos no Anexo B.

Assim como o método `contest.standings`, o método `contest.status` também aceita parâmetros na consulta, sendo que o `constestId` também é obrigatório. Porém, para esse método, são aceitos apenas outros 3 parâmetros: `from`, `count`, `handle`. Com o `handle` definido é possível restringir a consulta com apenas as submissões de um usuário. Os parâmetros `from` e `count` servem para limitar a consulta a um intervalo específico de submissões. O `from` é o índice base da primeira submissão a ser retornada e o `count` é o número de submissões a serem retornadas.

1.5 BROMS

O *Brazilian Online Marathon Scoreboard* (BROMS) é um software em desenvolvimento, com origem em um trabalho de conclusão de curso, sendo inicialmente desenvolvido por Makaha (2021) e agora neste trabalho, e tem como objetivo final ser usado para a apresentação dos eventos de programação competitiva.

Figura 4 – BROMS, visual antigo.



	TEAM	SCORE	PENALTY	A	B	C	D	E	F	G	H	I	J	K	L	M
1	Ops! Solução Trivial	2	22	0	0	1	0	0	0	0	1	0	0	0	0	0
2	Ginga com Tapioca	2	23	0	0	1	0	0	0	0	1	0	0	0	0	0
3	Trei Linha	2	24	0	0	0	0	1	0	0	1	0	0	0	0	0
4	ALT	2	25	0	0	1	0	0	0	0	1	0	0	0	0	0
5	Esse é meu time	2	25	0	0	1	0	0	0	0	1	0	0	0	0	0
6	¯_("/)_/¯	2	28	0	0	1	0	0	0	0	1	0	0	0	0	0
7	Choose difficulty: TITAN	2	28	0	0	1	0	0	0	0	1	0	0	0	0	0
8	dog hits dog	2	28	0	0	1	0	0	0	0	1	0	0	0	0	0
9	Teorema de Offson	2	28	0	0	0	0	1	0	0	1	0	0	0	0	0
10	Tô Nem Vendo	2	29	0	0	0	0	1	0	0	1	0	0	0	0	0
11	Teoria dos Papibaquígrafos	2	31	0	0	1	0	0	0	0	1	0	0	0	0	0
12	Wesley e os Safadões	1	2	0	0	0	0	0	0	0	1	0	0	0	0	0
13	Turkeys	1	3	0	0	0	0	0	0	0	1	0	0	0	0	0
14	fuze	1	3	0	0	0	0	0	0	0	1	0	0	0	0	0
15	Cvaicomnois kazuya	1	4	0	0	0	0	0	0	0	1	0	0	0	0	0
16	CalopsITA	1	4	0	0	0	0	0	0	0	1	0	0	0	0	0

Fonte: Makaha (2021)

O BROMS, exibido na Figura 4, consiste em uma aplicação de *front-end*, exibida em janela de navegador, capaz de exibir em tempo real o estado de um evento de programação competitiva. Inicialmente idealizado para competições realizadas no sistema BOCA com suporte para o sistema de pontuação do ICPC, o BROMS também conta com a *API mock*, uma aplicação extra desenvolvida para testar o placar.

Dentre os principais objetivos do BROMS está a minimização de dependências, propondo uma solução que permite executar o placar apenas utilizando um servidor mínimo que fornece as informações ao navegador. Este servidor depende apenas do Python, que atualmente já se encontra instalado por padrão na maioria dos sistemas operacionais.

Para a construção e manutenção do placar o BROMS possui quatro métodos que são responsáveis por fazer as requisições à API:

- `getContest()`: retorna os dados da competição,
- `getContestEnd()`: retorna um booleano que, se verdadeiro, a competição foi finalizada,
- `getRuns()`: retorna todas as submissões atuais de uma competição,
- `getNewRuns()`: retorna as submissões mais novas a partir da última submissão carregada.

Os Quadros 1 e 2 listam os campos dos objetos JSON que compõem o retorno do método `getContest()`. O Quadro 3 mostra os detalhes de um objeto `run`.

Quadro 1 – Detalhes do objeto `contest`

Campo	**Descrição**
`eventTitle`	Nome da competição.
`duration`	Duração da competição em minutos.
`frozen`	Minuto em que vai ocorrer o congelamento do placar da competição.
`blind`	Minuto em que vai ocorrer o *blind* na competição.
`penalty`	Valor da penalidade da competição.
`qtdProblems`	Quantidade de problemas da competição.
`teams`	Lista de times participantes da competição.

Fonte: (MAKAHA, 2021)

Quadro 2 – Detalhes do objeto `team`

Campo	**Descrição**
`teamId`	Identificador do time.
`college`	Nome da escola.
`name`	Nome do time.

Fonte: (MAKAHA, 2021)

Sinteticamente, no início do programa o BROMS solicita as informações da competição: com isso é possível a construção inicial do placar. Após a inicialização do placar o BROMS verifica se há submissões para desenhar no placar. Depois dessas etapas iniciais, o BROMS entra em um ciclo de requisições em que periodicamente verifica a conclusão

Quadro 3 – Detalhes do objeto `run`

Campo	Descrição
`runId`	Identificador do time.
`time`	Minutos passados após o início da competição até a submissão.
`teamUid`	Identificador do time.
`problem`	Problema relacionado à submissão.
`verdict`	Resultado da solução.

Fonte: (MAKAHA, 2021)

da competição para parar de fazer as requisições. Se a competição estiver em andamento o BROMS busca por novas submissões. Com a finalização da competição, o placar mostra o último estado da competição.

1.5.1 *API Mock*

Um problema comum no desenvolvimento de aplicações que precisam se comunicar com outras aplicações é a execução desta comunicação durante o estágio de desenvolvimento. Muitas vezes a execução de todas as aplicações em paralelo pode ser um problema quanto ao esforço computacional, ou em outras situações pode ocorrer que uma das aplicações ainda não implementou a comunicação.

Nestas situações é comum o uso do chamado *API mock* para simular os dados produzidos pela aplicação a qual pretende comunicar-se. São aplicações mais simples e geralmente limitadas, que objetivam simular de forma mais fidedigna possível as informações geradas pelas aplicações que substituem.

Apesar de geralmente produzirem dados mais genéricos, estes simuladores são úteis principalmente para dar prosseguimento ao desenvolvimento de uma aplicação sem ficar preso ao estado da implementação da outra aplicação.

A API mock desenvolvida para o BROMS foi gerada em um repositório separado do placar. Essa API foi implementada utilizando a linguagem de programação Python com ajuda do *framework fastAPI*. A API foi criada com o objetivo de simular uma competição e testar a comunicação com o placar. A simulação de uma competição é realizada a partir da leitura de dois arquivos estáticos, *contest e runs* (MAKAHA, 2021). Os dados carregados dos arquivos são disponibilizados em 4 rotas com propósitos distintos:

- `/contest`: rota do tipo `GET` que retorna as informações da competição,
- `/contest/finish`: rota do tipo `GET` que retorna um booleano que, se verdadeiro, a competição foi finalizada,
- `/runs`: rota do tipo `GET` que retorna todas as submissões atuais de uma competição,

- `/runs/diff`: rota do tipo `POST` que retorna as submissões mais novas a partir do identificador de uma submissão que é passado como parâmetro na requisição.

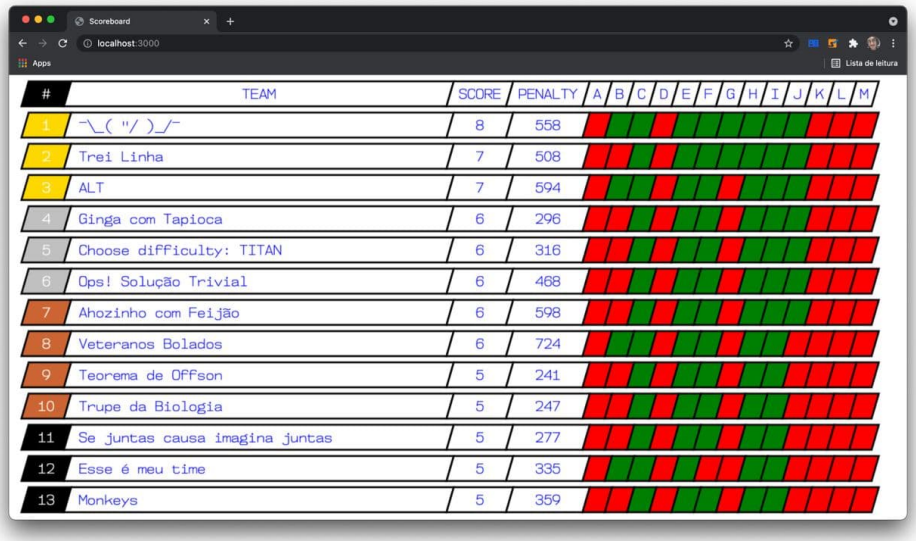
2 BROMS: Novas Implementações

Neste capítulo são apresentadas as funcionalidades e os códigos desenvolvidos neste trabalho. Na Seção Organização do código é apresentado o ponto de partida do desenvolvimento e seu impacto na estrutura do código, na Seção Arquitetura são apresentados os impactos que as mudanças tiveram sobre a arquitetura da aplicação, na Seção Animações são apresentados os detalhes referentes às animações, na Seção Visualizações são expostos as formas de visualização do placar, na Seção Comunicação com a API do Codeforces é detalhada a implementação da comunicação com a API do Codeforces, enquanto na Seção Comunicação com o BOCA apresentada a comunicação com o BOCA.

2.1 Organização do código

A princípio o BROMS apresentava uma aparência visual simples, como mostrado na Figura 4, que lembrava uma tabela estática. A primeira alteração então proposta no início deste período de desenvolvimento foi a substituição dos então retângulos que montavam o placar por paralelogramos, como exibido na Figura 5.

Figura 5 – BROMS, demonstração do novo layout.



Fonte: Makaha (2021)

Em sua versão inicial, o BROMS não possuía uma arquitetura bem definida. Este incremento teve como foco definir e implementar uma arquitetura voltada para lidar com tratamento de eventos e atualizações da tela. A proposta inicial foi de uma função que desenhava um paralelogramo com um texto interno, que logo evoluiu para a ideia de uma

classe com esta funcionalidade. Esta substituição foi de grande importância para expor a necessidade de desacoplamento dos componentes, que até então não vinha sendo bem planejada, causando então problemas como dificuldades na recuperação de informações.

A princípio, quando uma atualização na tela era solicitada, a classe `Scoreboard` era responsável por chamar a classe `Row`, que além de guardar todas as informações de uma das linhas do placar, era responsável por saber sua posição relativa dentro do `canvas` e por desenhar na tela todos os elementos pertencentes à linha. Desta forma o código da classe `Row` apresentava uma série de repetições de código, como apresentado no Código 1, definindo e desenhando cada um dos retângulos a ela pertencentes.

Código 1 – Código antigo, duplicações na classe `Row`

```
class Row {
    ...
    drawPosition(text, x, y) {
      this.c.strokeRect(x, y, this.size[0] * this.w, -this.h);;
      let [dx, dy] = align(text, 'center', this.size[0] * this.w, this.h);
      this.c.fillText(text, x + dx, y -dy);
    }
    drawName(text, x, y) {
      this.c.strokeRect(x, y, this.size[1] * this.w, -this.h);
      let [dx, dy, offset] = align(text, 'left', this.size[1] * this.w, this.h);
      this.c.fillText(text, x + dx, y -dy, this.size[1] * this.w - offset);
    }
    drawScore(text, x, y) {
      this.c.strokeRect(x, y, this.size[2] * this.w, -this.h);
      let [dx, dy] = align(text, 'right', this.size[2] * this.w, this.h);
      if (this.header) [dx, dy] = align(text, 'center', this.size[2] * this.w, this.h);
      this.c.fillText(text, x + dx, y - dy);
    }
    drawPenality(text, x, y) {
        this.c.strokeRect(x, y, this.size[3] * this.w, -this.h);
        let [dx, dy] = align(text, 'right', this.size[3] * this.w, this.h);
        if (this.header) [dx, dy] = align(text, 'center', this.size[3] * this.w, this.h);
        this.c.fillText(text, x + dx, y - dy);
    }
    draw(){
        ...
        this.drawPosition(this.position, x, y);

        x = x + this.size[0] * this.w;
        this.drawName(this.teamName, x, y);

        x += this.size[1] * this.w;
        this.drawScore(this.score, x, y);

        x += this.size[2] * this.w;
        this.drawPenality(this.penality, x, y);

        x += this.size[3] * this.w;
        const sum = this.size.reduce((a,b) => a+b);
        const problemWidth = this.w * (1 - sum)/(this.n -1);
        this.drawQuestions(x, y, problemWidth);
    }
    ...
}
```

Fonte: Autores

A classe `Parallelogram`, Código 2, foi então criada para representar um paralelogramo e, na sequência, recebeu um atributo da classe `Text`, criada também no contexto do problema do acoplamento. Um `Parallelogram` contém então toda informação necessária para se desenhar na tela, incluindo sua posição relativa.

