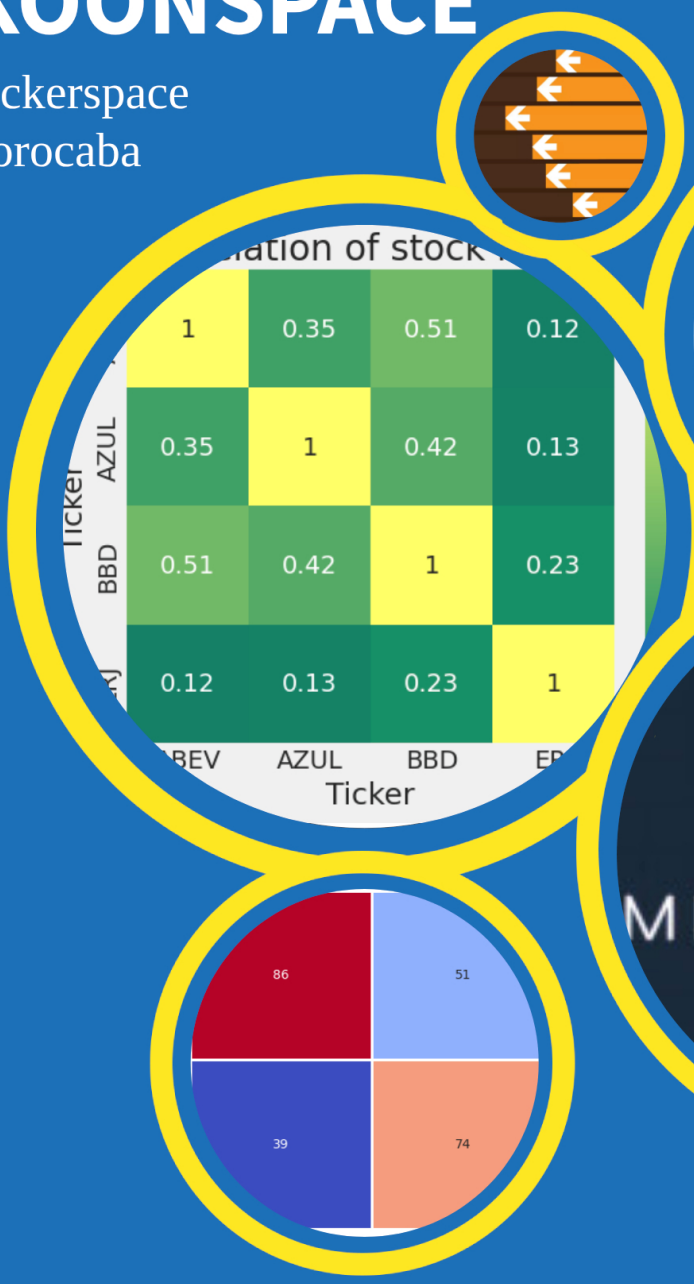
VOL. 6
2024

Revista HACKOONSPACE

Projeto Hackerspace
UFSCar Sorocaba



Apresentação dos artigos e projetos realizados na edição 2024 do HackoonSpace

Universidade Federal de São Carlos
Campus Sorocaba

PROEX
Pró-Reitoria de Extensão

Revista Hackoonspace

Vol. 6

Projeto Hackoonspace
2024

Apresentação

O Projeto Hackoonspace é um projeto de extensão promovido pela UFSCar Campus Sorocaba e realizado na forma de encontros que discutem a contracultura hacker, apresentando personagens, acontecimentos, aspectos socioculturais, artefatos e atividades com a finalidade de que os participantes sejam expostos à contracultura em questão e desenvolvam um projeto ou material acerca da mesma.

A principal intenção do projeto é desmistificar o conceito e figura do hacker, enquanto proporcionando um espaço de exposição, produção e compartilhamento de conteúdo que se configura como dentro da contracultura de forma que participantes de diferentes níveis de conhecimento técnico possam participar.

Esta revista tem como objetivo divulgar os trabalhos desenvolvidos pelos alunos participantes do Projeto Hackerspace no ano de **2024**. *Esperamos que esta revista possibilite a difusão dos conhecimentos adquiridos na atividade para a comunidade em geral. Gostaríamos de agradecer todos os alunos que contribuíram com artigos e projetos para esta edição da revista.*

Organização e edição:
Ana Luiza Salgado de Paula
Fernando Favareto Abromovick
Gustavo Eugênio de Souza Moraes
Marcus Vinicius Caruso Leite

Supervisão:
Gustavo M. D. Vieira

Conteúdo

Parte I

Artigos

Capítulo 1

Eficiência energética na programação

Vinicius Gabriel Nanini da Silva
Vitor Silveira

OBS: Para ver o artigo na íntegra, com imagens maiores, clique **aqui.**

Resumo — A eficiência energética na programação tem se tornado um aspecto fundamental na pesquisa em computação, impulsionada pela necessidade de desenvolver sistemas de alto desempenho com menor consumo de energia. Este estudo investiga o impacto da escolha da linguagem de programação, dos paradigmas adotados e das técnicas de otimização no consumo energético de aplicações computacionais. Para isso, foram analisadas diferentes linguagens e metodologias, considerando *benchmarks* padronizados e métricas de avaliação de eficiência. Os resultados indicam que linguagens compiladas, como C e *Rust*, apresentam melhor equilíbrio entre desempenho e consumo energético em comparação com linguagens interpretadas. Além disso, a programação paralela e a concorrência podem contribuir para a redução do consumo de energia, desde que bem implementadas. O estudo também aborda a influência da arquitetura de *hardware* na eficiência energética, destacando a importância da hierarquia de memória e das técnicas de gerenciamento de cache. Por fim, são discutidas estrategias de otimização de código e ferramentas de medição de consumo energético, ressaltando a necessidade de abordagens multidimensionais para promover a sustentabilidade na computação.

1.1 Introdução

Estrategias para aumentar a eficiência energética em sistemas computacionais tornaram-se um ponto importante na pesquisa em ciência da computação, impulsionadas pela crescente demanda por plataformas paralelas de alto desempenho e baixo consumo energético [1]. A escolha da linguagem de programação tem um impacto no consumo de energia.

Estudos recentes tem se dedicado a comparar o consumo de energia em diferentes linguagens de programação, buscando estabelecer rankings baseados na eficiência energética. Esses estudos revelam que a relação entre tempo de execução e consumo de energia nem sempre é direta, com linguagens mais rápidas, por vezes, consumindo mais energia do que as mais lentas. Além disso, o uso de memória também se mostra um fator relevante no consumo de energia.

A pesquisa nessa área abrange diversas abordagens, desde a análise do impacto de diferentes paradigmas de programação, como programação procedural e orientada a objetos, até a avaliação de práticas de codificação e estruturas de dados. Os resultados indicam que diversos fatores podem influenciar significativamente a eficiência energética de *softwares*, incluindo padrões de projeto, *technical debt*, e estruturas de dados.

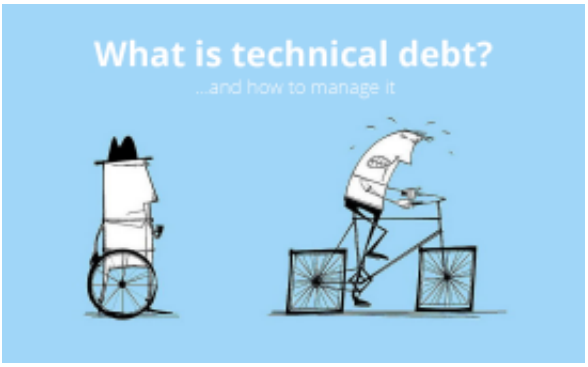


Figura 1.1: *Technical Debt* é o custo futuro de retrabalho causado por soluções rápidas ou ineficientes no desenvolvimento de *software*.

Figura 1.2: TI Verde (ou *Green* IT) refere-se a um conjunto de práticas e tecnologias voltadas para minimizar o impacto ambiental das atividades de Tecnologia da Informação (TI) [2]

A Sociedade Brasileira de Computação (SBC) identificou a gestão da informação em grandes volumes de dados multimídia distribuídos e o desenvolvimento tecnológico de qualidade como desafios cruciais para a pesquisa em computação no Brasil. Nesse contexto, o Grupo de Processamento Paralelo e Distribuído (GPPD) da UFRGS tem explorado a integração de técnicas avançadas de programação paralela com otimizações arquiteturais em máquinas multiprocessadas, visando criar um ambiente homogêneo para a execução de programas paralelos com alto desempenho, baixo consumo de memória e uso otimizado de recursos ociosos.

O paralelismo em diferentes níveis de computação surge como um fator chave para a construção de sistemas de computação verde. A programação paralela desempenha um papel fundamental na melhoria da eficiência energética, permitindo um melhor aproveitamento dos recursos computacionais e aumentando o desempenho das aplicações.

A eficiência energética na programação e um campo multidisciplinar que envolve a avaliação e otimização do consumo de energia em diferentes níveis, desde a escolha da linguagem de programação e o paradigma de programação ate as práticas de codificação e a arquitetura do sistema. A busca por soluções que conciliem alto desempenho e baixo consumo de energia é essencial para garantir a sustentabilidade e o futuro da computação.

1.2 Análise comparativa de linguagens de programação

A crescente preocupação com o consumo de energia em sistemas computacionais tem impulsionado a busca por linguagens de programação que ofereçam um bom desempenho com o menor gasto energético possível. A escolha da linguagem de programação, um dos primeiros passos no desenvolvimento de *software*, pode influenciar significativamente a eficiência energética da aplicação [3]. Um estudo investigou como diferentes linguagens de programação (C, C++ e Java) e paradigmas (orientado a objetos e procedural) influenciam o consumo de energia e o desempenho em aplicações paralelas [4], utilizando diferentes *kernels* do NAS *Parallel Benchmark*. Os resultados mostraram que a linguagem procedural C demonstrou melhor desempenho e menor consumo de energia em comparação com as lingua-

gens orientadas a objetos *Java* e *C++*. O estudo também avaliou o impacto do compilador *Just-In-Time* (JIT) na execução de aplicações *Java*, observando que *Java* depende muito da eficiência do JIT para obter um bom desempenho. Outro estudo comparou um conjunto grande de linguagens de programação com relação à sua eficiência, inclusive do ponto de vista energético [5]. O objetivo foi criar e analisar diferentes rankings para linguagens de programação com base em sua eficiência energética. Os resultados mostraram como linguagens mais lentas/rápidas podem consumir menos/mais energia e como o uso da memória influencia o consumo de energia. A análise comparativa foi realizada com base em estudos existentes que utilizam *benchmarks* padronizados, como o "*Computer Language Benchmarks Game*" (CLBG).

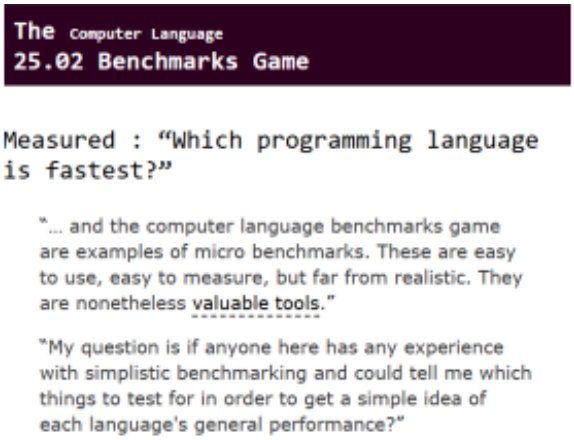


Figura 1.3: O *Computer Language Benchmarks Game* é um projeto *open source* que compara a implementação de algoritmos simples em diversas linguagens

Os *benchmarks* consistem em um conjunto de problemas computacionais que são implementados em diversas linguagens de programação, permitindo a comparação do tempo de execução, consumo de energia e uso de memória. O CLBG inclui um *framework* para executar, testar e comparar soluções implementadas para um conjunto de problemas de programação bem conhecidos e diversos [5]. Por outro lado, uma pesquisa detalhada sobre a relação entre a escolha da linguagem de programação e o consumo de energia corrigiu e aprimorou a metodologia de medição usada em trabalhos anteriores [6]. O estudo desenvolveu um modelo causal detalhado, identificando fatores críticos como a distinção entre linguagens de programação e suas implementações, o impacto das implementações de aplicativos, o número de núcleos ativos e a atividade da memória. Os resultados sugerem que a escolha da linguagem de programação não tem um impacto significativo no consumo de energia além do tempo de execução [6].

1.3 Impacto do paralelismo e da concorrência

O desenvolvimento de aplicações paralelas e concorrentes tem se tornado cada vez mais relevante no cenário da computação moderna, impulsionado pela crescente demanda por desempenho e eficiência energética. A escolha de paradigmas e interfaces de programação paralela (IPPs) pode influenciar significativamente o consumo energético e o desempenho de aplicações [4] [3].

A computação paralela envolve o uso simultâneo de múltiplas unidades de processamento para realizar cálculos, visando resolver problemas complexos, reduzir o tempo de solução, realizar cálculos mais precisos e utilizar eficientemente os recursos computacionais [4]. O paralelismo pode ser explorado em diferentes níveis, como no nível de instruções (ILP) e no nível de *threads* (TLP).

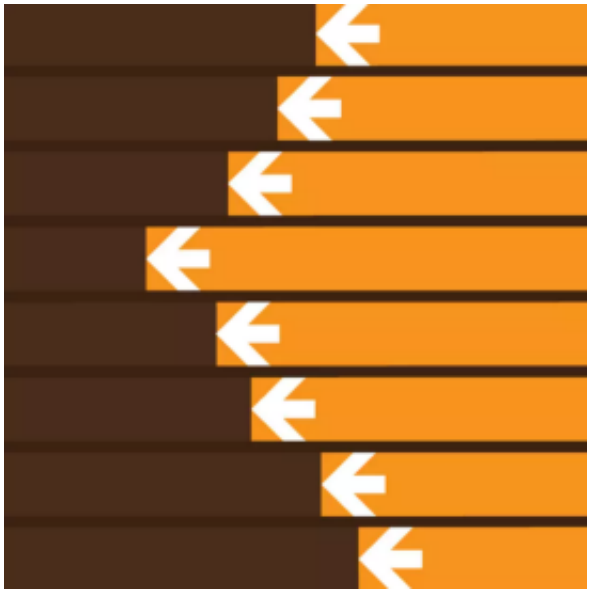


Figura 1.4: O paralelismo em programação permite a execução simultânea de múltiplas tarefas, melhorando desempenho e eficiência computacional.

