UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

JOÃO VICTOR FELISMINO FREIRES

Implementação de um Depurador Distribuído e os Desafios do Rastreamento de Dados

RIO DE JANEIRO
2023

JOÃO VICTOR FELISMINO FREIRES

Implementação de um Depurador Distribuído e os Desafios do Rastreamento de Dados

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Profa. Silvana Rossetto

RIO DE JANEIRO
2023

CIP - Catalogação na Publicação

F866i

Freires, João Victor Felismino
Implementação de um Depurador Distribuído e os Desafios do Rastreamento de Dados / João Victor Felismino Freires. -- Rio de Janeiro, 2023.
64 f.

Orientadora: Silvana Rossetto.
Trabalho de conclusão de curso (graduação) - Universidade Federal do Rio de Janeiro, Instituto de Computação, Bacharel em Ciência da Computação, 2023.

1. sistemas distribuídos. 2. depuradores. 3. rastremento de dados. 4. monitoramento de sistemas. I. Rossetto, Silvana, orient. II. Título.

Elaborado pelo Sistema de Geração Automática da UFRJ com os dados fornecidos pelo(a) autor(a), sob a responsabilidade de Miguel Romeu Amorim Neto - CRB-7/6283.

JOÃO VICTOR FELISMINO FREIRES

Implementação de um Depurador Distribuído e os Desafios do Rastreamento de Dados

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 4 de setembro de 2023

BANCA EXAMINADORA:

Silvana Rossetto, D.Sc.
Instituto de Computação/UFRJ
(orientadora)

Carolina Gil Marcelino, D.Sc.
Instituto de Computação/UFRJ

Hugo Musso Gualandi, D.Sc.
Instituto de Computação/UFRJ

Dedico este trabalho a todos aqueles a quem esta pesquisa possa ajudar de alguma forma. Ele é dedicado também a todos os professores e colegas do DCC-UFRJ, pois direta e indiretamente fizeram parte da jornada para que ele fosse um dia possível.

AGRADECIMENTOS

Gostaria de agradecer a toda minha família, que sempre esteve presente, me apoiou e se mostrou responsável. Agradeço especialmente a minha noiva Beatriz, que é essencial na minha vida e me deu todo o suporte, principalmente emocional, para que eu conseguisse passar por tempos difíceis e retomar a busca pela minha melhor versão (que atualmente ainda segue em alfa).

Agradeço também aos meus amigos, que em momentos de estresse me proporcionaram um pouco de descontração e me fizeram recuperar as energias para conseguir retornar aos trilhos. Em particular, agradeço ao Diogo Nolasco por toda sua amizade, suas críticas sinceras e sua disposição de ajudar até quando não é preciso.

Finalmente, agradeço à minha orientadora Silvana, que contribuiu bastante com conselhos, diretrizes, material de apoio, opiniões e debates que enriqueceram o projeto como um todo, tornando-o possível. Nossos debates não se limitaram apenas a temas técnicos, o que certamente permitiu o meu crescimento não apenas como profissional, mas também como pessoa.

"There is a race between the increasing complexity of the systems we build and our ability to develop intellectual tools for understanding their complexity. If the race is won by our tools, then systems will eventually become easier to use and more reliable. If not, they will continue to become harder to use and less reliable for all but a relatively small set of common tasks. Given how hard thinking is, if those intellectual tools are to succeed, they will have to substitute calculation for thought."

Leslie Lamport

RESUMO

Este trabalho aborda o projeto, o desenvolvimento e as técnicas de análise de ferramentas de depuração e rastreamento de dados em sistemas distribuídos. Discutimos o planejamento e a implementação de ferramentas de coleta de dados de *logs*, como o Dapper e o Pivot Tracing, exibindo os desafios e a multidisciplinaridade envolvidas nos processos de coleta, armazenamento e análise de dados massivos gerados por sistemas distribuídos. Em seguida, estudamos os depuradores, a importância e os desafios de depurar um sistema distribuído. Analisamos um depurador distribuído existente, o Friday, e implementamos o nosso próprio depurador distribuído com o foco em uma solução de finalidade didática. Por fim, relatamos os cenários de testes aos quais a implementação do depurador distribuído foi submetido e os resultados obtidos. Concluímos com as possíveis melhorias nesse depurador e a ligação entre análise de dados gerados por sistemas distribuídos e a depuração dos mesmos.

Palavras-chave: sistemas distribuídos; depuradores; rastremento de dados; monitoramento de sistemas.

ABSTRACT

This work addresses the design, development, and analysis techniques of tools for debugging and tracking data in distributed systems. We cover the design and implementation of log data collection tools such as Dapper and Pivot Tracing, showing the challenges and multidisciplinarity in the data life-cycle process that involves the collection, storage, and analysis of massive data generated by distributed systems. Next, we study debuggers, the importance and challenges of debugging a distributed system. We analyze an existing distributed debugger called Friday and implement our own distributed debugger focused on a didactic implementation. Finally, we report the test scenarios to which the distributed debugger implementation was submitted and the results obtained. We conclude with the possible improvements in this debugger and the link between an analysis of data generated by distributed systems and their debugging.

Keywords: distributed systems; debuggers; data tracing techniques; systems monitoring.

LISTA DE ILUSTRAÇÕES

LISTA DE CÓDIGOS

LISTA DE ABREVIATURAS E SIGLAS

GDB	GNU Debugger
GNU	GNU's Not Unix
ID	Identifier
LAN	Local Area Network
NTP	Network Time Protocol
pdb	The Python Debugger
RPC	Remote Procedure Call
UI	User Interface ou User interface design

SUMÁRIO

1 INTRODUÇÃO

Serviços de Internet com públicos-alvos distintos, desde aplicativos de celular até processamento de dados científicos, têm tirado proveito das arquiteturas de sistemas distribuídos. Um sistema distribuído, em termos gerais, é um conjunto de computadores independentes que aparece ao usuário como um único sistema coerente (STEEN; TANENBAUM, 2017). Esses sistemas têm como objetivo melhorar o desempenho das aplicações e/ou atender a um propósito específico dessas aplicações, como por exemplo, permitir que seus usuários estejam em localizações geográficas distintas. Muitos desses serviços, por mais básicos que sejam, utilizam diferentes componentes, como, por exemplo, um sistema de autenticação e um banco de dados para armazenamento de dados (PINHEIRO; APARICIO; COSTA, 2014).

As formas de projetar, separar, compor e/ou atender uma necessidade especial de um sistema distribuído não são únicas e são dependentes dos requisitos que devem ser atendidos (STEEN; TANENBAUM, 2017). Em muitas situações, podemos ter um número variado de computadores reais que atendem a um único objetivo. Por exemplo, podem variar de um único servidor a milhares de nós servidores envolvidos no atendimento de requisições web ao usar o mecanismo de busca Google (SIGELMAN et al., 2010).

À medida que esses sistemas crescem, identificar o caminho percorrido por um dado através do sistema e analisar esse caminho é uma tarefa importante que pode ser necessária em diferentes cenários, desde encontrar formas de otimizar o sistema até encontrar e depurar *bugs*. Para esse fim, diversas abordagens podem ser adotadas. A abordagem mais básica consiste em guardar o registro de cada ponto do sistema em que um dado foi processado e procurar por esse dado no histórico de cada um dos pontos. Essa abordagem simples é impraticável em grandes sistemas, onde o histórico de requisições pode gerar grandes volumes de dados, condição que pode facilmente impactar o desempenho do sistema e deixá-lo inviável de ser analisado, tornando difícil identificar a rota que um dado percorreu (JIA et al., 2017).

Por outro lado, temos abordagens mais sofisticadas que variam de acordo com o sistema e com cada necessidade de avaliação. Comumente, essas soluções requerem mudanças em parte da arquitetura atual do sistema e/ou nos protocolos de comunicação para encaminhar adequadamente os dados de rastreamento, como, por exemplo, o Google Dapper (SIGELMAN et al., 2010) e o Pivot Tracing (MACE; ROELKE; FONSECA, 2015), que são arquiteturas de *logs* e análises de *logs*, e ambas são inseridas em uma arquitetura principal. Outras ferramentas, como o banco de dados Cassandra (LAKSHMAN; MALIK, 2010), são usadas para lidar com grande volume de dados, principalmente quando se tem uma elevada quantidade de operações de escrita, que pode ser o caso de sistemas de armazenamento de *logs*.

Essas soluções dependem do tipo de análise a ser feita. Por exemplo, um administrador do sistema pode estar interessado em identificar problemas de latência e de desempenho e identificar quais requisições tiveram tais problemas. Um desenvolvedor, por outro lado, quer saber como o sistema se comportou ao receber um dado, se esse comportamento foi correto e se todas as informações foram transmitidas de maneira adequada.

1.1 MOTIVAÇÃO

A motivação para este trabalho surge da necessidade de facilitar o ensino de sistemas distribuídos e entender o fluxo de dados nesse contexto, especialmente em termos de depuração. A depuração em sistemas distribuídos pode ser desafiadora devido à complexidade de interações entre componentes e ao comportamento distribuído do sistema. Portanto, surge a necessidade de desenvolver uma ferramenta de depuração que seja prática e didática, visando tanto aplicações educacionais quanto industriais.

1.2 OBJETIVOS

O objetivo principal deste trabalho é desenvolver uma ferramenta de depuração para sistemas distribuídos que facilite o processo de identificação e correção de problemas. Essa ferramenta será projetada com uma abordagem didática, tornando-a adequada para fins educacionais, permitindo que estudantes e desenvolvedores compreendam melhor o funcionamento de sistemas distribuídos durante o processo de depuração.

Como objetivos específicos, temos:

1. Analisar e compreender os desafios e complexidades da depuração em sistemas distribuídos.
2. Projetar e implementar uma ferramenta de depuração que possibilite a visualização do fluxo de dados entre os componentes distribuídos.
3. Demonstrar o processo de depuração de sistemas distribuídos utilizando a ferramenta desenvolvida.

Ao atingir esses objetivos, esperamos contribuir para a área de sistemas distribuídos, tornando a depuração mais acessível e compreensível para estudantes e profissionais que lidam com essa tecnologia. Além disso, o trabalho visa promover um meio de compreensão do fluxo de dados em sistemas distribuídos.

1.3 METODOLOGIA

A metodologia utilizada neste trabalho consistiu em primeiro construir aplicações alvo baseadas na arquitetura clássica de cliente/servidor. Essas aplicações foram projetadas

com a intenção de criar cenários representativos que permitissem a identificação dos problemas a serem abordados pelo depurador. A heterogeneidade dessas arquiteturas de testes nos ajudou a definir o escopo da ferramenta e a compreender os possíveis fluxos de informações presentes em sistemas distribuídos.

Com as aplicações alvo e as demandas de depuração levantadas, partimos para o projeto e a implementação do depurador. Durante esse processo, foram realizados experimentos para auxiliar o desenvolvimento e para validar a eficácia da ferramenta em cenários reais. Os resultados dos experimentos foram analisados e utilizados para aprimorar a ferramenta, buscando torná-la uma solução prática e didática para a depuração de sistemas distribuídos.