Código 2 – Trecho principal da implementação da classe `Parallelogram`

```
class Parallelogram {
    constructor(Parent=undefined, x, y, wRel, h, text=undefined,
        fillColor='white', borderColor='black'){
        ...//Construtor padrão
    }

    setText(text, {selfAlign=this.text.selfAlign}={}){
        this.text.setText(text, selfAlign);
    }

    draw(){
        const ctx = canvasSingleton.getInstance().getContext();
        const ang = CONSTANTS.ang;
        ctx.beginPath();

        const dx = -this.h * Math.tan(ang) / 2;

        ctx.lineWidth = 3
        ctx.strokeStyle = this.borderColor;
        ctx.fillStyle = this.fillColor;

        ctx.moveTo(this.x, this.y);
        ctx.lineTo(this.x + this.w, this.y);
        ctx.lineTo(this.x + this.w - dx, this.y - this.h);
        ctx.lineTo(this.x - dx, this.y - this.h);
        ctx.lineTo(this.x, this.y);

        ctx.fill();
        ctx.closePath();
        ctx.stroke();

        if(this.text){
            this.text.draw();
        }
    }
    ...
}
```

Fonte: Autores

Analogamente, a classe `Text`, Código 3, apesar de não ser necessariamente instanciada como atributo de uma classe superior, podendo se desenhar por conta própria apenas com a informação nela contida, mantém a informação do possível `Parent`, a classe que o possui como membro, e é capaz de se desenhar em alinhada em relação ao mesmo.

A classe `Row`, Código 4, passou então a agregar apenas um conjunto de instâncias de paralelogramos, cuja implementação é independente da `Row` e que no momento solicitado serão responsáveis por desenhar a si próprios.

Na sequência da refatoração dos blocos menores e pensando no futuro da evolução

Código 3 – Trecho principal da implementação da classe `Text`

```
class Text {
    constructor(Parent=undefined, value='', color=COLORS.mainTextColor, selfAlign='center',
    ↪  font=FONTS.default,  {x=0, y=0}={}) {
        ... // Construtor padrão
    }

    setText(value, {selfAlign=this.selfAlign}={}){
        this.value = value;
        this.selfAlign = selfAlign;
    }

    draw(){
        const ctx = canvasSingleton.getInstance().getContext();
        ctx.fillStyle = this.color;
        if(this.Parent){
            const diff = this.Parent.h * Math.tan(CONSTANTS.ang) / 2;
            let [dx, dy, offset] = this.align(this.value, this.selfAlign, this.Parent.w,
            ↪  this.Parent.h, this.font);
            let maxWidth = this.Parent.w - diff - 2 * offset;
            ctx.fillText(this.value, this.Parent.x + dx, this.Parent.y - dy, maxWidth);
        } else {
            ctx.fillText(this.value, this.x + dx, this.y - dy, maxWidth);
        }
    }
}
```

Fonte: Autores

do software com a adição de animações e afins, foi decidido pela reestruturação do resto da arquitetura. Foi então proposta a refatoração de cada componente para que tenham a capacidade de se desenhar na tela independentemente dos outros elementos do placar, possibilitando a implementação de uma arquitetura voltada ao tratamento de eventos, onde apenas as partes interessadas devem atender a cada evento.

2.2 Arquitetura

Anteriormente ao início deste ciclo, o BROMS já possuía uma estrutura básica para tratamento dos eventos gerados. Eram então tratados os eventos de interrupção por parte do usuário, via mouse ou teclado, e os eventos da rede gerados pela submissões. Tais eventos eram tratados de forma genérica, no bloco principal do código, e tinham seu impacto limitado à execução de uma função definida.

Como citado anteriormente, a refatoração dos outros componentes permitiu a implementação de uma arquitetura mais bem definida, na qual toda atualização de informação parte de um evento e a tela é atualizada independentemente, obedecendo apenas ao fator tempo, como demonstrado no Código 5.

Ainda na refatoração desta arquitetura definiu-se pela implementação de um gerenciador de eventos, implementado na forma da classe `EventManager`, Código 6, dentro

Código 4 – Trecho da nova implementação da classe `Row`

```
class Row {
    ...
    buildParallelogs() {
        ...
        this.parallelogs.push(new Parallelogram(this, 0, this.y - this.marginY, this.size[2],
        ↪  this.h - 2*this.marginY, new Text(undefined, this.score)));
    }
    ...
    draw(){
        const a = Math.max(this.camera.y, this.y)
        const b = Math.min(this.camera.y + this.camera.h, this.y + this.h)

        if (b - a <= 0) return

        this.parallelogs.map((parallelog) => parallelog.draw())
    }

    align(value, type, width='', height='', font=FONTS.default) {
        const ctx = canvasSingleton.getInstance().getContext();
        ctx.font = `${font.size}px ${font.name}`;

        const textMetrics = ctx.measureText(value);
        const text_w = textMetrics.width;
        const text_h = textMetrics.actualBoundingBoxAscent;

        const offset = Math.max(width * 0.02, 1);
        const diff = height * Math.tan(CONSTANTS.ang) / 2

        const ans = {
          center: [diff + (width - diff - text_w)/2, (height - text_h)/2, offset],
          left: [offset + diff,(height - text_h)/2, offset],
          right: [width - text_w - offset < 0 ? width - text_w : width - text_w - offset,
          ↪  (height - text_h)/2, offset],
        }
        if (text_w > width - diff) return this.align(value, type, width, height, {...font, size:
        ↪  font.size - 1})
        return ans[type]
    }
}
```

Fonte: Autores

do padrão de projeto `Singleton`, o qual atua como ouvinte dos eventos e acumulador das inscrições a estes eventos.

Desta forma, cada objeto que deseja ouvir a um evento precisa apenas se inscrever na instância única do `EventManager` e implementar, dentro de seu método `onEvent`, como mostrado no Código 7, o tratamento para o evento desejado. Esta implementação também caracteriza a aplicação do padrão de projeto Observer.

2.3 Animações

As animações foram feitas de forma a tentar deixar mais claro para o telespectador a mudança de posição entre os times. Desta forma, sempre que há uma mudança nas

Código 5 – Atualização da tela feita com base no tempo

```
setInterval(async () => {
    redrawAll();
}, 40);
```

Fonte: Autores

Código 6 – Trecho principal da implementação da classe `EventManager`

```
class EventsManager {
    constructor(){
        this.events = {}
    }

    registerListener(eventType, listener){
        if(this.events[eventType])
            this.events[eventType] = [...this.events[eventType], listener];
        else
            this.events[eventType] = [listener];
    }

    unregisterListener(eventType, listener){
        if(this.events[eventType])
            this.events[eventType].pop(listener);
    }

    onEvent(eventType, event){
        this.events[eventType].map(listener => listener.onEvent(eventType, event));
    }

    notify(eventType, event) {
        this.onEvent(eventType, event);
    }
}
```

Fonte: Autores

colocações, ao invés de simplesmente trocarem de posições, é possível ver cada linha – que representa um time – se deslocando em direção à sua nova posição.

Esta funcionalidade foi desenvolvida a partir de um novo atributo dado a cada linha para identificar sua posição de destino. Assim sendo, cada linha agora guarda sua posição atual e sua posição de destino. A posição destino é recalculada no momento do processamento da submissão, onde caso haja uma mudança nas colocações, cada linha que teve sua colocação alterada tem sua posição de destino alterada, dando início ao processo de movimento.

Na sequência, quando cada linha houver de se desenhar, sua posição será alterada de forma a mover-se em direção à posição de destino, sendo este cálculo feito através da divisão da distância das posições por 30, com o movimento mínimo de 0,01 pixel - caso o resultado da divisão seja menor que este valor a posição é truncada para a posição de

Código 7 – Trecho de código da classe `Camera` implementando o método `onEvent`

```
onEvent(eventType, event){
        if(eventType === 'keydown'){
            if (event.code === "ArrowUp") {
                this.move(0, this.y - 50);
            }else if (event.code === "ArrowDown") {
                this.move(0, this.y + 50);
            }
            let manager = eventsManager.getInstance();
            manager.notify('cameraMovement', this);
        }else if(eventType === 'wheel'){
            if (event.deltaY < 0) {
                this.move(0, this.y - 25);
            }
            else if (event.deltaY > 0) {
                this.move(0, this.y + 25);
            }
            let manager = eventsManager.getInstance();
            manager.notify('cameraMovement', this);
        }else if(eventType === 'resize'){
            this.resize(event.target.innerWidth, event.target.innerHeight);
        }else if(eventType === 'scoreboardResize'){
            this.setYMax((event.totalRows + 1) * event.rowHeight + 4);
        }
    }
```

Fonte: Autores

destino. O objetivo desta forma de calcular é gerar uma variação gradual da velocidade, onde quanto maior a distância que a linha está da sua posição final mais rápido ela se moverá até que se aproxime o suficiente e, suavemente, pare em sua posição de destino.

Ademais, com esta organização do processamento como descrita acima, a linha não deverá realizar movimentos bruscos, mesmo que haja outras mudanças nas colocações enquanto uma animação ainda está em andamento. Nestes casos, as linhas simplesmente continuarão a mover-se em direção às suas próximas posições de destino, mesmo que na direção contrária ao seu movimento anterior.

2.4 Visualizações

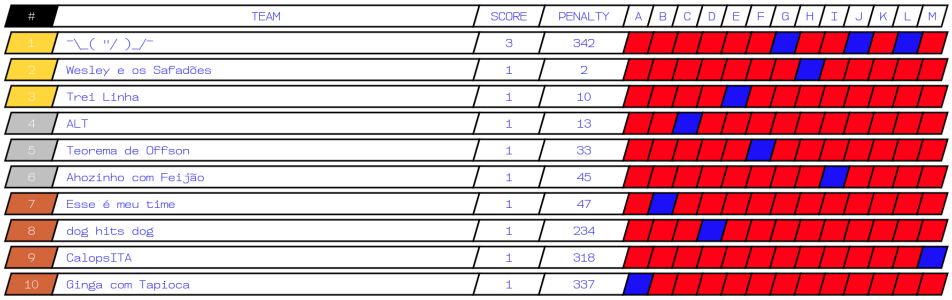
Além das animações, outro grande objetivo do trabalho era disponibilizar mais informações ao telespectador. Para tal, foram desenvolvidas novas formas de visualização do placar de acordo com as demandas do cliente professor Dr. Edson Júnior.

Estas visualizações são processadas em simultâneo à visualização padrão e consequentemente têm os seus dados sempre atualizados, estando sempre prontas para serem desenhadas. Através do teclado o usuário pode selecionar qual visualização será exibida.

2.4.1 Primeiros a acertar

Nesta visualização são apresentados apenas os times que foram os primeiros a acertar cada questão do evento, demonstrada na Figura 6.

Figura 6 – BROMS, Forma de Visualização: primeiros a acertar.

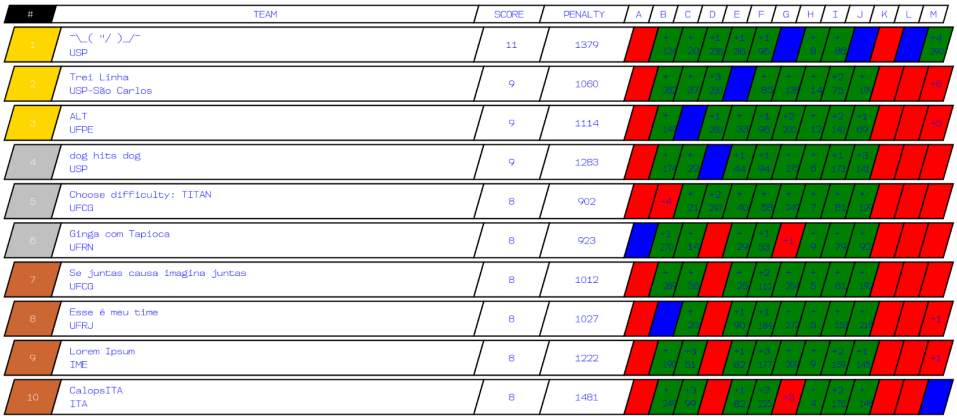


Fonte: Autores

2.4.2 Visualização completa

Esta visualização busca dar mais informações aos usuários, nela, além das informações padrões, são exibidos também a quantidade de tentativas de cada time em cada questão e a penalidade aplicada a cada uma. Quando disponível, são exibidos também o nome da organização ao qual o time pertence – geralmente é o nome da universidade. Demonstrada na Figura 7.

