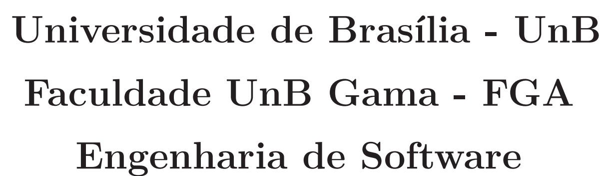
Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Engenharia de Software



BROJ - Juiz eletrônico e online para ensino e aprendizagem

Autores: Caio Nardelli Maranhão,
Simião Correia Lima de Carvalho

Orientador: Dr. Edson Alves da Costa Júnior

Brasília, DF

2017

Caio Nardelli Maranhão,
Simião Correia Lima de Carvalho

BROJ - Juiz eletrônico e online para ensino e aprendizagem

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: Dr. Edson Alves da Costa Júnior

Coorientador: Dr. Fábio Macedo Mendes

Brasília, DF
2017

Caio Nardelli Maranhão,
Simião Correia Lima de Carvalho
BROJ - Juiz eletrônico e online para ensino e aprendizagem/ Caio Nardelli Maranhão,
Simião Correia Lima de Carvalho. – Brasília, DF, 2017-
56 p. : il. (algumas color.) ; 30 cm.

Orientador: Dr. Edson Alves da Costa Júnior

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2017.

1. juiz eletrônico. 2. aplicação educacional. I. Dr. Edson Alves da Costa Júnior. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. BROJ - Juiz eletrônico e online para ensino e aprendizagem

CDU 02:141:005.6

Caio Nardelli Maranhão,
Simião Correia Lima de Carvalho

BROJ - Juiz eletrônico e online para ensino e aprendizagem

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, 07 de julho de 2017:

Dr. Edson Alves da Costa Júnior
Orientador

Dr. Fábio Macedo Mendes
Coorientador

Dr. Fernando William Cruz
Convidado 1

Prof. Felipe Duerno do Couto Almeida
Convidado 2

Brasília, DF
2017

Dedicamos este trabalho aos que merecem.

Agradecimentos

Eu, Caio Nardelli, agradeço aos meus pais e à minha família pela educação e paciência. Agradeço também à minha namorada pelo carinho imensurável. Por fim, agradeço à todos que de alguma forma contribuiram na minha jornada até aqui.

Eu, Simião Carvalho, agradeço primeiramente à minha família, principalmente meus pais. Todos os amigos que fiz ao longo do caminho desde os que ficaram do Ensino Médio: Larissa Moreira e Thiago Von Grapp aos que conheci na faculdade e permaneceram por toda a graduação: Paula Gurgel, Matheus Godinho e João Paulo Ribeiro. E todos que me ajudaram a chegar aqui.

Agradecemos ao nosso orientador Prof. Edson Alves, pelo conhecimento e pelas virtudes passadas, aos professores da banca, e aos professores Paulo Meirelles, Maurício Serrano e Milene Serrano. Agradecemos também aos nossos amigos, especialmente do grupo SACC.

"O que temos que ter sempre em mente é que a hegemonia do ambiente social acarreta um processo de transformação e modernização de todos os recursos funcionais envolvidos para a sociedade como um todo."

(Desconhecido)

Resumo

Criadas inicialmente com o intuito de elevar o nível de conhecimento dos programadores no mercado, competições de programação tomaram proporções inimagináveis se espalhando por todo mundo em eventos de diferentes tamanhos e propósitos. A partir disso as universidades começam a aderir à cultura da programação competitiva como forma de motivação para que os alunos aprendam programação de forma eficiente e estejam preparados para o mercado. Começa a se tornar um desafio em termos de infraestrutura dar suporte à essa demanda crescente de programadores participando de competições pelo mundo. Esse trabalho tem como objetivo facilitar a operação da infraestrutura tanto dos eventos de competições como das aplicações educacionais de correção automatizada através da implementação de um juiz eletrônico capaz de suportar grandes cargas de processamento e tolerante à falhas, e que pode ser estendido para diversas aplicações.

Palavras-chaves: juiz eletrônico. tecnologias educacional. programação competitiva.

Abstract

Initially created to raise the level of programmers' knowledge in the marketplace, programming competitions have taken unimaginable proportions, spreading throughout the world in events of different sizes and purposes. From that, the universities begin to adhere to the culture of competitive programming as a form of motivation for students to learn and be prepared for the market. It's starts becoming a challenge with respect to infrastructure to support this growing demand of programmers participating in competitions around the world. This work aims to facilitate the infrastructure operations of both competitions and educational applications of automated grading through the implementation of a fault tolerant electronic judge capable of withstanding large processing loads, which can be extended to several applications.

Key-words: online judge. educational technologies. competitive programming.

Lista de ilustrações

Figura 1 – A pilha de protocolos da Internet 28
Figura 2 – Requisições síncronas e assíncronas 28
Figura 3 – Virtualização baseada em *containers* 30
Figura 4 – Visão lógica arquitetural 35
Figura 5 – Visão de múltiplos juízes eletrônicos 35
Figura 6 – Visão de múltiplos clientes 36
Figura 7 – MER do juíz eletrônico 43
Figura 8 – Tela de um contest no BROJ 43
Figura 9 – Tela de um problema no BROJ 44
Figura 10 – Lista de problemas no BROJ, visão administrador 44
Figura 11 – Cadastro de um problema 45
Figura 12 – Cadastro de um *contest* 46

Lista de tabelas

Tabela 1 – Vereditos comuns de um juiz eletrônico 25
Tabela 2 – Linguagens comuns aceitas em um juiz eletrônico 26
Tabela 3 – Especificação das máquinas utilizadas 33
Tabela 4 – Comandos no banco de dados do juiz eletrônico 36

Lista de abreviaturas e siglas

AC	*Accepted*
WA	*Wrong Answer*
TLE	*Time Limit Exceeded*
RE	*Runtime Error*
CE	*Compilation Error*
PE	*Presentation Error*
ACM	*Association for Computing Machinery*
ICPC	*International Collegiate Programming Contest*
SBC	Sociedade Brasileira de Computação
TLS	*Transport Layer Security*
SSL	*Secure Sockets Layer*
TCP	*Transmission Control Protocol*
UDP	*User Datagram Protocol*
JSON	*JavaScript Object Notation*
AMQP	*Advanced Message Queueing Protocol*
MOM	*Message Oriented Middleware*
UTC	*Universal Time Coordinated*

Sumário

Introdução 23

1 FUNDAMENTAÇÃO TEÓRICA 25
1.1 Juiz Eletrônico 25
1.2 Competição de Programação 26
1.2.1 Juízes Online 26
1.3 Conceitos Elementares de Redes de Computadores 27
1.3.1 Camada de Transporte 27
1.3.2 Comunicação Síncrona e Assíncrona 28
1.4 *Advanced Message Queueing Protocol* 29
1.5 Virtualização baseada em *containers* 29
1.5.1 Docker 30
1.6 Engenharia de software 30
1.6.1 *Extreme programming* 30
1.6.2 *CRUD* 31

2 METODOLOGIA 33
2.1 Ferramentas 33
2.2 Metodologia de Desenvolvimento 33
2.3 Arquitetura 34
2.3.1 Juiz eletrônico 34
2.3.1.1 Cliente 35
2.3.1.2 Script de Administração 36
2.3.1.3 Servidor de Correção 36
2.3.1.4 Courier 37
2.3.2 Juiz online 37
2.3.2.1 Problemas 37
2.3.2.2 Contests 38
2.3.2.3 Submissões 38
2.3.2.4 Usuários 38

3 RESULTADOS 39
3.1 Juiz Eletrônico 40
3.1.1 Troca de mensagens 41
3.1.2 Isolamento da correção 41
3.1.2.1 SELinux 42

3.1.2.2 AppArmor 42
3.1.3 Banco de dados 42
3.2 Juiz Online 42
3.3 Comunicação interface-correção 46

4 CONSIDERAÇÕES FINAIS 47

REFERÊNCIAS 49

APÊNDICES 53

APÊNDICE A – CONFIGURAÇÃO DO AMBIENTE DO BROJ . . 55
A.1 Juiz eletrônico 55
A.1.1 Requisitos 55
A.1.2 Configuração 55
A.1.3 Execução 56
A.2 Juiz online 56
A.2.1 Requisitos 56
A.2.2 Configuração 56
A.2.3 Execução 56

Introdução

Contextualização

Para cada exercício dado a um estudante, existe um gabarito. Segundo Sadler (2005), “estudantes merecem ser classificados apenas com base na qualidade de seu trabalho, sem influência do desempenho de outros estudantes do curso nas mesmas tarefas ou não, e sem considerar o nível anterior de desempenho de cada estudante”. A decisão de quando e como avaliar o desempenho de um estudante é extremamente relevante no que tange o desenvolvimento do conhecimento dele (CASE; SWANSON, 1998).