1.4 RESULTADOS

O depurador proposto foi implementado na linguagem de programação Python e apresenta a funcionalidade de suportar múltiplas conexões com os serviços sendo depurados. Esse recurso permite relacionar diferentes conexões por meio de identificadores, simulando o caminho percorrido por um determinado dado através dos múltiplos sistemas sendo depurados. Cada conexão representa um serviço ou componente distribuído que se conecta ao depurador, permitindo a visualização do fluxo de dados entre eles.

Nos testes realizados, abrangendo componentes distribuídos simples e componentes mais complexos (que contém múltiplas interações entre os sistemas), o depurador atendeu aos objetivos colocados inicialmente. Durante os experimentos, o depurador demonstrou estabilidade e nos permitiu acessar e analisar os ambientes de testes, conforme o esperado.

A funcionalidade de múltiplas conexões do depurador é um aspecto crucial para compreender e depurar sistemas distribuídos complexos, pois possibilita o rastreamento do fluxo de dados entre diferentes componentes em um cenário realista. Essa funcionalidade é abordada com mais detalhes no capítulo 33.

1.5 ORGANIZAÇÃO DO TEXTO

Além desta introdução, este texto está organizado nos seguintes capítulos:

- **Capítulo 2:** Discussão sobre o fluxo de dados em sistemas distribuídos, os desafios para entender esse fluxo e a importância disso no contexto de depuração.
- **Capítulo 3:** Apresentação de como um depurador pode ser aplicado em um sistema distribuído e a implementação do depurador de aplicações distribuídas com Python.
- **Capítulo 4:** Demonstração dos resultados obtidos com o depurador, por meio da descrição de cenários de testes e análise dos resultados.

- **Capítulo 5:** Conclusão com considerações finais, revisão dos resultados, discussão das limitações e sugestões para trabalhos futuros.

2 COLETA DE TRACING

Neste capítulo, discutiremos em profundidade os metadados gerados sobre a aplicação, focando principalmente nas técnicas de *tracing* e *tracking*. Embora mencionamos brevemente os metadados relevantes coletados sobre o hardware na seção 2.3, nosso principal interesse são os metadados que dizem respeito a aplicação, cuja variedade pode ser adaptada para atender às suas necessidades específicas. A identificação dos pontos de avaliação e a distinção entre os objetivos dos dados coletados – seja para otimizar o desempenho do sistema ou para compreender o comportamento e as interações do usuário com o sistema – são questões essenciais da nossa investigação sobre como avaliar o funcionamento de um sistema distribuído. Optamos por utilizar o termo "metadados" de forma sinônima a "dados" com o propósito de concisão. Entretanto, é válido ressaltar que quando se alude a um dado concreto, esta distinção será devidamente elucidada.

Para começar, vamos diferenciar os termos *tracing* e *tracking*. Essas duas técnicas, embora distintas, são frequentemente associadas devido à sua terminologia e padrões compartilhados. A principal diferença entre elas reside na direção temporal em que operam. No *tracking*, o caminho de um objeto é rastreado a partir de um ponto inicial até a sua posição atual, enquanto no *tracing*, o caminho é seguido de trás para frente, do ponto atual do objeto até o local onde foi inicialmente gerado. Em nosso texto, nos concentraremos principalmente na técnica de *tracing*, embora o termo *tracing* seja usado para se referir a ambas as técnicas, visto que nosso foco está na coleta de dados, e não na direção de sua coleta. A escolha do termo *tracing* se justifica pelo fato de que, em aplicações de sistemas distribuídos, o ponto final das requisições, geralmente o usuário, é normalmente o ponto de partida para a análise de erros.

O rastreamento de código, também conhecido como *tracing*, é uma técnica que permite ao programador rastrear manualmente a execução de um programa ou segmento de código, essa técnica é uma extensão do termo abordado préviamente por suas similaridades, porém a técnica se aprofunda em entender a execução de um programa. Isso é realizado com a ajuda de alguma ferramenta que permite observar as alterações nos valores das variáveis durante a execução do programa e determinar sua saída. Em sistemas mais complexos, o *tracing* pode envolver o acompanhamento de aspectos mais genéricos, como a execução de uma chamada de função. Nesses casos, a ausência de um padrão ou fórmula específica para realizar esse rastreamento implica a necessidade de técnicas que ajudem o programador a identificar o código responsável por uma funcionalidade específica ou associado a um defeito específico.

Em sistemas distribuídos, a técnica de *tracing* distribuído é usada para rastrear e observar a utilização de recursos à medida que as requisições se deslocam pelo sistema. Os dados coletados por meio dessa técnica permitem identificar padrões de uso, falhas de

software ou de hardware e problemas de desempenho do sistema. As técnicas modernas de *tracing* têm como objetivo fornecer aos usuários um meio mais sofisticado de coletar, acessar e analisar os registros de execução em um sistema distribuído, correlacionando eventos que ocorreram em diferentes componentes. Essas informações podem ser úteis em diversos níveis do sistema: para gerar conhecimento sobre o sistema, para depurar e reproduzir falhas de aplicações com sucesso e para indicar a necessidade de expansão do sistema, detectar horários de pico de uso ou tentativas de ataques.

Apesar das muitas vantagens que a coleta desses dados pode oferecer, existem desafios significativos que acompanham esse processo. Entre esses desafios estão: a necessidade de desenvolver métodos para correlacionar os eventos em um sistema; entender quais partes do sistema devem ser monitoradas e quais não; e a questão de se os dados gerados podem ser armazenados e analisados posteriormente.

Neste capítulo, abordaremos diversos pontos de avaliação relacionados à coleta e análise de dados em sistemas distribuídos. Na seção de "Pontos de avaliação", discutiremos a importância de entender e priorizar quais eventos devem ser avaliados, bem como a compreensão da topologia e interação dos componentes distribuídos para identificar áreas de melhoria e definir pontos de coleta de dados. Em seguida, na seção de "Ferramentas e infraestruturas de registros de eventos", exploraremos duas ferramentas sobre rastreamento distribuído: Dapper e Pivot Tracing, e como suas contribuições influenciaram outras ferramentas contemporâneas. Já na seção de "Monitoramento de Infraestrutura", discutiremos o monitoramento do sistema e da rede em sistemas distribuídos, bem como a importância da análise de *tracing*. Por fim, na seção de "Armazenamento de Informações", abordaremos os desafios e as estratégias para armazenar dados em sistemas distribuídos, considerando a distribuição das máquinas e a busca por eficiência no acesso aos registros de eventos.

2.1 PONTOS DE AVALIAÇÃO

Uma das principais questões a serem consideradas ao coletar dados sobre um sistema é entender e priorizar quais eventos devem ser avaliados, quando e onde os dados coletados devem ser armazenados. Compreender a topologia do sistema e a interação entre seus componentes é crucial para identificar possíveis áreas de melhoria, tanto nos componentes individuais quanto no sistema distribuído como um todo. Além disso, a compreensão da organização e interação dos componentes distribuídos é fundamental para a implementação de pontos de coleta de dados. Entretanto, é importante notar que essas informações podem influenciar positiva ou negativamente o sistema: enquanto os dados coletados podem ajudar a aprimorar o entendimento do uso do sistema, a coleta de dados pode sobrecarregar o sistema e afetar sua funcionalidade ou adicionar complexidade excessiva, tornando difícil a manutenção ou expansão do sistema.

Na sequência, dividiremos nossa discussão sobre os pontos de avaliação em várias subseções. Na subseção 2.1.1, falaremos sobre as métricas destinadas a identificar gargalos no sistema. Em 2.1.2, discutiremos as métricas que têm como objetivo compreender o funcionamento do sistema. Em 2.1.3, abordaremos as informações adicionais que podem ser relevantes para a coleta de dados principal e que podem complementar — e às vezes ser essenciais — as ferramentas de coleta de *tracing*. Finalmente, em 2.1.4, discutiremos a sincronização de relógio, uma técnica que pode ser usada para relacionar eventos em sistemas distribuídos e que é crucial na análise de como os eventos ocorrem em um sistema específico.

2.1.1 Métricas de Desempenho

Ao coletar dados de desempenho, estamos interessados em diversos aspectos relacionados ao tempo de acesso ao sistema. Para melhorar o desempenho de um sistema distribuído, precisamos coletar uma amostra de dados suficientemente grande que nos permita compreender quais partes do sistema necessitam de otimização. Os problemas de desempenho podem estar associados a vários fatores, incluindo latência, baixa vazão do sistema, falhas de hardware, entre outros (TIERNEY et al., 1998).

Quando a análise tem como foco a obtenção de informações sobre acesso, disponibilidade e desempenho de um recurso, essa coleta tende a gerar mais dados genéricos do que informações específicas de um evento. Esta quantidade de dados pode ser reduzida, armazenando apenas estatísticas e métricas desses eventos, ao invés dos dados brutos.

Por exemplo, a avaliação do tempo de execução de um serviço pode ser realizada de forma probabilística, com coletas periódicas dos dados dos eventos. Por outro lado, para avaliar a latência de um sistema, é necessário coletar dados referentes à conexão de rede do usuário do recurso e/ou à sua localização. Adicionalmente, podemos restringir a coleta de dados aos recursos que apresentaram algum tipo de falha, seja de aplicação ou de sistema. Desta forma, o foco é resolver falhas existentes, minimizando os dados armazenados e o tempo de resposta do sistema.

2.1.2 Métricas de Comportamento

Um sistema que adota um protocolo de coleta de eventos de erros necessita de informações mais detalhadas e específicas que possam impactar na sua correção. A coleta de eventos ocorridos em um sistema distribuído, visando ter uma visão aprofundada do comportamento do sistema e do fluxo de dados através dele, acarreta em maior gasto de armazenamento e processamento, podendo impactar tanto a aplicação como tornar inviável a visualização desses dados(DAVIDSON; WALL; MACE, 2023)(PARKER et al., 2020). Este tipo de coleta é amplamente utilizado para a reconstrução do grafo de eventos em um sistema. Ao avaliar o armazenamento ou coleta de dados de um comportamento

geral do sistema, é necessário calcular o impacto dessa abordagem e das ferramentas que a implementam. Um exemplo de uma métrica de comportamento seria o número de vezes que uma requisição fez uma chamada específica de função ou os eventos em que uma requisição retornou um erro específico.

2.1.3 Coleta de Metadados

Metadados são, essencialmente, dados que fornecem informações sobre outros dados. Eles desempenham um papel crucial na coleta de eventos em sistemas distribuídos, uma vez que concedem a capacidade de recriar a estrutura e a sequência de eventos que um dado inicial atravessou no sistema. Essa capacidade de reconstruir a relação entre eventos confere flexibilidade à maneira como as informações são armazenadas, concede liberdade na seleção da arquitetura de coleta de eventos e simplifica a transição para outras arquiteturas, quando necessário(PARKER et al., 2020). No entanto, essa vantagem vem com o custo de aumentar o espaço de armazenamento necessário para acomodar os metadados associados a cada novo evento.