Figura 7 – BROMS, Forma de Visualização: informações extra.



Fonte: Autores

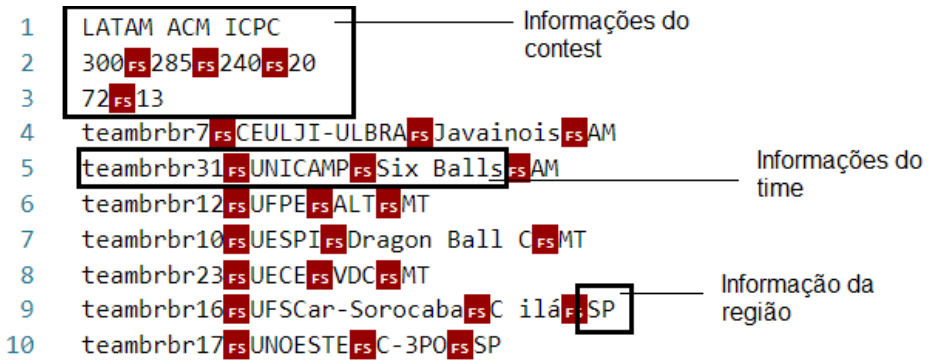
2.4.3 Visualização por agrupamento

Esta visualização é visualmente idêntica a padrão, a diferença se dá pela amostra de times exibidos. Nela, são exibidos apenas times de determinado agrupamento, sendo o

objetivo principal permitir a separação de times por regiões ou sedes. A divisão acontece através de um parâmetro adicional que deve ser fornecido ao placar junto com os dados da competição.

Atualmente os agrupamentos são apenas para demonstração da funcionalidade, sendo limitados à definição por dois caracteres, que devem ser fornecidos ao placar junto às informações do evento, como mostrado na Figura 8. Os atalhos para acessar essas visualizações são gerados automaticamente a partir destes caracteres, sendo necessário apenas pressionar as teclas referentes às siglas da mesma forma que foi definido na configuração do evento. Desta forma, atualmente só é possível que um time pertença a um único agrupamento, sendo necessário um pequeno ajuste no código remover esta restrição.

Figura 8 – Trecho do arquivo `contest` que foi alterado para simulação.



Fonte: Autores

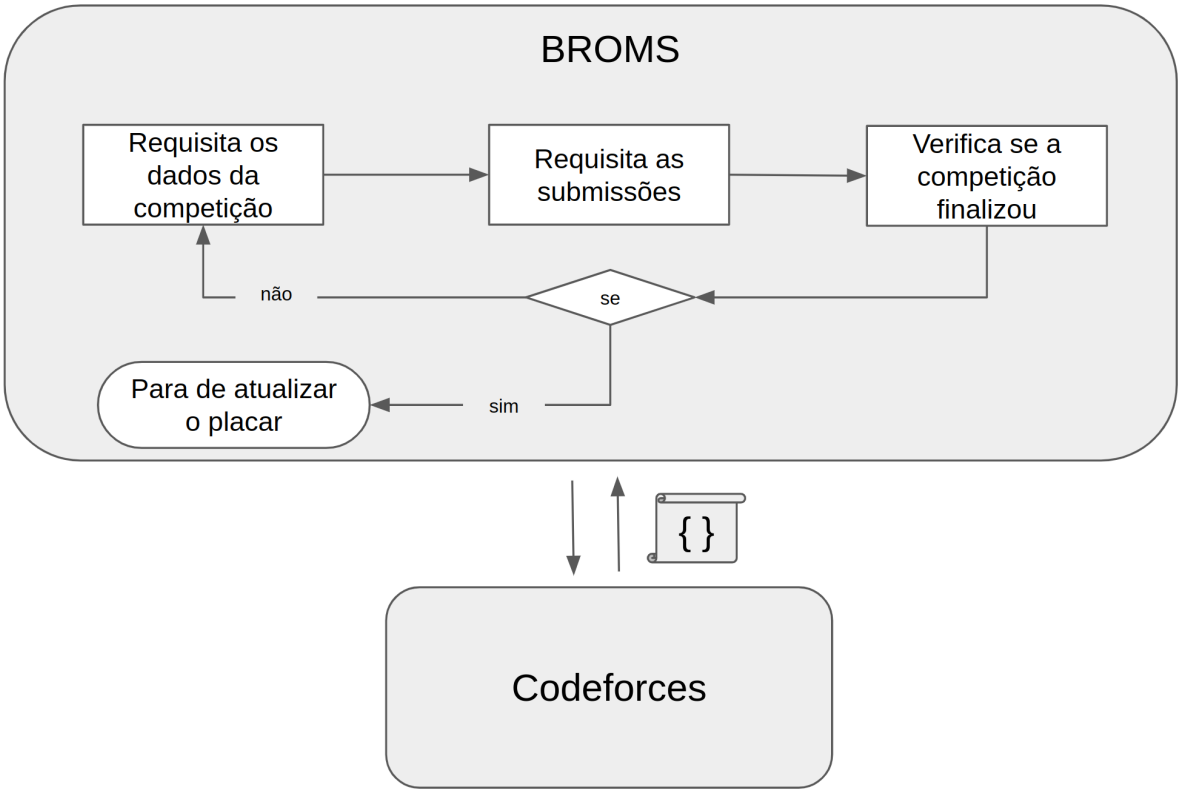
2.5 Comunicação com a API do Codeforces

Com o objetivo de manter o objetivo do BROMS de possuir quantidade baixa de dependências, o módulo de comunicação com a API do Codeforces foi desenvolvido em Javascript sem a necessidade de um serviço extra para realizar a comunicação. A partir dessa abordagem não foi preciso adicionar novas dependências, pois foram utilizadas apenas interfaces padrões da biblioteca do navegador. A Figura 9 apresenta de forma sintética os processos realizados na comunicação com a API do Codeforces.

Os antigos métodos responsáveis pelas requisições web foram incorporados em um novo módulo destinado apenas à *API mock*. Porém foram mantidas as interfaces, nas quais foi introduzido um mapeamento para diferenciar as chamadas de API de acordo com o modo de execução, *API mock* ou API do Codeforces, como pode ser visto nos Códigos 8 a 10.

Foi desenvolvido um método responsável por montar a URL para poder acessar a API do Codeforces via requisições do tipo `GET` (Código 11). Para montar a URL é preciso

Figura 9 – Comunicação com o Codeforces



Fonte: Autores

Código 8 – Interface `getContest()`

```
5  export const getContest = () => {
6    switch (MODE) {
7      case "MOCK":
8        return getMockContest();
9      case "CF":
10       return getCFContest();
11   }
12 }
```

Fonte: Autores

passar a rota da API, `standing` ou `status`. Também é possível passar um contador e um índice para paginação.

A API do Codeforces possui limitações relacionadas à paginação e à ordenação dos itens. A API não disponibiliza uma forma de inferir a quantidade total dos itens e nem possibilita a escolha da ordenação, então quando há a necessidade de carregar os últimos itens é preciso carregar, em algum momento, todos os itens para manter os dados consistentes. A paginação é feita apenas passando o índice do primeiro elemento e a quantidade de itens esperada para resposta.

Para o BROMS seria interessante que na consulta de novas submissões fosse pos-

Código 9 – Método `getMockContest()`

```
4  export const getMockContest = () => {
5    const url = `${MOCK_BASE_URL}/contest`;
6    return new Promise((resolve, reject) => {
7      fetch(url)
8      .then(response => response.json())
9      .then(responseJson => mockToContestModel(responseJson))
10     .then(resolve)
11     .catch(reject)
12   })
13 }
```

Fonte: Autores

Código 10 – Método `getCFContest()`

```
4  export const getCFContest = () => {
5    return new Promise((resolve, reject) => {
6      getApiUrl('standings')
7      .then(url => fetch(url))
8      .then(response => response.json())
9      .then(responseJson => toContestModel(responseJson.result))
10     .then(resolve)
11     .catch(reject)
12   })
13 }
```

Fonte: Autores

sível passar o identificador do último item carregado e o retorno fosse constituído apenas por novos itens, não carregados por requisições anteriores, como é feito na *API mock*. Para utilizar a mesma interface para carregar as novas solicitações foi implementada uma função interna que faz essa abstração. Para verificar se há novas submissões a função faz uma requisição inicial na rota `status` para pegar as ultimas submissões realizadas. Nessa requisição é definida a quantidade de itens a serem carregados. Se é identificado que todos os itens carregados são novos, são feitas sucessivas requisições a partir do índice do último item carregado até identificar uma solicitação já processada, então são retornadas as novas solicitações.

Após receber a resposta do servidor, o objeto JSON é passado para um método responsável por transformar o objeto e manter apenas os dados necessários para o funcionamento do BROMS. O Código 12 apresenta os métodos conversores. Também foi criado um conversor para resposta da *API mock*, com isso, o tratamento dos dados do `contest` foi removido da função `main`.

As propriedades esperadas para o objeto `contest` do BROMS que podem ser extraídas dos objetos retornados pelo método `contest.standings` da API do Codeforces

Código 11 – Método getApiUrl()

```
4  export async function getApiUrl(api, count = null, index = null) {
5      let params = `contest.${api}?apiKey=${CF_KEY}&contestId=${CF_CONTEST_ID}`;
6
7      if (count && index) {
8          params = params.concat(`&count=${count}&from=${index}`);
9      }
10
11     const time = Math.trunc(new Date().getTime() / 1000);
12
13     params = params.concat(`&time=${time}`)
14
15     const rand = generateRandomString();
16     let hash = '';
17
18     await sha512(`${rand}/${params}#${CF_SEC}`).then(h => hash = h);
19
20     const url = `https://codeforces.com/api/${params}&apiSig=${rand}${hash}`;
21
22     return url;
23 }
24
25 const generateRandomString = () => {
26     return Math.random().toString(36).substring(2, 8);
27 }
28
29 const sha512 = (str) => {
30     return crypto.subtle.digest("SHA-512", new TextEncoder("utf-8").encode(str)).then(buf => {
31         return Array.prototype.map.call(new Uint8Array(buf), x => (('00' +
           ↪  x.toString(16)).slice(-2))).join('');
32     });
33 }
```

Fonte: Autores

são:

- `eventTitle`: o nome do evento é obtido pela propriedade `name` do objeto `contest`,
- `duration`: a duração do evento é obtida pela conversão para minutos do valor da propriedade `durationSeconds` do objeto `contest`,
- `qtdProblems`: a quantidade de problemas é obtida pela propriedade `length` da lista de problemas: `problems`,
- `teams`: os times são obtidos da lista `rows`, em que de cada objeto `party` presente na lista são extraídos o `teamId` e o `teamName`. Para participações individuais é utilizado o *nickname* do usuário como identificador e nome do time. O *nickname* é obtido da propriedade `handle` do primeiro item da lista de `members` do objeto `party`.

De forma análoga é feito o mapeamento da lista de objetos `runs` retornadas pelo método `constest.status` da API do Codeforces para construção das submissões para serem processadas pelo BROMS:

Código 12 – Métodos Conversores

```
import { FROZEN, BLIND, PENALTY } from '../appSettings.js';

export const toContestModel = (model) => {
    return {
        eventTitle: model.contest.name,
        duration: Math.trunc(model.contest.durationSeconds / 60),
        frozen: model.contest.durationSeconds - FROZEN,
        blind: model.contest.durationSeconds - BLIND,
        penalty: PENALTY,
        qtdProblems: model.problems.length,
        teams: toTeamModel(model.rows)
    }
}

export const mockToContestModel = (model) => {
    return {
        eventTitle: model.name,
        duration: model.duration,
        frozen: model.frozen,
        blind: model.blind,
        penalty: model.penalty,
        qtdProblems: model.n_questions,
        teams: model.teams
    }
}

export const toTeamModel = (model) => {
    return model.map(r => {
        return {
            teamId: r.party.teamId ?? r.party.members[0].handle,
            college: '',
            name: r.party.teamName ?? r.party.members[0].handle
        }
    })
}

export const toRunModel = (model) => {
    if (!model) {
        return []
    }
    return model.map(r => {
        return {
            runId: r.id,
            time: Math.trunc(r.relativeTimeSeconds / 60), // sec => min
            teamUid: r.author.teamId ?? r.author.members[0].handle,
            problem: r.problem.index,
            verdict: r.verdict == 'OK' ? 'Y' : 'N'
        }
    })
}
```

Fonte: Autores

- `runId`: o identificador da submissão é obtido pela propriedade `id`,
- `time`: o tempo da submissão é obtido pela conversão para minutos do valor da propriedade `relativeTimeSeconds`,
- `teamUid`: é obtido pela propriedade `teamId` do objeto `author` da submissão. Para participações individuais é utilizado o *nickname* do usuário como identificador do

time. O *nickname* é obtido da propriedade `handle` do primeiro item da lista de `members` do objeto `author`,

- `problem`: é obtido pela propriedade `index` do objeto `problem` da submissão,
- `verdict`: é obtido pela propriedade `verdict` da submissão.