Conforme o número de exercícios e de estudantes cresce, a correção manual destes exercícios torna-se cada vez mais difícil, sendo propensa a erros, especialmente em um contexto de programação (CHEANG et al., 2003). Segundo Eckerdal, Thuné e Berglund (2005), “aprender programação é um jeito de pensar, que permite *problem solving*, e é experimentado como um ‘método’ de pensamento”. Esse aprendizado vem através de vários exercícios de programação, desenvolvendo raciocínio lógico e familiarizando conceitos de computação (NORONHA et al., 2015). Uma forma de automatizar a correção de exercícios beneficia tanto quem os realiza quanto quem os elaboram e corrigem.

Algumas plataformas tentam suprir o problema de correção automática, como a Pex4Fun, um ambiente de *serious gaming* (jogos educacionais) lançado pela *Microsoft Research* para ensinar ciência da computação em escala e o POJ (*Peking University Online Judge*). Desenvolvido pelo laboratório de inteligência artificial da universidade de Peking, o POJ foi inicialmente desenvolvido para ser uma plataforma de treino para competições ACM *International Collegiate Programming Contest (ACM/ICPC)* (WEN-XIN; WEI, 2005).

O juiz *online* é uma alternativa para a correção automática. Ele conhece os problemas propostos aos alunos. Quando um estudante submete o código de sua solução o juiz checa a corretude do programa submetido. Este juiz deve ser seguro o suficiente de modo que códigos maliciosos não o afetem e sejam descartados. Um programa só deve ser considerado certo se gerar a resposta correta para um conjunto de entradas dentro do tempo limite. O conjunto de entradas precisa ser secreto para que o código produzido não se aproveite do conhecimento dos casos de teste (KURNIA; LIM; CHEANG, 2001).

O envolvimento dos alunos com sistemas de juízes *online* melhora o desempenho destes ao longo do curso de computação em relação aos que não tiveram tal experiência, inferindo-se que os fundamentos necessários para o aprendizado de conceitos mais avançados tornaram-se melhor assimilados por estes alunos (PEREIRA, 2015).

Para que tais sistemas de juízes *online* sejam efetivamente utilizados no ambiente acadêmico eles não podem apenas automaticamente corrigir os problemas, eles precisam ser seguros e tolerantes à falhas (KURNIA; LIM; CHEANG, 2001). Essas questões podem ser vistas como atendidas, se o sistema tem dependabilidade, ou seja, ser capaz de atender as especificações durante todo o período de funcionamento, estar operacional e executar suas funções corretamente de forma íntegra (WEBER, 2003).

Objetivos

O objetivo geral deste trabalho é implementar um juiz eletrônico capaz de julgar exercícios de programação, com capacidade para altas cargas e tolerante à falhas, e um juiz online capaz de administrar uma prova.

Os objetivos específicos são:

- Especificar arquitetura de um juiz eletrônico e um juiz *online* desacoplados;
- Implementar uma aplicação de teste para a validação do uso de um juiz eletrônico em um contexto de programação;
- Implementar a correção automática em um ambiente seguro.

É importante que os juízes sejam desacoplados por dois motivos: facilitar a entrada de colaboradores da comunidade, dado que o possível colaborador pode se concentrar na sua área de preferência, *front end* (juiz online) ou *back end* (juiz eletrônico), e principalmente, para facilitar o uso de múltiplos nós de servidores de correção e tentar garantir a disponibilidade, requisito prioritário nesse trabalho.

Organização do Trabalho

Este trabalho está organizado em capítulos. No Capítulo 1 encontra-se o referencial teórico, com os conceitos abordados neste trabalho, como o juiz eletrônico, e fundamentos da computação. No Capítulo 2 encontra-se o desenvolvimento deste trabalho, descrevendo os passos metodológicos utilizados. No Capítulo 3 encontra-se um relato dos resultados, e no Capítulo 4 estão as considerações finais sobre este estudo e o futuro deste trabalho.

1 Fundamentação Teórica

Este capítulo expõe os conceitos necessários para o entendimento do resto do trabalho e passa de conceitos de programação competitiva aos fundamentos computacionais requeridos na implementação do BROJ (*Brazilian Online Judge*).

1.1 Juiz Eletrônico

Juízes eletrônicos são sistemas automatizados que compilam, executam e testam os códigos-fonte baseados em um conjunto de entradas e saídas pré-determinadas (ZHIGANG et al., 2001). Esse processo de correção automática acontece da seguinte maneira: o juiz executa o código recebido e alimenta-o com as entradas do problema; o programa resultante do código processa esses dados e responde a saída da questão; por fim o juiz compara a saída do programa com a saída correta e emite um veredito (CHAVES et al., 2013).

Para o juiz uma questão pode ser vista como um conjunto de entradas e o que este conjunto significa, como o programa processa as entradas e qual deve ser o conjunto de saídas. A validação da entrada geralmente não é o foco das questões, já que se pode assumir que a entrada segue as especificações da questão corretamente (KURNIA; LIM; CHEANG, 2001).

Os juízes eletrônicos têm vereditos divididos em duas categorias: a de acerto, geralmente representados por AC, YES ou OK, e as de erro, que tenta dar alguma informação a respeito do que o usuário está errando. Os vereditos mais comuns para um juiz eletrônico são: AC, WA, TLE, RE, PE e CE (SKIENA; REVILLA, 2006) conforme apresentado na Tabela 1.

Tabela 1: Vereditos comuns de um juiz eletrônico

Sigla	**Significado**	**Descrição**
AC	*Accepted*	Solução correta
WA	*Wrong Answer*	Difere do gabarito
TLE	*Time Limit Exceed*	Não responde dentro do tempo limite
RE	*Runtime Error*	Erro durante a execução
PE	*Presentation Error*	Espaços ou quebra de linha sobrando ou faltando
CE	*Compilation Error*	Erro de compilação

As linguagens mais utilizadas nos juízes eletrônicos são as adotadas pela ACM/ICPC (*Association for Computing Machinery – International Collegiate Programming Contest*) as quais estão listadas na Tabela 2.

Tabela 2: Linguagens comuns aceitas em um juiz eletrônico

Linguagem	**Versão**	**Compilador/Interpretador**
Java	8	*OpenJDK* 1.8.091
C	11	*GCC* 5.4.0
C++	11	*G++* 5.4.0
Python	2.7 ou 3	*PyPy 5.1.2*

1.2 Competição de Programação

Uma frase que define bem uma competição de programação é "Dados problemas conhecidos de Ciência da Computação, resolva-os o mais rápido possível" (HALIM et al., 2010). A maior e mais antiga competição de programação é a ACM-ICPC sendo dividida em várias fases eliminatórias até a Final Mundial. Esta competição chegou a envolver, em 2017, 46.381 estudantes de 2.948 universidades vindo de 103 países diferentes de seis continentes, quebrando o recorde da competição. A competição promove a criatividade, o trabalho em equipe e a inovação, fazendo com que alunos de graduação testem suas habilidades trabalhando sobre pressão em um curto espaço de tempo (ACM-ICPC, 2016).

No Brasil a Maratona de Programação é um evento organizado pela SBC desde 1996, e que faz parte da ACM-ICPC. É uma das competições regionais que classificam as melhores equipes brasileiras para a Final Mundial. No caso do Brasil, antes da fase regional existem as sedes sub-regionais, que são organizadas pelos estados para classificar os melhores times para a etapa regional.

A dificuldade dos problemas varia, não só de acordo com a fase da prova, desde o sub-regional até a fase mundial, mas em cada prova existem problemas facilmente resolvíveis até problemas complexos. Pelas regras da ICPC, os times são classificados pela quantidade de problemas resolvidos, não existindo pesos diferentes entre os problemas. O desempate é feito através de penalidades de tempo por submissão (FERRASA; SOUZA, 2012).

A quantidade de código produzido em uma competição de programação, na maioria dos casos, torna inviável a correção manual, fazendo com que os juízes eletrônicos tenham um papel fundamental nesse tipo de evento.