A concorrência, por sua vez, se refere à capacidade de um sistema lidar com múltiplas tarefas simultaneamente, mesmo que não haja execução paralela real. A concorrência pode ser implementada através de *threads*, processos ou outras técnicas, permitindo que um programa execute diferentes partes do código de forma aparentemente simultânea [4].

A paralelização de aplicações geralmente leva a um ganho de desempenho em relação à versão sequencial, pois o tempo total de computação é distribuído entre os processadores. No entanto, o ganho de desempenho pode ser limitado por fatores como a dependência de dados, a comunicação entre processadores e o *overhead* da criação e gerenciamento de threads.

Ao contrário do desempenho, o consumo de energia em aplicações paralelas nem sempre acompanha o ganho de desempenho. A energia total consumida pela execução paralela corresponde a soma da energia consumida em cada processador, adicionada a energia consumida para comunicação entre os processadores. Aplicações com grande quantidade de comunicação tendem a ter um maior consumo de energia.

O *overhead* da execução em uma maquina virtual e com o compilador JIT também pode aumentar o consumo de energia. Além disso, o gerenciamento de múltiplas *threads* pode superar o ganho com o paralelismo, resultando em pouca escalabilidade no consumo de energia.

1.4 Otimização de código

A otimização de código é uma área crucial na ciência da computação, com o objetivo de aprimorar o desempenho e a eficiência de *softwares* em diversas métricas, como tempo de execução, consumo de energia e uso de memória [4]. A crescente preocupação com a sustentabilidade e a popularização de dispositivos móveis e sistemas embarcados têm impulsionado o interesse na **eficiência energética** do *software*[4].

A. Técnicas de otimização de código

Diversas técnicas podem ser aplicadas para otimizar o código e reduzir o consumo de energia:

- **Paralelização**: Explorar o paralelismo em aplicações pode melhorar o desempenho em processadores *multicore* [4]. A escolha de bibliotecas de *threads* e interfaces de programação paralela (IPPs) adequadas e importante para otimizar o uso de recursos computacionais.
- **Otimização de *loops***: Remover ou refatorar *loops* desnecessários pode reduzir o consumo de energia.
- **Gerenciamento de dados**: Adotar políticas de cache eficientes, minimizar a troca de dados e gerenciar o ciclo de vida dos dados armazenados são praticas importantes.
- **Ajuste de precisão computacional**: Limitar a precisão computacional ao nível necessário para a operação pode economizar energia.
- **Monitoramento em tempo real**: Monitorar o consumo de energia em tempo real permite identificar gargalos e áreas de melhoria.

- **Controle de granularidade**: O controle da granularidade em programação paralela, ajustando o tamanho das tarefas para **maximizar o paralelismo e diminuir o *overhead***, contribui para o uso eficiente dos recursos e minimização do consumo de energia [3].

B. Ferramentas e Metodologias

A avaliação e comparação de linguagens de programação e técnicas de otimização requerem o uso de *benchmarks* padronizados e ferramentas de medição de energia [4].

- O **NAS *Parallel Benchmark* (NPB)** é um conjunto de *benchmarks* desenvolvido pela NASA para avaliar o desempenho de computadores paralelos [4].

- O ***Computer Language Benchmarks Game* (CLBG)** é utilizado para comparar linguagens em termos de consumo energético, tempo de execução e uso de memória [4].

- Ferramentas como o **Intel *Running Average Power Limit* (RAPL)** permitem medir o consumo de energia de diferentes componentes do sistema [4].

Tabela I
COMPARAÇÃO DE LINGUAGENS DE PROGRAMAÇÃO

	Energy	Time	Mb
(c) C	1.00	1.00	1.17
(c) Rust	1.03	1.04	1.54
(c) C++	1.34	1.56	1.34
(c) Ada	1.70	1.85	1.47
(v) Java	1.98	1.89	6.01
(c) Pascal	2.14	3.02	1.00
(c) Chapel	2.18	2.14	4.00
(v) Lisp	2.27	3.40	1.92
(c) Ocaml	2.40	3.09	2.82
(c) Fortran	2.52	4.20	1.24
(c) Swift	2.79	4.20	2.71
(c) Haskell	3.10	3.55	2.45
(v) C#	3.14	3.14	2.85
(c) Go	3.23	2.83	1.05
(i) Dart	3.83	6.67	8.64
(v) F#	4.13	6.30	4.25
(i) JavaScript	4.45	6.52	4.59
(v) Racket	7.91	11.27	3.52
(i) TypeScript	21.50	46.20	4.69
(i) Hack	24.02	26.99	3.34
(i) PHP	29.30	27.64	2.57
(v) Erlang	42.23	36.71	7.20
(i) Lua	45.98	82.91	6.72
(i) Jruby	46.54	43.44	19.84
(i) Ruby	69.91	59.34	3.97
(i) Python	75.88	71.90	2.80
(i) Perl	79.58	65.79	6.62

Figura 1.5: Estudo de comparação da eficiência de linguagens de programação

C. Estudos e Resultados

Estudos têm demonstrado que a escolha da linguagem e as técnicas de otimização podem ter um impacto significativo no consumo de energia.

- Um estudo comparou 27 linguagens de programação e para cada linguagem foi feito o melhor código para vários algoritmos conhecidos como: *fasta*, *binary-trees*, *k-nucleotide* e outros. Com isso identificou-se que *C*, *Rust* e *C++* são as mais eficientes em termos de energia (Tabela 1) [1].

- A versão C do NAS *Parallel Benchmark* obteve melhor desempenho e menor consumo energético em relação as implementações em *C++* e *Java* [4].

- O uso do compilador JIT em *Java* pode melhorar o desempenho, mas também aumentar o consumo de energia. [4]

Figura 1.6: *Rust*, *C* e *C++*: Linguagens mais bem ranqueadas no estudo de comparação de eficiência das linguagens

1.5 Arquiteturas de *Hardware*

As arquiteturas de *hardware* desempenham um papel fundamental no desempenho e na eficiência energética de sistemas computacionais. A escolha e o design da arquitetura influenciam diretamente o consumo de energia, o tempo de execução e a capacidade de processamento de aplicações.

A. Computação Paralela e *Multicore*

- A **computação paralela** e o uso de **processadores *multicore*** são abordagens importantes para aumentar o desempenho sem aumentar o consumo de energia de forma linear. Ao invés de elevar a frequência do *clock*, o que leva a um aumento significativo no consumo de energia, o paralelismo permite que varias unidades de processamento trabalhem simultaneamente [4].

- **Arquiteturas *multicore***, que integram múltiplos núcleos de processamento em um único chip, tornaram-se populares como uma forma de contornar as limitações físicas da miniaturização de transistores e do aumento da frequência [4].

- A eficiência da computação paralela depende da **sincronização** e **comunicação** entre os processadores, que podem ser realizadas por meio de memória compartilhada ou troca de mensagens. A escolha do modelo de comunicação e das interfaces de programação paralela (IPPs) adequadas pode impactar o desempenho e o consumo de energia [4].

B. Hierarquia de Memória *Cache*

- A **hierarquia de memória *cache*** é um componente crítico das arquiteturas de *hardware* modernas. Ela visa reduzir a latência no acesso aos dados, armazenando em níveis mais rápidos (*cache* L1, L2, L3) os dados mais frequentemente utilizados [4].

- O **compartilhamento de memória *cache*** entre os núcleos de processamento pode melhorar o desempenho, mas também apresenta desafios em termos de organização e gerenciamento [3]. Estudos mostram que a integração entre os projetos de arquitetura de memória *cache* e o projeto físico da memória são importantes para obter o melhor equilíbrio entre tempo de acesso à memória *cache* e redução de faltas de dados [3].

- **Arquiteturas de *cache* não uniforme (NUCA)** são uma alternativa às arquiteturas de cache tradicionais, visando reduzir a latência no acesso à memória [3].

C. Otimização de *Hardware* para eficiência energética

- ***Green Computing*** busca desenvolver soluções de *hardware* com menor consumo energético. Isso envolve a otimização de componentes, o desligamento de unidades ociosas e o uso de técnicas de gerenciamento de energia [3].

- A **gestão de energia** em processadores modernos permite o ajuste dinâmico da frequência e tensão dos núcleos, adaptando o consumo de energia a carga de trabalho [3].

- O uso de ***hardware* especializado**, como GPUs e FPGAs, pode acelerar o desempenho de tarefas específicas com maior eficiência energética.

D. Impacto da Arquitetura no *Software*

- As características da arquitetura de *hardware* influenciam a forma como o *software* é desenvolvido e otimizado [4]. O programador deve considerar a **organização da memória** e os **recursos de paralelismo** disponíveis para obter o máximo desempenho e eficiência energética.
- A **granularidade** das tarefas paralelas, o **escalonamento** e o **balanceamento de carga** são aspectos importantes da programação paralela que devem ser adaptados à arquitetura de *hardware* [3].

E. Tendências futuras

- • A pesquisa em arquiteturas de *hardware* para eficiência energética continua a ser uma área ativa, com foco em **novas tecnologias de integração, memórias não voláteis e arquiteturas heterogêneas**.
- • O uso eficiente da memória, tanto no acesso quanto no armazenamento de dados, é um aspecto importante a ser considerado em futuras arquiteturas para ***Green Computing***.

1.6 Ferramentas e Metodologias para Medição de Energia

A medição precisa do consumo de energia é crucial para avaliar a eficiência energética de *software* e hardware, permitindo identificar áreas de otimização e comparar diferentes abordagens. Diversas ferramentas e metodologias estão disponíveis para este fim, cada uma com suas características e aplicações.

A. Ferramentas de Medição

- **Intel RAPL (*Running Average Power Limit*)**: Utilizada para medir o consumo de energia em CPUs da *Intel*, oferecendo precisão e sendo amplamente confiável.
- ***Unix time***: Ferramenta nativa de sistemas *Unix* usada para monitorar o uso de memória, fornecendo informações sobre o consumo máximo de memória (*peak memory usage*).
- **Bibliotecas de *software***: A biblioteca *"Running Average Power"*foi utilizada para obter o consumo energético nos experimentos descritos.
- ***Intel Software Development Assistant***: Permite medir o consumo de energia durante a execução de cargas de trabalho específicas.

B. Metodologias de Medição

- ***Benchmarks* Padronizados**: O uso de *benchmarks* como o *Computer Language Benchmarks Game* (CLBG) e o NAS *Parallel Benchmarks* (NPB) garante comparabilidade entre diferentes linguagens, paradigmas e arquiteturas.
- **Métricas de Avaliação**: Além do tempo de execução e do consumo de energia, o *Energy-Delay Product* (EDP) é uma métrica útil para combinar desempenho e consumo energético em um único valor. O EDP é calculado como Energia × Tempo.
- **Repetições múltiplas**: Para garantir a confiabilidade dos resultados, é importante realizar múltiplas repetições das medições.
- **Tempo de Inatividade**: Entre cada execução, um tempo de inatividade (*idle time*) de dois minutos pode ser utilizado para permitir que o sistema arrefeça e execute a coleta de lixo (*garbage collecting*).
- **Análise de Diferentes Tipos de Execução e Paradigmas**: Inclui a análise por tipo de execução (compiladas, interpretadas e em maquina virtual) e paradigmas (imperativo, funcional, orientado a objetos, *script*).

C. Considerações sobre a Validade

- **Viés de Mono-método**: Utilizar apenas uma ferramenta para medir energia e tempo pode ser uma limitação. No entanto, ferramentas como RAPL e *time* são conhecidas por sua precisão.
- **Interação de Diferentes Tratamentos**: É importante usar programas independentes para avaliar as linguagens.

- **Escolha Representativa de *Benchmarks***: O *benchmark* selecionado deve refletir os cenários reais de uso.
- **Monitoramento Contínuo**: Acompanhar o consumo de energia em tempo real durante o desenvolvimento pode contribuir para otimizações mais eficazes.

Ao combinar ferramentas de medição precisas com metodologias rigorosas, é possível obter *insights* valiosos sobre o consumo de energia em sistemas computacionais, orientando o desenvolvimento de *software* e *hardware* mais eficientes.

1.7 Conclusão

A análise realizada ao longo deste estudo evidencia a importância da eficiência energética no desenvolvimento de *software*, considerando fatores como a escolha da linguagem de programação, o paradigma adotado, as técnicas de otimização de código e as arquiteturas de *hardware* utilizadas. Os resultados demonstram que linguagens compiladas, como *C* e *Rust*, tendem a apresentar um melhor equilíbrio entre desempenho e consumo energético quando comparadas a linguagens interpretadas, como *Python* e *Perl*. No entanto, a relação entre tempo de execução e consumo energético nem sempre é linear, indicando a necessidade de uma abordagem multidimensional na avaliação da sustentabilidade do *software*.

O uso do paralelismo e da concorrência se mostrou um fator relevante para a eficiência energética, embora sua eficácia dependa da granularidade das tarefas, do modelo de comunicação entre processadores e do *overhead* introduzido pela gestão de *threads*. Além disso, a hierarquia de memória e a escolha de técnicas de *cache* desempenham um papel fundamental na redução do consumo energético, tornando a otimização da alocação e do acesso a dados um aspecto crítico para sistemas de alto desempenho.