Alguns dados esperados para o BROMS não são disponibilizados pela API do Codeforces: `frozen`, `blind` e `penalty`. Esses dados devem ser inseridos manualmente antes de iniciar a aplicação. Os nomes das escolas dos times também não podem ser carregados dinamicamente e são iniciados em branco.

2.6 Comunicação com o BOCA

Para a comunicação com o BOCA foi utilizado o módulo de comunicação desenvolvido para a *API mock*. Com essa abordagem a comunicação é feita de forma indireta. A *API mock* funciona como cliente do BOCA e como servidor para o BROMS.

O BOCA disponibiliza uma pasta compactada chamada `webcast.zip`. Esta pasta contém cinco arquivos: `runs`, `contest`, `time`, `version` e `icpc`.

O `webcast.zip` pode ser acessado fazendo uma requisição do tipo `GET` na url:

`http://<IP_BOCA>/boca/admin/report/webcast.php?webcastcode=<CHAVE>`

A chave é disponibilizada pelo administrador do BOCA que configura um arquivo `webcast.sep` que fica em:

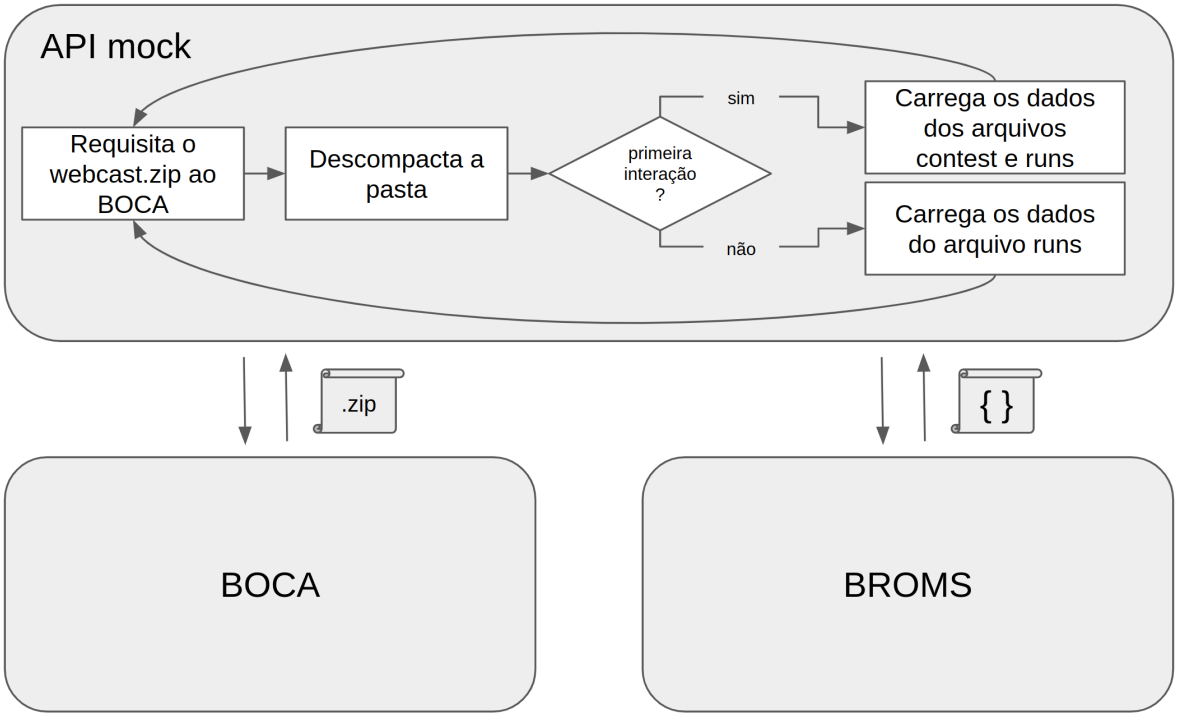
`/var/www/boca/src/private/webcast.sep`

A sintaxe desse arquivo é: `CHAVE 1/ID_INICIAL/ID_FINAL` onde a chave é uma *string* para resgatar o placar, e `ID_INICIAL` e `ID_FINAL` são os identificadores de um grupo de usuários que devem aparecer para essa chave. Para adicionar mais de um grupo de usuários, basta usar a mesma sintaxe e separar por espaço.

Como dito anteriormente, a *API mock* já utilizava os arquivos `runs` e `contest` para simular uma competição e expor os dados para o BROMS. Então para prover os dados de uma competição em tempo real para o BROMS foi criado um módulo na *API mock* que funciona como cliente do BOCA. Esse módulo fica responsável por requisitar periodicamente o `webcast.zip` e extrair os arquivos `runs` e`contest`. Após carregar os arquivos os dados ficam disponíveis e o BROMS pode realizar as requisições. A Figura 10 mostra os processos realizados pela *API mock* na comunicação com o BOCA.

Por parte do BROMS o processo é parecido com o que acontece na comunicação com o Codeforces, mas neste caso, como todos os times são carregados na primeira inte-

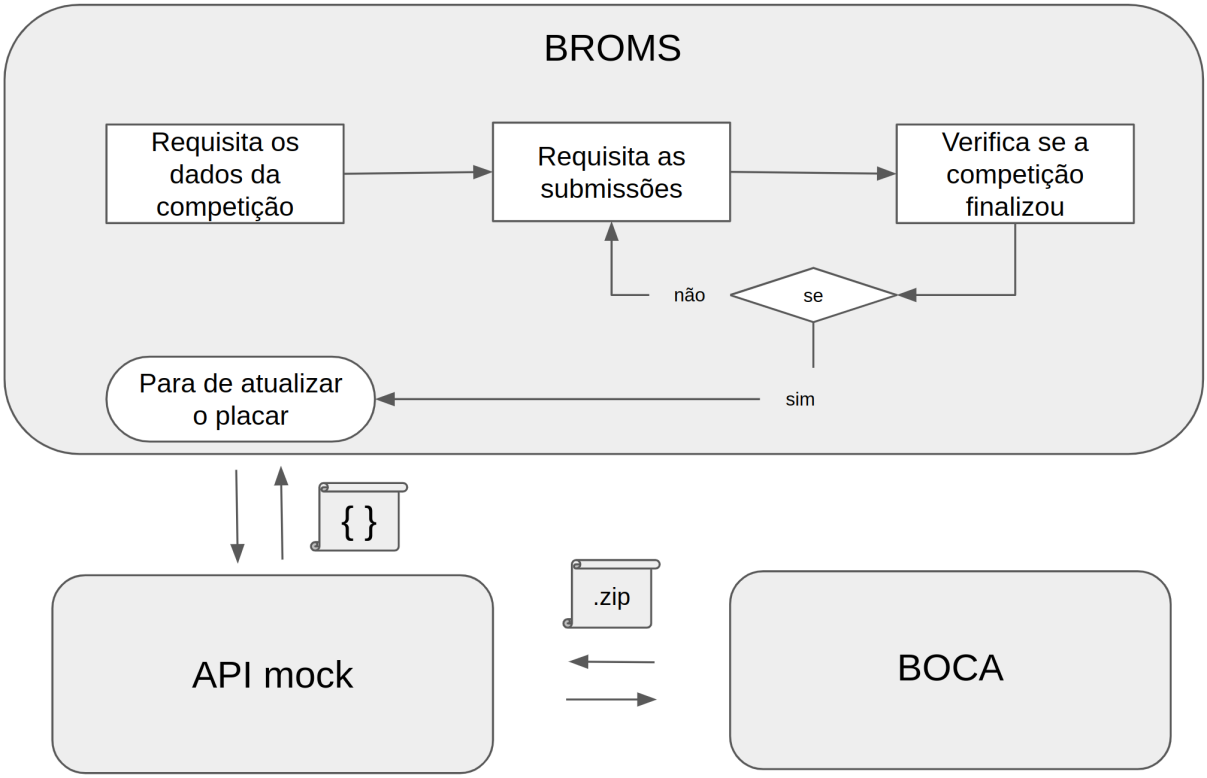
Figura 10 – Comunicação com o Boca - *API mock*



Fonte: Autores

ração, o ciclo se concentra nas requisições de novas submissões até que a competição se encerre (Fig. 11).

Figura 11 – Comunicação com o Boca - *API mock*



Fonte: Autores

3 Metodologia

Neste capítulo são apresentadas as metodologias adotadas no desenvolvimento deste trabalho. Nesta perspectiva, são detalhados os métodos aplicados na implementação do software e os processos realizados na coleta de dados para validação dos resultados. Na Seção Ferramentas de Desenvolvimento e Ambiente são apresentadas as ferramentas usadas e a configuração do ambiente de desenvolvimento. A Seção Metodologia de Desenvolvimento apresenta as técnicas utilizadas no desenvolvimento do trabalho. Por fim, as Seções Ensaios de Integração com a API do Codeforces e Ensaio de Integração com o BOCA definem os processos de execução dos ensaios experimentais para validação do placar e para a coleta de dados.

3.1 Ferramentas de Desenvolvimento e Ambiente

O BROMS, desde sua origem, foi desenvolvido usando as ferramentas Javascript e HTML5, tecnologias que, além de atender ao critério de requisitos mínimos, eram capazes de dar suporte ao que se pretendia para as etapas futuras, e por isso não foi considerado alterá-las, sendo necessário para visualização do placar apenas um navegador que possuir suporte a estas tecnologias.

Ainda no contexto de dependências mínimas, neste trabalho não é implementada uma solução complexa para isolamento e gerenciamento de ambientes, sendo considerada desnecessária a aplicação de técnicas como conteinerização. Logo, não serão abordados tópicos como sistema operacional, IDES e afins, usados no desenvolvimento, considerando que estes não deveriam impactar na execução do placar e nem no desenvolvimento do mesmo.

Para configurar o ambiente de desenvolvimento desta aplicação é necessário apenas um interpretador Python, versão 3.8.5 ou superior, usado para o servidor mínimo que fornece as informações ao navegador. O serviço pode ser iniciado pelo comando apresentado no Código 13.

Código 13 – Executando o servidor mínimo Python

```
1    python -m http.server 3000
```

Fonte: Autores

O BROMS, salvo o modo integrado ao Codeforces, necessita ainda de um servidor

para alimentá-lo com as informações de um suposto evento, feito atualmente em forma de um *API mock* separada do placar. Desenvolvida com o FastAPI, utilizando o pacote `Uvicorn`, este servidor simula as submissões de uma competição e pode ser iniciado pelo comando apresentado no Código 14.

Código 14 – Executando a *API mock*

```
1    uvicorn main:app
```

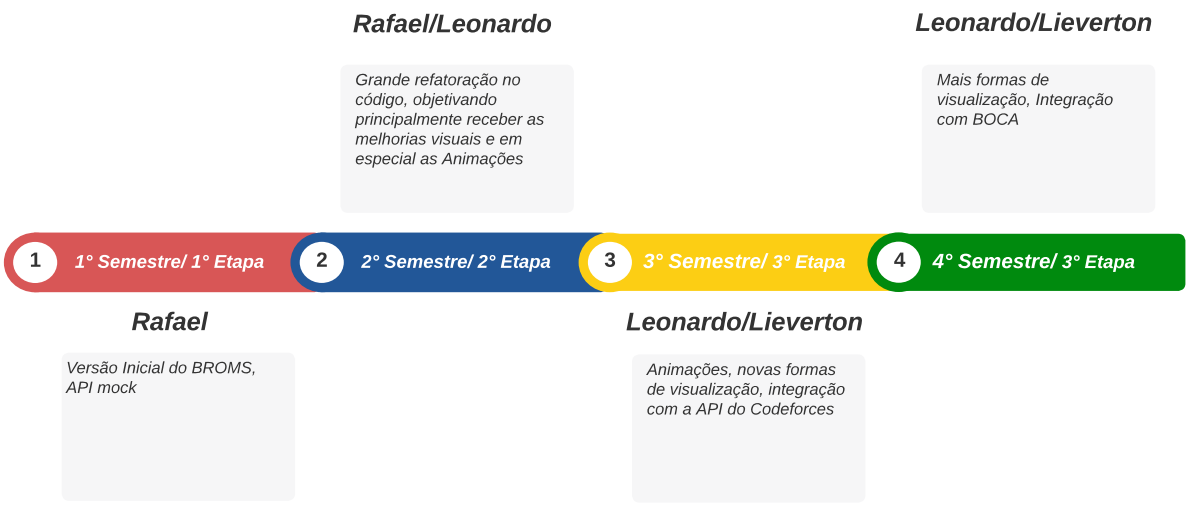
Fonte: Autores

Todo o processo de desenvolvimento e testes foi realizado utilizando os navegadores Google Chrome v94 e o Mozilla Firefox v92.0, que dão suporte aos requisitos mínimos supracitados.