1.2.1 Juízes Online

Juízes *online* são sites que fazem uma interface para um banco de questões e um juiz eletrônico. Além disso, alguns deles mantém fóruns e comunidades que promovem discussões acerca das questões e competições regulares. A seguir estão alguns dos mais populares juízes online.

Codeforces é um *site* russo dedicado à programação competitiva. Foi criado e é

mantido por um grupo da *Saratov State University* liderado por Mikhail Mirzayanov. Desde 2013 o Codeforces é a maior plataforma em termos de usuário ativos. No dia 29 de agosto de 2016 o *Codeforces Round #369 Div*.2 quebrou o recorde do *site* com 8.670 usuários registrados em um mesmo *contest* (CODEFORCES, 2016).

Um dos maiores bancos de questões do mundo, a plataforma UVa *Online Judge*, mantida pela *Universidad de Valladolid*, na Espanha, categoriza os problemas, disponibiliza *rankings* dos usuários a partir da quantidade de exercícios resolvidos entre outras funcionalidades. É possível criar *contests* rapidamente através dos identificadores das questões (UVA, 2016). O uHunt é uma ferramenta desenvolvida a partir da base de dados do UVa para acompanhar estatísticas dos seus problemas resolvidos e encontrar novos problemas (UHUNT, 2016).

O URI *Online Judge* é um juiz online brasileiro desenvolvido e mantido pelo departamento de Ciência da Computação da universidade URI e tem como objetivo prover práticas de programação e compartilhamento de conhecimento entre seus usuários. O URI conta com mais de 1.500 problemas divididos entre 8 categorias disponíveis em inglês e português. O URI Acadêmico é um módulo dedicado aos técnicos de times e professores, onde é possível criar disciplinas e listas de exercícios (URI, 2017).

Esses juízes *online* têm seus códigos licenciados para o direito exclusivo do produtor, dificultando não só o uso no ambiente acadêmico, como a realização de modificações, para se adequar às necessidades do usuário.

1.3 Conceitos Elementares de Redes de Computadores

Uma parte fundamental no BROJ é a comunicação de dados. A comunicação de dados, no contexto da computação, é a troca de dados entre dois dispositivos através de algum meio de transmissão. Para que ela ocorra é necessário que os dispositivos sejam parte de um sistema de comunicação (FOROUZAN, 2006). O maior exemplo de uma rede de computadores é a Internet.

Nas subseções seguintes serão descritos alguns tópicos relevantes sobre redes de computadores.

1.3.1 Camada de Transporte

Para organizar arquiteturalmente os protocolos de rede existe uma separação por camadas (KUROSE; ROSS, 2006), que pode ser visto na Figura 1.

Entre a camada de aplicação e a de rede, a camada de transporte tem um papel muito importante na arquitetura de redes, que é prover comunicação entre dois processos rodando em *hosts* diferentes (KUROSE; ROSS, 2006). Alguns dos protocolos da camada

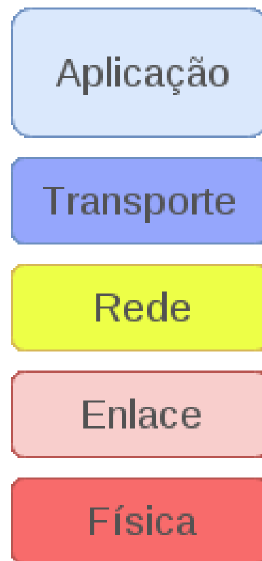


Figura 1 – A pilha de protocolos da Internet

de transporte, como o TCP (*Transmission Control Protocol*), tentam garantir que a informação vai ser trocada entre os processos de forma correta, já outros como o UDP (*User Datagram Protocol*) são protocolos mais simples que abrem mão de controle para terem um desempenho maior (SAWASHIMA et al., 1997).

1.3.2 Comunicação Síncrona e Assíncrona

A diferenciação entre a comunicação síncrona e assíncrona é feita em relação ao tratamento do tempo. Na comunicação síncrona as mensagens respeitam uma fila onde cada mensagem só é disparada após a anterior ter sido recebida, já na comunicação assíncrona a mensagem é enviada sem nenhuma espera como pode ser visto na Figura 2.

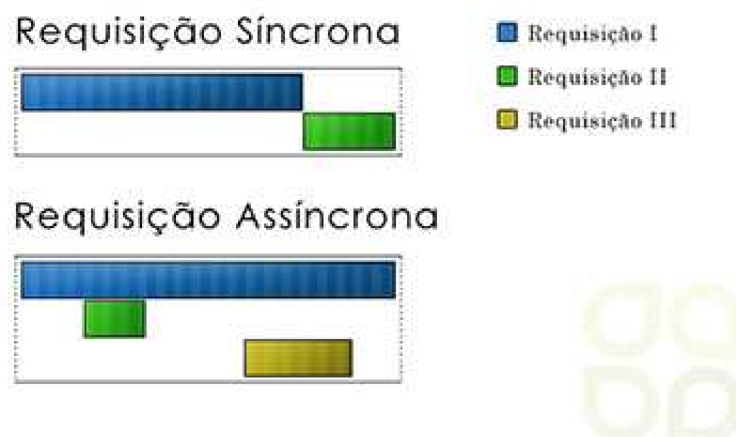


Figura 2 – Requisições síncronas e assíncronas

Na comunicação síncrona é garantido que toda mensagem chega no seu destino em um tempo pré-definido. Já na comunicação assíncrona a comunicação é incerta, se a mensagem não chega ao seu destino não é possível determinar o motivo desta falha: se a comunicação está lenta, o receptor não está disponível, entre outros. Isto torna necessário na comunicação assíncrona mecanismos de réplicas de mensagens e detecção de falhas (CRISTIAN, 1996).

A comunicação assíncrona tem a vantagem da liberdade na trocas de mensagens que ficam independentes entre si, fazendo com que uma mensagem que não pode ser processada ou demore muito tempo para tal, não congestione o fluxo e interrompa o envio das próximas mensagens. Porém, para que isso seja possível, as mensagens precisam ser independentes. No caso em que uma mensagem precisa que uma outra anterior já tenha sido processada, o uso da comunicação síncrona é o ideal.

1.4 *Advanced Message Queueing Protocol*

O AMQP (*Advanced Message Queueing Protocol*) é um protocolo que tem como objetivo tornar-se o padrão de interoperabilidade entre todos os MOM (*Message Oriented Middleware*) (VINOSKI, 2006). A comunicação do AMQP ocorre através de um protocolo de transporte *peer-to-peer* sobre o TCP (ISO/IEC, 2014).

O diferencial do AMQP é que ele permite a especificação das mensagens a serem recebidas, e de parâmetros que controlam um *trade-off* entre segurança, desempenho e confiabilidade (FERNANDES et al., 2013). Além disso, é um protocolo que foi especificado para ser independente de plataformas, facilitando a comunicação entre qualquer aplicação capaz de usar uma implementação do AMQP (ISO/IEC, 2014).

1.5 Virtualização baseada em *containers*

A virtualização baseada em *containers* compartilha recursos físicos do computador, sem o custo de operar através de um sistema operacional *guest* (convidado) emulado (XAVIER et al., 2013). Isto é evidenciado na Figura 3. Por isso, é possível executar dezenas de *containers* a mais em um computador *host* (hospedeiro), comparados com máquinas virtuais comuns (FINK, 2014).

Como aponta Seo et al. (2014), sistemas de virtualização por *containers* em relação à máquinas virtuais ocupam menos espaço físico em disco rígido, inicializam seu sistema em torno de seis vezes mais rápido, e realizam operações computacionais de forma mais eficiente. Entretanto, máquinas virtuais são mais fáceis de administrar quando é necessário gerenciar individualmente políticas de redes, segurança e usuários.

1.5.1 Docker

Docker é um *container* que extende a habilidade de construir e executar aplicações de forma rápida e fácil para o usuário. É usado em aplicações populares como Spotify, Yelp e Ebay (DOCKER, 2017).

Ele usa as vantagens de duas funcionalidades do kernel Linux, *namespaces* e *cgroup*, para criar um ambiente virtual seguro para seus *containers*. Os *cgroup* provem um mecanismo para limitar os recursos do processo em cada componente e os *namespaces* envolvem o sistema operacional em instâncias diferentes dos recursos do sistema operacional (BUI, 2015).