A implementação de ferramentas e metodologias de medição, como o *Intel* RAPL e *benchmarks* padronizados, permite uma análise mais precisa do impacto energético das aplicações, fornecendo subsídios para decisões fundamentadas no desenvolvimento de *software* sustentável. Estratégias de otimização, como a refatoração de código, o ajuste da precisão computacional e a adoção de paradigmas que minimizem o uso excessivo de recursos, são essenciais para promover a eficiência energética

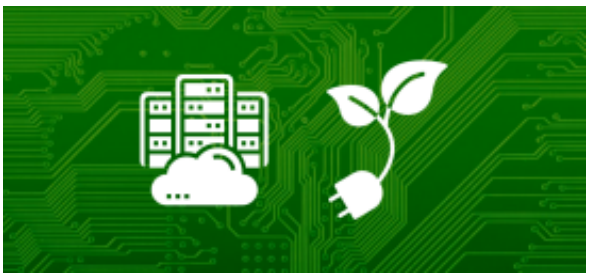


Figura 1.7: A busca por eficiência energética no desenvolvimento de *software* é essencial para a sustentabilidade (*Green IT*).

Como trabalhos futuros, sugere-se a ampliação do escopo da pesquisa para incluir novas linguagens emergentes e arquiteturas especializadas, como aceleradores baseados em inteligência artificial. Além disso, a criação de *frameworks* automatizados para avaliação e recomendação de práticas sustentáveis pode contribuir significativamente para o avanço da computação verde. A busca por um equilíbrio entre desempenho, consumo energético e viabilidade econômica seguirá como um dos principais desafios para a área de desenvolvimento de *software*, demandando esforços contínuos para a construção de sistemas mais eficientes e sustentáveis.

1.8 Bibliografia

[1] S. georgiou, m. kechagia, and d. spinellis, "analyzing programming languages' energy consumption: An empirical study," proceedings of the 21st pan-hellenic conference on informatics, pp. 1–6, 2017.

[2] S. murugesan, "harnessing green it: Principles and practices," it professional, vol. 10, no. 1, pp. 24–33, 2008.

[3] S. d. mor, m. a. alves, j. v. lima, n. b. maillard, and p. o. navaux, "eficiência energética em

computação de alto desempenho: Uma abordagem em arquitetura e programação para green computing,"xxxvii seminario integrado de software e hardware (semish) , pp. 346–360, 2010.

[4] G. g. d. magalhaes, "como linguagens de programação e paradigmas afetam desempenho e consumo energético em aplicac oes paralelas," dissertação de mestrado, universidade federal do rio grande do sul, 2016.

[5] R. pereira, m. couto, f. ribeiro, r. rua, j. cunha, j. p. fernandes, and j. saraiva, "ranking programming languages by energy efficiency," science of computer programming, vol. 205, p. 102609, 2021.

[6] N. van kempen, h.-j. kwon, d. t. nguyen, and e. d. berger, "it's not easy being green: On the energy efficiency of programming languages," arxiv preprint arxiv:2410.05460, 2024.

Capítulo 2

APIs: a Diplomacia no Mundo da Tecnologia

Ryan Guerra Sakurai

OBS: Para ver o artigo na íntegra, com imagens maiores, clique **aqui.**

2.1 Introdução

No universo da tecnologia, alguns conceitos têm papeis essenciais nos bastidores, mas são quase invisíveis aos olhos do usuário final. Um deles é a API (*Application Programming Interface*), ou Interface de Programação de Aplicações. Assim como diplomatas facilitam a comunicação entre nações, as APIs atuam como intermediárias silenciosas, garantindo que *softwares* distintos possam "conversar", trocar informações e trabalhar em harmonia.

Para entender as APIs, é preciso primeiro compreender o que é uma interface. Interfaces operam com base em dois pilares: abstração e encapsulamento. A abstração esconde complexidades, permitindo que utilizemos algo sem precisar entender seu funcionamento interno. Já o encapsulamento protege os detalhes técnicos, expondo apenas o que é necessário para a interação.

Um controle remoto de TV, por exemplo, é uma interface física: você aperta botões para mudar de canal sem precisar entender como ele funciona por dentro. Da mesma forma, um aplicativo de banco, que é uma interface gráfica (GUI, *Graphical User Interface*), simplifica operações complexas: com alguns cliques, você transfere dinheiro sem ver os sistemas de segurança ou bancos de dados por trás. As APIs funcionam da mesma maneira, porém, ao contrário das interfaces citadas, que são feitas para usuários finais, as APIs são feitas para serem usadas por programas

No *Spotify*, por exemplo, quando um desenvolvedor cria o botão de *play* no aplicativo, ele utiliza uma API de uma biblioteca de interfaces gráficas para mostrar o botão visualmente. A essa interface pré-construída, ele adiciona a lógica específica da aplicação. Quando o usuário clica no botão, a aplicação aciona a API do servidor do *Spotify* para obter os dados da música (arquivo de áudio, metadados, etc.). Em seguida, para reproduzir o áudio, o aplicativo utiliza a API do sistema operacional, como o *Android*, que controla o *hardware* do dispositivo, garantindo que o som saia pelo fone do usuário. Paralelamente, se a letra da música aparece na tela, isso ocorre porque o *Spotify* integra uma API externa, a do *Musixmatch*, que fornece as letras sincronizadas em tempo real. Cada etapa desse processo depende de APIs distintas e de diferentes tipos, todas trabalhando em conjunto para que a aplicação funcione.

2.2 Tipos de API

Quando falamos de APIs, normalmente nos referimos a APIs *web*, que permitem a comunicação entre aplicações via internet. No entanto, as APIs estão presentes em qualquer contexto de desenvolvimento e em qualquer tipo de *software*, atu-

ando como pontes entre componentes de um sistema. Abaixo, exploro alguns dos principais tipos.

APIs de Sistema Operacional

Essas APIs permitem que aplicativos interajam com recursos do sistema operacional, como *hardware*, serviços de arquivo ou redes. No exemplo anterior usando o *Spotify*, o aplicativo utiliza a API do *Android* ou iOS para acessar o controle de áudio do dispositivo. A existência dessas APIs é a razão pela qual muitos programas podem ser independentes de sistema operacional: elas abstraem detalhes de implementação específicos de cada SO, permitindo que desenvolvedores criem aplicações compatíveis com múltiplas plataformas.

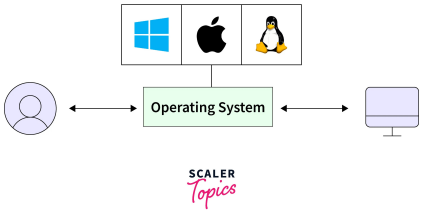


Figura 2.1: Ilustração de uma API de Sistema Operacional

APIs de Linguagens de Programação

Linguagens de programação oferecem APIs nativas para simplificar operações rotineiras na linguagem. Um exemplo é a *Collections* API do Java, que fornece estruturas de dados prontas, como listas (‘*ArrayList*‘, ‘*LinkedList*‘), conjuntos (‘*HashSet*‘, ‘*TreeSet*‘) e mapas (‘*HashMap*‘). Essas estruturas são acessadas por meio de interfaces padronizadas, permitindo que desenvolvedores manipulem dados sem se preocupar com detalhes de implementação. Por exemplo, ao usar a interface ‘*List*‘, abstraem-se como os elementos são armazenados internamente. Isso garante flexibilidade: é possível alternar entre ‘*ArrayList*‘ e ‘*LinkedList*‘ sem alterar a lógica principal do programa.

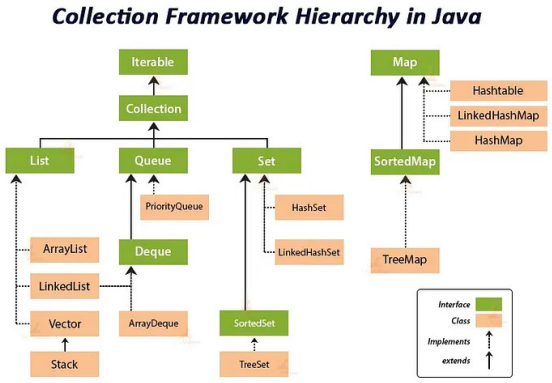


Figura 2.2: Ilustração da *Collections* API, sendo que as interfaces (verde escuro) representam APIs

APIs de Bibliotecas e *Frameworks*

Bibliotecas e *frameworks* também disponibilizam APIs para simplificar problemas recorrentes. Um exemplo é a biblioteca *React*, que oferece funções e componentes pré-definidos (como ‘*Fragment*‘ e *hooks*) para construção de interfaces de usuário. No caso do *Spotify*, o botão de *play* pode ser criado usando a API de uma biblioteca gráfica, que encapsula a renderização visual e eventos de clique.

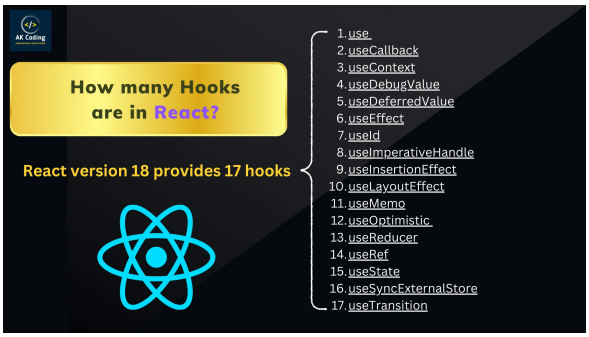


Figura 2.3: Funções que a biblioteca *React* disponibiliza através de sua API

APIs Remotas

São APIs acessíveis via rede, geralmente utilizando protocolos como HTTP. O *Spotify*, por exemplo, usa a API do *Musixmatch* para exibir letras de músicas em tempo real. As API web previ-

amente citadas são um tipo de API remota e serão exploradas com mais profundidade a seguir.

2.3 APIs *Web*

As APIs *web* são interfaces que permitem a comunicação entre aplicações através da internet, utilizando protocolos como HTTP. Sua popularidade cresceu com a computação em nuvem, microsserviços e aplicações multiplataforma, que exigem interoperabilidade e escalabilidade. *Endpoints* são elementos fundamentais nesse contexto: trata-se de URIs (*Uniform Resource Identifier*), ou Identificadores Uniformes de Recurso, que uma API expõe para acesso a recursos ou funcionalidades. Por exemplo, um *endpoint* como '*/musicas*' pode ser acessado para retornar uma lista de músicas. Para garantir a segurança no uso dessas APIs, são frequentemente utilizados *tokens* de autenticação para controle de acesso e para prevenir mal uso das APIs.

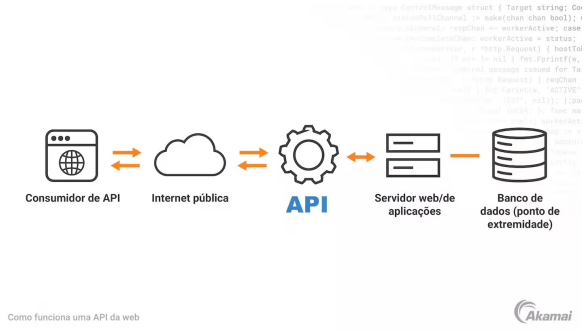


Figura 2.4: Ilustração do funcionamento de uma API *web*

Três motivos principais impulsionam sua adoção: delegação de processamento, que alivia a carga computacional nos dispositivos finais ao transferir tarefas complexas para o servidor; economia de memória, já que dados podem ser acessados sob demanda, reduzindo a necessidade de armazenamento local; e facilidade de integração, graças à padronização que simplifica a conexão entre sistemas heterogêneos.

Dentre as tecnologias relacionadas, abordaremos *SOAP*, *REST*, *GraphQL*, *WebSocket* e *Webhook*. Embora nem todas se enquadrem estritamente na definição técnica de APIs Web, todas tem participação e estão interligadas no ecossistema moderno. Além disso, há variações no nível de abstração dos conceitos e em suas classificações.

SOAP

O SOAP (*Simple Object Access Protocol*), ou Protocolo Simples de Acesso a Objetos, é um protocolo padronizado para troca de dados estruturados entre aplicações. Projetado para ser independente de plataforma e linguagem, ele usa XML para formatar mensagens e é majoritariamente usado através de HTTP, apesar de algumas aplicações mais antigas utilizarem outros protocolos. A seguir está um exemplo de troca de mensagens SOAP:

```xml
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap=
"http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <GetMusicaInfos xmlns="http://api.com/soap">
      <musicaId>1234</musicaId>
    </GetMusicaInfos>
  </soap:Body>
</soap:Envelope>
```

—

```xml
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap=
"http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <m:GetMusicaInfosResposta xmlns:m=
    "http://api.com/soap">
      <m:Musica>
        <m:Titulo>Purity Weeps</m:Titulo>
        <m:Artista>Invent Animate</m:Artista>
        <m:Lancamento>Heavener</m:Lancamento>
        <m:Duracao>04:08</m:Duracao>
        <m:Generos>
          <m:Genero>Metalcore</m:Genero>
          <m:Genero>Metal Progressivo</m:Genero>
        </m:Generos>
      </m:Musica>
    </m:GetMusicaInfosResposta>
  </soap:Body>
</soap:Envelope>
```

Ele é baseado em serviços e sua estrutura rígida é definida por um contrato formal escrito em WSDL (*Web Services Description Language*), ou Linguagem de Descrição de Serviços *Web*, que descreve as operações disponíveis e os formatos de entrada/saída da API. Por exemplo, um serviço bancário pode usar WSDL para especificar como uma transferência deve ser solicitada, incluindo campos como valor, conta destino e autenticação.

O SOAP se destaca em ambientes que exigem alta segurança e confiabilidade, como sistemas financeiros ou governamentais. Ele suporta padrões avançados de criptografia e transações ACID (Atomicidade, Consistência, Isolamento, Durabilidade), essenciais para operações críticas. Além disso, é utilizado em sistemas legados, onde a estabilidade e a adesão a regras bem-definidas são prioritárias.