3.2 Metodologia de Desenvolvimento

Quanto ao tempo, o desenvolvimento do BROMS pode ser dividido em três etapas. A primeira, o início do trabalho, onde o estudante Rafael Makaha trabalhou sozinho. A segunda com o ingresso do estudante Leonardo Medeiros no projeto. E por último, a terceira, onde não houve participação do estudante Rafael Makaha, entretanto houve o ingresso do estudante Lieverton Santos ao projeto. A Figura 12 apresenta a linha do tempo do desenvolvimento do BROMS.

Figura 12 – Linha do tempo do desenvolvimento do BROMS



Fonte: Autores

Como dito, o primeiro ciclo de desenvolvimento não foi realizado pelos autores e consequentemente não será o foco deste trabalho, sendo descrito e detalhado por Makaha (2021) em seu trabalho.

Como metodologia para este trabalho foi feita uma adaptação e mesclagem de práticas do Scrum e do *Extreme Programming* (XP). Com o objetivo de adaptação no desenvolvimento foi usada a técnica de programação em pares do XP.

O segundo ciclo de desenvolvimento, que teve duração equivalente a um semestre letivo, teve início com uma breve avaliação comparativa do BROMS em relação aos outros placares disponíveis, em especial o atualmente usado para a transmissão da Maratona SBC de Programação, o Maratona Rustrimeitor (RUSTRIMEITOR, 2021). Foram avaliados principalmente o formato visual do placar e como são demonstradas as informações para os espectadores, em especial a forma como lidam com as interrupções do usuário. Esta avaliação serviu para melhor entender os objetivos do BROMS e o quão distante ele estava de alcançá-los, além de identificar possíveis incrementos ao mesmo.

Nesta etapa, o processo de desenvolvimento era realizado em pequenos ciclos de evolução, com duração de uma semana e reuniões de revisão/planejamento realizadas todas as terças-feiras na plataforma Teams com a presença do Professor Dr. Edson Júnior, que atuou também como um cliente neste produto. Nestas reuniões eram avaliadas as alterações realizadas, sendo observado se atendiam ao esperado para o objetivo definido. Em sequência, eram então propostos os novos objetivos para a *sprint* seguinte, de acordo com as prioridades do BROMS.

Por fim, na terceira e última etapa de desenvolvimento, com duração equivalente a dois semestres letivos, manteve-se a estratégia de reuniões semanais. No primeiro semestre desta etapa as reuniões foram realizadas às segundas-feiras, geralmente com a equipe completa presente. Já no segundo semestre as reuniões de revisão/planejamento foram realizadas às quintas-feiras apenas com a presença do estudante Lieverton e do orientador. E para manter o estudante Leonardo alinhado eram realizadas reuniões suplementares com a participação dos dois estudantes.

Nas primeiras semanas foi utilizada a técnica de pareamento para alinhar o novo integrante no desenvolvimento e foram desenvolvidas animações no placar. Após essa interação inicial, o estudante Leonardo continuou no desenvolvimento das animações e das novas visualizações do placar, enquanto o estudante Lieverton focou na integração do placar com os sistema Codeforces.

No segundo semestre da terceira etapa o foco do estudante Lieverton foi na implementação da comunicação com o sistema BOCA. Nessa etapa o Professor Dr. Bruno Ribas, mantenedor do BOCA, auxiliou o estudante no desenvolvimento do módulo de comunicação com o BOCA. Foram trocados *e-mails* a respeito da configuração do sistema e

da autenticação para requisitar os arquivos necessários para comunicação. As informações fornecidas pelo Professor Dr. Bruno Ribas estão sistematizadas na Seção Comunicação com o BOCA. O Quadro 4, apresenta o resumo das tarefas realizadas em cada uma das *sprints* ao longo das duas etapas finais.

Quadro 4 – Atividades por sprint

Sprint	**Tarefa**	**Participante(s)**
01	Familiarização com o BROMS	Leonardo
02	Análise comparativa com outros placares	Leonardo e Rafael
03	Novo layout: Paralelogramos	Leonardo e Rafael
04	Reorganização: Constantes e métodos	Leonardo e Rafael
05	Refatoração: Renderização de Paralelogramos	Leonardo e Rafael
06	Refatoração: Renderização de Textos	Leonardo e Rafael
07	Refatoração: Eventos	Leonardo e Rafael
08	Documentação TCC1	Leonardo
09	Apresentação TCC1	Leonardo
10	Familiarização com o BROMS	Lieverton
11	Animações no placar	Leonardo e Lieverton
12	Animações no placar	Leonardo e Lieverton
12	Estudo sobre a API do CF	Lieverton
13	Integração com a API do CF	Lieverton
13	Preparação do código para receber novas formas de visualização	Leonardo
14	Integração com a API do CF	Lieverton
14	Desenvolvimento do ensaio	Lieverton
14	Acompanhamento do experimento 1	Lieverton
14	Nova forma de visualização: Primeiro time a acertar cada questão	Leonardo
15	Resultados do experimento 1	Lieverton
15	Nova forma de visualização: Filtro pro região	Leonardo
16	Nova forma de visualização: Filtro pro região	Leonardo
16	Documentação TCC1	Lieverton
17	Documentação TCC1	Lieverton
17	Documentação TCC2	Leonardo
18	Documentação TCC2	Leonardo
18	Ajustar integração	Lieverton
19	Acompanhamento do experimento 2	Lieverton
19	Resultados do experimento 2	Lieverton
20	Apresentação TCC1	Lieverton
21	Junção entre as *branchs* do código	Leonardo e Lieverton
22	Forma de visualização: Informações extras dos times	Leonardo
23	Acompanhamento do experimento 3	Lieverton
23	Resultados do experimento 3	Lieverton
23	Forma de visualização: Informações extras dos times	Leonardo e Lieverton
24	Forma de visualização: Informações extras dos times	Leonardo
24	*Deploy* do BOCA	Lieverton
25	Forma de visualização: Informações extras dos times	Leonardo
25	Documentação TCC2	Leonardo e Lieverton
26	Forma de visualização: Informações extras dos times	Leonardo
26	Documentação TCC2	Leonardo e Lieverton
27	Documentação TCC2	Leonardo e Lieverton
27	Forma de visualização: Informações extras dos times	Leonardo
27	Integração com o BOCA	Lieverton
28	Integração com o BOCA	Leonardo e Lieverton
28	Acompanhamento do experimento 4	Lieverton
29	Correção do código	Leonardo e Lieverton
30	Correção do código	Leonardo e Lieverton
30	Documentação TCC2	Leonardo e Lieverton
31	Documentação TCC2	Leonardo e Lieverton

Fonte: Autores

3.3 Ensaios de Integração com a API do Codeforces

A realização dos ensaios experimentais tiveram como objetivo validar o acesso à API do Codeforces e verificar a eficiência do placar. Para isso, foram escolhidos eventos em tempo real gerenciados pelo Professor Dr. Edson Júnior. Esses eventos foram monitorados de forma remota.

Uma prova de Algoritmos e Programação de Computadores (APC), disciplina ministrada na FGA, foi escolhida como evento para realização do primeiro experimento. A prova foi realizada no dia 18 de março de 2022 às 16 horas, horário de Brasília, e teve duração de 90 minutos. A pontuação foi dada de acordo com a pontuação do regulamento da OBI. Devido o placar não prever o sistema de pontuação da OBI, foi aferido apenas o número de questões que tiveram acerto em sua totalidade para o cálculo da pontuação conforme o sistema de pontuação do ICPC.

Um segundo evento teste, uma prova de Tópicos Especiais em Programação (TEP), foi realizado no dia 23 de abril de 2022 às 08 horas, horário de Brasília, com duração de 4 horas. A pontuação seguiu o regulamento de competição do ICPC. A partir disso, foi melhor visível o resultado obtido pelo placar em comparação com o resultado final da competição disponibilizado na plataforma do Codeforces.

Por último, foi escolhida outra prova de TEP. O terceiro evento foi realizado no dia 15 de julho de 2022 às 16 horas e 20 minutos , horário de Brasília, com duração de 90 minutos. O evento ocorreu nos mesmos moldes do primeiro evento.

Para um melhor entendimento dos resultados obtidos, foram gerados dados que possam apontar prováveis pontos no software com falhas ou ineficiências. Neste sentido, foram usadas as seguintes métricas:

- número de requisições;
- número de requisições com falhas;
- tempo de resposta das requisições;
- tamanho das requisições; e
- número de exceções.

Os *logs* de console e de rede do navegador foram usados para levantar os dados necessários para as métricas. Com base nos *logs*, também foi possível verificar as causas que levaram a uma falha de requisição e a fonte das exceções.

3.3.1 Parâmetros Utilizados

Para solicitação de novas submissões no caso da comunicação com a API do Codeforces é necessário definir o número de submissões por requisição e o tempo entre cada ciclo de requisições.

A Tabela 3 apresenta os parâmetros definidos para a realização do primeiro experimento, enquanto a Tabela 4 apresenta os parâmetros definidos para a realização do segundo experimento. Por fim, a Tabela 5 apresenta os parâmetros definidos para a realização do terceiro experimento.

Tabela 3 – Parâmetros utilizados no experimento 1

Parâmetro	**Valor**
Número de submissões por requisição	50 submissões
Tempo esperado a cada ciclo de requisições	10 segundos

Fonte: Autores

Tabela 4 – Parâmetros utilizados no experimento 2

Parâmetro	**Valor**
Número de submissões por requisição	50 submissões
Tempo esperado a cada ciclo de requisições	30 segundos

Fonte: Autores

Tabela 5 – Parâmetros utilizados no experimento 3

Parâmetro	**Valor**
Número de submissões por requisição	50 submissões
Tempo esperado a cada ciclo de requisições	30 segundos

Fonte: Autores

3.3.2 Procedimento

Com a finalidade de prover um processo adequado para execução dos ensaios experimentais foi desenvolvimento um procedimento. Esse procedimento mostra as etapas necessárias para a correta execução da aplicação e coleta dos dados. As etapas são apresentadas a seguir:

1. Passar os parâmetros necessários para o correto funcionamento das requisições: `key`, `secret` e `contestId`.

2. Iniciar o servidor: `python3 -m http.server 3000`.
3. Acessar o endereço `http://localhost:3000/` no navegador.
4. Acessar as ferramentas de desenvolvimento do navegar: `F12`.
5. Habilitar a opção de preservar *log* nas abas de console e rede.
6. Atualizar a página do navegador: `F5`.

3.3.3 Possíveis Erros

Durante o desenvolvimento os servidores do Codeforces apresentaram instabilidade. Com isso é possível que, durante a realização do experimento, algumas requisições falhem. No caso do primeiro ciclo de requisições falhar a aplicação não iniciará corretamente e para contornar esse problema será necessário atualizar a página da aplicação.

A API do Codeforces pode bloquear o acesso e ficar indisponível se muitas requisições forem feitas em um curto intervalo de tempo. Se forem abertas várias abas do navegador com instâncias do BROMS é provável que a API fique indisponível durante um tempo, retornando erro nas requisições. Para evitar esse problema deve haver apenas uma aba com o BROMS. No caso de ocorrer esse problema, devem ser fechadas todas as abas secundárias e a única aba que deve continuar em execução deve ser atualizada.

3.4 Ensaio de Integração com o BOCA

Com finalidade de verificar o funcionamento do BROMS em uma competição real gerenciada no BOCA foi escolhido um evento produzido pela UnB, a X Maratona UnB de Programação. O evento foi realizado no dia 31 de agosto de 2022 no campus Darcy Ribeiro às 13 horas, horário de Brasília, e teve duração de 5 horas.