Figura 3 – Virtualização baseada em *containers*

1.6 Engenharia de software

Quando se trata de projetos de software, muitos dos problemas encontrados já foram previamente discutidos. Nas subseções seguintes serão descritos dois tópicos relevantes sobre Engenharia de Software.

1.6.1 *Extreme programming*

Extreme programming ou XP é uma disciplina de desenvolvimento de software desenhada para projetos pequenos (2 à 10 participantes) que tenta resolver alguns dos maiores problemas do desenvolvimento de um projeto de software através de uma série de técnicas como ciclos de produção reduzidos e programação em pares (BECK, 2000).

Pair programming (programação em pares) é uma técnica datada de 1996 como parte das práticas do *Extreme programming*, onde dois programadores em conjunto produzem um artefato (projeto, algoritmo, código, etc.). Os dois programadores funcionam como um organismo sendo um o “piloto”, o que tem o controle da atividade e está escrevendo o projeto ou código. A outra pessoa está procurando por melhores alternativas

ou defeitos no que o piloto fez. Os papéis são trocados periodicamente sendo os dois igualmente ativos durante a atividade (WILLIAMS et al., 2000).

1.6.2 *CRUD*

CRUD - *Create, Read, Update and Delete* (criar, ler, atualizar e apagar) são as quatro operações básicas em uma entidade em um banco de dados. Alguns *frameworks* para criação de *websites* possuem uma forma simples de realizar essas operações, além de controlar as permissões dado que muitas vezes é interessante que nem todos os usuários consigam realizar todas as quatro operações.

2 Metodologia

Este capítulo exibe o desenvolvimento deste trabalho, descrevendo os passos metodológicos utilizados.

2.1 Ferramentas

Para a implementação do juiz eletrônico, a linguagem escolhida foi o Python pelo desempenho, pelas bibliotecas e ferramentas da linguagem (como o *pika*[1] e o *pony*[2]). A versão do interpretador utilizada é a 3.6 ou superior.

Para o juiz *online*, foi utilizado o *framework* Rails, na versão 5, com a linguagem Ruby, na versão 2.3, pela velocidade no desenvolvimento dado o conhecimento da equipe. Para o *front-end* foi utilizado o *framework* Bootstrap, na versão 4.

O desenvolvimento aconteceu em distribuições *debian-based* do Linux, Linux Mint 17 e Debian 8. As máquinas usadas têm suas especificações listadas na Tabela 3.

O sistema de versionamento utilizado foi o git [3] , e como *forge* foram utilizados o GitHub e o GitLab, com repositórios duplicados nas duas plataformas.

Tabela 3: Especificação das máquinas utilizadas

Máquina	**CPU**	**Disco**	**Memória RAM**
I	Intel Core i5-2500K @ 3.30 GHz	250 GB (SSD) & 1 TB (HDD)	8 GB
II	Intel Core i7-2617M @ 1.50 GHz	500 GB (HDD)	8 GB
III	Intel Core i7-3610QM @ 2.30 GHz	500 GB (HDD)	8 GB

2.2 Metodologia de Desenvolvimento

Um planejamento inicial sobre os requisitos e arquitetura básicos do sistema foi realizado entre os dias 2 e 14 de setembro de 2016, e os principais requisitos podem ser

[1] Pika é uma implementação em Python do protocolo AMQP 0-9-1 que mantém independência da biblioteca de rede utilizada.
[2] Pony é um mapeador objeto-relacional em Python fácil de usar que traduz consultas para SQL.
[3] Git é um sistema de controle de versão distribuído e um sistema de gerenciamento de código fonte, com ênfase em velocidade.

vistos a seguir:

- Um servidor de correção que possa ser replicado, *multi-threaded*, e que receba requisições via rede;
- O usuário deve ser capaz de consumir o serviço de correção através de um cliente;
- Um juiz online básico precisa manter os problemas e fazer submissões para o teste do juiz eletrônico
- No caso de indisponibilidade de um dos componentes do juiz eletrônico (servidor, cliente, *courier*) o serviço deve poder ser restaurado.
- O ambiente de correção deve ser seguro, de forma que um código malicioso não afete o funcionamento do servidor

Pelo fato de a equipe ser bem pequena, contando com apenas dois desenvolvedores, duas principais estratégias foram utilizadas. A primeira foi ter um desenvolvedor resolvendo uma *issue*, e o outro revisando. Outra era desenvolver através do *pair programming*. Aproximadamente uma vez por semana era conduzida uma reunião com o orientador para definir os caminhos a serem seguidos.

O projeto foi organizado de forma à integrar um *framework* de testes facilmente. O *framework* de testes unitários *PyUnit* foi usado como prova de conceito e na entrega desse trabalho apresenta 53% de cobertura de código.

Também foi utilizado para integração contínua o Travis CI [4] que automaticamente roda todos os testes e verifica a *build* a cada *commit* no repositório.

2.3 Arquitetura

Nessa seção serão apresentados os subsistemas do BROJ e suas respectivas responsabilidades. Serão discutidas as decisões arquiteturais e o funcionamento do sistema.

2.3.1 Juiz eletrônico

O sistema do juiz eletrônico foi dividido em quatro componentes: cliente, juiz (servidor de correção), um *script* de administração e um *courier*.

A Figura 4 apresenta a visão lógica simplificada da arquitetura do sistema de modo geral. O usuário, que pode tanto ser alguém manualmente corrigindo um problema ou um juiz online, acessa um cliente que se comunica diretamente com o juiz eletrônico através

[4] Travis CI é uma ferramenta de integração contínua que roda um script de verificação a cada modificação no repositório.

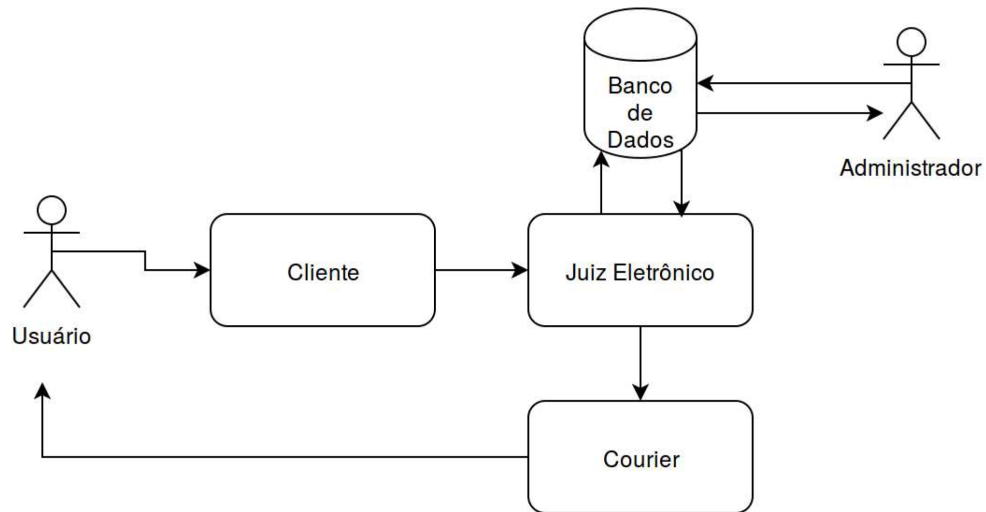


Figura 4 – Visão lógica arquitetural

da comutação de pacotes. O juiz corrige o problema baseado nos problemas cadastrados e retorna um veredito. O *courier*, que está sempre observando as atualizações do juiz, captura o veredito e o envia ao usuário.

Tanto é possível que os juízes sejam replicados (Figura 5) quanto múltiplos clientes podem requisitar ao mesmo juiz (Figura 6), existindo N cópias distribuídas. O *courier* pode estar observando a resposta de vereditos de uma lista de juízes.