Um exemplo de metadado é o Timestamp (Carimbo de Data/Hora) um dos metadados mais comum em logs de eventos. Ele indica o momento em que o evento ocorreu. Esse metadado é importante para a análise temporal dos registros, permitindo a ordenação cronológica dos eventos.

2.1.4 Sincronização de relógio

Em um sistema distribuído, geralmente não temos um relógio global que suporte todas as aplicações do sistema. Em vez disso, devemos lidar com vários relógios locais conflitantes ou construir relógios lógicos (LAMPORT, 1978). Os relógios lógicos permitem ordenar os eventos ocorridos, mas não fornecem informações sobre o tempo absoluto em que os eventos ocorrem. Para depurar um sistema em busca de um erro, a determinação da relação de causalidade entre os eventos e a ordem de sua execução é mais relevante do que o tempo absoluto em que ocorreram. Isso torna a adoção de um relógio lógico uma solução adequada para resolver conflitos de sincronização. No entanto, para a depuração de desempenho, precisamos que as informações sobre o tempo físico sejam as mais precisas possíveis. A abordagem clássica para a sincronização do tempo, o Network Time Protocol (NTP) (MILLS, 1989), pode sincronizar relógios dentro de $100\mu s$ a $2ms$. Para sistemas de *tracing* que capturam eventos cuja variação do tempo pode ter um intervalo de ocorrência maior, essa precisão pode ser suficiente, como, por exemplo, em muitos sistemas Web. Contudo, quando temos chamadas RPC (THURLOW, 2009) que levam menos de $10\mu s$, com eventos separados por apenas alguns μs, como em um sistema distribuído de alto desempenho ligado por uma LAN, a latência medida pode ser inferior a $50\mu s$, o que torna

a precisão oferecida pelo NTP insuficiente e requer melhores técnicas de sincronização de tempo.

2.2 FERRAMENTAS E INFRAESTRUTURAS DE REGISTROS DE EVENTOS

O desenvolvimento e a evolução das tecnologias para o rastreamento de sistemas distribuídos são de suma importância para garantir a eficiência, a confiabilidade e a escalabilidade desses sistemas. No cerne dessa progressão, duas ferramentas em particular desempenharam um papel histórico: Dapper (SIGELMAN et al., 2010) e Pivot Tracing (MACE; ROELKE; FONSECA, 2015). Ambos os projetos foram pioneiros em suas abordagens para rastreamento distribuído e forneceram uma base sólida sobre a qual muitas das ferramentas atuais foram construídas.

Nesta seção, iremos detalhar as características principais e os pontos fortes dessas duas ferramentas. A escolha de estudo sobre o Dapper e Pivot Tracing para esta análise se baseia no seu histórico, no status de *open-source* e nos resultados apresentados. Deve-se observar que, embora existam outras ferramentas privadas disponíveis no mercado, muitas não são facilmente acessíveis ou possuem restrições de uso e entre as ferramentas *open-source* disponíveis, as disponíveis se baseiam no Dapper e Pivot Tracing, porém apenas o Dapper e Pivot Tracing possuem artigos descrevendo sua arquitetura, funcionalidade e resultados. Portanto, optamos por concentrar nossa atenção nessas duas para permitir que os leitores possam verificar de maneira independente o que é apresentado aqui.

Vamos primeiro explorar as características individuais do Dapper e do Pivot Tracing, antes de discutir como suas contribuições influenciaram o desenvolvimento de ferramentas de rastreamento contemporâneas, como Jaeger (CLOUD NATIVE COMPUTING FOUNDATION, 2017), Zipkin (OPENZIPKIN, 2021) e OpenTelemetry (CLOUD NATIVE COMPUTING FOUNDATION, 2019). A análise comparativa destas ferramentas tem como objetivo oferecer uma visão geral de como a área de rastreamento distribuído evoluiu e continua a evoluir.

2.2.1 Dapper

O Dapper (SIGELMAN et al., 2010), desenvolvido pela equipe do Google, foi projetado para fornecer aos desenvolvedores dados e relatórios sobre o comportamento dos sistemas distribuídos complexos da empresa. Esses sistemas consistem em grandes conjuntos de servidores menores. Para entender o comportamento desses sistemas, é necessário observar as atividades em muitos programas e máquinas diferentes e como eles interagem entre si. Dapper visa alcançar três metas de projeto principais: baixa sobrecarga, transparência no nível da aplicação (programadores não precisam alterar ou inserir códigos para o funcionamento do coletor de dados) e escalabilidade.

A arquitetura do Dapper, por natureza, funciona como outro sistema distribuído escalável. A troca de informações é feita através de chamadas de procedimento remoto (RPC) entre os serviços-alvo e os coletores do Dapper. Esses coletores trocam esses dados de rastreamento com um serviço de reconstrução de dados, liberando os coletores dessa responsabilidade para reduzir a sobrecarga sobre eles e permitir que respondam às chamadas o mais rápido possível.

2.2.2 Pivot Tracing

Pivot Tracing (MACE; ROELKE; FONSECA, 2015) é uma estrutura de monitoramento destinada a sistemas distribuídos, capaz de correlacionar estatísticas de aplicações, componentes e máquinas em tempo real, sem a necessidade de modificar ou reimplementar o código do sistema. Este sistema oferece aos usuários a flexibilidade de definir e implementar consultas de monitoramento de maneira dinâmica, permitindo a coleta de estatísticas diversificadas de pontos específicos do sistema.

Uma característica central do Pivot Tracing é a sua abordagem de rastreamento por causalidade e o modo como utiliza essa relação entre os eventos. A causalidade refere-se à relação de causa e efeito entre diferentes eventos em um sistema. No contexto do Pivot Tracing, essa relação de causalidade é utilizada para entender como os eventos estão interconectados e como um evento específico pode levar a outros eventos subsequentes.

Essa relação de causalidade é crucial para compreender não apenas os eventos individualmente, mas também como eles contribuem para o comportamento global e os resultados observados em um sistema distribuído complexo. Ao entender as relações de causa e efeito entre eventos, os engenheiros e desenvolvedores podem identificar padrões, gargalos e pontos problemáticos no sistema.

Para representar essa relação de causalidade e os eventos, o Pivot Tracing utiliza a noção de "tuplas". Uma tupla é uma estrutura de dados que contém diversos valores, cada um representando um aspecto específico de um evento. Esses valores podem incluir identificadores únicos, informações temporais, detalhes sobre a causa do evento e informações sobre os efeitos resultantes.

As tuplas formam um conjunto de dados distribuído em fluxo, significando que as informações são continuamente atualizadas e processadas à medida que os eventos ocorrem no sistema. Isso possibilita que os usuários definam consultas de monitoramento de forma dinâmica e coletem estatísticas e informações relevantes em tempo real.

Em resumo, o Pivot Tracing utiliza a relação de causalidade entre os eventos para modelar o comportamento do sistema. Isso é alcançado por meio de tuplas, que representam os eventos e suas características. Essas tuplas formam um conjunto de dados distribuído em fluxo, permitindo o monitoramento e a análise em tempo real de sistemas distribuídos complexos.

2.2.3 Análise das ferramentas descritas

Dapper e Pivot Tracing, embora compartilhem o mesmo propósito geral de rastrear o fluxo de requisições em arquiteturas de sistemas distribuídos, apresentam abordagens de implementação distintas. Enquanto o Dapper adota uma estrutura de processamento centralizada, otimizada para a eficiência na entrega de dados consultados, o Pivot Tracing, por sua vez, oferece uma flexibilidade maior em termos de consultas. Vale destacar que essas ferramentas não apenas operam de maneira independente, mas também têm a capacidade de serem combinadas entre si, ampliando ainda mais as possibilidades de análise e monitoramento do fluxo de requisições em ambientes distribuídos.

A escolha entre Dapper e Pivot Tracing pode depender mais da natureza da arquitetura que se pretende monitorar. O Dapper é indicado quando se necessita de baixa latência e não se requer consultas muito complexas. Pivot Tracing, por outro lado, é mais adequado quando se deseja realizar consultas avançadas em tempo real. É importante lembrar que ambas as ferramentas, sendo sistemas distribuídos, carregam a complexidade associada à manutenção de tais sistemas.

2.2.4 Comparando Dapper e Pivot Tracing com Ferramentas Recentes

Com o avanço da ciência e tecnologia da computação, várias novas ferramentas e frameworks foram introduzidas, oferecendo diferentes níveis de detalhes e facilidade de uso para o rastreamento distribuído. Algumas dessas ferramentas são Jaeger, Zipkin e OpenTelemetry.

2.2.4.1 Zipkin

Zipkin(OPENZIPKIN, 2021) é uma ferramenta de rastreamento distribuída baseada na mesma ideia do Dapper. Ele fornece uma maneira de rastrear solicitações através de vários sistemas, ajudando a identificar onde ocorre a latência em uma cadeia de solicitações. Como o Dapper, o Zipkin prioriza uma sobrecarga mínima, tornando-o adequado para sistemas que requerem alto desempenho. Ele também fornece uma interface de usuário para visualizar os rastreamentos e latências de cada serviço. No entanto, ao contrário do Dapper, ele tem uma abordagem mais modular, o que pode torná-lo mais fácil de ser adaptado a necessidades específicas.

2.2.4.2 Jaeger

Jaeger(CLOUD NATIVE COMPUTING FOUNDATION, 2017) é uma ferramenta de rastreamento distribuído de código aberto, inspirada no Dapper e no Zipkin. Foi projetada para monitorar microserviços e facilitar a solução de problemas que envolvem latência de rede. O Jaeger coleta dados de rastreamento de sua aplicação e armazena-os de uma

maneira que facilita a visualização e a análise desses dados. Ele fornece uma UI para visualizar rastreamentos de uma maneira muito detalhada, permitindo que os usuários vejam a latência de cada componente individual da solicitação. A flexibilidade do Jaeger permite que ele seja usado em vários cenários de rastreamento, tornando-o adequado para ambientes complexos e em grande escala.

2.2.4.3 OpenTelemetry

OpenTelemetry(CLOUD NATIVE COMPUTING FOUNDATION, 2019) é um conjunto de APIs, bibliotecas e agentes de coleta de dados para observabilidade que inclui rastreamento, logs e métricas de execução. É um projeto de código aberto que combina os projetos OpenCensus(OPENCENSUS, 2018) e OpenTracing(OPENTRACING, 2016). Sua finalidade é tornar estável as capacidades de rastreamento e observabilidade, e fornecer suporte unificado para todos os aspectos da coleta de dados. É a ferramenta mais recente e moderna comparada ao Dapper e ao Pivot Tracing, e é construída com um foco maior na interoperabilidade e flexibilidade.

Tanto o Jaeger quanto o Zipkin podem ser comparados com o Dapper e o Pivot Tracing em termos de funcionalidade e objetivos. Ambos oferecem rastreamento distribuído com foco em baixa sobrecarga (como o Dapper) e possuem interfaces de usuário que permitem visualizar e analisar os rastreamentos. O OpenTelemetry, por sua vez, é uma ferramenta mais completa, que fornece não apenas rastreamento distribuído, mas também outras formas de observabilidade, como logs e métricas. Ele oferece maior flexibilidade e interoperabilidade, podendo ser integrado a uma variedade de sistemas e ferramentas.