Apesar de sua robustez, o SOAP é criticado por sua verbosidade (devido ao XML) e complexidade, o que faz com que não seja tão amplamente usado atualmente. No entanto, em cenários onde a precisão e a segurança superam a necessidade de agilidade, ele permanece uma escolha relevante.

REST

REST (*Representational State Transfer*), ou Transferência de Estado Representacional, é o tipo de API mais utilizado na atualidade, destacando-se pela flexibilidade e leveza em comparação a protocolos mais rígidos como o SOAP. Enquanto o SOAP depende de especificações formais, o REST segue princípios flexíveis, priorizando simplicidade e eficiência na comunicação entre sistemas.

Sua arquitetura é centrada em recursos e coleções, entidades acessíveis por meio de URLs (tipos de URI). Por exemplo, ao acessar um *endpoint* como '*https://api.com/musicas*', o cliente pode criar um novo recurso na coleção de músicas e, ao acessar '*https://api.com/musicas/1234*', o cliente pode solicitar dados de uma música específica cujo identificador é '1234'. Essas entidades são geralmente representadas em JSON (*JavaScript Object Notation*), ou Notação de Objeto *Javascript*, um formato leve, legível e fácil de processar.

A comunicação em REST utiliza métodos HTTP para definir ações sobre os recursos, seguindo uma lógica intuitiva: 'GET' recupera recursos, 'POST' os cria, 'PUT' os atualiza por completo, 'PATCH' os modifica parcialmente e 'DELETE' os remove. Para especificar como e o quê deve ser acessado, os URLs podem incluir parâmetros de caminho (ex.: '*/musicas/123*', onde '123' especifica o ID da música) para identificar recursos específicos e parâmetros de consulta (ex.: '*/musicas?genero=metal*', onde '*genero=metal*' filtra as músicas por gênero), usados para filtrar, ordenar ou paginar resultados.

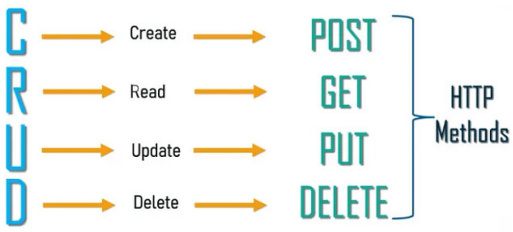


Figura 2.5: Relação entre as operações de CRUD e os métodos HTTP

Complementando os métodos e parâmetros, os *headers* (cabeçalhos) HTTP transmitem metadados, como autenticação ou tipo de conteúdo. Enquanto isso, os códigos de status indicam resultados: 2xx para sucesso (como '*200 OK*'), 4xx para erros do cliente (como o famoso '*404 Not Found*') e 5xx para falhas do servidor (como '*500 Internal Server Error*'). Esses códigos ajudam a identificar rapidamente problemas, como um usuário tentando acessar uma página inexistente.

Para ser considerada *RESTful*, uma API deve seguir os seguintes princípios:

- Arquitetura Cliente-Servidor: separação clara de responsabilidades entre cliente e servidor;
- *Stateless* (Sem Estado): cada requisição deve ser autossuficiente, contendo todas as informações necessárias para seu processamento,

sem que o servidor armazene contexto entre requisições;

- *Cacheabilidade*: clientes ou intermediários podem reutilizar respostas anteriores para requisições idênticas, reduzindo a carga no servidor e melhorando o desempenho (essa capacidade deve ser especificado pelo servidor);
- Sistema em Camadas: intermediários (como *gateways* e balanceadores de carga) podem atuar entre cliente e servidor sem afetar a comunicação, sem que o cliente precise saber disso;
- Interface Uniforme: a interação entre cliente e servidor é padronizada, aderindo a quatro subprincípios:
  - Identificação de Recursos em Requisições: cada recurso envolvido na interação deve ser identificado de maneira única por um URI, sendo conceitualmente distintos das representações retornadas ao cliente;
  - Manipulação de Recurso Através de Representações: se um cliente possui uma representação de um recurso e seus metadados, ele tem informações suficientes para modificar ou excluir o recurso;
  - Mensagens Autodescritivas: mensagens devem incluir informações suficientes para descrever como processá-las;
  - Hipermídia como Motor do Estado de Aplicação (HATEOAS): tendo acessado um URI inicial da aplicação, o cliente deve podem acessar todos os recursos que precise através de *links* providos pelo servidor.

A seguir está um exemplo de troca de mensagens REST:

```
GET /musicas/1234 HTTP/1.1
Host: api.com
Accept: application/json
```

```
HTTP/1.1 200 OK
Content-Type: application/json
```

```
{
  "id": "1234",
  "titulo": "Purity Weeps",
  "artista": "Invent Animate",
  "lancamento": "Heavener",
  "duracao": "04:08",
  "generos": ["Metalcore", "Metal Progressivo"],
  "_links": {
    "self": { "href": "/musicas/1234" },
    "lancamento": { "href": "/lancamentos/heavener" },
    "artista": { "href": "/artistas/4321" }
  }
}
```

Para facilitar a documentação e padronização de APIs REST, ferramentas como *OpenAPI* (antigo *Swagger*) são amplamente usadas. Elas permitem descrever *endpoints*, métodos e parâmetros em um formato estruturado, possibilitando gerar documentação, código, casos de teste, etc. Isso ajuda a reduzir erros e acelerar o desenvolvimento.

GraphQL

O GraphQL é uma linguagem de consulta e *runtime* (ambiente de execução) que transfere o controle dos dados para o cliente. Ele oferece uma alternativa flexível aos paradigmas tradicionais como REST e SOAP, especialmente em cenários onde os requisitos de dados são mais complexos. Sua principal vantagem reside na capacidade para o cliente definir exatamente quais dados deseja receber, eliminando problemas comuns em outras arquiteturas, como o *overfetching* (receber mais informações do que o necessário) e o *underfetching* (obter menos dados, exigindo requisições adicionais).

Em APIs REST, por exemplo, ao acessar um *endpoint* como '*/musicas/1234*', o servidor retorna todos os campos da música, mesmo que o cliente precise apenas do título e do artista. Com o *GraphQL*, isso é evitado: a consulta especifica apenas os campos desejados, como titulo e artista, resultando em uma resposta enxuta. Além disso, o *underfetching* é resolvido com consultas aninhadas. Em vez de fazer múltiplas requisições para

obter a música e seu lançamento (como ocorreria em REST, com *endpoints* separados), o *GraphQL* permite buscar ambos em uma única requisição, incluindo a estrutura de relacionamento diretamente na *query* (consulta).

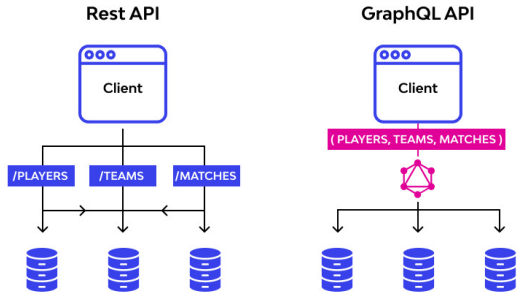


Figura 2.6: Comparação entre REST e GraphQL

A eficiência de rede é outra vantagem significativa. Como as respostas são moldadas pelas consultas, há menos transferência de dados redundantes. O *GraphQL* opera através de um único *endpoint* (como '*/graphql*'), onde todas as operações são tratadas. Isso contrasta com a proliferação de *endpoints* em REST, cada um dedicado a um recurso específico.

No cerne do *GraphQL* está um sistema de tipos definido no servidor por meio de um esquema. Esse esquema descreve todos os recursos, campos, relações e operações permitidas, garantindo validação tanto em fase de desenvolvimento quanto em execução. A seguir está um exemplo de esquema e uma troca de mensagem entre cliente e servidor:

```
type Query {
  musica(id: ID!): Musica
}

type Musica {
  id: ID!
  titulo: String!
  artista: Artista!
  lancamento: Lancamento!
  duracao: String!
  generos: [String!]!
}

type Artista {
  id: ID!
  nome: String!
  pais: String!
  seguidores: Int!
}

type Lancamento {
  id: ID!
  nome: String!
  tipo: String!
  anoDeLancamento: Int!
}
```

```
query {
  musica(id: "123") {
    titulo
    duracao
    artista {
      nome
      pais
    }
    lancamento {
      nome
      tipo
    }
  }
}
```

```
{
  "musica": {
    "titulo": "Purity Weeps",
    "duracao": "04:08",
    "artista": {
      "nome": "Invent Animate",
      "pais": "EUA"
    },
    "lancamento": {
      "nome": "Heavener",
      "tipo": "Álbum"
    }
  }
}
```

O versionamento é simplificado em comparação ao REST. Em vez de criar novas *URIs* (como '*/v2/musicas*'), o *GraphQL* permite adicionar novos campos ao esquema sem afetar consultas existentes. Campos obsoletos podem ser marcados como '*@deprecated*', permanecendo acessíveis para compatibilidade, mas ocultos em documentações. Isso facilita a evolução da API sem rupturas.

Entretanto, a flexibilidade tem custos. A curva de aprendizado é mais acentuada, exigindo familiaridade com conceitos como esquemas e *resolvers* (funções que conectam consultas aos dados). Além disso, consultas complexas podem sobrecarregar o servidor, já que a liberdade do cliente em solicitar dados aninhados ou profundos exige maior poder de processamento.

WebSocket

Enquanto estilos como REST e *GraphQL* operam sobre HTTP com um modelo de requisição-resposta, o *WebSocket* oferece uma abordagem alternativa para cenários que exigem comunicação contínua e em tempo real entre cliente e servidor. Protocolos baseados em HTTP, por dependerem de requisições individuais, tornam-se ineficientes em casos como *chats* ao vivo e jogos *multiplayer*. Isso ocorre porque os sistemas precisariam recorrer a técnicas como *polling* (consultas periódicas ao servidor) ou *long polling* (mantendo requisições abertas até que haja uma resposta), o que consome recursos desnecessários e introduz latência.

Embora o *WebSocket* seja um protocolo distinto, ele foi projetado para coexistir com HTTP. Ele utiliza a mesma porta ('80' para HTTP não criptografado e '443' para HTTPS) e permitindo que *firewalls* e *proxies* intermediários tratem o tráfego sem problemas. Além disso, a conexão inicia-se com um *handshake* HTTP, onde o cliente envia uma requisição especial incluindo o cabeçalho '*Upgrade: websocket*'. Se o servidor aceitar, responde com o status '*101 Switching Protocols*', convertendo a conexão para *WebSocket*, estabelecendo um canal de comunicação bidirecional, persistente e independente de HTTP entre cliente e servidor. Uma vez aberta a conexão, ambas as partes podem enviar dados a qualquer momento, sem a necessidade de repetir *handshakes* ou esperar por requisições.

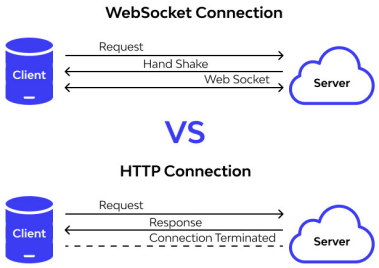


Figura 2.7: Comparação entre conexão WebSocket e conexão HTTP

O *WebSocket* é um protocolo de camada baixa, o que significa que não impõe formatos específicos para os dados trafegados. Ao contrário de REST (que geralmente usa JSON) ou SOAP (baseado em XML), as mensagens podem ser enviadas como texto simples, binários ou até mesmo estruturas personalizadas. Essa flexibilidade permite adaptação a diversos cenários, mas também exige que desenvolvedores definam suas próprias convenções para interpretar os dados. Para simplificar o desenvolvimento, bibliotecas como *Socket.IO* (para *JavaScript*) abstraem detalhes técnicos, oferecem reconexão automática e suporte a *fallbacks* para HTTP em casos de incompatibilidade.

Dois exemplos emblemáticos de uso do *WebSocket* são os jogos *Agar.io* e *Slither.io*, que o utilizam para atualizações em tempo real. Em *Agar.io*, cada movimento de um jogador é transmitido imediatamente para todos os demais, mantendo a sincronia do mapa. Já em *Slither.io*, a posição da serpente e a coleta de itens são processadas sem atrasos, criando uma experiência fluida.

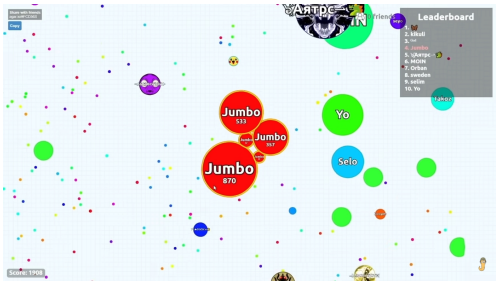


Figura 2.8: Jogo de navegador *Agar.io*

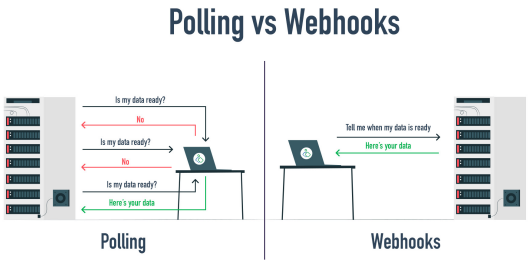


Figura 2.9: Comparação entre *polling* e *Webhooks*

Webhook

Enquanto APIs como REST e *GraphQL* operam em um modelo síncrono de requisição-resposta, e o *WebSocket* mantém uma conexão bidirecional contínua, os *Webhooks* oferecem uma abordagem complementar para cenários que demandam notificações assíncronas e *event-driven* (orientadas a evento). Imagine um serviço de *e-commerce* onde o sistema precisa atualizar o status de um pedido assim que o pagamento via PIX for confirmado. Utilizar técnicas como *polling* (consultas periódicas ao servidor) ou *long polling* (manter uma requisição aberta até que haja uma resposta) seria possível, mas ineficiente: geraria tráfego redundante e consumiria recursos do servidor e do cliente desnecessariamente. É nesse contexto que os *Webhooks* se destacam como solução.