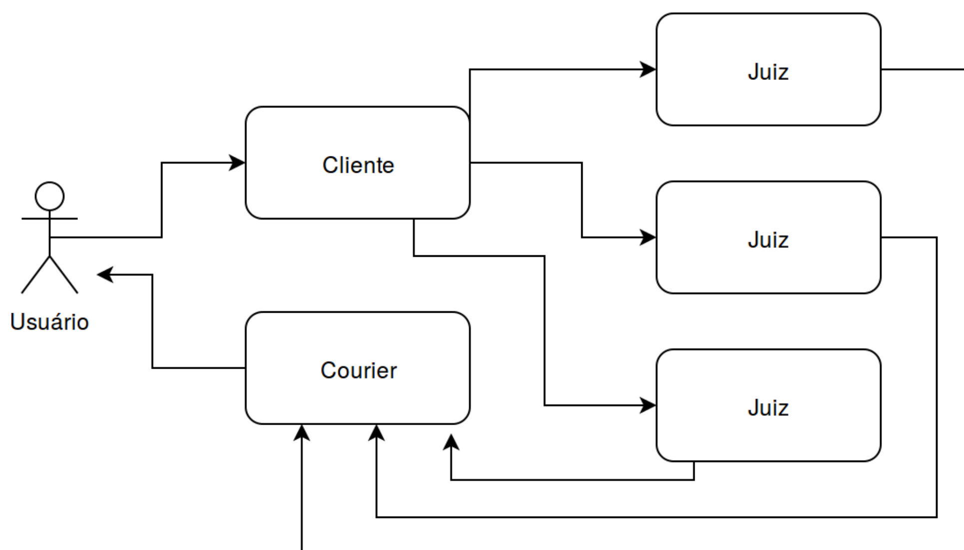


Figura 5 – Visão de múltiplos juízes eletrônicos

2.3.1.1 Cliente

O cliente é o subsistema que provê uma interface para o usuário para que o mesmo possa utilizar os serviços disponibilizados.

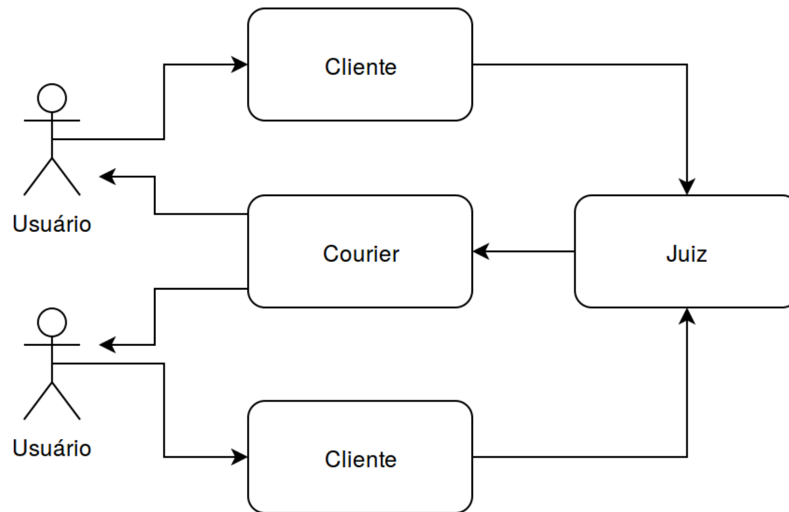


Figura 6 – Visão de múltiplos clientes

Ele é capaz de se comunicar com o juiz eletrônico requisitando correções. Para tal ele deve ser capaz de abrir uma conexão com o juiz, localizar e transmitir o código do usuário, além de identificar o usuário e o problema a ser corrigido.

Para simplificar o uso de múltiplos juízes o cliente deve ser capaz de especificar o endereço do juiz no qual ele quer destinar dada submissão. Dessa forma desacoplada, cada chamada de correção no cliente pode ser destinada a um juiz diferente.

2.3.1.2 Script de Administração

O *script* de administração é responsável pela interface de CRUD (criar, ler, atualizar e deletar) dos problemas no banco de dados do juiz. Os comandos que podem ser executados através deles se encontram na Tabela 4.

Tabela 4: Comandos no banco de dados do juiz eletrônico

Comando	**Descrição**
create	Cadastra um exercício
update	Atualiza um exercício
remove	Remove um exercício
read	Lista os exercícios

Assim como o cliente, a administração deve ser desacoplada do juiz, sendo possível ser usado em qualquer um dos N juízes. Esta flexibilidade permite o uso deste *script* como ferramenta de sincronização entre os juízes.

2.3.1.3 Servidor de Correção

Os servidores de correção são onde os exercícios são de fato corrigidos, comparando a entrada recebida com a saída desejada.

O servidor de correção conta com uma fila de requisições a serem corrigidas e um banco de dados com as entradas e saídas corretas para os problemas. Quando vai julgar o próximo problema da fila ele copia o código para um arquivo temporário, compila o código (se for o caso), executa e resgata possíveis exceções de *runtime* ou erro no juiz (RTE, JE). No caso em que o código não resgata nenhuma exceção e executa dentro do tempo estabelecido, evitando um veredito de TLE, o juiz responde de acordo com o gabarito (AC, WA ou PE).

O funcionamento do servidor de correção é assíncrono de forma que um problema que foi recebido primeiro pode não ser o primeiro a ter seu veredito, o que é muito importante pois alguns problemas podem levar minutos para serem corrigidos enquanto outros apenas poucos segundos.

A compilação e execução devem levar em conta a confiabilidade do código executado. No caso de um código malicioso o servidor deve ser capaz de manter sua execução sem erros ou interrupção. Isto pode ser alcançado através de políticas de segurança e de usuários, e do isolamento da execução do programa do usuário.

2.3.1.4 Courier

O *Courier* é encarregado de observar os vereditos que são emitidos pelo juiz. Ele recebe um ou mais endereços e registra os vereditos desses juízes numa fila, para a partir dela ser capaz de responder o status de dada submissão.

Ele concentra as respostas das submissões de vários nós (juízes) o que facilita para o controle das submissões antes de enviar para o usuário. Isto permite sua substituição em caso de problemas, em tempo real, sem prejudicar o andamento de uma avaliação.

2.3.2 Juiz online

O juiz online é o responsável por ser a interface para o usuário dos problemas, provas, submissões além dos próprios usuários.

2.3.2.1 Problemas

Deve ser possível através da interface do juiz *online* criar, editar e ler problemas, além de submeter soluções.

Na submissão do problema o juiz online é livre para decidir como o código será corrigido, desacoplado do juiz eletrônico anteriormente especificado. Caso nosso juiz eletrônico seja utilizado o juiz online precisa de uma instância do cliente instalada na mesma máquina e o conhecimento do endereço do servidor de correção.

Os problemas de avaliações passadas formam um banco de problemas que podem ser resolvidos posteriormente, contribuindo para o treinamento e aprendizado do usuário.

2.3.2.2 Contests

Contests são formados por um subconjunto dos problemas previamente cadastrados, e fornecem ao usuário uma interface para acompanhar todos esses problemas, o *scoreboard* (placar), estatísticas na resolução dos problemas e clarificações acerca dos problemas ou do *contest*.

Este é o foco principal do juiz online. Dado que as competições de programação acontecem em forma de um *contest*.

2.3.2.3 Submissões

O juiz online recebe as submissões através de uma página de formulário, onde o usuário envia o código para determinada questão.

Também é possível que o usuário conheça o veredito de suas submissões.

2.3.2.4 Usuários

Os usuários tem que ser mantidos pelo juiz online, assim como os seus dados, problemas resolvidos, submissões e seus vereditos e *contests* no qual ele está participando e já participou.

É interessante que o usuário esteja restrito de algumas funcionalidades atreladas ao seu perfil até que faça *login* no sistema, como realizar novas submissões, assim como verificar suas submissões antigas ou se inscrever e participar de *contests*.

Também é importante a diferenciação de papéis, onde o administrador tem mais privilégios que o usuário comum, podendo criar e editar problemas e *contests*.

3 Resultados

Com o desenvolvimento desse trabalho até o presente momento, é possível o uso do BROJ para a aplicação completa de um *contest*, desde o cadastro de problemas, cadastro do *contest*, até a sua própria execução. Os seguintes resultados foram obtidos:

- implementação de um juiz online como prova de conceito;
- especificação de uma arquitetura para um juiz eletrônico escalável e seguro;
- implementação de um juiz eletrônico e um juiz online livres (licença GPLV3+ (GNU, 2007));

Baseada nas propostas de arquitetura especificadas no Capítulo 2, nesse capítulo iremos apresentar a implementação do juiz eletrônico, do juiz online e a comunicação entre eles.