Enquanto o Dapper e o Pivot Tracing são projetos de pesquisa significativos e influentes na área de rastreamento distribuído, as ferramentas mais recentes, como Jaeger, Zipkin e OpenTelemetry, oferecem mais funcionalidades e são mais adaptáveis às necessidades modernas de desenvolvimento de aplicações. No entanto, é importante notar que as ideias e conceitos fundamentais dessas ferramentas atuais foram em grande parte inspirados e influenciados pelo trabalho feito no Dapper e no Pivot Tracing.

2.3 MONITORAMENTO DE INFRAESTRUTURA

Um dos principais desafios em sistemas distribuídos é assegurar que todos os componentes estão operando corretamente e têm recursos suficientes para manter esta operação. Para garantir a escalabilidade e disponibilidade do sistema, é crucial coletar dados sobre os recursos de cada componente. A metodologia de coleta desses dados assemelha-se à coleta simples de eventos mencionada na seção anterior, porém não nos preocupamos com sincronização ou causalidade neste contexto. A recriação pode ser relevante, mas não de imediato. O monitoramento da infraestrutura, por sua vez, é frequentemente dinâmico e ocorre instantaneamente. Se uma máquina não está respondendo em um dado momento,

o foco é fazer com que ela volte a operar, sem nos preocuparmos diretamente com a corretude do sistema, mas com seu funcionamento.

Nesta seção, discutiremos duas questões fundamentais do monitoramento da infraestrutura: o monitoramento do sistema e da rede. Também exploraremos como o armazenamento de informações pode impactar o desempenho do sistema distribuído e, finalmente, abordaremos a importância da análise de *tracing*.

2.3.1 Monitorando um Sistema

Todo monitoramento gera sobrecarga, mesmo no caso mais simples de requisição e resposta, utilizado apenas para sinalizar que o sistema está operante. Mesmo em sistemas críticos, onde a transmissão de informações de eventos que ocorreram pode não ser possível, é prudente manter informações de estado na máquina e aceitar essa sobrecarga de monitoramento, ainda que seja em serviços periféricos que possam determinar se um serviço crítico está operante ou não. Por exemplo, em uma situação em que um serviço crítico precisa se comunicar periodicamente com outro serviço e existe uma estimativa de tempo para a comunicação entre eles (JIA et al., 2017).

2.3.2 Monitorando a Rede

A latência de comunicação é um aspecto importante do monitoramento. Embora alguns sistemas web possam não ser muito afetados por problemas de alta latência, em serviços como videoconferências e jogos online, uma latência alta resulta em uma experiência ruim para os usuários. Além disso, um componente com alta latência pode impactar todo um sistema distribuído. Por exemplo, se temos um sistema A com latência de 500ms para se conectar com um componente do sistema B, e esse componente faz 100 requisições de A para B, com cada requisição demorando em média 1 ms para ser respondida, então teremos um atraso significativo causado pela latência, afetando a interação entre os sistemas.

2.4 ARMAZENAMENTO DE INFORMAÇÕES

Armazenar informações sobre eventos em um sistema distribuído é um desafio. A escolha do método de armazenamento depende do problema a ser resolvido. Escrever dados em um arquivo local é o método mais simples, mas em um sistema distribuído, essa abordagem pode dificultar a recuperação desses dados. Por exemplo, se o sistema está distribuído através de um *load balancer*, cada máquina terá seu próprio arquivo de registro de eventos (MA et al., 2018). Para verificar se um evento ocorreu, será necessário verificar cada registro de cada máquina, o que pode sobrecarregar máquinas críticas que não contêm o registro, impedindo-as de atender a requisições importantes. Além disso,

problemas de sincronização podem ocorrer quando várias instâncias estão executando o mesmo serviço simultaneamente.

Em serviços distribuídos independentes (que operam em apenas uma instância), salvar dados localmente com estratégias de confiabilidade e tolerância a falhas pode ser uma escolha aceitável. No entanto, quando temos réplicas e arquiteturas complexas e a reconstrução do grafo de eventos se faz necessária, uma arquitetura independente para armazenar esses dados e garantir um formato padrão é a abordagem mais adequada.

2.5 ANÁLISE DE TRACING

A análise de *tracing* é uma ferramenta poderosa para identificar a trajetória de uma requisição em um sistema. Comumente utilizada para reproduzir ou entender uma falha, ela também pode ser empregada para melhorar o desempenho do sistema (DUMAIS et al., 2014), além de identificar e prevenir futuros problemas no mesmo. A análise de *tracing* torna-se um desafio complexo devido à grande quantidade de dados que um sistema pode gerar, tornando uma simples busca por um dado específico uma tarefa desafiadora. Ferramentas como o Dapper, que indexa o grafo de chamadas aos sistemas por um identificador de *tracing* e de serviço, podem auxiliar em análises mais profundas. No entanto, a busca por todos os registros em que um sistema foi ativado pode resultar em um volume muito alto de dados. Portanto, a atenção também deve ser direcionada para a análise de dados e para as ferramentas e infraestruturas de Big Data (SAGIROGLU; SINANC, 2013).

3 UM DEPURADOR DISTRIBUÍDO COM PYTHON

Um depurador, ou ferramenta de depuração, é um programa de computador usado para testar e identificar falhas em outros programas. A funcionalidade principal de um depurador é executar o programa alvo sob condições controladas, em tempo real, e passo a passo de forma interativa. Isso permite ao programador acompanhar as operações em andamento e monitorar alterações no estado dos recursos utilizados pelo programa, tais como áreas de memória utilizadas pelo programa alvo ou pelo sistema operacional do computador. Tais alterações podem indicar defeitos no código (AGANS, 2002; LENCEVICIUS, 2012).

Os depuradores também servem como ferramentas importantes para fins didáticos. A possibilidade de navegar passo a passo pela execução de um aplicativo e observar as alterações causadas por cada linha de código pode facilitar o aprendizado. Além das funcionalidades descritas anteriormente, alguns depuradores são limitados a reproduções determinísticas de um programa, ou seja, eles registram os passos de uma execução e permitem que o usuário interaja e visualize as alterações realizadas pelo código em cada passo. Outros depuradores, por sua vez, oferecem suporte para depuração em tempo real, onde as reproduções podem exibir aspectos mais realísticos do funcionamento do programa. As funcionalidades típicas de depuração incluem a capacidade de executar ou interromper o programa alvo em pontos específicos, exibir e modificar o conteúdo da memória ou variáveis, e chamar funções.

Os depuradores funcionam, em linhas gerais, através de *breakpoints*. Os *breakpoints* indicam ao depurador em quais pontos do programa a execução deve ser interrompida, transferindo ao usuário o controle e o acesso aos dados existentes no ambiente naquele momento. Após a interrupção pelo *breakpoint*, o depurador fornece acesso ao programa através de sua interface interativa, que permite realizar alterações no estado do código em execução.

A forma como os *breakpoints* são definidos depende da linguagem de programação em uso. Programas compilados normalmente se é utilizado um programa externo ao compilador e argumentos específicos para depuração ao compilar o programa. Esse é o caso de algumas linguagens que compilam para linguagem de máquina, como a linguagem C e o depurador GDB (GNU Operating System, 1986) (que também suporta C++, Fortran, Java, Chill, assembly, e Modula-2). Já programas interpretados, geralmente se têm acesso aos depuradores como uma extensão do próprio interpretador ou como uma biblioteca, fazendo com que os *breakpoints* sejam chamadas de funções dentro do código, como no caso do Python pdb (Python Software Foundation, 1996), do `JavaScript debugger` (Mozilla Foundation, 1996) e do `Ruby debug.rb` (The Ruby Programming Language, 2006). No caso do depurador JavaScript, ele pode ser facilmente acessado por qualquer navegador

que forneça acesso ao console de desenvolvimento, invocando o depurador com o comando *debugger*, uma palavra reservada da linguagem.

Cada linguagem e cada depurador apresentam diferentes interfaces e funcionalidades adicionais. Por exemplo, o `Python pdb` permite visualizar o código completo do arquivo que está sendo depurado, enquanto o `JavaScript debugger` é mais enxuto no número de comandos do depurador expostos para a linguagem, porém ele conta com um ambiente que o complementa. A maior distinção ocorre entre os depuradores de programas interpretados e compiladas. Nos depuradores de executáveis (programas compilados), normalmente a definição de *breakpoints* é realizada após a compilação do programa. Por outro lado, nos depuradores de programas interpretados, é necessária uma alteração direta no código seja para inserir ou remover um *breakpoint* ou invocar o depurador, e em algumas linguagens é possível sinalizar ao depurador para que ele insira esses breakpoints. Enquanto os depuradores de executáveis apresentam limitações quanto às alterações realizadas durante a execução: não é fácil alterar referências de funções ou tipos de variáveis, sendo que em alguns casos isso pode se mostrar muito difícil. Em contrapartida, em interpretadores, essas alterações não são um problema. Em muitos casos é possível definir novas funções dentro do interpretador e tratá-las como variáveis, alterando-as conforme necessário. Como a verificação de tipos em um interpretador é dinâmica, essa operação é considerada trivial.

Este trabalho foi parcialmente desenvolvido com o objetivo de aplicar os métodos estabelecidos por um depurador em um ecossistema distribuído. Assim, cada sistema seria capaz de enviar dados que poderiam recriar um depurador funcional de forma remota.

3.1 TIPOS DE DEPURADORES E SUAS DEFINIÇÕES

1. **Depurador de Linha Única:** Um depurador de linha única permite a execução passo a passo do código, permitindo que o programador examine o estado das variáveis e a saída a cada etapa. Isso é útil para rastrear erros simples e compreender o fluxo de execução. Um exemplo é o Python Pdb (Python Software Foundation, 1996).

2. **Depurador de Ponto de Interrupção:** Um depurador de ponto de interrupção permite definir pontos específicos no código onde a execução será interrompida. Isso permite inspecionar o estado do programa em momentos críticos e facilita a depuração de comportamentos complexos. Como exemplos temos o GDB (GNU Debugger) (GNU Operating System, 1986) e o depuradores acoplados a ambientes integrados de desenvolvimento, como o Visual Studio Code (MICROSOFT, 2015).

3. **Depurador de Memória:** Um depurador de memória é usado para detectar vazamentos de memória e acessos inválidos. Ele rastreia alocações e desalocações de memória, ajudando a identificar problemas de gerenciamento de memória. Como exemplos temos o Valgrind (SEWARD, 2002) e o AddressSanitizer (SEREBRYANY et al., 2012).

4. **Depurador Baseado em Reprodução/Replay**: Um depurador baseado em re-

petição ou reprodução permite que os desenvolvedores gravem a execução de um programa em um cenário específico e depois reproduzam essa execução para examinar o comportamento do programa e identificar problemas. Isso é útil para depurar erros difíceis de reproduzir manualmente (RONSSE; BOSSCHERE; KERGOMMEAUX, 2000).

5. **Depurador Simbólico**: Um depurador simbólico trabalha com informações simbólicas, como nomes de variáveis e funções em vez de endereços de memória. Isso torna a depuração mais intuitiva, uma vez que os desenvolvedores podem se concentrar nos elementos do código em vez de lidar diretamente com endereços de memória (KING, 2009).