Diferente dos experimentos realizados com o Codeforces, o monitoramento do evento foi realizado presencialmente, com oportunidade de expor o placar para a equipe responsável por gerenciar o evento.

3.4.1 Procedimento

O procedimento definido para o ensaio com o BOCA é similar ao do com o Codeforces, com a diferença de credenciamento e a adição da etapa para iniciar a *API mock*.

1. Atualizar o link e a chave do BOCA na *API mock*.
2. Iniciar a *API mock*: `uvicorn main:app`

3. Iniciar o servidor: `python3 -m http.server 3000`.
4. Acessar o endereço `http://localhost:3000/` no navegador.
5. Acessar as ferramentas de desenvolvimento do navegar: `F12`.
6. Habilitar a opção de preservar *log* nas abas de console e rede.
7. Atualizar a página do navegador: `F5`.

4 Resultados

Este capítulo apresenta os resultados obtidos durante os ensaios experimentais. Divide-se em duas seções, onde, na Seção Integração com a API do Codeforces são apresentados os resultados obtidos nos ensaios experimentais de integração com a API do Codeforces, enquanto na Seção Integração com o BOCA são apresentados os resultados obtidos no ensaio experimental de integração com o BOCA.

4.1 Integração com a API do Codeforces

Após o primeiro ensaio experimental foram obtidos resultados que apontaram falhas no carregamento de dados essenciais à aplicação. Com isso, foi possível perceber pontos de melhoria nas requisições dos dados. Também foi possível validar a construção do placar.

Ao iniciar o primeiro evento teste, verificou-se incompatibilidade entre o método de inicialização dos times e a API do Codeforces que disponibiliza os dados a respeito dos times. Para o correto funcionamento do BROMS, era necessário que as informações dos times fossem carregadas na primeira interação, logo que a primeira requisição *web* busca os times e as questões, informações fundamentais para a definição e dimensionamento da aplicação gráfica (MAKAHA, 2021). Entretanto, a API do Codeforces disponibiliza apenas informações de times que efetuaram alguma submissão. Com isso, ao iniciar um evento, não há dados a respeito dos times e por conseguinte, quando há atualização das submissões, não é possível fazer o mapeamento das submissões aos seus respectivos times. Para dar prosseguimento no experimento, a página foi atualizada periodicamente de forma manual para carregar novos participantes. Uma possível solução é fazer requisições para atualização dos times e participantes a cada ciclo de requisições.

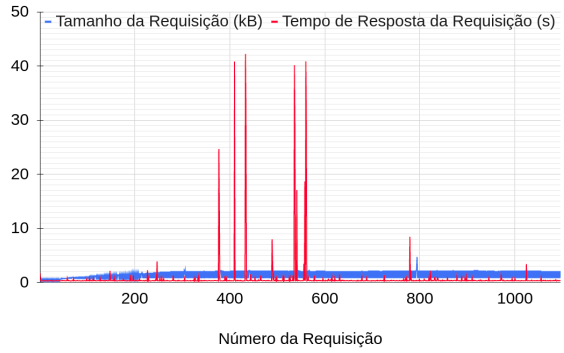
Foram feitas 1096 requisições em uma instância do BROMS durante todo o evento, todas elas bem sucedidas. Em uma instância paralela, para testar a saturação da API do Codeforces, apenas uma requisição retornou um erro 503 (*Service Temporarily Unavailable*), mas o erro não resultou em mau funcionamento da aplicação.

As Figuras de 13a a 13c apresentam os tamanhos e os tempos de resposta de todas as requisições. Após analisar os dados obtidos, constatou-se a ocorrência de alternância entre as requisições dos tipos *standings* e *status*, no total de 548 requisições para cada tipo de requisição. Esse efeito ocorreu por conta de não haver necessidade de mais do que uma requisição do tipo *status* para manter os dados atualizados. O efeito de alternância pode ser vista na Figura 13d, logo que cada tipo de requisição tende a um valor diferente

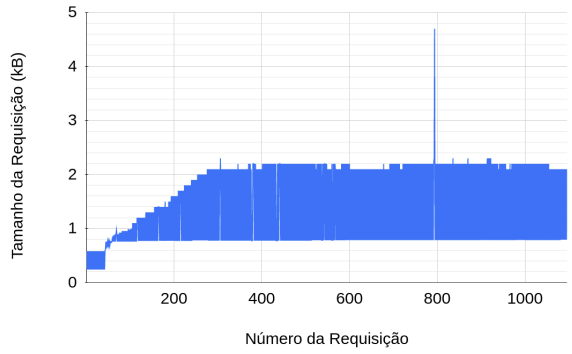
Figura 13 – Requisições feitas à API do Codeforces

(a) Tempo de resposta das requisições feitas à API do Codeforces

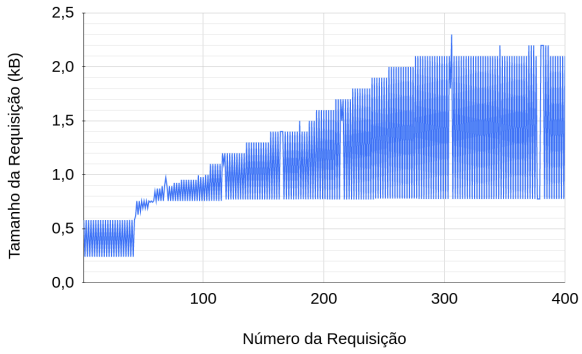
(b) Tamanho e tempo de resposta das requisições feitas à API do Codeforces



(c) Tamanho das requisições feitas à API do Codeforces



(d) Tamanho das primeiras 400 requisições feitas à API do Codeforces



Fonte: Autores

para o tamanho da requisição.

Para melhor compreensão do comportamento de cada tipo de requisição, os dados foram separados pelo tipo (Figs. 14a a 16b). A Figura 15a mostra que a moda do tamanho das requisições do tipo *standings* se aproxima de 800 bytes, enquanto a Figura 15b mostra que a moda das requisições do tipo *status* se aproxima de 2,2 quilobytes. Pode ser visto alguns picos no tamanho das requisições nos gráficos. Os picos ocorreram devido ao fato de que, na atualização manual das páginas, não há limitador no número de itens para as respostas das requisições na primeira interação, comportamento esperado para manter a aplicação consistente. Os picos de tamanho não coincidem com os picos de tempo de resposta, como é visível nas Figuras 16a e 16b, assim, o tamanho das requisições não impactou no tempo de resposta, porém, em eventos com uma maior quantidade de participantes, por exemplo mais de 300 participantes, o tamanho das respostas das requisições pode impactar no tempo de resposta na ocorrência de atualização da página da aplicação, levando um tempo considerável para renderizar o placar.

Figura 14 – Tempo de resposta das requisições feitas à API do Codeforces

(a) Tempo de resposta das requisições feitas à API *standings* do Codeforces

(b) Tempo de resposta das requisições feitas à API *status* do Codeforces

Fonte: Autores

Figura 15 – Tamanho das requisições feitas à API do Codeforces

(a) Tamanho das requisições feitas à API *standings* do Codeforces

(b) Tamanho das requisições feitas à API *status* do Codeforces

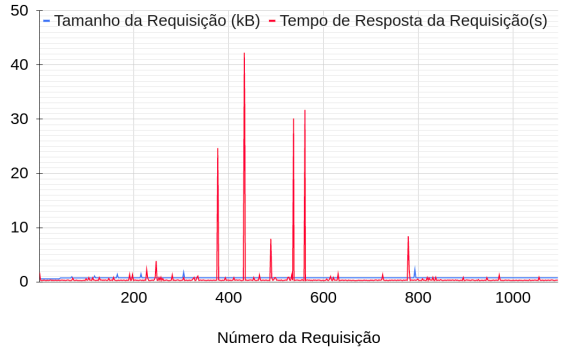
Fonte: Autores

Para testar o impacto de carregar os dados de uma competição através de uma única requisição, foi feita uma série de requisições às API's *standings* e *status*. Foram feitas 100 requisições aos endereços <https://codeforces.com/api/contest.standings?contestId=566&from=1> e <https://codeforces.com/api/contest.status?contestId=566&from=1>. Os endereços citados possuíssem dados de uma competição pública já finalizada, esse tipo de competição no Codeforces conta com uma grande participação de usuários, nesse caso 625 usuários. Em média, a resposta com os dados do tipo *standings* demorou 600 milissegundos para carregar 28,7 quilobytes, enquanto do tipo *status* demorou 25,6 segundos para carregar 946 quilobytes.

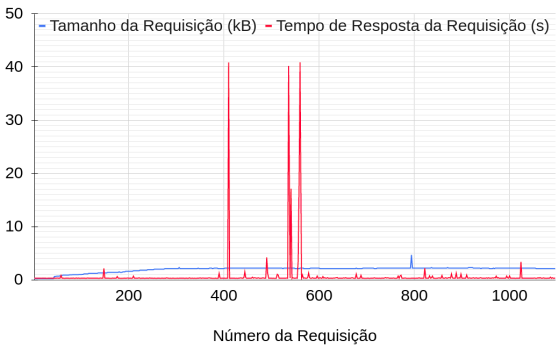
De forma geral, os resultados obtidos no primeiro ensaio, tanto no tempo de resposta das requisições quanto no tamanho, foram satisfatórios. Além do mais, o placar final

Figura 16 – Tamanho e tempo de resposta das requisições feitas à API do Codeforces

(a) Tamanho e tempo de resposta das requisições feitas à API *standings* do Codeforces



(b) Tamanho e tempo de resposta das requisições feitas à API *status* do Codeforces



Fonte: Autores

apresentou a pontuação correta de acordo como era esperado para o sistema de pontuação do ICPC (Fig. 17). A Figura 18 apresenta o resultado final da prova obtido através da captura de tela da interface do Codeforces.

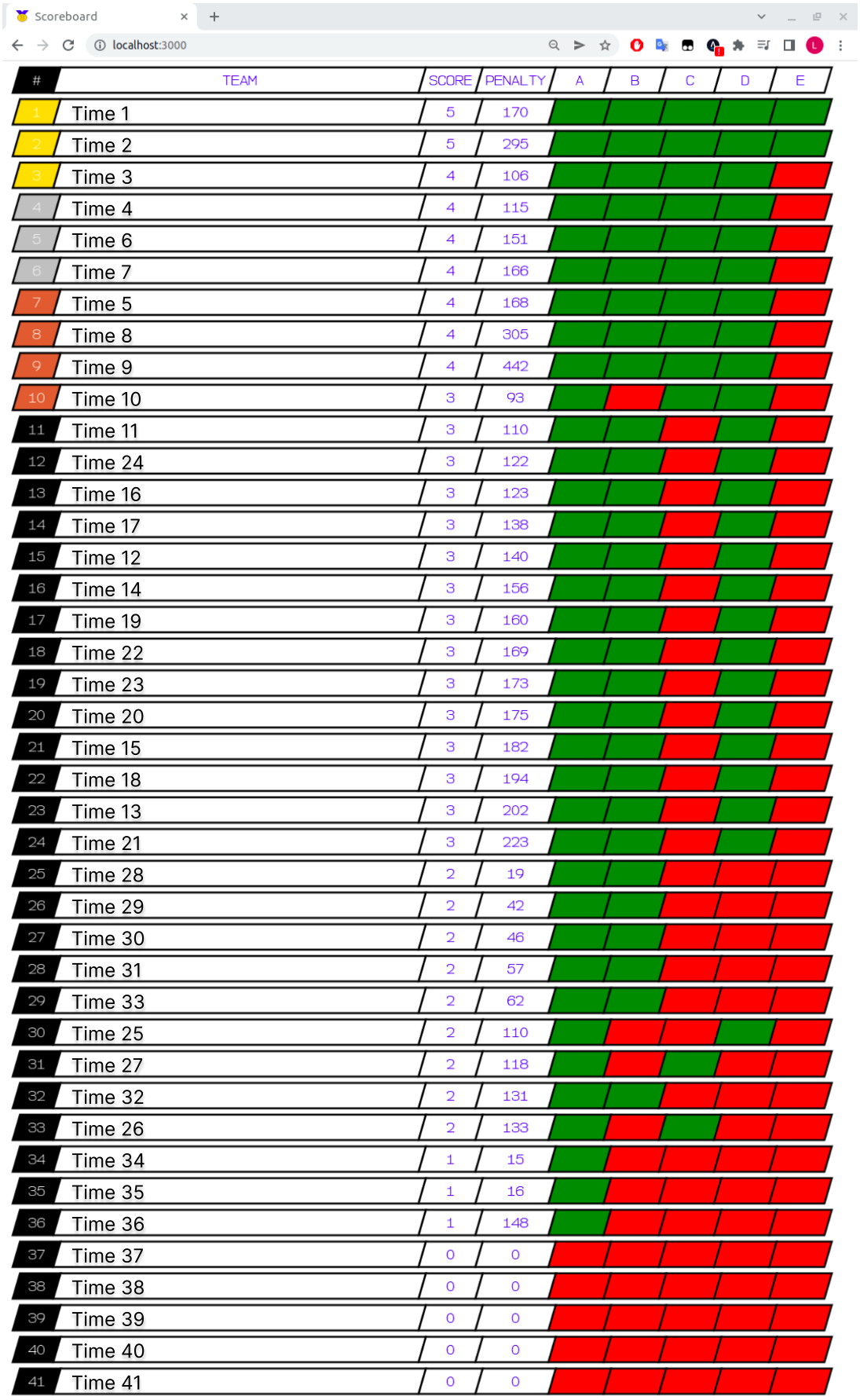
Para o segundo ensaio experimental foi feito um ajuste no programa que ao verificar a existência de novas submissões é feita uma nova requisição para atualizar a lista de times. Logo após as primeiras submissões verificou-se que havia duplicidade de times no placar (Fig. 19). Constatou-se que o erro acontecia na atualização de uma variável de controle e a implementação foi corrigida durante o evento.