Os *Webhooks* funcionam como *callbacks* HTTP, mecanismos que permitem a um servidor notificar automaticamente um cliente quando um evento específico ocorre. Para isso, o cliente registra previamente um URL de *callback* em um *endpoint* da API, informando ao servidor para onde enviar a notificação. Quando o evento desejado acontece, o servidor dispara uma requisição HTTP (geralmente 'POST') para o URL registrado, enviando dados relevantes sobre o evento. Essa abordagem elimina a necessidade de consultas repetitivas, transferindo a iniciativa da comunicação para o servidor. Por isso, os *Webhooks* são frequentemente chamados de APIs reversas ou APIs de *push*, em contraste com as APIs tradicionais baseadas em *pull*, onde o cliente sempre inicia a interação.

Um exemplo conhecido é o *GitHub*, que utiliza *Webhooks* para notificar sistemas externos sobre eventos em repositórios, como *push*, *pull request* ou *merge*. Ao configurar um *Webhook*, um desenvolvedor pode fazer com que cada *push* no repositório dispare uma requisição para um serviço de *deploy*, iniciando automaticamente a implantação da aplicação em produção. Essa integração contínua exemplifica como os *Webhooks* conectam sistemas heterogêneos de maneira desacoplada, permitindo que fluxos complexos sejam automatizados com mínimo esforço.

2.4 Conclusão

O artigo explorou desde APIs de sistema operacional e linguagens até estilos arquiteturais modernos como REST, *GraphQL* e *Webhooks*, demonstrando que sua força reside justamente na capacidade de serem usadas em conjunto. O exemplo do *Spotify* ilustra isso claramente: um simples clique no botão *play* envolve APIs de interface gráfica, acesso ao *hardware* do dispositivo, comunicação com servidores remotos e integração com serviços externos. Ao abstrair detalhes técnicos, elas permitem que desenvolvedores foquem em funcionalidades criativas, enquanto garantem eficiência, escalabilidade e segurança. Em um cenário tecnológico cada vez mais interconectado, compreender e integrar diferentes APIs torna-se vital.

2.5 Bibliografia

[1] Api // dicionário do programador - youtube. URL: `https://youtu.be/`

vGuqKIRWosk?si=umEoamAHZRe0Otd4.

[2] Apis for beginners - how to use an api (full course / tutorial) - youtube. URL: https://youtu.be/WXsD0ZgxjRw?si=zsrBGSIdyCh1_wgv.

[3] Api - wikipedia. URL: https://en.wikipedia.org/wiki/API.

[4] Api collections em java: fundamentos e implementação básica - devmedia. URL: https://www.devmedia.com.br/api-collections-em-java-fundamentos-e-implementacao-basica/28445.

[5] Built-in react apis - react. URL: https://react.dev/reference/react/apis.

[6] Top 6 most popular api architecture styles - youtube. URL: https://youtu.be/4vLxWqE94l4?si=dVwWvpkeA3E04J_5.

[7] Difference between rest api vs web api vs soap api explained - youtube. URL: https://youtu.be/2mqN7ZhDsUA?si=68YaT2LbSwbMZXGE.

[8] Rest vs soap | differences between soap and rest web services | nodejs training | edureka - youtube. URL: https://youtu.be/_fq8Ye8kodA?si=rapL5smiCPF6WTan.

[9] What is a soap api and how does it work? | postman blog. URL: https://blog.postman.com/soap-api-definition/#:~:text=SOAP%20(also%20known%20as%20Simple,text%2C%20JSON%2C%20and%20more.

[10] Restful apis in 100 seconds // build an api from scratch with node.js express - youtube. URL: https://youtu.be/-MTSQjw5DrM?si=wqC1AaDuNdw6p7Zn.

[11] What is rest api? examples and how to use it: Crash course system design 3 - youtube. URL: https://youtu.be/-mN3VyJuCjM?si=4JbcaOw4D4fpREE8.

[12] Rest - wikipedia. URL: https://en.wikipedia.org/wiki/REST.

[13] Fielding dissertation: Chapter 5: Representational state transfer (rest). URL: https://ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

[14] Openapi specification - wikipedia. URL: https://en.wikipedia.org/wiki/OpenAPI_Specification.

[15] Graphql explained in 100 seconds - youtube. URL: https://youtu.be/eIQh02xuVw4?si=H0VpdCOQaes73XSs.

[16] Graphql | a query language for your api. URL: https://graphql.org/.

[17] Websocket - wikipedia. URL: https://en.wikipedia.org/wiki/WebSocket.

[18] Websockets in 100 seconds beyond with socket.io - youtube. URL: https://youtu.be/1BfCnjr_Vjg?si=-8CYaAePGCv4GRm6.

[19] Websocket // dicionário do programador - youtube. URL: https://youtu.be/T4unNrKogSA?si=kS-kAPV5LA5PGVmg.

[20] Top 3 things you should know about webhooks! - youtube. URL: https://youtu.be/x_jjhcDrISk?si=5SI0YLS9EVvUv6k8.

[21] Webhooks (os melhores apps usam) // dicionário do programador - youtube. URL: https://youtu.be/2JHKIrComW0?si=Um2JWEHGTkUm4fmS.

Parte II

Projetos

Capítulo 3

MicroTorr: Peer to Peer file sharing inspired by BitTorrentV1

Rafael Gimenez Barbeta

Acesse o repositório do projeto clicando aqui

Read this paper in english

3.1 Sobre esse projeto

Implementação simplificada de uma rede de compartilhamento de arquivos *peer to peer* (P2P), inspirada no *BitTorrent V1*, escrita em *Go*. Vem com uma interface de linha de comando (CLI) feita com *Cobra* CLI.

Este projeto é capaz de executar uma simulação completa de um enxame mínimo de P2P, incluindo o servidor *Tracker*, o cliente *torrent* e a codificação de um arquivo para compartilhamento com este software. Ele implementa a maioria dos conceitos básicos do protocolo *BitTorrent*, tais como:

- Geração de arquivos de metadados "*.torrent*", com o algoritmo de codificação *bencode*
- Descoberta de *peers* com *links* de *Tracker* e *timeouts*
- Vários 'enxames' de *torrent* no mesmo *tracker*
- Estratégia de *download* "*Rarest piece first*"
- Mudança automática para modo de *seeding* uma vez que o *download* seja concluído
- Seleção de *peers* com base na estimativa de largura de banda e velocidade de conexão
- Verificação de integridade com o algoritmo de *hash* SHA1

Além disso, o *MicroTorr* inclui seus próprios recursos para facilitar a execução:

- Definição de velocidade máxima de *upload* e *download*
- Esperar por X quantidade de *seeders* e Y de *leechers* antes de começar o *download*
- Barra de progresso para o andamento do *download*

- Diferentes níveis (e cores) de verbosidade do programa
- Auto-completar comandos, fornecido pela *Cobra* CLI

3.2 Dependências e Recursos Auxiliares

Este projeto é feito inteiramente em *Go*, e implementa a maioria das funcionalidades do *BitTorrent* "do zero". Aqui está uma lista dos módulos de terceiros usados na construção da aplicação:

1. *Cobra* CLI
2. Bencode Encoding
3. Bandwidth Limiting
4. Progress Bar

Para referência, aqui está a especificação do protocolo *BitTorrent v1* que inspirou este projeto.

3.3 Sobre a Implementação

Este código é composto por 3 partes diferentes: um gerador e carregador de metadados (*.mtorrent*), *tracker* e o cliente *torrent*. Aqui está uma breve explicação de cada parte.

.mtorrent generator

```
# Create a .mtorrent
MicroTorr createMtorr test_file

# Show .mtorrent info
MicroTorr loadMTorr
```

Ele é responsável por gerar os metadados necessários para "dividir" um arquivo em pedaços e ajudar os *peers* a se conectarem. Usa *bencode*, assim como o *BitTorrent*, para codificar estes campos:

- *announce*: Contém a URL do *tracker*.
- *info*, que contém os valores: tamanho, nome do arquivo, tamanho dos pedaços, concatenação de *sha1* dos pedaços, *id_hash*.

Exceto por *id_hash*, que é um *sha1* do arquivo inteiro, todos os campos são os mesmos da estrutura de arquivo *.torrent* do *BitTorrent*.

Tracker

```
# Create a Tracker
MicroTorr tracker
```

Fornece apenas um *endpoint*: "GET */announce*" com os parâmetros:

- *info_hash*: *hash* SHA-1 de 20 bytes do dicionário *info* do arquivo *.mtorrent*. Neste caso, é o *id_hash*.
- *peer_id*: ID de 20 *bytes* gerado aleatoriamente.
- *ip*: IP do *peer*.
- *port*: O número da porta na qual o *peer* está ouvindo.
- *event*: O tipo de evento. Pode ser "*started*", "*stopped*", "*completed*", "*alive*".

A resposta é um JSON que contém apenas os endereços IP dos *peers*, suas portas de escuta e *IDs*. Uma vez que um *peer* faz essa solicitação, ele é adicionado à lista de *peers* do enxame de *info_hash*.

Os *peers* no *MicroTorr* sempre se conectam a todos os outros *peers* disponíveis. Eles devem continuar enviando solicitações GET, com os mesmos parâmetros e evento "*alive*". *Peers* que não enviam uma solicitação GET em um intervalo de 30 segundos serão considerados mortos e removidos da lista de *peers*.

Eles também podem alertar o *tracker* sobre uma saída voluntária, com "*completed*" quando tiverem todas as peças, e "*stopped*" quando o *download* for cancelado pelo usuário (enviando um sinal de interrupção).

Cliente *Torrent*

```
# Leech mode or download
MicroTorr download test_file.mtorrent

# Seed mode or upload
MicroTorr download test_file.mtorrent
-s test_file
```

Responsável por baixar peças de outros *peers* para obter o arquivo solicitado. Os *peers* podem se conectar ao enxame tanto em modo *leech* quanto em modo *seed*, sendo que o último tem o arquivo completo carregado e dividido na memória. Isso é composto por três componentes principais: *core*, *peerWire* e *trackerController*.

Tracker Controller

Envia o "*keep alive*" para o *tracker*. Também se comunica com o componente *Core* para informar o *Tracker* sobre o status "*completed*" ou "*stopped*".

Fornece uma maneira de recuperar dados do *tracker*.

Peer Wire

Responsável por gerenciar *sockets* brutos, conexões TCP, limitações de largura de banda, conectar novos *peers*, desconectar *peers*, serializar mensagens e enviar e receber dados. Ele é executado em uma *goroutine* separada e serve como uma abstração para o componente "*core*", permitindo que o *core* envie dados estruturados em um canal, com um *peerId* como destino, e receba uma resposta em outro canal. Todo o processo de lidar com a rede subjacente é ocultado por este componente.

Também realiza o *handshake* inicial para cada nova conexão e gera uma mensagem de controle para o *core* com o novo *peerId* a ser adicionado.

Core

Executa a lógica central do programa, como determinar qual peça baixar, lidar com as atualizações dos *peers* e suas próprias atualizações, além de enviar e receber peças.

No total, há 4 tipos de mensagens que este protocolo pode enviar:

- *have*: Anuncia aos outros *peers* uma nova peça baixada.
- *bitfield*: Indica todas as peças que um *peer* possui ou não. É enviado sempre que uma nova conexão é feita e apenas uma vez pelo proprietário.
- *request*: Usado para solicitar um bloco.
- *piece*: O bloco real da peça.

Request e *Piece* são tratados por duas *goroutines* separadas, chamadas *PieceRequester* e *PieceUploader*, respectivamente.

PieceRequester solicitará continuamente peças no canal com o componente *Peer Wire*, seguindo a estratégia de "*rarest piece first*". Um contador ocorre para determinar qual peça (ou peças) tem o mínimo de *peers* que as possuem. Uma vez encontradas essas peças, o *PieceRequester* escolherá aleatoriamente uma peça entre todas as peças raras e escolherá um *peer* que possui essa peça e também é o *peer* mais rápido conhecido.

Este último passo acontece cerca de 90% das vezes, e nos 10% restantes, ele escolhe um *peer* aleatório. Isso acontece para, esperançosamente, escolher um *peer* que seja mais rápido do que o esperado, já que a velocidade dos *peers* só é medida quando uma peça é baixada. Uma vez selecionada, a peça é solicitada e o *PieceRequester* aguarda sua chegada.

No outro extremo, uma *goroutine PieceUploader* atenderá a essa solicitação e sempre enviará a peça. Uma verificação de integridade SHA1 é feita para garantir que a peça é igual à esperada.

Quando todas as peças chegam, o *PieceRequester* chamará *AssemblePieces* para gravar o arquivo completo no disco e alerta o *Tracker Controller* de que o *download* foi concluído.

3.4 Instalação

Instalar a partir do código-fonte

Para instalar a partir do código-fonte, você precisará do compilador *Go*. Você pode seguir os passos da documentação oficial: *Go Download and install*

Em seguida, clone o repositório, instale o binário e adicione o *script* de *autocomplete* ao seu *shell* preferido. Supondo que você use o *shell bash*, estes são os comandos necessários para configurar o *MicroTorr*:

```
git clone
https://github.com/rafaelbarbeta/MicroTorr
cd MicroTorr
go install
# Check your PATH variable if this fails!
MicroTorr completion bash > microtorr
sudo cp microtorr
/etc/bash_completion.d/microtorr
```

Reinicie seu *shell*. Agora você pode executar o *MicroTorr* com autocompletar!

Instalar com pacote *debian* (Recomendado)

Basta baixar o pacote *.deb* em "*Releases*"e executar:

```
sudo dpkg -i microtorr.deb
```

Isso colocará o binário em */usr/local/bin* e também configurará o autocompletar no *shell bash*.

Sem necessidade de instalar *Go* ou qualquer outra coisa.