6. **Depurador Simbólico Baseado em Reprodução/Replay:** Um depurador simbólico baseado em reprodução é uma ferramenta de depuração que une a ideia do depurador simbólico e o depurador baseado em reprodução. Ele trabalha com a ideia de reproduzir a execução de um programa de maneira controlada e simbólica para analisar seu comportamento (RONSSE; BOSSCHERE; KERGOMMEAUX, 2000). Ao invés de executar o programa original, um depurador simbólico baseado em reprodução executa o programa em um ambiente controlado, onde ele mantém um registro das operações e dos valores dos dados manipulados pelo programa, sem necessariamente executar cada instrução no hardware real. A cada reprodução, o depurador coleta informações sobre o estado dos dados, as condições de execução e as operações realizadas (KING, 2009). Um exemplo de depurador que implementa essa abordagem é o Friday (GEELS et al., 2007).

3.1.1 Combinação de Abordagens de Depuração

Os tipos de depuradores mencionados anteriormente não são mutuamente exclusivos e frequentemente suas funcionalidades podem ser combinadas para obter uma análise mais abrangente dos programas. A abordagem simbólica baseada em repetição, por exemplo, pode ser integrada a depuradores tradicionais para aprofundar a detecção de erros em execuções complexas. Ao executar o programa sob uma execução simbólica, é possível explorar diferentes cenários que podem levar a erros sutis em iterações posteriores. Da mesma forma, a combinação de um depurador de memória com um depurador de ponto de interrupção pode revelar vazamentos de memória que ocorrem em momentos específicos da execução.

3.2 DEPURADORES DISTRIBUÍDOS

Apesar de o aparato de monitoramento, a coleta de eventos e a análise de dados serem ferramentas indispensáveis, essas soluções lidam apenas com situações passadas e podem não ser suficientes para depurar um erro completamente. Surge, então, a necessidade de um depurador capaz de atuar de maneira interativa nesses sistemas. Tradicionalmente, após a ocorrência de um erro, os registros padrões conseguem retratar apenas a pilha de

chamadas atuais. Consequentemente, apenas parte dos métodos que foram executados e que são impactados pela chamada do método com erro estão na pilha. Os métodos que já finalizaram a execução não estão mais disponíveis. Esse é um problema em que um depurador pode auxiliar.

Depuradores citados anteriormente enfrentam a limitação de estar vinculados a um processo local, e sua execução está atrelada a uma única máquina. Em sistemas distribuídos, chamadas a outros componentes são encapsuladas em cada componente do sistema. Isso torna difícil que o depurador de uma máquina acesse o código sendo executado em outra.

Um depurador remoto, deve ser capaz de depurar um código de maneira remota. E um depurador distribuído, portanto, é a extensão de um depurador remoto, e deve ser capaz de relacionar chamadas de múltiplos sistemas e seja capaz de atender a diversos sistemas. Assim, ele pode auxiliar na reconstrução parcial ou total do *tracing* de uma chamada, através da inspeção de vestígios de chamadas remotas durante a depuração. Além disso, deve ser capaz de acessar múltiplos ambientes, dado que conhecemos a rota que uma chamada tomou (por exemplo, por meio de uma ferramenta de coleta e análise de *tracing*). Mesmo quando parte do funcionamento do sistema é ocultado por ferramentas ou abstrações, o conhecimento da arquitetura subjacente pode ser valioso para uma depuração mais eficaz, e o depurador distribuído pode ajudar a obter esse conhecimento ao permitir que os desenvolvedores examinem a interação entre diferentes componentes.

Com o avanço dos *middlewares* e das abstrações de programação distribuída, a utilização e a implementação de programas distribuídos se tornou mais simples, como é o caso do RPC. Entretanto, em bases de código complexas, distinguir as camadas de abstrações das reais chamadas de recursos distribuídos pode não ser uma tarefa trivial.

Concomitantemente aos desafios de entender o funcionamento por trás da abstração de programação distribuída, uma outra motivação para este trabalho é a aplicação da ferramenta para finalidade didática. Ou seja, a capacidade de executar a ferramenta em um ambiente distribuído, entender seu funcionamento e acompanhar a interação entre os serviços distribuídos. No ambiente didático de sistemas distribuídos, convive-se com a oposição entre um código verboso e complexo, que mostra todos os recursos necessários para realizar uma tarefa simples, e um código simples e altamente abstraído, que executa tarefas complexas. Por exemplo, em Python, temos nativamente a biblioteca HTTPServer (Python Software Foundation, 2021a), que abstrai completamente o funcionamento de um servidor HTTP (FIELDING et al., 1999) que serve arquivos de um diretório especificado. Por outro lado, temos um suporte completo de uma biblioteca de socket (Python Software Foundation, 2021c), onde também podemos implementar de forma trabalhosa o mesmo servidor.

A proposta deste trabalho é poder unir os dois cenários de forma mais transparente. O depurador deverá ser capaz de caminhar através das chamadas e mostrar o que está

por dentro da abstração. Essa não é a principal ideia ao se implementar um depurador, mas, num contexto tão complexo, é um uso válido para o mesmo.

Um exemplo de um depurador distribuído é o projeto Friday, que será abordado na próxima subseção. Ele ilustra os desafios e soluções encontrados ao se desenvolver um depurador de sistemas distribuídos. O Friday foi escolhido por ser uma ferramenta já estável, de código aberto e com uma pesquisa acadêmica associada a ele.

3.2.1 Friday - um depurador distribuído

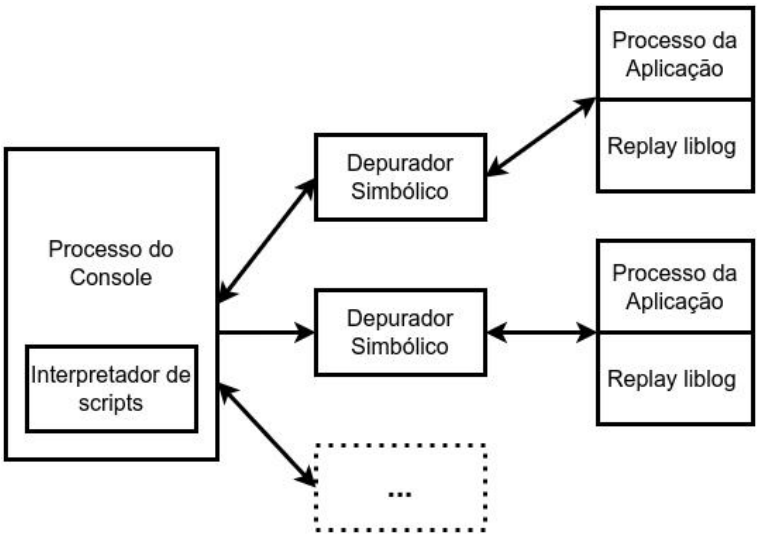
Friday (GEELS et al., 2007) é um depurador simbólico baseado em repetição para aplicativos distribuídos, permitindo que desenvolvedores mantenham visões globais e abrangentes do estado do sistema. Friday fornece aos programadores recursos para verificar invariantes globais em execuções distribuídas. Ele é uma ferramenta poderosa para tarefas de depuração distribuídas, porém possui limitações que devem ser consideradas pelos usuários.

O Friday concentra-se em dois aspectos. Primeiro, ele fornece características primitivas para detectar eventos no sistema, sejam eventos baseados em dados (*watchpoints*) ou em fluxos de controle (*breakpoints*). Esses *watchpoints* e *breakpoints* são distribuídos, coordenando a detecção em todos os nós do sistema. Os *watchpoints* acionam o depurador sempre que uma variável monitorada é alterada, e os *breakpoints* atuam sempre que uma linha/função é executada. Em segundo lugar, o Friday permite que os usuários enviem comandos arbitrários a pontos de controle e pontos de interrupção distribuídos. Friday dá a esses comandos acesso a todo o estado do programa, bem como a um armazenamento compartilhado e persistente para salvar estatísticas de depuração, construir modelos comportamentais ou analisar o estado global.

Friday oferece aos usuários um console central de depuração, que está conectado a processos de repetição. Cada processo de repetição executa uma instância de um depurador simbólico tradicional, como o GDB (veja a Figura 1). O console inclui um interpretador de linguagem de script incorporado, que interpreta ações. Isso pode causar lentidão para a execução, porque o código binário de depuração pode não ter o mesmo desempenho que o código binário de produção por realizar mais operações necessárias para a depuração. Além disso, também pode causar incompatibilidade com diferentes versões do GDB, uma vez que, para diferentes linguagens, temos diferentes versões de GDB compatíveis.

Apesar dessas limitações, é importante salientar que Friday é uma ferramenta muito relevante quando pensamos em depuração de sistemas distribuídos. Essas limitações não minimizam a relevância do trabalho, mas apontam para a complexidade do problema de se desenvolver uma ferramenta de depuração que seja eficaz para uma ampla variedade de sistemas distribuídos. Tais limitações também oferecem caminhos para trabalhos futuros na área.

Figura 1 – Arquitetura do Friday (GEELS et al., 2007)



3.3 A IMPORTÂNCIA DA DEPURAÇÃO DE SISTEMAS DISTRIBUÍDOS E A COLETA E ANÁLISE DE LOGS

A depuração de sistemas distribuídos está diretamente relacionada à coleta e análise de logs, bem como ao entendimento do funcionamento do sistema como um todo. A coleta de logs é uma prática comum em sistemas distribuídos, onde múltiplos componentes e serviços podem estar interagindo e executando em diferentes máquinas. Os logs são registros detalhados de eventos que ocorrem durante a execução do sistema e são essenciais para entender o comportamento e detectar possíveis problemas.

Como visto no capítulo 2, através da análise de logs, é possível identificar falhas, erros e gargalos, o que pode ser fundamental para a manutenção e otimização do sistema. No entanto, em sistemas distribuídos, a coleta e análise de logs podem ser complexas devido à natureza distribuída do ambiente. Cada componente pode gerar logs independentemente, e é difícil correlacionar esses registros para obter uma visão completa do sistema.

É neste ponto que a depuração de sistemas distribuídos se torna essencial. O depurador distribuído permite acompanhar e analisar a execução do sistema em tempo real, identificando a interação entre os diferentes componentes distribuídos e suas dependências. Ele permite que os desenvolvedores rastreiem o fluxo de execução, estabeleçam pontos de depuração e inspecionem variáveis e estado do sistema em diferentes pontos da aplicação distribuída.

Com o depurador distribuído, é possível entender como as diferentes partes do sistema se comunicam e como eventos específicos se propagam por toda a arquitetura. Isso facilita a identificação de problemas de comunicação, erros de lógica, conflitos de dados e outras questões que podem surgir em sistemas distribuídos.

Além disso, o depurador distribuído também pode ser usado para obter informações detalhadas sobre o desempenho do sistema, identificando gargalos e pontos de melhoria.