Durante o evento a API do Codeforces demonstrou-se bastante instável, ficando inacessível durante um período aproximado de 25 minutos, além de outros instantes menos impactantes. Com isso o placar não era atualizado em muitos momentos. Além do mais, apesar de receber novas submissões da API, algumas submissões também não impactaram na atualização do placar.

O resultado final do placar apresentou valores semelhantes ao resultado obtido pela interface do Codeforces, com exceção da pontuação de alguns times, devido ao Codeforces não contabilizar submissões com erro de compilação para o cálculo da penalidade e a regra de negócio implementada, que está de acordo com ICPC, contabiliza. As Figuras 20 e 21 apresentam os resultados finais obtidos pelos cinco primeiros times e pode ser vista a diferença de penalidade para o time 5.

O terceiro evento teste serviu para validar a atualização do placar de forma automática. Percebeu-se que o erro não acontecia devido a um problema de concorrência e sim por uma estrutura que alterava as ordens das submissões ao pegar novas submissões. Como era utilizado o identificador da submissão como parâmetro para procurar as novas submissões e este campo é dado em ordem decrescente, ao usar o identificador da su-

Figura 17 – Placar da prova de APC



#	TEAM	SCORE	PENALTY	A	B	C	D	E
1	Time 1	5	170					
2	Time 2	5	295					
3	Time 3	4	106					
4	Time 4	4	115					
5	Time 6	4	151					
6	Time 7	4	166					
7	Time 5	4	168					
8	Time 8	4	305					
9	Time 9	4	442					
10	Time 10	3	93					
11	Time 11	3	110					
12	Time 24	3	122					
13	Time 16	3	123					
14	Time 17	3	138					
15	Time 12	3	140					
16	Time 14	3	156					
17	Time 19	3	160					
18	Time 22	3	169					
19	Time 23	3	173					
20	Time 20	3	175					
21	Time 15	3	182					
22	Time 18	3	194					
23	Time 13	3	202					
24	Time 21	3	223					
25	Time 28	2	19					
26	Time 29	2	42					
27	Time 30	2	46					
28	Time 31	2	57					
29	Time 33	2	62					
30	Time 25	2	110					
31	Time 27	2	118					
32	Time 32	2	131					
33	Time 26	2	133					
34	Time 34	1	15					
35	Time 35	1	16					
36	Time 36	1	148					
37	Time 37	0	0					
38	Time 38	0	0					
39	Time 39	0	0					
40	Time 40	0	0					
41	Time 41	0	0					

Fonte: Autores

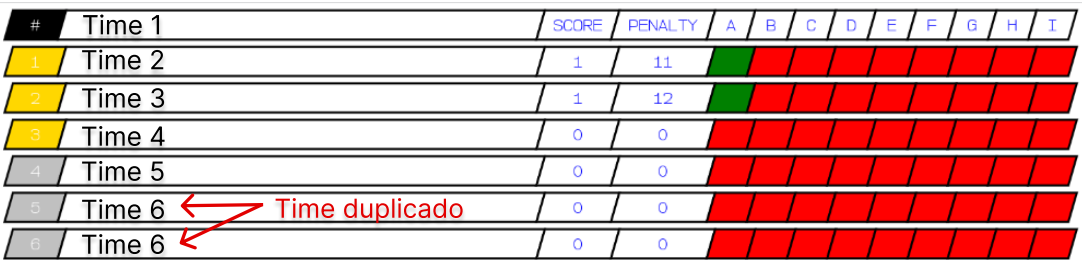
Figura 18 – Resultado final da prova de APC

Standings

#	Who	=	A	B	C	D	E
1	Time 1	100	10 00:03	15 00:09	20 01:14	25 00:22	30 01:02
1	Time 2	100	10 00:32	15 00:37	20 00:42	25 00:48	30 01:16
3	Time 3	90	10 00:03	15 00:09	20 00:19	25 01:15	20 01:25
4	Time 4	70	10 00:03	15 00:20	20 00:35	25 00:57	
4	Time 5	70	10 00:05	15 00:32	20 01:00	25 01:11	
4	Time 6	70	10 00:12	15 00:22	20 00:41	25 00:56	
4	Time 7	70	10 00:05	15 00:20	20 00:38	25 01:03	0
4	Time 8	70	10 00:10	15 00:30	20 01:29	25 01:16	
4	Time 9	70	10 00:30	15 00:41	20 01:29	25 01:22	
10	Time 10	58	10 00:05	3 01:20	20 00:38	25 00:50	
11	Time 11	50	10 00:08	15 00:26		25 01:16	
11	Time 12	50	10 00:11	15 00:32	0	25 01:17	
11	Time 13	50	10 00:09	15 00:54		25 01:19	
11	Time 14	50	10 00:10	15 00:29	0	25 01:17	0
11	Time 15	50	10 00:06	15 01:16		25 00:40	
11	Time 16	50	10 00:13	15 00:32	0	25 01:18	
11	Time 17	50	10 00:11	15 00:24	0	25 01:23	
11	Time 18	50	10 00:15	15 00:55		25 01:24	
11	Time 19	50	10 00:19	15 00:56		25 01:25	
11	Time 20	50	10 00:04	15 00:23	0	25 01:28	
11	Time 21	50	10 00:54	15 00:21		25 01:28	
11	Time 22	50	10 00:08	15 01:13		25 01:28	
11	Time 23	50	10 00:09	15 00:35		25 01:29	
11	Time 24	50	10 00:08	15 00:25	0	25 01:29	
25	Time 25	35	10 00:06			25 01:24	
26	Time 26	33	10 00:05	3 01:29	20 01:28		
27	Time 27	30	10 00:11		20 01:27	0	
28	Time 28	28	10 00:05	15 00:14			3 01:24
29	Time 29	25	10 00:04	15 00:38			
29	Time 30	25	10 00:07	15 00:39			
29	Time 31	25	10 00:14	15 00:43			
29	Time 32	25	10 00:17	15 01:14			
29	Time 33	25	10 00:11	15 00:51		0	
34	Time 34	13	10 00:15	3 00:59			
34	Time 35	13	10 00:16	3 01:28		0	
36	Time 36	10	10 00:48	0			
37	Time 37	0	0				
37	Time 38	0	0				
37	Time 39	0	0	0			
37	Time 40	0	0				
37	Time 41	0	0				
	Accepted Tried		36 41	29 35	12 18	25 28	2 7

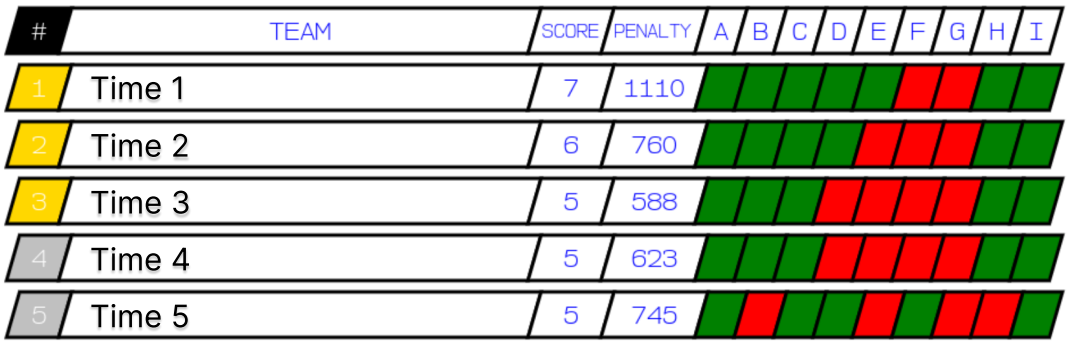
Fonte: Codeforces (2022)

Figura 19 – Erro de time duplicado



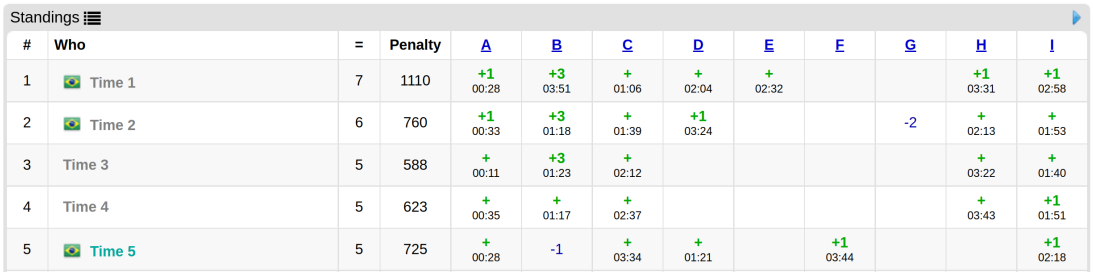
Fonte: Autores

Figura 20 – *Top* 5 da prova de TEP no placar



Fonte: Autores

Figura 21 – *Top* 5 da prova de TEP no Codeforces



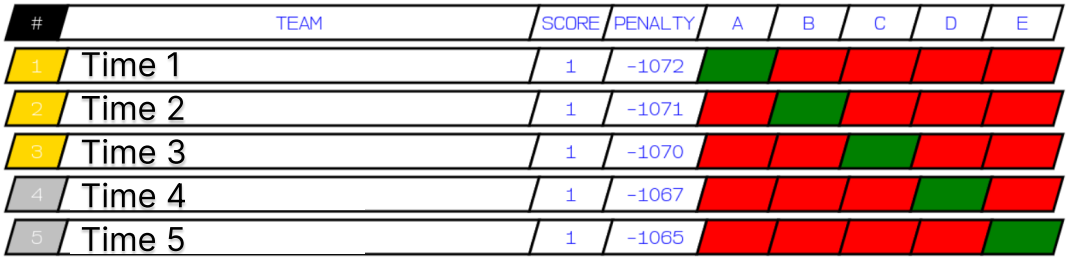
Fonte: Codeforces (2022)

posta última submissão encontrada algumas submissões eram descartadas, pois possuíam o identificador menor que a submissão que era usada como parâmetro e que estava fora de ordem.

Ao iniciar o evento verificou-se que já havia submissões feitas e o placar mostrava uma pontuação negativa. Esse erro ocorreu devido a algumas submissões realizadas no dia anterior do evento para validar os pacotes das questões. Esse erro foi rapidamente corrigido filtrando as submissões com o valor negativo em relação ao tempo relativo ao início do evento. A Figura 22 mostra o erro ocorrido ao iniciar o placar.

Apesar do imprevisto, com apenas um erro encontrado durante todo o evento, e do sistema de pontuação ser diferente do ICPC o experimento foi suficiente para validar

Figura 22 – Erro de pontuação negativa



Fonte: Autores

a atualização do placar de forma automática, cumprindo com seu objetivo.

4.2 Integração com o BOCA

Com a realização do ensaio experimental na X Maratona UnB de Programação foi possível verificar o funcionamento do BROMS e mapear alguns problemas.