As funcionalidades desenvolvidas até esse momento no BROJ são:

1. Servidor de correção assíncrono;
2. Servidor de correção com correção isolada em *container*;
3. Suporte à correção de código em C++ e Python;
4. Correção customizada via *script*;
5. Cliente para submissão de problemas;
6. Administração da base de problemas do servidor de correção via *script*;
7. Gerenciador de submissões centralizado (*courier*);
8. Juiz online responsivo (com *bootstrap*);
9. Login e permissão baseada em perfis;
10. Cadastro de problemas e *contests* via interface;
11. Leitura e submissão de problemas via interface;
12. Registro individual de usuários em *contests*;
13. Histórico de submissões;
14. Horário do *contest* localizado através do timeanddate.com;

3.1 Juiz Eletrônico

O juiz eletrônico desenvolvido é capaz de receber um código referente a uma questão e responder se este código resolve ou não a questão corretamente. Por mais simples que esse processo pareça, algumas complicações surgem na implementação quando se leva em conta o contexto de competições de programação. Problemas como: ser capaz de responder múltiplas requisições, garantir que o código não irá se aproveitar de vulnerabilidades existentes no ambiente no qual o juiz é executado, e continuar funcionando mesmo que um dos servidores de correção se torne indisponível, serão tratados nessa seção.

A implementação do juiz eletrônico foi baseada na troca de mensagens através do RabbitMQ e virtualização em *containers* do Docker. O RabbitMQ faz a comunicação via conexão TCP e abstrai a troca de mensagens, tratando exceções, controlando filas de mensagens, entre outros, além de aumentar a velocidade do desenvolvimento. Os *containers* Docker são razoavelmente seguros: mesmo com a configuração padrão, o nível de segurança ainda pode ser aumentado executando os processos docker “sem privilégios”e habilitando funcionalidades do kernel Linux, como AppArmor e SELinux (BUI, 2015), práticas adotadas nesse trabalho.

Os vereditos do juiz são determinados a partir do Código 3.1:

```
1  try:
       test_cases = problem.test_cases
3      if len(test_cases) < 1:
           logger.error('Problem without test cases on RUN CODE')
5          return Verdict.JE

7      args = []
       if(language in consts.runners):
9          args.append(consts.runners[language])
       args.append(executable)
11     for test_case in test_cases:
           output = subprocess.check_output(args=args,
13                                          timeout=problem.time_limit,
                                            encoding='utf-8',
15                                          input=test_case.input_)
           if problem.check_code:
17             exec(problem.check_code, globals())
               try:
19                 if not check(test_case.input_,
                               test_case.output, output):
21                     return Verdict.WA
               except:
23                 return Verdict.JE
           elif not equal_test_cases(output, test_case.output):
25             logger.info((f'Output [{output!r}] did not match'
```

```
                                f'[{test_case.output!r}]'))
27
                if equal_test_cases(output, test_case.output,
29                                  normalize=True):
                    return Verdict.PE
31
                return Verdict.WA
33
except subprocess.TimeoutExpired as tle:
35      logger.info(tle)
        return Verdict.TLE
37  except subprocess.CalledProcessError as rte:
        logger.info(rte)
39      return Verdict.RTE
except FileNotFoundError as fnfe:
41      logger.error(fnfe)
        return Verdict.JE
43
return Verdict.AC
```

Código 3.1 – Vereditos do juiz

3.1.1 Troca de mensagens

Um ponto de conexão é definido neste trabalho como uma fila de mensagens para a qual podem ser enviadas mensagens, e da qual mensagens podem ser consumidas. Para o juiz, existem dois pontos de conexão: o *JudgeConnection* e o *CourierConnection*.

O *JudgeConnection* é o ponto para o qual o cliente envia as submissões de questões para serem corrigidas, e do qual o servidor de correção consome tais mensagens. O *CourierConnection* é o ponto para o qual o servidor de correção manda o resultado de uma correção de uma submissão, e do qual o cliente recebe tal mensagem.

Ambos os pontos de conexão foram implementados com o RabbitMQ usando o Pika, uma implementação em Python do protocolo AMQP. As filas e as mensagens foram configuradas para serem duráveis e persistentes, respectivamente. Estas configurações visam garantir que as filas e mensagens não sejam esquecidas ou excluídas quando o RabbitMQ fechar.

3.1.2 Isolamento da correção

No juiz eletrônico o processo de compilação do código recebido, assim como o da execução, são isolados do *host* através de *containers* Docker, cada um com seu *container*. Os *containers* pré-configurados a partir de imagens são instanciados no momento da correção.

O Docker foi escolhido pela facilidade de isolamento dos processos e pela performance superior em relação à máquinas virtuais especialmente em uso extensivo de *Input/Output* (FELTER et al., 2015).

3.1.2.1 SELinux

O kernel do Linux administra permissões por usuário, enquanto no SELinux o controle é feito por objeto, atribuindo tipos (*labels*) aos objetos, onde cada tipo só pode se relacionar com objetos do próprio tipo. Sendo assim, se o atacante ganha controle de um usuário ele ainda precisa ganhar controle dos objetos que ele quer atacar. Usando o SELinux, os *containers* podem separar os objetos em duas categorias: dentro e fora do *container*.

O Docker utiliza o SELinux de duas formas. Na primeira ele atribui a todos os processos do *container* o tipo *svirt_lxc_net_t* e todo o conteúdo do *container* com o tipo *svirt_sandbox_file_t*, a partir daí os processos do tipo *svirt_lxc_net_t* só tem permissão para ler/escrever em objetos do tipo *svirt_sandbox_file_t*. Na segunda ele escolhe um tipo aleatório para todos os processos e conteúdos do *container*, de forma que processos de diferentes *containers* não conseguem comprometer um ao outro.

3.1.2.2 AppArmor

AppArmor trata do escopo de programas individualmente através de perfis que limitam suas capacidades. O Docker tem uma interface para carregar perfis pré-definidos quando um novo *container* vai ser iniciado. Se nenhum perfil for especificado ele carrega um perfil padrão que protege o acesso a arquivos importantes, como o $/sys/fs/cgroups/$ e o $/sys/kernel/security/$.

3.1.3 Banco de dados

O servidor de correção mantém uma base de problemas que ele é capaz de corrigir. Como pode ser observado na Figura 7, o problema tem um título, tempo e memória limite e opcionalmente um código de correção, além de um conjunto de N casos de testes. Um caso de teste é formado por um par *input/output*.

3.2 Juiz Online

O juiz online é a interface para o usuário final ler e submeter problemas, dentro ou fora de um *contest*. Nesse trabalho, o juiz online é completamente desacoplado do juiz eletrônico, como já foi apontado. Entretanto, nessa seção serão explicados o funcionamento do juiz *online*, assim como a comunicação entre os dois juízes.

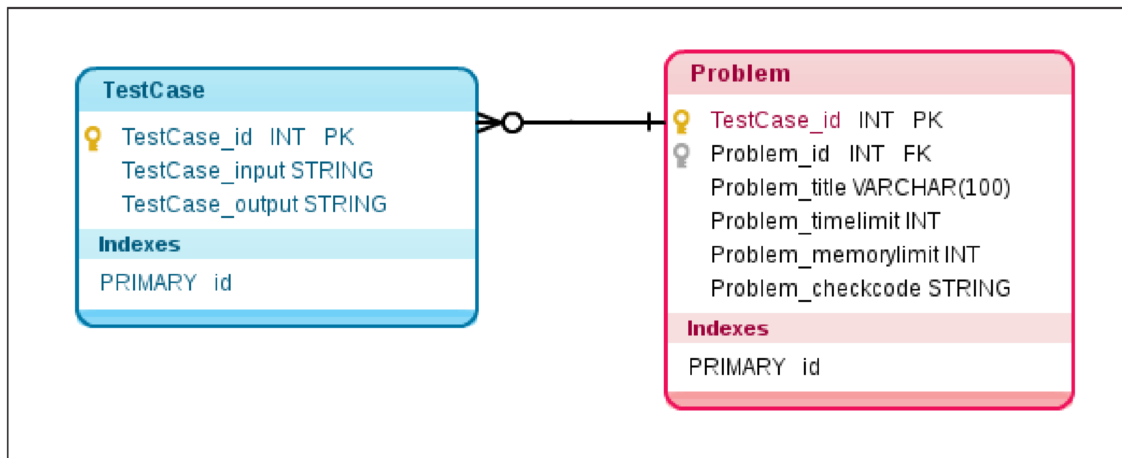


Figura 7 – MER do juíz eletrônico

Nesse trabalho o juiz *online* não foi o foco da implementação, resultando em uma interface ainda primitiva e com muitos aspectos a serem aprimorados.

O principal trabalho do juiz *online* é manter *contests*. Como resultado desse trabalho, temos o possível cadastro, edição, registro e participação em um *contest*. A interface pode ser vista na Figura 8.