3.5 Executando o *MicroTorr*

Vamos construir um cenário simples de *torrent* para explorar o *MicroTorr* localmente. Primeiro, vamos criar um arquivo de 100M com o comando *dd* chamado *'freeware'*:

```
dd if=/dev/urandom of=freeware bs=1M
count=100
```

Em seguida, precisamos gerar um arquivo de metadados para *freeware*:

```
MicroTorr createMtorr freeware

# Checking if .mtorrent was
# created successfully
MicroTorr loadMtorr freeware.mtorrent
```

Uma vez que o arquivo de metadados seja criado, prosseguimos para criar um enxame contendo três *peers*: um "*seeder*" (que possui o arquivo completo) e dois "*leechers*" (que não possuem todos as peças). Eles se descobrirão com a ajuda de um *Tracker*:

```
MicroTorr tracker -v 2
```

O terminal ficará suspenso. Vamos abrir três novos terminais, um para o *seeder* e dois para os *leechers*.

Crie os diretórios "*home*" para *peer1* e *peer2*:

```
mkdir -p peer/peer1
mkdir -p peer/peer2
```

Acesse cada um deles nos diferentes terminais. Inicie-os especificando o arquivo *.mtorrent* junto com o modo de verbosidade, interfaces *loopback* e portas de escuta.

Leecher 1:

```
MicroTorr download ../../freeware.mtorrent
-i lo -p 1111 -v 2
```

Leecher 2:

```
MicroTorr download ../../freeware.mtorrent
-i lo -p 2222 -v 2
```

Agora você verá que o *tracker* está recebendo solicitações "*alive*" de ambos os *peers*. Você pode perceber que nada aconteceu, exceto pelas mensagens "*New Connection*", porque eles estão esperando que pelo menos um *seeder* entre no enxame. Nesta demonstração, queremos limitar a largura de banda do *seeder* para 3 MB/s no máximo, então, cedo ou tarde, os *leechers* poderão cooperar para alcançar uma melhor velocidade de *download*.

No terminal do *seeder*, execute:

```
MicroTorr download freeware.mtorrent
-i lo -p 3333 -v 2 -s freeware -u 3000
```

Agora observe a barra de progresso se preenchendo. Como você verá, a barra de progresso começa a se preencher lentamente e, com cerca de 50% do *download*, a velocidade aumenta rapidamente.

Nas estatísticas impressas, você pode ver que aproximadamente metade do arquivo foi baixado do *seeder*, e a outra metade foi baixada de outro *peer*. Isso acontece porque, seguindo a estratégia de *download* "*rarest piece first*", cada *leecher* baixa inicialmente um pedaço que o mínimo de *peers* atualmente possui no enxame. Isso leva a um comportamento interessante, onde cada *leecher* faz *download* de peças complementares. Depois disso, haverá um momento em que cada peça será de propriedade tanto do *seeder* quanto de um dos outros *peers*. Como o outro *leecher* agora também possui as peças restantes do arquivo, o *leecher* 1, por exemplo, pode tentar baixar uma peça do *leecher* 2. Ele notará que o *leecher* 2 tem uma velocidade de *upload* muito mais rápida do que o *seeder*, então ele fará o *download* das peças restantes dele. O inverso também é verdadeiro.

Se você quiser garantir que o arquivo recebido é realmente o mesmo que o do diretório raiz, basta executar este comando para o "*freeware*" em cada "diretório de *peer*":

```
sha1sum < freeware
```

Com sorte, a saída será a mesma para todos eles. E é isso!

Experimente com outros cenários também. Note que isso pode ser executado entre diferentes computadores na mesma LAN ou, se você tiver acesso a diferentes IPv4 públicos, pela internet!

3.6 Limitações

Observe que este código não funciona como cliente *BitTorrent*, e portanto não pode baixar *torrents* da internet. Em vez disso, é uma simulação didática de como o protocolo funciona internamente. Eu o fiz como uma forma de me desafiar com programação concorrente, redes e, claro, a linguagem *Go* e seu ecossistema. Além disso, eu queria aprender como esse protocolo realmente funcionava, pois eu desconhecia completamente seu funcionamento interno até construir este código. Tenha em mente que alguns recursos do protocolo *BitTorrent* foram retirados desta implementação, tais como:

- *Choking* e *Unchoking* Oportunista
- Comportamento *Tit-for-Tat* (embora ele se comporte dessa forma, de certa maneira)
- Slots de *Download* e *Upload*
- Retransmissão de peças quebradas (detectadas por *sha1*)
- Técnicas de NAT *traversal* para *peers* sob NAT
- DHT (Tabela *Hash* Distribuída)
- Criptografia
- *Links Magnet*
- E outros.

Por essas razões, ele não é otimizado para baixar arquivos realmente grandes pela internet. Veja o qbittorrent para isso (use com responsabilidade! XD).

3.7 Screenshots

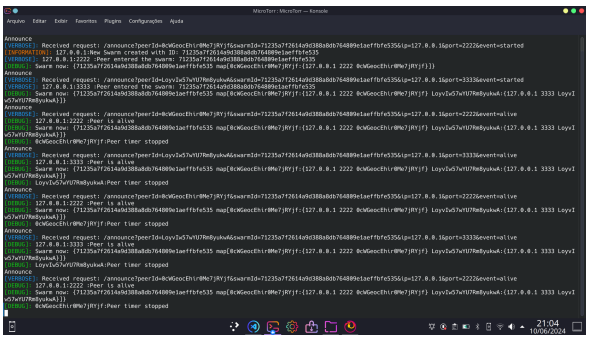


Figura 3.1: *Tracker* em funcionamento

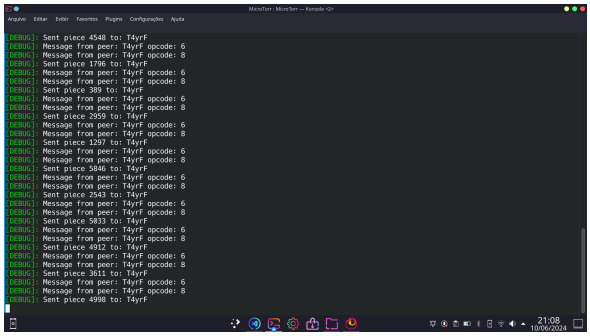


Figura 3.2: Comentários de *Debug* emitidos no console

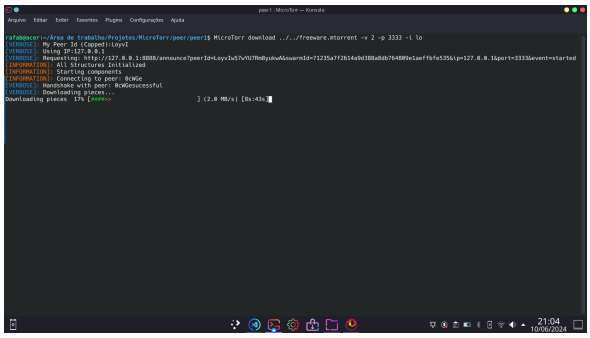


Figura 3.3: Leecher em funcionamento, fazendo o download de uma das peças do arquivo

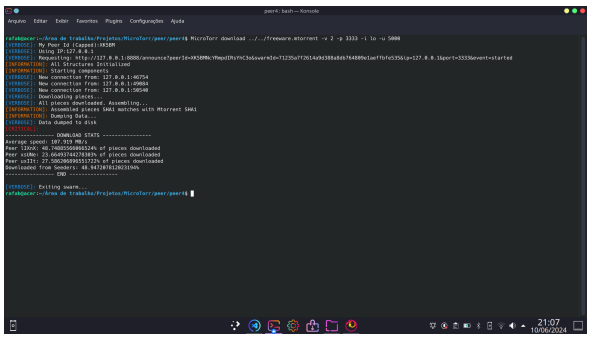


Figura 3.4: Estatísticas do download feito pelo enxame. Note que o arquivo foi baixado de diversas fontes diferentes ao mesmo tempo

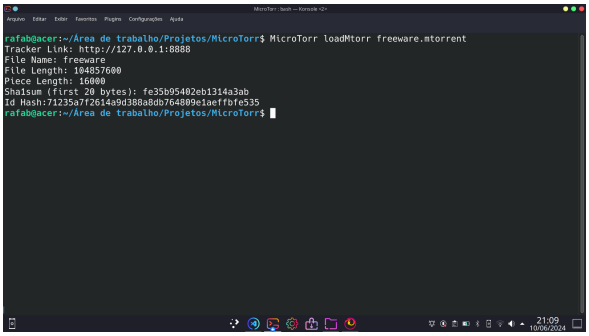


Figura 3.5: MicroTorr com informações do arquivo .torrent

3.8 Repositório

O repositório com os arquivos e demais informações sobre o projeto se encontra em:

`https://github.com/rafaelbarbeta/MicroTorr`

Capítulo 4

phishing-detector

André luís de Souza Oliveira
Marcos Antonio de Santana Júnior

Acesse o repositório do projeto clicando aqui

4.1 Conceito do projeto

Phishing é uma técnica de ataque cibernético que busca enganar pessoas para que revelem informações sensíveis, como senhas, dados bancários ou números de cartões de crédito, através de comunicações fraudulentas que aparentam ser confiáveis. Esses ataques geralmente ocorrem via *e-mails*, mensagens de texto (SMS), sites falsos ou até por meio de URLs aparentemente legítimas, que direcionam as vítimas a páginas que coletam suas informações pessoais. O atacante se passa por uma entidade confiável, como um banco, uma loja online ou uma instituição governamental, para ganhar a confiança da vítima e convencê-la a fornecer esses dados.

O *phishing* é considerado uma das formas mais comuns de cibercrime devido à sua simplicidade e à alta taxa de sucesso, afetando tanto indivíduos quanto empresas. Por isso, métodos avançados de detecção de *phishing*, como processamento de linguagem natural (*Natural Processing Language* - NLP), como também o uso de IA generativa para classificar automaticamente comunicações suspeitas, são essenciais para prevenir esses ataques e proteger usuários e organizações.

Este projeto foi desenvolvido com o objetivo de explorar e analisar o potencial da Inteligência Artificial Generativa na classificação de textos de diferentes tipos (*Email*, SMS, HTML, URL) como *phishing* ou não. A ferramenta utilizada para esse estudo foi a API do *Gemini* 1.5 Pro, que representa um dos avanços mais recentes em modelos de IA generativa. Para maximizar a precisão e a eficiência do modelo na tarefa de classificação, foi empregada a técnica de engenharia de *prompt Chain-of-Thought* (CoT), que permite ao modelo abordar o problema de forma sequencial, raciocinando passo a passo durante o processo de tomada de decisão. Essa abordagem não apenas melhora a compreensão da IA sobre o problema, mas também torna o processo de classificação mais transparente e explicável.

A aplicação deste método fornece *insights* sobre como a IA pode ser usada em cenários de cibersegurança, especialmente na detecção de tentativas de *phishing*, que continuam a ser uma das ameaças mais prevalentes no ambiente digital.

4.2 Pré-requisitos e recursos utilizados

O projeto foi implementado em linguagem *Python* para tratamento dos dados, chamada da API do *Gemini* e geração de gráficos. Tudo isso feito no ambiente de desenvolvimento do *Google Colab*, as bibliotecas e módulos utilizadas estão presentes no código. Os dados foram extraídos das seguintes fontes:

1. *Kaggle*: `https://www.kaggle.com/datasets/jackksoncsie/spam-email-dataset`

2. *Huggingface*: `https://huggingface.co/datasets/ealvaradob/phishing-dataset`

4.3 Passo a passo

Para se construir o projeto como um todo, seguimos os seguintes passos:

1. Estudo de artigos relacionados a classificação com o uso de IA Generativa;
2. Estudo de *notebooks Python* relacionados a engenharia de *prompt*;
3. Busca por *datasets*;
4. Manipulação dos *datasets*;
5. Randomização das entradas;
6. Conexão com a API do *Gemini*;
7. Construção do *prompt* usado com a API;
8. Execução do código e análise prévia do desempenho das chamadas à API;
9. Análise de resultados;

4.4 Instalação

Passos necessários para se recriar o projeto:

1. Clonar/abrir o *python notebook* via *Google Colab*;
2. Criação de chave de API Gemini. Isso é necessário para gerar as respostas da IA;
3. Instalação do *dataset* "**dataset_final_250.csv**", que está na pasta *datasets*;
4. (Opcional) Caso queira recriar a partir dos *datasets* iniciais, baixe os *datasets* referenciados na seção de Pré-requisitos. *Webs.json*, *urls.json* e *texts.json* da fonte *HuggingFace* e *emails.csv* do *Kaggle*.
5. Caso coloque o *dataset* no Drive, conceda acesso a ele para o *Google Colab* durante a execução do *notebook*. Nós fizemos dessa maneira.

4.5 Execução

Esses são os passos necessários para se reproduzir os resultados, a partir dos códigos presente na pasta '*notebooks*':

1. Importação de bibliotecas;
2. Conexão com *Gemini* API;
3. Montar ou importar *dataset*;
4. *Prompt* do modelo;
5. Execução do modelo;
6. Tratamento de resultados;
7. Análise de resultados;
8. Criação de gráficos.

Há duas maneiras de se executar o projeto, mas todas seguem os passos de execução acima. No número 3 - *Montar/importar dataset* - é preciso decidir qual opção seguir:

1. (Padrão) Seguir com o "**dataset_final_250.csv**", e executar passo-a-passo do *dataset* "**exec_model.ipynb**";
2. Criar sua própria entrada, utilizando o notebook "**build_dataset.ipynb**" e utilizar o resultado no notebook "**exec_model.ipynb**". Na seção de instalação há os detalhes sobre o que precisa ser feito caso se opte por essa opção.

Cada célula do notebook possuí mais informações sobre o que foi feito em cada etapa.