No desenvolvimento da *API mock* foi assumido de forma equivocada que as submissões estavam ordenadas pelo identificador. O resultado disso foi que algumas submissões não eram carregadas. Como solução provisória, durante o evento foi feita uma alteração no código para que todas as submissões fossem carregadas. Mas essa solução não sanou o problema por completo, logo que dependendo da última submissão enviada alguma outra submissão podia ser processada mais de uma vez, causando erro no cálculo da penalidade, sendo necessária a atualização da página do placar para corrigir o estado. Como solução final foi feita uma alteração no parâmetro que é passado para consultar as novas submissões, no lugar do identificador da última submissão processada é utilizado o índice para a próxima submissão a ser solicitada. Com isso a *API mock* verifica se há submissões a partir do índice informado: se existir a API retorna as submissões, caso contrário retorna uma lista vazia.

Durante a maratona verificou-se que um dos times estava com o final do nome errado. O nome do time terminava com aspas duplas, mas no lugar de aparecer o carácter das aspas duplas era mostrado o código: `&quot;`. Como solução foi utilizada a função `unscape` da biblioteca HTML do Python ao carregar os nomes dos times.

O evento serviu também para testar outras duas *branchs* de desenvolvimento, as das animações e das novas visualizações. Não foi possível validar as visualizações, mas foi identificado que as submissões de um time não eram contabilizadas. O problema aconteceu devido a um erro no intervalo considerado na lista que continha as informações a serem desenhadas na tela. Então foi corrigido o intervalo contabilizado e o problema foi solucionado.

Apesar dos erros encontrados foi possível validar a comunicação com o sistema

BOCA, em que o sistema não demonstrou erros nas requisições. Também foi possível validar a animação feita na troca das posições dos times.

5 Considerações Finais

Em suma, o trabalho feito culminou em promover a escalabilidade do projeto, havendo grande melhoria na qualidade geral do código. As refatorações realizadas contribuíram também para melhor definir as responsabilidades de cada classe e desacoplar as partes. Além disso, obteve-se um impacto positivo na diminuição da barreira de entrada para contribuir com o projeto. Com o desacoplamento das partes os componentes passaram a ser mais autônomos e melhorias podem ser implementadas de forma mais específica, sem exigir um entendimento de toda aplicação.

As funcionalidades apresentadas neste trabalho foram integradas e estão disponíveis para competições gerenciadas tanto pelo Codeforces quanto pelo BOCA. Com isso, o BROMS pode ser usado na transmissão dos eventos de ambas plataformas. As funcionalidades *blind* e *frozen* não foram implementadas neste trabalho, porém o desenvolvimento pode ser feito sem grandes dificuldades em futuros incrementos.

A maior dificuldade em desenvolver o BROMS é respeitar o requisito de manter as dependências mínimas e usar apenas as bibliotecas padrões dos navegadores para implementar as funcionalidades. Por conta da complexidade de descompactar um arquivo utilizando as bibliotecas padrões dos navegadores, preferiu-se adaptar a *API mock* para a comunicação com o BOCA. Em um possível futuro incremento, com a disponibilização dos arquivos `contest` e `runs` de forma direta sem a necessidade de descompactar, os arquivos poderão ser carregados diretamente no BROMS, sem a necessidade da *API mock* como intermediária na comunicação. Porém essa alteração tem um maior custo na requisição dos dados, podendo mais que dobrar bytes transmitidos. Uma outra abordagem seria implementar uma nova API no BOCA para que não sejam transmitidos dados redundantes.

Por fim, o BROMS ainda necessita de uma documentação no que se refere aos guias de como contribuir com o projeto, manual de uso e algumas definições do repositório[1], como licença aplicada sobre o código, que devem ser definidos para possibilitar o desenvolvimento do BROMS pela comunidade.

[1] link do repositório do BROMS: <https://github.com/rafaelmakaha/maratona-BROMS>

Referências

CAMPOS, C. P. de; FERREIRA, C. E. Boca: um sistema de apoio a competições de programação. 2004. Citado 3 vezes nas páginas 21, 26 e 27.

CODEFORCES. *Codeforces*. 2022. <https://codeforces.com/>. Acessado em: 2022-07-15. Citado 2 vezes nas páginas 64 e 65.

IEEE. *IEEEXtreme*. 2021. <https://ieeextreme.org/>. Acessado em: 2021-10-10. Citado na página 23.

LAAKSONEN, A. *Competitive Programmer's Handbook*. [S.l.]: Finland, 2018. Citado na página 23.

MAKAHA, R. Broms: Brazilian online marathon scoreboard. Universidade de Brasília, 2021. Citado 9 vezes nas páginas 21, 27, 28, 30, 31, 32, 35, 53 e 59.

MIRZAYANOV, M. *Codeforces*. 2011. Disponível em: <https://codeforces.com/>. Acessado em: 11 mar. 2022. Citado 10 vezes nas páginas 21, 23, 24, 27, 29, 30, 75, 76, 77 e 79.

PREIBISCH, S. *API Development A Practical Guide for Business Implementation Success*. [S.l.]: Apress, 2018. Citado na página 28.

RUSTRIMEITOR, M. *Maratona Rustrimeitor*. 2021. <https://github.com/wuerges/maratona-animeitor-rust>. Acessado em: 2021-10-10. Citado 2 vezes nas páginas 25 e 53.

SKIENA, S. S.; REVILLA, M. A. *Programming challenges: the programming contest training manual*. [S.l.]: Springer, 2003. Citado na página 26.

UNICAMP. *OBI*. 2022. Disponível em: <https://olimpiada.ic.unicamp.br/info/regulamento/>. Acessado em: 11 mar. 2022. Citado na página 24.

Anexos

ANEXO A – Objetos que compõem o retorno do método `contest.standings` da API do Codeforces

O método `contest.standings` retorna dados sobre a competição e a classificação. O retorno é composto por um objeto JSON formado por três outros objetos: `contest`, `problems` e `rows`. Os atributos `problems` e `rows` são listas de objetos de `problem` e `ranklistRow` respectivamente. Os Quadros de 5 a 10 apresentam os detalhes de cada objeto que compõe o retorno do método `contest.standings`.

Quadro 5 – Detalhes do objeto `contest`

Campo	**Tipo**	**Descrição**
`id`	`integer`	Identificador da competição.
`name`	`string`	Nome da competição.
`type`	`enum`	Sistema de pontuação: CF, IOI, ICPC.
`phase`	`enum`	Fase da competição: `BEFORE`, `CODING`, `PENDING_SYSTEM_TEST`, `SYSTEM_TEST`, `FINISHED`.
`frozen`	`boolean`	Se verdadeiro, a lista de classificação da competição será congelada e mostrará apenas os envios criados antes do congelamento.
`durationSeconds`	`integer`	Duração da competição em segundos.
`startTimeSeconds`	`integer`	Hora de início da competição no formato Unix. Pode ser nulo.
`parentTimeSeconds`	`integer`	Número de segundos após o início. Pode ser nulo.
`preparedBy`	`string`	Identificador do usuário que criou a competição. Pode ser nulo.
`webSiteUrl`	`string`	URL da página da competição. Pode ser nulo.
`description`	`string`	Descrição da competição. Pode ser nulo.
`difficulty`	`integer`	Dificuldade de 1 a 5, quanto maior o número maior a dificuldade. Pode ser nulo.
`kind`	`string`	Tipo legível da competição. Pode ser nulo.
`icpcRegion`	`string`	Nome da região para competições oficiais do ICPC. Pode ser nulo.
`country`	`string`	País da competição. Pode ser nulo.
`city`	`string`	Cidade da competição. Pode ser nulo.
`season`	`string`	Seção da competição. Pode ser nulo.

Fonte: Mirzayanov (2011)

Quadro 6 – Detalhes do objeto `problem`

Campo	**Tipo**	**Descrição**
`contestId`	`integer`	Identificador da competição. Pode ser nulo.
`problemsetName`	`string`	Nome abreviado do conjunto de problemas ao qual o problema pertence. Pode ser nulo.
`index`	`string`	Uma letra ou letra com dígito(s) indicando o índice do problema.
`name`	`string`	Nome do problema.
`type`	`enum`	`PROGRAMMING`, `QUESTION`.
`points`	`float`	Quantidade máxima de pontos para o problema. Pode ser nulo.
`rating`	`integer`	Dificuldade do problema. Pode ser nulo.
`tags`	`string[ ]`	*Tags* do problema.

Fonte: Mirzayanov (2011)

Quadro 7 – Detalhes do objeto `ranklistRow`

Campo	**Tipo**	**Descrição**
`party`	`party object`	Time participante da competição.
`rank`	`integer`	Lugar do time na competição.
`points`	`float`	Quantidade total de pontos marcados do time.
`penalty`	`integer`	Penalidade total do time (ICPC).
`successfulHackCount`	`integer`	Quantidade de *hacks* bem sucedidos.
`unsuccessfulHackCount`	`integer`	Quantidade de *hacks* mal sucedidos.
`problemResults`	`problemResult[ ]`	Resultado do time em cada problema. Pode ser nulo.
`lastSubmissionTimeSeconds`	`integer`	Tempo em segundos desde o início da competição até a última submissão que adicionou algum ponto à pontuação total do time (IOI). Pode ser nulo.

Fonte: Mirzayanov (2011)

Quadro 8 – Detalhes do objeto `party`

Campo	Tipo	Descrição
`contestID`	`integer`	Identificador da competição. Pode ser nulo.
`members`	`member[ ]`	Lista de membros do time.
`participantType`	`enum`	`CONTESTANT, PRACTICE, VIRTUAL, MANAGER, OUT_OF_COMPETITION`.
`teamId`	`integer`	Id do time. Pode ser nulo.
`teamName`	`string`	Nome do time. Pode ser nulo. Pode ser nulo. Pode ser nulo.
`ghost`	`boolean`	Se verdadeiro, participou da competição, mas não do Codeforces.
`room`	`integer`	Sala do time. Pode ser nulo. Pode ser nulo.
`startTimeSeconds`	`integer`	Tempo em segundos que o time iniciou a competição. Pode ser nulo.

Fonte: Mirzayanov (2011)

Quadro 9 – Detalhes do objeto `member`

Campo	Tipo	Descrição
`handle`	`string`	Identificador do usuário.
`name`	`string`	Nome do usuário. Pode ser nulo.

Fonte: Mirzayanov (2011)

Quadro 10 – Detalhes do objeto `problemResult`

Campo	Tipo	Descrição
`points`	`float`	Pontuação obtida.
`penalty`	`integer`	Penalidade do time no problema (ICPC). Pode ser nulo.
`rejectedAttemptCount`	`integer`	Número de submissões incorretas.
`type`	`enum`	`PRELIMINARY, FINAL`.
`bestSubmissionTimeSeconds`	`integer`	Número de segundos após o início da competição antes da submissão que trouxe a quantidade máxima de pontos o problema. Pode ser nulo.

Fonte: Mirzayanov (2011)

ANEXO B – Objeto que compõe o retorno do método `contest.status` da API do Codeforces

O método `contest.status` retorna uma lista em ordem decrescente de envio com dados sobre as submissões feitas na competição. O Quadro 11 mostra os detalhes do objeto do tipo `submission`.

Quadro 11 – Detalhes do objeto `submission`

Campo	**Tipo**	**Descrição**
`id`	`integer`	Identificador da submissão.
`contestId`	`integer`	Identificador da competição. Pode ser nulo.
`creationTimeSeconds`	`integer`	Hora em que a submissão foi criada em formato Unix.
`relativeTimeSeconds`	`integer`	Número de segundos passados após o início da competição até a submissão.
`problem`	`problem`	Problema respondido pela submissão.
`author`	`party`	Autor da submissão.
`programmingLanguage`	`string`	Linguagem utilizada na submissão.
`verdict`	`enum`	`FAILED`, `OK`, `PARTIAL`, `COMPILATION_ERROR`, `RUNTIME_ERROR`, `WRONG_ANSWER`, `PRESENTATION_ERROR`, `TIME_LIMIT_EXCEEDED`, `MEMORY_LIMIT_EXCEEDED`, `IDLENESS_LIMIT_EXCEEDED`, `SECURITY_VIOLATED`, `CRASHED`, `INPUT_PREPARATION_CRASHED`, `CHALLENGED`, `SKIPPED`, `TESTING`, `REJECTED`.
`testset`	`enum`	`SAMPLES`, `PRETESTS`, `TESTS`, `CHALLENGES`, `TESTS1`, `...`, `TESTS10`.
`passedTestCount`	`integer`	Número de testes aprovados.
`timeConsumedMillis`	`integer`	Tempo máximo em milissegundos de uma solução para um teste.
`memoryConsumedBytes`	`integer`	Memória máxima em *bytes* de uma solução para um teste.
`points`	`float`	Número de pontos marcados (IOI). Pode ser nulo.

Fonte: Mirzayanov (2011)