[BROJ] PROBLEMS CONTESTS SUBMISSIONS Log out

Problems Standings

#	Name	Time Limit	Memory Limit	Status
A	String de Dígitos	1s	512mb	Not Tried
B	Defendendo Alamo	32s	256mb	Not Tried
C	Dia da Vovo	3s	256mb	Not Tried

Contest 1

Start: 2017-07-04 22:44:00 UTC

Duration: 300 minutes

Statistics

A: 1/2

B: 0/1

Announcements

Announcements fake 1

Figura 8 – Tela de um contest no BROJ

Como o *contest* é composto de problemas, que devem ser lidos e submetidos, é preciso uma página para apresentação dos problemas: esta é apresentada na Figura 9. Os problemas continuam disponíveis para os usuários mesmo após o *contest*, como mostra a Figura 10, para que seja possível o estudo dos problemas mesmo após o término do *contest*.

Figura 9 – Tela de um problema no BROJ

Figura 10 – Lista de problemas no BROJ, visão administrador

A aba de submissões mostra, para cada usuário, apenas suas próprias submissões, para que ele tenha informações relevantes acerca do código enviado, como tempo de execução, veredito, memória utilizada e a possível recuperação do código enviado. Por isso, é possível o uso dessa aba somente quando logado no sistema.

Todas as funcionalidades de CRUD, com exceção do acesso aos *contests* e problemas, são restritos apenas ao papel de administrador, por isso é importante a manutenção de perfis de usuários.

O cadastro de um problema pode ser visto na Figura 11, onde o administrador precisa entrar com o nome único do problema, a descrição (texto que apresenta o problema a ser resolvido), o exemplo de *input* e *output*, o tempo limite para que a solução seja considerada válida, em segundos, a quantidade limite de memória, em Kilobytes e

opcionalmente alguma nota para deixar o problema mais claro.

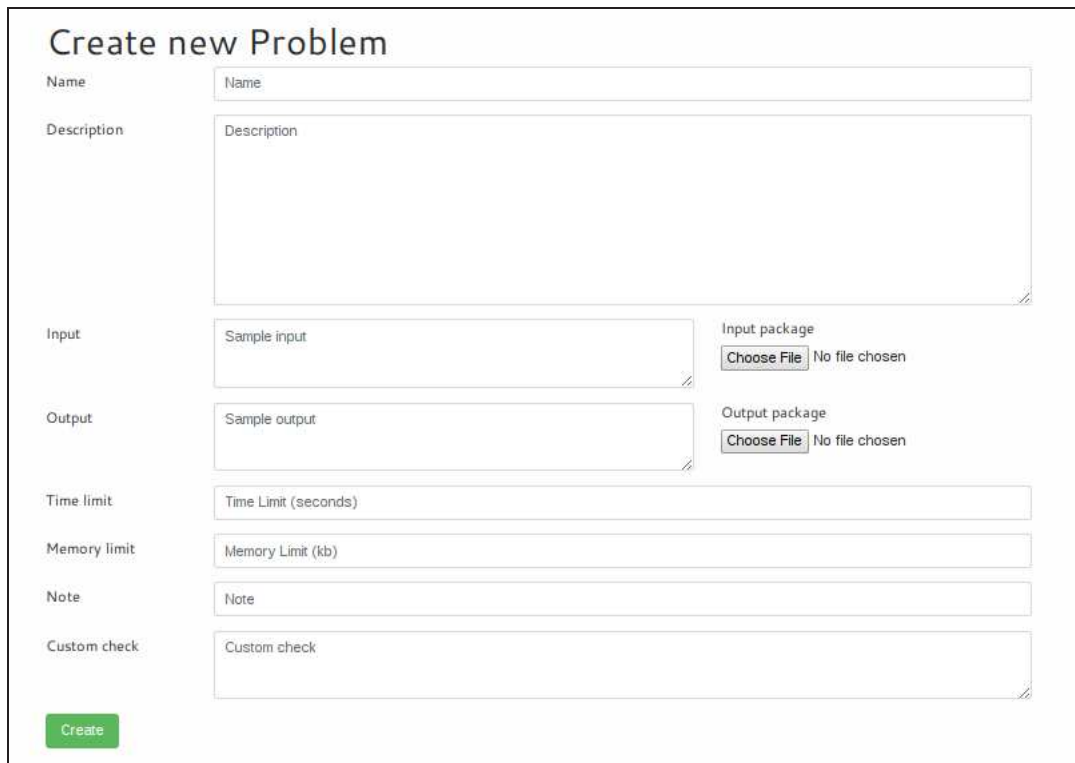


Figura 11 – Cadastro de um problema

É possível que o criador do problema escreva o próprio verificador ao invés de uma simples comparação por igualdade. Para tal, na criação do problema no juiz *online*, precisa ser definido um método em Python que recebe 3 *strings* como parâmetro: o *input* do caso de teste, o *output* esperado e o *output* do usuário para o caso de teste, respectivamente. A partir disso, quaisquer critérios podem ser aplicados para definir se o resultado do usuário está certo ou não. Um simples exemplo para verificar se o numero é corretamente dividido por 2 com precisão customizada pode ser visto no Código 3.2.

```
def check(input, output, user_output):
    import io
    inbuf = io.StringIO(input)
    a = float(inbuf.read())
    outbuf = io.StringIO(user_output)
    b = float(outbuf.read())
    return (abs(a/2.0 - b) < 1e-6)
```

Código 3.2 – Validação customizada

O cadastro de um *contest* pode ser visto na Figura 12, onde o administrador tem que escolher o nome único do *contest*, a data e hora (UTC - *Universal Time Coordinated*) de inicio, a duração em minutos e o conjunto de questões, escolhidos a partir de uma lista.

New Contest

Name

Start 2017 June 21 – 18 : 27

Duration

Problem ids Dia da Vovo Defendendo Alamo

Create Contest

Figura 12 – Cadastro de um *contest*

3.3 Comunicação interface-correção

O juiz online precisa se comunicar com o servidor de correção nos momentos descritos a seguir:

- quando o *contest* é cadastrado, onde é necessário a criação dos problemas na base de dados dos juízes eletrônicos que vão fazer a correção. O *script admin* faz o cadastro em cada um dos juízes;
- quando existe uma submissão para um problema, duas requisições são necessárias: a primeira para recuperar o *id* do problema, a segunda submete o código através do *script* de cliente para o *id* extraído no primeiro passo;
- quando o *courier* tem um novo veredito, ele comunica ao juiz online para que este atualize o placar.

Toda a comunicação é feita usando o protocolo TCP através do RabbitMQ, permitindo que as máquinas do juiz eletrônico e do juiz online não estejam necessariamente no mesmo local físico. Entretanto é necessário para o juiz online ser capaz de executar os *scripts* de administrador e cliente.

O endereço do juiz eletrônico é definido na criação do *contest*, podendo ser posteriormente editado. Dessa forma, no caso de um servidor indisponível o endereço pode ser facilmente modificado para um outro servidor de correção. Existe também um botão de sincronização para o caso de adição de novos problemas ao *contest*, onde o *script* de administrador é novamente acionado para cadastrar os novos problemas no juiz eletrônico. O mesmo botão pode ser utilizado quando o endereço do servidor de correção é modificado.

4 Considerações Finais

O trabalho foi desenvolvido do zero, desde as especificações dos requisitos do sistema até a escolha de tecnologias e programação. Alguns problemas surgiram e alguns estudos tiveram que acontecer para se tomar a melhor decisão para cada problema. Por exemplo, o problema de isolar a compilação e execução do código para que códigos maliciosos não afetem o funcionamento do juiz, onde alternativas de *sandboxing* e redução de privilégios foram discutidas até se chegar no uso do Docker como solução.

Por fim o resultado obtido, mesmo com suas limitações, é capaz de realizar um *contest* por completo, além de permitir recuperação e ressubmissão de problemas após o término do mesmo. O servidor de correção tem módulos desacoplados e assíncronos, seguindo o planejamento inicial de forma satisfatória e permitindo o uso de redundância em cada parte do sistema.

O juiz eletrônico e juiz online desacoplados, de forma que o servidor de correção pode ser utilizado via linha de comando ou em conjunto com outros juízes online, é um ponto positivo da solução. Porém esta característica dificulta a configuração do ambiente, dado que é necessário um processo de configuração para o juiz eletrônico e outro para o juiz online, além da comunicação entre os dois.