Ele permite que os desenvolvedores monitorem o sistema em tempo real e obtenham um entendimento valioso sobre seu comportamento e uso de recursos. Vale ressaltar que essas informações também podem ser obtidas pelo coletor de logs.

A combinação de coleta de logs e depuração distribuída torna-se uma abordagem holística para entender, monitorar e manter sistemas complexos e distribuídos, fornecendo uma visão mais abrangente e detalhada do funcionamento do sistema e facilitando a identificação e resolução de problemas em ambientes distribuídos. A continuidade da pesquisa nessa área é crucial para o aprimoramento das ferramentas e técnicas de depuração em sistemas distribuídos, proporcionando melhorias significativas na manutenção e desempenho desses sistemas.

3.4 DEPURADOR DISTRIBUÍDO COM PYTHON

Neste trabalho, desenvolvemos um depurador distribuído simples utilizando a linguagem de programação Python. Ao contrário do exemplo do Friday, nossa implementação utiliza uma linguagem interpretada na qual o depurador faz parte do ambiente de execução. Dessa forma, conseguimos remover as dependências externas ao código, como as *flags* especiais para as bibliotecas Friday e GDB. Com isso, aproveitamos as características de um depurador de programas interpretados, tais como: acesso direto ao código-fonte, possibilidade de realizar alterações em tempo real, exploração de trechos de código não executados e a capacidade de depuração dinâmica do sistema, conforme delineado no início deste capítulo. A linguagem Python já possui uma interface de depuração, e nosso trabalho consiste em expandir essa interface e adaptar suas funcionalidades para atender às necessidades de um depurador distribuído.

Uma limitação ao escrever um depurador utilizando o suporte de uma linguagem específica é ficar restrito à linguagem escolhida. No caso de um depurador distribuído, pode ser necessário que todos os componentes estejam padronizados na mesma linguagem para tirar maior proveito de funcionalidades específicas. Essas exigências dificultam a implementação em um ambiente real já existente. Entretanto, em se tratando de uma ferramenta para finalidades didáticas, essa restrição torna-se mais aceitável.

3.4.1 Estendendo o depurador de Python

Para estender o depurador padrão da linguagem Python, *pdb*, é necessário compreender seu funcionamento em detalhes. O *pdb* possui um amplo conjunto de funcionalidades para depuração. Podemos utilizá-lo como uma biblioteca Python e invocar seus pontos de interrupção (*breakpoints*) como chamadas de função (neste trabalho, utilizamos o método **pdb.set_trace()**). Após a interrupção em um *breakpoint*, a interface de depuração é aberta e temos acesso aos comandos do *pdb* (veja o apêndice A). Todos os métodos da biblioteca *pdb* são de código aberto, assim como todo o código-fonte do Python. Explo-

rando o paradigma de programação orientada a objetos, podemos extender a classe que representa a interface do depurador e modificar seu comportamento padrão.

3.4.2 Interface de depuração

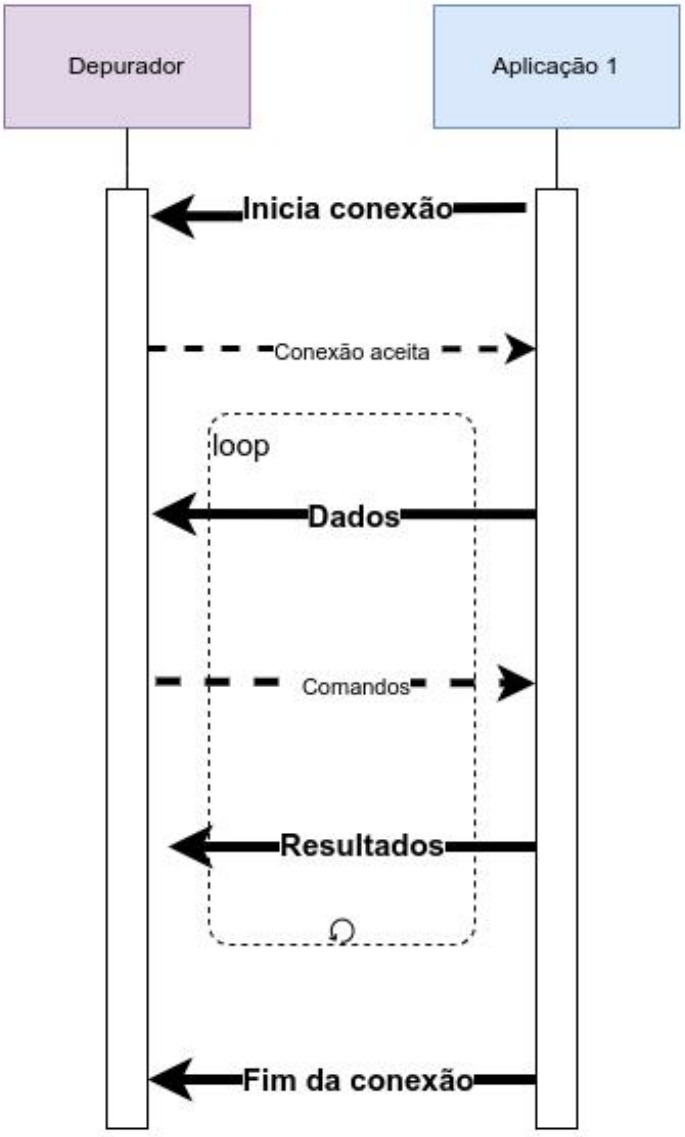
Ao iniciar a construção de um depurador, foi necessário definir uma interface de depuração de alto nível para controlar os programas e extrair informações de seus estados. Parte desta interface foi herdada do *pdb*. Implementamos uma abstração com acesso remoto para controlar a execução de programas em máquinas distribuídas, extrair informações da tabela de símbolos e ler/escrever variáveis no espaço de endereçamento do processo dessas máquinas. Para criar essa abstração foi necessário transmitir a interface de controle de forma remota para receber comandos de um terminal e enviar informações do programa que está sendo depurado.

Nosso depurador foi desenvolvido visando simplicidade em cenários de sistemas distribuídos. Ele permite uma depuração remota detalhada e facilita a identificação de relações entre chamadas de diferentes sistemas. A interação com uma máquina remota é retratada na Figura 2. Neste fluxo, a aplicação estabelece uma conexão com o depurador e inicia um ciclo contínuo onde envia seus dados de estado, enquanto o usuário por meio do depurador emite comandos de controle e recebe os resultados da execução desses comandos na aplicação alvo.

Deve-se também levar em conta que a sessão de depuração pode ser interrompida de duas maneiras principais: Intencionalmente pelo depurador, em resposta a comandos do usuário, ou inesperadamente pela aplicação em depuração, como por exemplo, devido a erros como estouro de memória, exceções não tratadas, ou falhas na comunicação com uma parte específica do sistema distribuído. Ambas as situações podem emergir em qualquer etapa da sessão de depuração, o que reforça a necessidade de resiliência e adaptabilidade do depurador em ambientes distribuídos.

Durante a depuração de uma aplicação, uma das vantagens de um depurador é a capacidade de entrar em métodos e compreender sua execução. No entanto, em sistemas distribuídos, uma função pode estar sendo chamada em outro ambiente, e depurar essa situação exigiria a abertura de vários terminais remotos. Nossa implementação busca relacionar as chamadas abertas de forma hierárquica, utilizando pontos de interrupção em todas as aplicações-alvo em depuração. Essas aplicações podem compartilhar um mesmo ID de depuração, permitindo que a aplicação estabeleça relações entre contextos e retorne a um contexto de uma camada superior quando uma subchamada é finalizada, conforme mostrado na Figura 3, que é uma extensão recursiva da Figura 2. Vale ressaltar que na prática, chamadas encadeadas podem ser bloqueantes, assim sendo necessário que a conexão "filha"encerre e retorne o contexto com algum resultado para a conexão que a invocou.

Figura 2 – Diagrama de sequência de uso do depurador



3.4.3 Arquitetura do depurador proposto

A arquitetura do depurador é dividida em duas partes importantes: uma biblioteca que serve como base para o depurador e é capaz de adicionar *breakpoints*, analisar/alterar dados da aplicação-alvo, e uma aplicação que atua como interface para receber dados do depurador e enviar comandos de forma remota.

A interface do depurador é um processo que é executado de forma passiva em uma máquina, ou seja, é um servidor que aguarda que as aplicações entrem em contato para ativar as conexões de depuração, como mostrado na Figura 2. Cada aplicação estabelece uma conexão com esse servidor, mas apenas uma conexão está ativa para interação com o usuário. A cada nova conexão, o usuário pode aceitá-la ou rejeitá-la. Caso uma conexão esteja relacionada a um ID de uma conexão que ainda está ativa, essas conexões com o mesmo ID são colocadas na mesma lista e, de acordo com a ordem de ocorrência, a segunda

conexão é uma chamada direta da primeira conexão. As conexões que compartilham o mesmo ID se comportam como uma pilha, pois uma chamada mais antiga aguarda o retorno da chamada mais recente para que o depurador continue sua execução, como destacado na Figura 3.

Como o depurador é uma ferramenta que suporta a execução passo a passo, uma aplicação pode estar pausada, aguardando a execução de uma linha de código (chamadas de métodos ou execução de instruções variadas), e essa linha de código também pode estar relacionada a uma nova chamada ao depurador. Esse comportamento de dependência e espera pela finalização de uma chamada deve ser replicado na interface de depuração.

A Figura 4 mostra os componentes principais do depurador distribuído. O gerenciador e a interface do usuário formam um componente do sistema, e os sistemas que estão sendo depurados formam componentes dinâmicos. A hierarquia de componentes e as conexões entre eles são construídas logicamente pelo gerenciador.

3.4.4 Implementação do depurador proposto

O *pdb* é projetado para ser utilizado em programas comuns e utiliza as interfaces padrões para operações de entrada e saída (I/O). Uma das principais tarefas foi substituir a forma como o *pdb* utiliza essas interfaces. Nosso objetivo inicial foi redirecionar a entrada e saída para um terminal remoto, permitindo o acesso por meio do *telnet* (ALEXANDER, 1994) que é uma ferramenta de interação simples com terminais remotos via texto. Redirecionando o I/O padrão do interpretador por meio de uma conexão de leitura e escrita usando a interface de *socket*, podemos nos conectar remotamente a esse *socket* por meio do *telnet*, conforme mostrado na Figura 5, onde o comando tenta acessar um terminal remoto que está sendo executado no localhost e na porta 6899. Neste ponto o processo estará pausado e esperando por essa conexão do usuário via terminal remoto para que continue sua execução. Na figura 5 podemos ver o comando *longlist*A, que faz parte do depurador pdb, e mostra o código fonte do código que está sendo depurado.

No entanto, a implementação principal do gerenciador de depuração utiliza um serviço que emprega uma conexão passiva e centralizada para receber e gerenciar as conexões de depuração. Ter compatibilidade simultânea com os dois tipos de conexões não seria adequado. Ao optar por uma abordagem de conexão ativa para o nosso depurador, precisamos lidar com endereços de conexão, identificadores de serviço e garantir que a conexão seja iniciada e que as informações iniciais sejam recebidas antes de redirecionar completamente o I/O padrão.