4.6 *Bugs*/problemas conhecidos

Sendo um projeto com IA gratuita, nossa expectativa já não estava tão alta em relação ao seu desempenho, tanto que nosso principal objetivo é analisar o potencial, ver se gratuitamente conseguimos obter um bom resultado. Ademais, a quantidade de chamadas a API é limitada, o que nos obrigou a reduzir o tamanho do *dataset*, devido a isso, não conseguimos trabalhar com novos reprocessamentos ou melhorias no modelo. A capacidade de generalização que nosso protótipo pode ter é questionada por esse ponto, a quantidade de amostras. Com mais testes poderíamos

ser mais assertivos em relação a sua efetividade, principalmente entendendo os pontos de melhoria após a análise de resultados feita no fim do projeto.

4.7 Referências

- Gráficos/matriz de confusão
- ChatSpamDetector: Leveraging Large Language Models for Effective Phishing Email Detection
- SecureNet: A Comparative Study of DeBERTa and Large Language Models for Phishing Detection

4.8 Imagens

- Cálculos métricas:

Essas métricas de avaliação nos ajudam a avaliar o desempenho do modelo. Tivemos **64% de acurácia de classificação**.

	precision	recall	f1-score	support
0	0.69	0.63	0.66	137
1	0.59	0.65	0.62	113
accuracy			0.64	250
macro avg	0.64	0.64	0.64	250
weighted avg	0.64	0.64	0.64	250

Figura 4.1: Métricas de avaliação do desempenho do modelo

- Matriz de confusão:

A matriz de confusão nos ajuda a entender como está a distribuição do nosso resultado. Utilizamos essas informação para guiar na geração dos gráficos e analisar especificamente entradas.

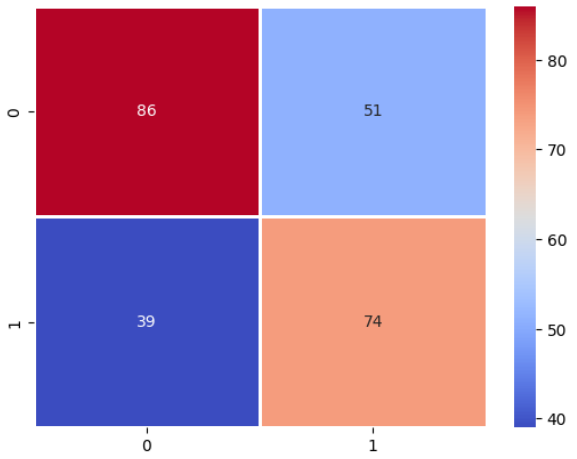


Figura 4.2: Matriz de confusão gerada com os resultados obtidos pelo modelo

- Gráficos gerados:

Distribuição de tipos

Buscamos adicionar diferentes tipos de entradas para diversificar o *dataset*, visto que, a maioria dos *datasets* são somente de *E-mails*. A classificação foi feita pelo modelo.

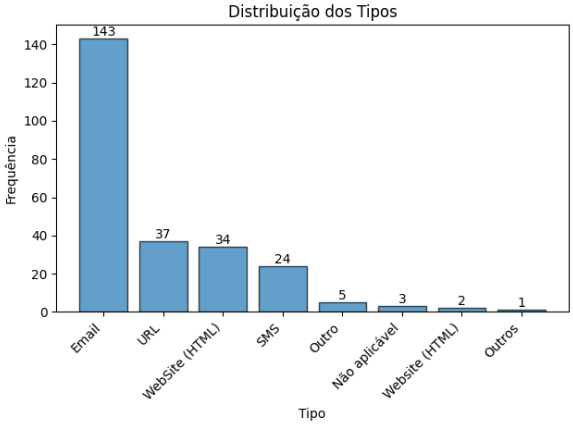


Figura 4.3: Tipos de mensagens analisadas pelo modelo

Distruibição de pontuação do phishing

Interessante observar que a API não nos retornou nenhuma resposta na faixa de 3-7, indicando

que ele sempre busca indicar que suas escolhas são assertivas.

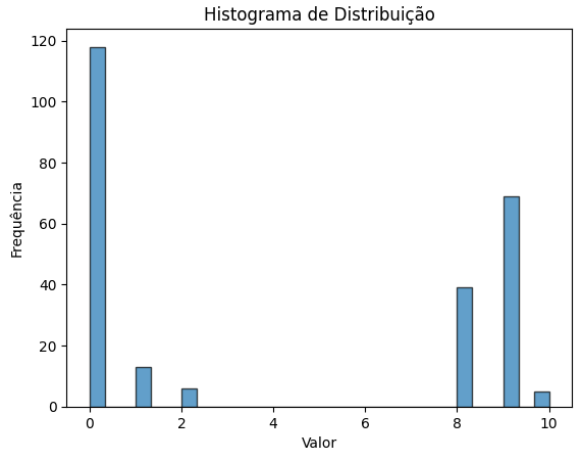


Figura 4.4: Distribuição das avaliações feitas pelo modelo

Nuvem de palavras

A nuvem de palavras nos ajuda a ter uma ideia de quais são os principais temas abordados nos textos presentes do *dataset*, palavras como "*Enron*" e "*Dynegy*" se referem a empresas do ramo de energia, outras palavras presentes na nuvem como "*energy*" e "*power*" nos ajuda a entender que há bastante palavras que se referem a utilização de energia. Lembrando, nosso *dataset* é randomizado, então, há diversos temas e a cardinalidade de temas/palavras é grande.

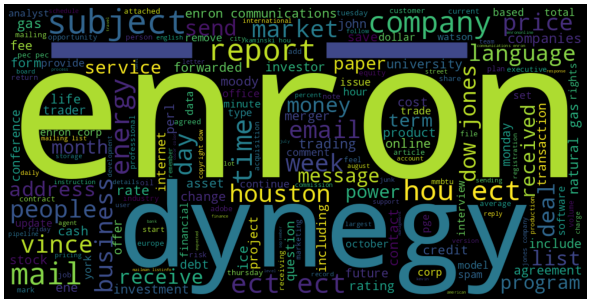


Figura 4.5: Nuvem de palavras gerada com o conteúdo analisado pelo modelo

Análise Falso Positivos e Falsos Negativos

Ao analisar as mensagens classificadas como falsos negativos e falsos positivos, ficou evidente que o modelo apresenta dificuldades específicas em classificar corretamente textos que combinam linguagem natural com trechos de código, como HTML e *JavaScript*. Isso demonstra uma limitação do modelo em processar e interpretar de forma eficaz conteúdos que misturam estruturas de linguagem humana com sintaxes formais de programação, evidenciando um desafio ao lidar com a fusão de texto e código em uma única mensagem.

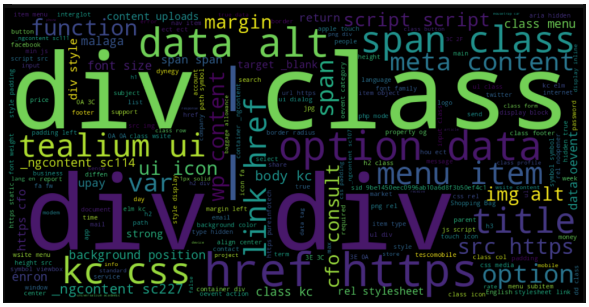


Figura 4.6: Nuvem de palavras gerada com o conteúdo analisado pelo modelo quando usado para análise de HTMLs e JavaScript. Percebe-se que a nuvem formada confunde o conteúdo das mensagens com as estruturas do texto

4.9 Repositório

O repositório com os arquivos e demais informações sobre o projeto se encontra em:

`https://github.com/akamarc0s/phising-detector`

Capítulo 5

Webscraping Word Searcher w/ Homoglyphs

Gustavo Eugênio de Souza Moraes

Acesse o repositório do projeto clicando aqui

5.1 Conceito do projeto

Este projeto tem como objetivo realizar uma busca por palavras em páginas da web considerando variações com homóglifos, ou seja, caracteres que se parecem visualmente, mas que pertencem a alfabetos diferentes (por exemplo, a letra latina "a" e a letra cirílica "а"). Isso permite detectar conteúdos maliciosos ou tentativas de enganar sistemas automatizados de moderação, que normalmente não reconhecem essas variações.

5.2 Pré-requisitos e recursos utilizados

Foi utilizado a linguagem Java 8 para desenvolver a implementação geral do projeto, além de utilizar-se [outro projeto], inacabado na época (2021), de base para definir quais seriam os homóglifos base:

5.3 Passo a passo

O projeto foi executado em algumas etapas breves:

1. Implementação do mecanismo de *webscraping*, utilizado para escanear páginas HTTP a serem analisadas pela aplicação. (HTMLPageReader e SafetyCheck)
2. Definir mapeamento estático de letras para homóglifos. (Homoglyphs)
3. Definir e implementar estratégia de comparação de letras com homóglifos. (StringComparator e HomoglyphsWordMatcher)
4. Integrar as implementações e receber os parâmetros de busca por *args*. (Application)

5.4 Instalação

Para compilar e rodar localmente o programa, é recomendado utilizar o *Maven* como compilador do projeto. Caso não esteja habituado com a ferramenta, é possível aprender um pouco mais aqui.

Para compilar o projeto, utilize o comando *mvn install* no diretório raiz do projeto (o mesmo diretório onde se encontra o arquivo pom.xml).

OBS: É recomendada uma versão Java 8 ou superior para executar o projeto.

5.5 Execução

Para executar o programa, basta executar o arquivo JAR no diretório *target* gerado pela execução do Maven com os argumentos necessários.

Exemplo de chamada

```
java -classpath ./target/Word-Search-w-Homoglyphs
-1.0-SNAPSHOT.jar Application <URL> <blacklist>
```

Sendo:

- *java* – O comando padrão para executar arquivos JAR gerados pelo compilador Java.
- *-classpath* – Uma flag que indica que passaremos o arquivo JAR que contém nossa classe executável.
- *./target/Word-Search-w-Homoglyphs-1.0-SNAPSHOT.jar* – O caminho padrão do executável do programa gerado pelo Maven.
- *Application* – Nome da nossa classe executável (Main).
- <*URL*> – O primeiro argumento, em formato URL (ex: `https://www.google.com/`) da página onde deve se realizar a busca.
- <*blacklist*> – Uma lista de palavras a serem buscadas, separadas por espaço.
  - **Obs:** Utilize aspas (*"palavra1 palavra2"*) para buscar por frases literais ao invés de palavras individuais.

5.6 Imagens/screenshots

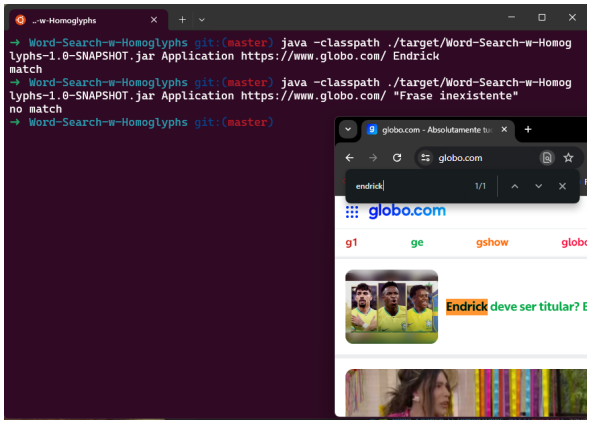


Figura 5.1: Imagem de exemplo 1, mostrando a busca pelo termo *"Endrick"* no site do GE

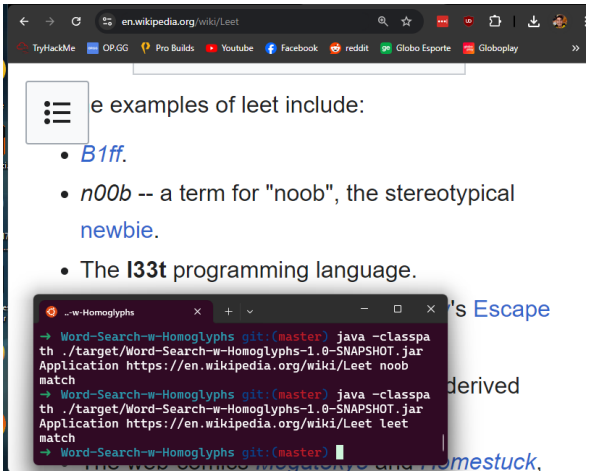


Figura 5.2: Imagem de exemplo 2, buscando pelo termo "*noob*", na página da *wikipedia* sobre *leet*

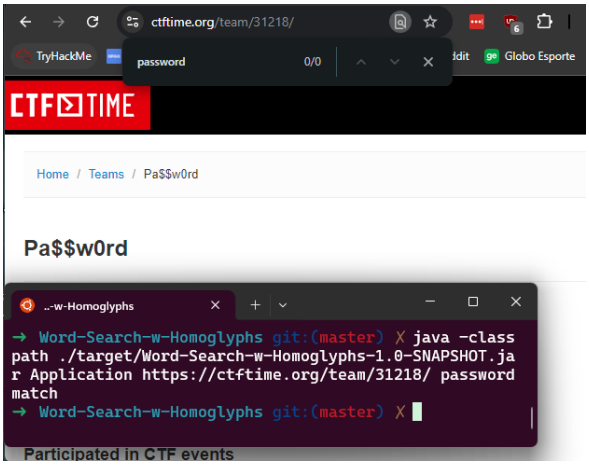


Figura 5.3: Imagem de exemplo 3, buscando pelo termo "*password*" em um site de CTF

5.7 *Bugs*/problemas conhecidos

- Como o projeto se baseia em um mapa de letra ↔ homóglifo para fazer a tradução nas buscas, o escopo de funcionamento do projeto fica delimitado à precisão e completude deste mapa. Caracteres não inseridos no mapa não serão considerados para a busca.
- Para acrescentar caracteres ao mapa de tradução, insira o caractere desejado ao final da

string correspondente à letra no mapa de tradução.