O trabalho apresenta potenciais melhorias a serem trabalhadas futuramente, tais como:

- melhorar interface gráfica, ampliando a usabilidade e aplicando padrões de *design*;
- múltiplos nós de servidores de correção para cada *contest* funcionando com balanceamento de carga;
- “dockerizar”o servidor de correção e o juiz *online* para facilitar o *deploy*.

Dado o estado atual do BROJ os objetivos desse trabalho foram parcialmente cumpridos, havendo um juiz eletrônico e um juiz online sob licença livres capazes de conduzir um *contest* e com alguns mecanismos de tolerância à falhas. Fica faltando um teste de *stress* para garantir a capacidade de altas cargas, além de uma solução de balanceamento de carga entre os nós de servidores de correção.

Além dos pontos já mencionados, para trabalhos futuros os principais pontos a serem trabalhados são os de tolerância à falhas de um modo mais automatizado além de testes de *stress* e estratégias de funcionamento com alta capacidade de cargas.

Referências

ACM-ICPC. *About ACM-ICPC*. 2016. Disponível em: <https://icpc.baylor.edu/>. Citado na página 26.

BECK, K. *Extreme programming explained: embrace change*. [S.l.]: addison-wesley professional, 2000. Citado na página 30.

BUI, T. Analysis of docker security. *arXiv preprint arXiv:1501.02967*, 2015. Citado 2 vezes nas páginas 30 e 40.

CASE, S. M.; SWANSON, D. B. *Constructing written test questions for the basic and clinical sciences*. [S.l.]: National Board of Medical Examiners Philadelphia, PA, 1998. Citado na página 23.

CHAVES, J. O. M. et al. Integrando moodle e juízes online no apoio a atividades de programação. In: *Anais do Simpósio Brasileiro de Informática na Educação*. [S.l.: s.n.], 2013. v. 24, n. 1, p. 244. Citado na página 25.

CHEANG, B. et al. On automated grading of programming assignments in an academic institution. *Computers & Education*, Elsevier, v. 41, n. 2, p. 121–131, 2003. Citado na página 23.

CODEFORCES. *About Codeforces*. 2016. Disponível em: <http://codeforces.com/>. Citado na página 27.

CRISTIAN, F. Synchronous and asynchronous. *Communications of the ACM*, ACM, v. 39, n. 4, p. 88–97, 1996. Citado na página 29.

DOCKER. *Docker*. 2017. Disponível em: <https://www.docker.com/>. Citado na página 30.

ECKERDAL, A.; THUNÉ, M.; BERGLUND, A. What does it take to learn 'programming thinking'? In: ACM. *Proceedings of the first international workshop on Computing education research*. [S.l.], 2005. p. 135–142. Citado na página 23.

FELTER, W. et al. An updated performance comparison of virtual machines and linux containers. In: IEEE. *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*. [S.l.], 2015. p. 171–172. Citado na página 42.

FERNANDES, J. L. et al. Performance evaluation of restful web services and amqp protocol. In: IEEE. *Ubiquitous and Future Networks (ICUFN), 2013 Fifth International Conference on*. [S.l.], 2013. p. 810–815. Citado na página 29.

FERRASA, M.; SOUZA, M. Competições de raciocínio lógico e programação de computadores: um relato de experiência. *Anais do 10º CONEX–Conversando Sobre Extensão*, 2012. Citado na página 26.

FINK, J. Docker: a software as a service, operating system-level virtualization framework. *Code4Lib Journal*, v. 25, 2014. Citado na página 29.

FOROUZAN, A. B. *Data communications & networking*. [S.l.]: Tata McGraw-Hill Education, 2006. Citado na página 27.

GNU. *GNU License*. 2007. Disponível em: <http://www.gnu.org/licenses/gpl.html>. Citado na página 39.

HALIM, S. et al. *Competitive programming*. [S.l.]: Lulu, 2013, 2010. Citado na página 26.

ISO/IEC. *Advanced Message Queuing Protocol (AMQP) V1.0 Specification*. [S.l.], 2014. Disponível em: <https://www.iso.org/standard/64955.html>. Citado na página 29.

KURNIA, A.; LIM, A.; CHEANG, B. Online judge. *Computers & Education*, Citeseer, v. 36, p. 299–315, 2001. Citado 3 vezes nas páginas 23, 24 e 25.

KUROSE, J. F.; ROSS, K. W. Redes de computadores e a internet. *São Paulo: Person*, 2006. Citado na página 27.

NORONHA, T. F. et al. Uma nova metodologia para o desenvolvimento de sistemas de correção automática de listas de exercıcios de programação baseada em testes de unidade. 2015. Citado na página 23.

PEREIRA, T. Autilização de juízes eletrônicos e problemas oriundos da maratona de programação no ensino de programação da faculdade unb gama: Um estudo de caso. 2015. Citado na página 23.

SADLER, D. R. Interpretations of criteria-based assessment and grading in higher education. *Assessment & Evaluation in Higher Education*, Taylor & Francis, v. 30, n. 2, p. 175–194, 2005. Citado na página 23.

SAWASHIMA, H. et al. Characteristics of UDP packet loss: Effect of TCP traffic. In: *Proceedings of INET'97: The Seventh Annual Conference of the Internet Society*. [S.l.: s.n.], 1997. Citado na página 28.

SEO, K.-T. et al. Performance comparison analysis of linux container and virtual machine for building cloud. *Advanced Science and Technology Letters*, v. 66, n. 105-111, p. 2, 2014. Citado na página 29.

SKIENA, S. S.; REVILLA, M. A. *Programming challenges: The programming contest training manual*. [S.l.]: Springer Science & Business Media, 2006. Citado na página 25.

UHUNT. *uHunt*. 2016. Disponível em: <http://uhunt.felix-halim.net/>. Citado na página 27.

URI. *URI Online Judge*. 2017. Disponível em: <https://www.urionlinejudge.com.br>. Citado na página 27.

UVA. *About UVa Online Judge*. 2016. Disponível em: <https://uva.onlinejudge.org/>. Citado na página 27.

VINOSKI, S. Advanced message queuing protocol. *IEEE Internet Computing*, IEEE, v. 10, n. 6, 2006. Citado na página 29.

WEBER, T. S. Tolerância a falhas: conceitos e exemplos. *Apostila do Programa de Pós-Graduação–Instituto de Informática-UFRGS. Porto Alegre*, 2003. Citado na página 24.

WEN-XIN, L.; WEI, G. Peking university oneline judge and its applications [j]. *Journal of Changchun Post and Telecommunication Institute S*, v. 2, 2005. Citado na página 23.

WILLIAMS, L. et al. Strengthening the case for pair programming. *IEEE software*, IEEE, v. 17, n. 4, p. 19–25, 2000. Citado na página 31.

XAVIER, M. G. et al. Performance evaluation of container-based virtualization for high performance computing environments. In: IEEE. *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*. [S.l.], 2013. p. 233–240. Citado na página 29.

ZHIGANG, S. et al. Moodle plugins for highly efficient programming courses. 2001. Citado na página 25.

Apêndices

APÊNDICE A – Configuração do ambiente do BROJ

A.1 Juiz eletrônico

A.1.1 Requisitos

- Python >= 3.6
- RabbitMQ-server

A.1.2 Configuração

Deve ser criado o ambiente virtual Python para se trabalhar:

```
$ mkvirtualenv --python=python3.6 broj
$ pip install -r requirements.txt
```

É necessário um arquivo de configuração no caminho **/opt/broj/config.ini** que deve seguir o seguinte padrão:

```
[db]
user = <user>
password = <password>
name = broj_dev
```

Uma tabela chamada **broj_dev** precisa existir no postgres, podendo ser criada assim:

```
$ sudo -u postgres psql
$ CREATE DATABASE broj_dev;
```

O banco pode ser populado com um problema exemplo executando o comando:

```
$ ./admin.py create
```

Para checar os dados no banco rode:

```
$ ./admin.py read
```

A.1.3 Execução

Cliente:

```
$ ./client.py -h
$ ./client.py -l cpp -f ./test_code_cpp.cpp -u 1 -p 1
```

Servidor de correção:

```
$ ./judge.py -l cpp
```

Courier:

```
$ ./courier.py
```

Executando os testes automatizados:

```
$ ./runtests.sh
```

A.2 Juiz online

A.2.1 Requisitos

- Rails >= 5.0
- Ruby >= 2.3

A.2.2 Configuração

Todas as gems podem ser instaladas com o comando:

```
$ bundle install
```

A.2.3 Execução

O servidor pode ser executado com o comando:

```
$ rails s -b 0.0.0.0
```