Separamos o objeto que representa uma conexão do objeto que representa o gerenciador. Cada objeto de conexão é separado por uma *thread* diferente e esses objetos recebem comandos do terminal do usuário por meio de uma fila assíncrona que representa uma estrutura de multi-produtor e multi-consumidor. Essa estrutura permite que cada *thread*/conexão espere por seus dados de entrada de forma encapsulada. Dessa forma,

o gerenciador fica livre para enviar seus dados sob demanda, sem precisar compartilhar sua entrada padrão e sem afetar quem tem acesso aos dados de entrada, ao contrário da implementação do depurador na máquina-alvo.

Durante a implementação dessa estrutura, enfrentamos alguns desafios. Uma fonte inicial de assincronismo no projeto foi a alternância entre a entrada padrão e o recebimento de novas conexões pelo gerenciador. Para solucionar esse problema, utilizamos a biblioteca padrão *select* (Python Software Foundation, 2021b), que nos permitiu separar as interrupções da entrada padrão e as interrupções de novas solicitações de conexão por meio do *socket*. Também nos deparamos com um problema semelhante ao do problema do produtor-consumidor (MAZIERO, 2019), já que temos n processos, em que $n - 1$ podem escrever dados em seus *sockets*, mas não recebem dados da entrada padrão, pois não estão ativos para interação. O único terminal ativo tem o fluxo de interação normal. Para resolver essa questão, utilizamos múltiplas filas bloqueadoras, em que cada fila pode entrar em um estado de bloqueio em que o consumidor espera até que a fila esteja livre. Por meio dessas filas, transmitimos comandos da interface do depurador para o processo remoto ativo que está sendo depurado.

O diagrama da Figura 6 apresenta a arquitetura do depurador. No coração desta arquitetura, encontramos o *Distribuidor de Eventos*, implementado usando a biblioteca *select* (Python Software Foundation, 2021b). Ele atua como ponto central de coordenação, gerenciando os eventos ou comandos provenientes tanto do *Gerenciador de Entrada do Usuário* quanto do *Gerenciador de Conexões*, direcionando-os aos serviços pertinentes.

O *Gerenciador de Entrada do Usuário* é o ponto crucial de interação com os sistemas que serão depurados. Ele capta os comandos do usuário e os envia ao *Distribuidor de Eventos* direcionados ao sistema alvo. Por outro lado, a *Interface de Entrada do Usuário* é o canal através do qual as instruções são fornecidas e os retornos são exibidos ao usuário. No código, o retorno é realizado diretamente das interfaces dos serviços para a saída padrão. À direita e acima do *Distribuidor de Eventos*, encontra-se o *Gerenciador de Conexões*. Sua função é atuar como uma ponte, mantendo uma comunicação direta com o *Distribuidor de Eventos*, transmitindo e recebendo mensagens. Além disso, este gerenciador tem a responsabilidade de determinar qual serviço está ativo e qual está em modo de espera.

Na parte inferior do diagrama, são destacados três serviços: *Serviço A*, *Serviço B* e *Serviço* . Eles representam sistemas distribuídos conectados ao depurador. Na figura, o *Serviço A* está ativo, recebendo as instruções do usuário. Em contrapartida, *Serviço B* e *Serviço C* estão em repouso. No entanto, mesmo inativos, eles mantêm uma conexão ativa com o depurador, demonstrando que estão prontos para serem ativados conforme a necessidade. Foi ainda introduzida uma *Fila Assíncrona*, que estabelece uma conexão entre o *Gerenciador de Entrada do Usuário* e o *Gerenciador de Conexões*. Este elemento indica que os comandos ou eventos podem ser organizados de forma não síncrona, aguar-

dando o processamento adequado. Por fim, as flechas no diagrama representam o fluxo de informações e comandos entre os componentes. Flechas contínuas simbolizam fluxos ativos em andamento, enquanto flechas tracejadas denotam fluxos ou conexões potenciais que não estão em uso ativo no momento atual.

Além das limitações impostas pelas linguagens específicas (Seção 3.4), embora nossa interface de comando e gerenciamento esteja desacoplada da máquina-alvo e de seu interpretador, a implementação é baseada na mesma interface exposta pelo *pdb*. Como nosso depurador atua em tempo real, em vez de adotar a abordagem de gravação e reprodução determinística, ele captura eventos que ocorrem em tempo real. Isso permite que o usuário observe problemas de concorrência e falhas em outros componentes, o que pode ser extremamente útil durante a depuração. Essas decisões contribuem para a eficácia e flexibilidade do nosso depurador distribuído, tornando-o uma ferramenta viável para depurar sistemas distribuídos em Python. O código fonte do projeto pode ser encontrado no Gitlab(FREIRES, 2023).