- A etapa de *webscraping* analisa todo o documento HTML em busca das palavras-chave, incluindo *tags* HTML, cabeçalhos, *scripts* e respectivos parâmetros. Atualmente não há disponibilidade de buscar apenas no *innerHTML*/corpo do documento.

5.8 Repositório

O repositório com os arquivos e demais informações sobre o projeto se encontra em:

`https://github.com/Gustavoesm/Word-Search-w-Homoglyphs`

Capítulo 6

Análise e predição de valores de ações

Renan Oliveira de Barros Lima

Acesse o repositório do projeto clicando aqui

6.1 Conceito do projeto

Este projeto foi desenvolvido com o intuito de expor conhecimentos de aprendizado de máquina na prática para fazer análise e predição de valores de ações da bolsa de valores. Primeiro fazemos exploração dos dados usando gráficos para visualizar a variação dos valores, média móvel, risco ao investir e descobrir a correlação entre as ações analisadas, para assim fazer o pré-processamento dos dados para treinar o modelo Long Short-Term Memory layer (LSTM) para prever seus possíveis valores ao longo do tempo.

6.2 Pré-Requisitos e recursos utilizados

- Linguagem *Python*
- *Jupyter Notebook*
- *Docker*
- *VScode* com a extensão *Dev Container*

6.3 Instalação

1. Clone o repositório na sua máquina e abra no *VSCode*:

```
git clone <URL\_DO\_REPOSITORIO>
```

2. Reabra o projeto no *Dev Container* quando solicitado pelo *pop-up* do *VSCode*.
3. O *Dev Container* pode pedir pra instalar *python* ou extensões do *Jupyter*/*Anaconda*

6.4 Execução

1. Abra o notebook *'main.ipynb'*.
2. Execute as células para rodar o código.

6.5 Passo a passo

Para realizar esse projeto seguir os seguintes passos:

1. Pesquisei sobre o assunto de ações e procurei soluções existentes.
2. A melhor solução que encontrei foi o notebook *Stock Market Analysis + Prediction using LSTM* no *Kaggle*, que é uma excelente plataforma para aprender ciências de dados, IA e *Machine Learning* com os notebooks feitos pela comunidade, e também nele é possível participar de competições.

3. Estudei o código e me aprofundei nos conceitos de ações e no modelo de Redes Neurais *Long Short Term Memory* (LSTM), links de artigos que li e vídeo podem ser encontrados nas referências.

4. Atualizei o código para funcionar com a versão mais recente do *Yahoo Finance*

5. Traduzi o notebook pra português e modifiquei o código para usar ações de empresas brasileiras conhecidas: Ambev (ABEV), Bradesco (BBD), AZUL (AZUL) e EMBRAER (ERJ)

6.6 Análise das Ações

Iremos explorar dados do mercado de algumas ações populares do Brasil (Ambev, Bradesco, Azul e EMBRAER). Utilizando *Yahoo Finance* para obter informações sobre ações e visualizaremos diferentes aspectos desses dados com *Seaborn* e *Matplotlib*. Também analisaremos algumas formas de avaliar o risco de uma ação com base em seu histórico de desempenho. Além disso, faremos previsões de preços futuros da Ambev utilizando o método *Long Short-Term Memory* (LSTM).

1. Como o preço da ação variou ao longo do tempo?

Nesta seção, abordaremos como lidar com a solicitação de informações sobre ações com pandas e como analisar atributos básicos de uma ação.

Preço de Fechamento

O preço de fechamento é o último preço pelo qual a ação é negociada durante o dia de negociação regular. O preço de fechamento de uma ação é o *benchmark* padrão usado por investidores para rastrear seu desempenho ao longo do tempo.

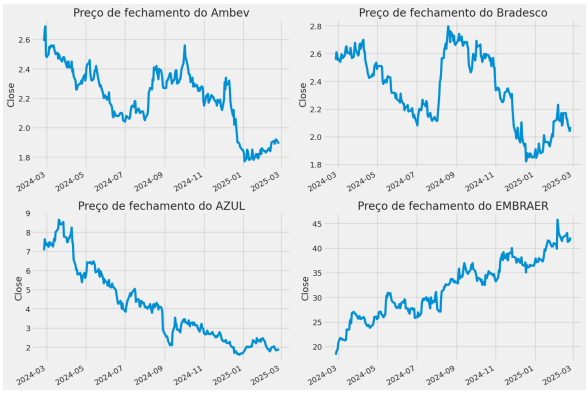


Figura 6.1: Gráfico do preço de fechamento para as empresas testadas no projeto

Volume de Vendas

Volume é a quantidade de um ativo ou título que muda de mãos ao longo de um período de tempo, geralmente ao longo de um dia. Por exemplo, o volume de negociação de ações se referiria ao número de ações de título negociadas entre sua abertura e fechamento diários. O volume de negociação e as mudanças no volume ao longo do tempo são entradas importantes para *traders* técnicos.

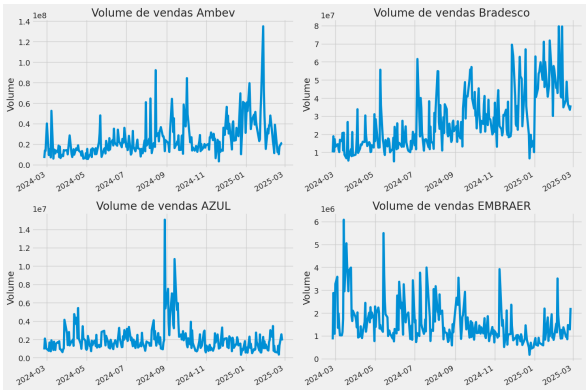


Figura 6.2: Gráfico do volume de vendas para as empresas testadas no projeto

2. Qual foi o retorno diário médio da ação?

A média móvel (MA) é uma ferramenta simples de análise técnica que suaviza dados de preço criando um preço médio constantemente atualizado. A média é tomada em um período de tempo es-

pecífico, como 10 dias, 20 minutos, 30 semanas ou qualquer período de tempo que o *trader* escolher. Por exemplo, se a média móvel de 10 dias cruzar acima da de 50 dias, pode ser um sinal de alta, e vice-versa.

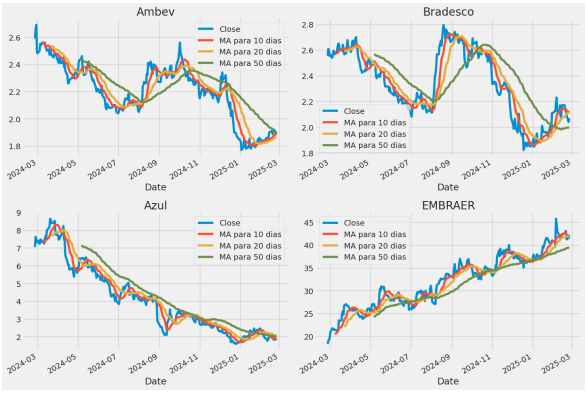


Figura 6.3: Gráfico da média móvel para as empresas testadas no projeto. Vemos no gráfico que os melhores valores para medir a média móvel são 10 e 20 dias porque ainda capturamos tendências nos dados sem ruído.

3. Qual foi a média móvel das diferentes ações?

Agora que fizemos algumas análises de base, vamos em frente e mergulhar um pouco mais fundo. Agora vamos analisar o risco da ação. Para fazer isso, precisaremos dar uma olhada mais de perto nas mudanças diárias da ação, e não apenas em seu valor absoluto.

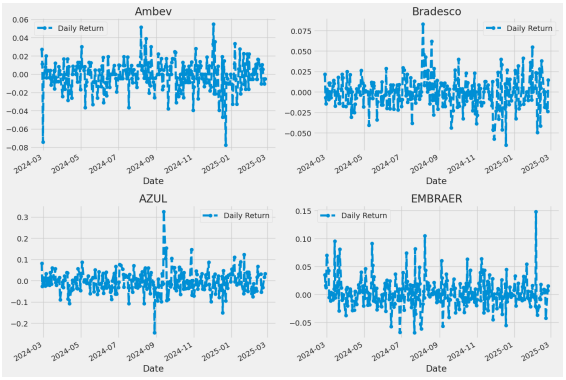


Figura 6.4: Gráfico do retorno diário para as empresas testadas no projeto

- Ambev: Os retornos diários variam entre -0.08 e 0.06, com alguns picos e quedas ao longo do tempo.
- Azul: Apresenta retornos mais voláteis, com variações entre -0.2 e 0.3.
- Bradesco: Retornos mais estáveis, variando entre -0.05 e 0.075.

Ótimo, agora vamos dar uma olhada geral no retorno médio diário usando um histograma.

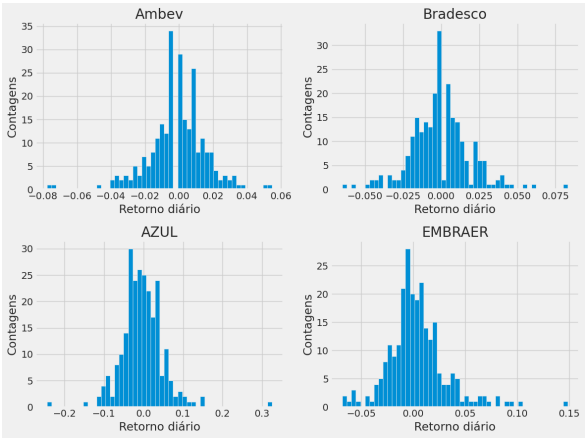


Figura 6.5: Gráfico do retorno diário, agora em formato de histograma, para as empresas testadas no projeto

- Ambev: A maioria dos retornos diários está concentrada em torno de 0.00 a 0.02, indicando uma volatilidade relativamente baixa.
- Azul e Embraer: Ambas mostram uma distribuição mais ampla, com retornos variando mais significativamente, o que pode indicar maior volatilidade.
- Bradesco: A distribuição de retornos parece mais concentrada, sugerindo menor volatilidade comparada às outras empresas.

4. Qual foi a correlação entre as ações analisadas?

Correlação é uma estatística que mede o grau em que duas variáveis se movem em relação uma à outra, que tem um valor que deve estar entre

−1,0 e +1,0. A correlação mede a associação, mas não mostra se x causa y ou vice-versa — ou se a associação é causada por um terceiro fator.

Segue gráfico de correlação, para obter valores numéricos reais para a correlação entre os valores de retorno diário das ações.

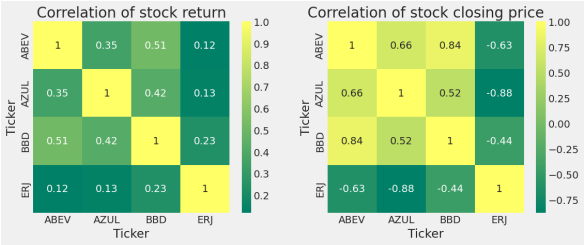


Figura 6.6: Gráfico de correlação entre as ações das empresas testadas no projeto, para o retorno diário e preço de fechamento, respectivamente. Vemos aqui numericamente e visualmente que a Bradesco e a Ambev tiveram a correlação mais forte de retorno diário de ações.

5. Quanto valor está em risco ao investir em uma ação específica?

Há muitas maneiras de quantificar o risco. Uma das maneiras mais básicas de usar as informações que coletamos sobre retornos percentuais diários é comparar o retorno esperado com o desvio padrão dos retornos diários.

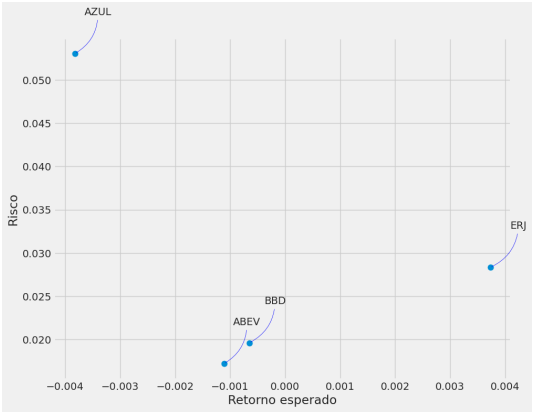


Figura 6.7: Gráfico de desvio padrão por retorno esperado. Podemos ver que a AZUL possui um maior risco.

6. Prevendo o preço de fechamento da Ambev S.A. usando LSTM

Treinamos o modelo LSTM com as ações da Ambev e visualizando os resultados graficamente, podemos ver que o modelo conseguiu prever aproximadamente os valores válidos das ações.

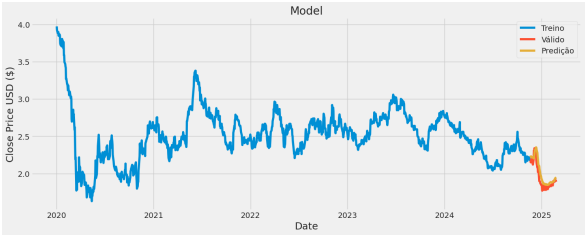


Figura 6.8: Previsão do modelo LSTM para o preço de fechamento das ações da Ambev.

6.7 *Bugs*/problemas conhecidos

- O modelo LSTM pode não prever com precisão os valores das ações devido à natureza volátil do mercado de ações.

6.8 Referências e links úteis para se aprofundar

- Arquitetura de Redes Neurais Long Short Term Memory (LSTM)
- Vídeo Long Short-Term Memory (LSTM), claramente explicado
- Vídeos sobre cuidados ao usar LSTM ao prever valores de ações:
  1. Predicting Stock Prices with LSTMs: One Mistake Everyone Makes (Episode 16)
  2. Stock Price Prediction with Machine Learning Mistakes: Prices As Inputs (Episode 20)
  3. Common Mistakes in Stock Price Prediction: Prices As Targets (Episode 21)
- Notebook usado como Base Stock Market Analysis + Prediction using LSTM
- Biblioteca do Yahoo finance para obter dados financeiros

6.9 Repositório

O repositório com os arquivos e demais informações sobre o projeto se encontra em:

`https://github.com/Renan04lima/stock-market-prediction`