Figura 3 – Diagrama de sequência de uso do depurador com chamadas internas

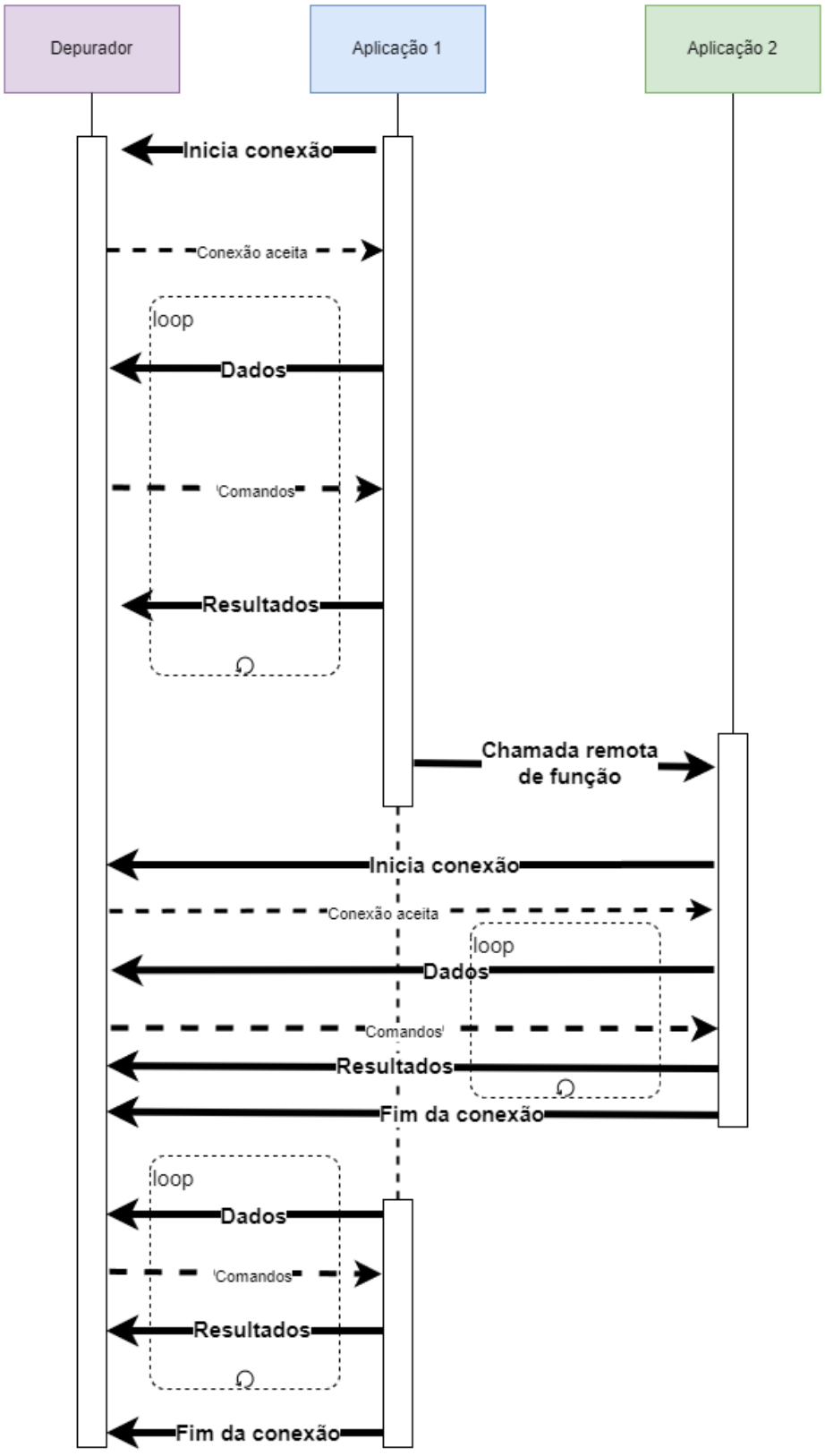


Figura 4 – Componentes da arquitetura-alvo

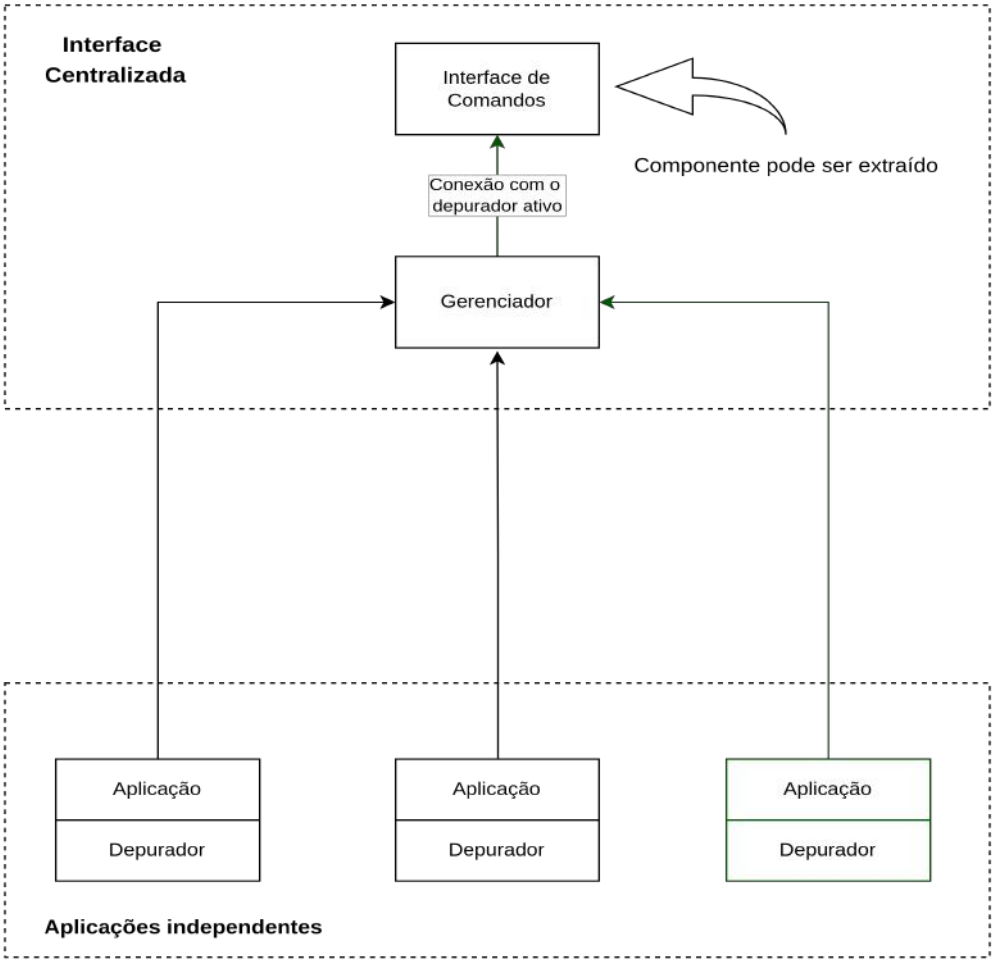
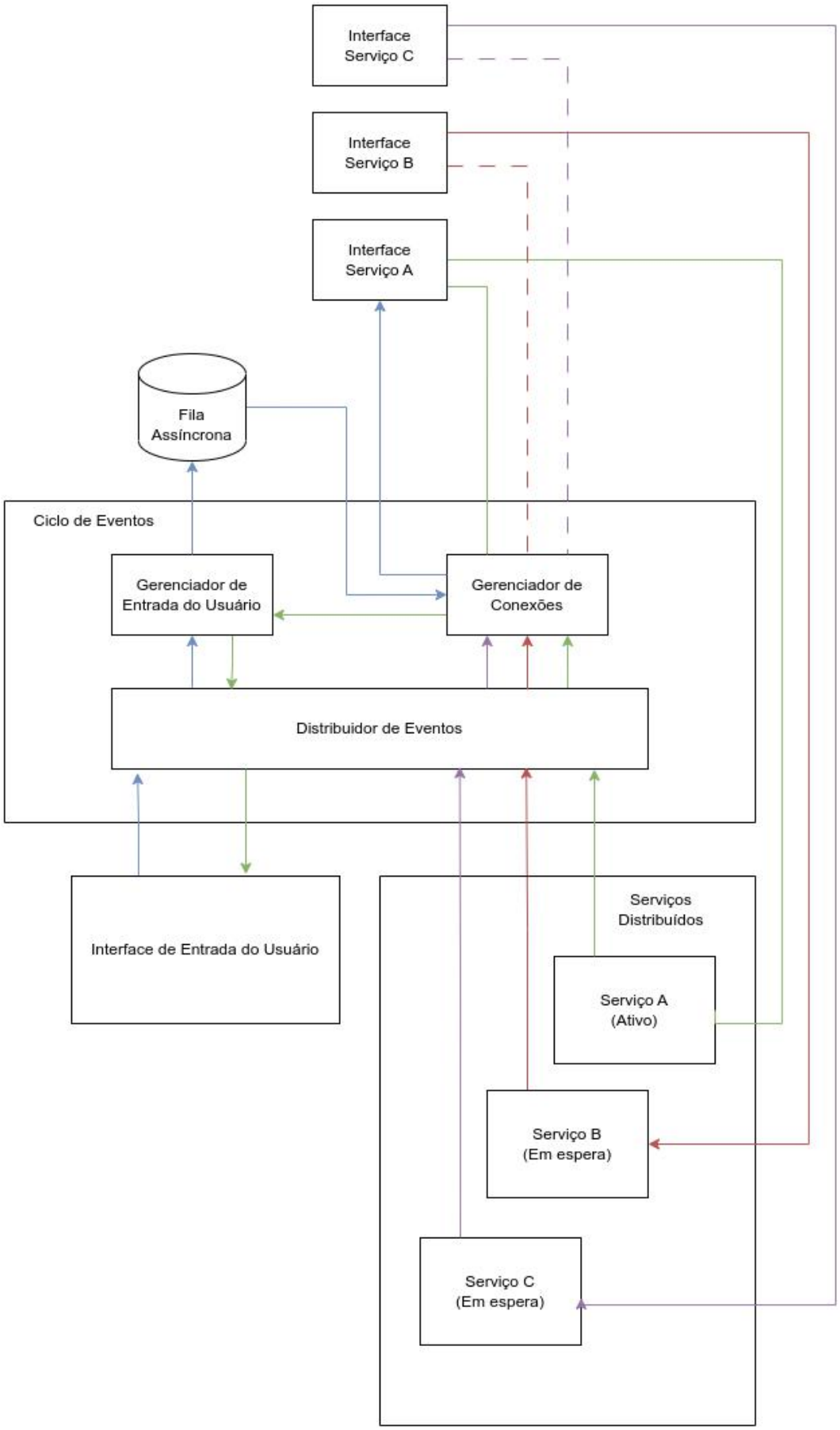


Figura 5 – Acesso ao depurador via telnet

```
➜  ~ telnet localhost 6899
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
> /home/joao/dev/TCC/examples/telnet_script.py(6)test()
-> arr = [x, 2, 3]
(Pdb) longlist
  3     def test():
  4         x = 1
  5         Rdb().set_trace()
  6  ->     arr = [x, 2, 3]
  7         return arr
(Pdb)
```

Figura 6 – Esquema da Arquitetura do Depurador

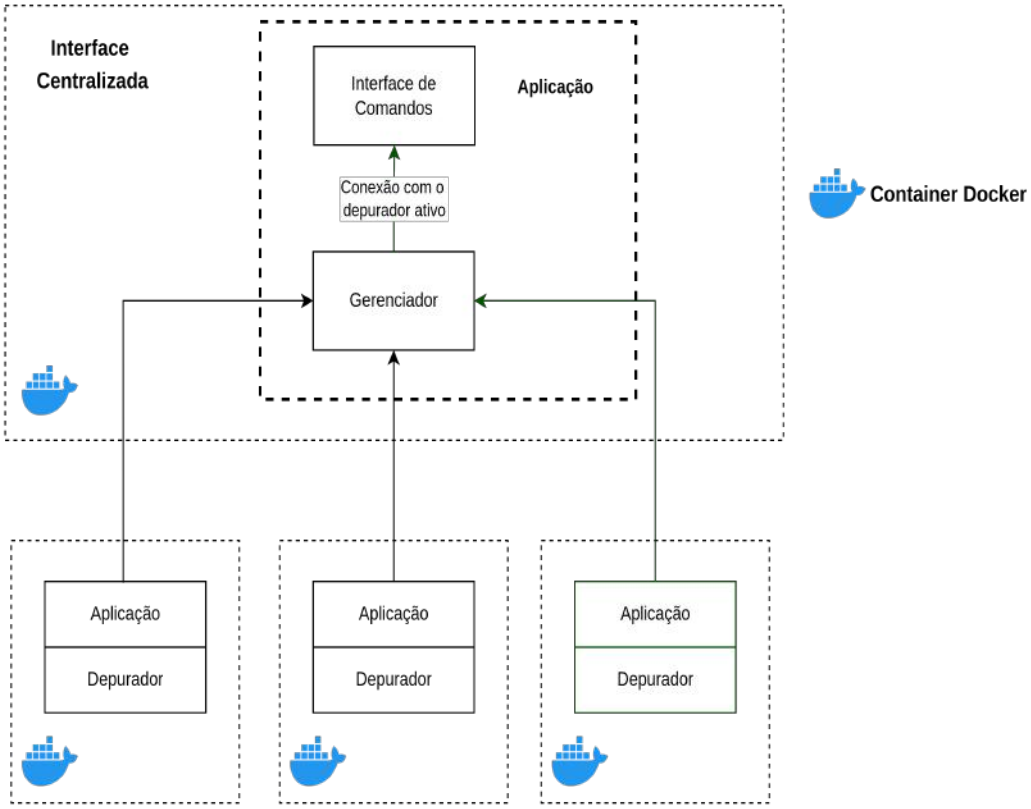


4 TESTES E RESULTADOS

Junto com o desenvolvimento do depurador, foram definidos previamente os casos de teste e os objetivos a serem alcançados. Buscamos focar nosso projeto em casos de teste com aplicações reais e de fácil utilização, a fim de atender a um dos principais objetivos: o uso didático da ferramenta. Dessa forma, nossos resultados poderão refletir claramente nossos objetivos.

Neste capítulo apresentaremos inicialmente os dois cenários a partir dos quais geramos os testes, e os scripts correspondentes. Em seguida, analisaremos com mais detalhe os scripts usados nos testes e como as funcionalidades do depurador foram exploradas. Por fim, discutiremos os resultados obtidos.

Figura 7 – Organização da arquitetura com Docker



4.1 TESTES

Os testes foram desenvolvidos com o objetivo de alcançar dois principais objetivos: depurar uma aplicação distribuída de forma remota e gerenciar conexões remotas que compartilham o mesmo identificador. Para isso, criamos serviços cliente/servidor simples,

mas capazes de colocar os conceitos em prática e demonstrar o potencial do depurador proposto.

4.1.1 Arquitetura de Teste com Docker

A implementação da arquitetura de teste foi fortemente fundamentada na utilização do Docker (MERKEL, 2014). Esta escolha permitiu simular condições que espelham ambientes reais de execução (NAIK, 2016). Como é evidenciado na Figura 7, diversos componentes da arquitetura operam em contêineres Docker distintos. A adoção deste design simplificou o processo de reprodução dos cenários testados com o docker, que precisa apenas de poucos comandos para iniciar todo o ambiente e expor apenas os serviços alvos. E também reforçou a confiabilidade e validade de que os resultados que alcançamos poderão ser reproduzidos em um ambiente em nuvem. Adicionalmente, os cenários de testes descritos foram encapsulados em arquivos de exemplo e estão disponíveis no repositório de código-fonte da aplicação (FREIRES, 2023).

4.1.2 Cenário 1 - Script de Requisição

O primeiro cenário de teste envolve apenas a depuração de uma aplicação que contém um script que se comunica com o depurador (Código 1). O objetivo deste teste é verificar a capacidade do depurador desenvolvido de executar corretamente e utilizar sua interface. Durante a execução desse script de teste, é invocado um *breakpoint* que permite que o depurador interrompa a execução e forneça informações relevantes para a depuração. Os resultados dessa depuração podem ser visualizados na Figura 8.

Podemos observar que o exemplo do Código 1 corresponde ao código exibido na Figura 8, a qual ilustra as funcionalidades do depurador, parte das funcionalidades herdadas do depurador *pdb*. No uso da biblioteca `rdb`, que é a biblioteca do depurador, podemos visualizar a função de *breakpoint* `set_trace` do depurador. Embora não esteja contido na imagem, essa função aceita argumentos adicionais. como o `host` e `port`, para alterar o *host* e a porta de comunicação com o gerenciador de depuração, bem como os argumentos mencionados na figura, um nome para o sistema depurado e um token de identificação para a chamada do sistema que vai ser depurada. Mais detalhes desse script serão detalhados na subseção 4.2. Além disso, podemos verificar na Figura 8 que o ponto de parada do depurador ocorreu na linha seguinte ao *breakpoint* `rbd.set_trace`. Essa capacidade de depuração passo a passo permite uma análise minuciosa do código e facilita a detecção de erros ou comportamentos indesejados.

Código 1 – Script de requisição

```
# Script 1

import rdb
import requests
import uuid

# cria um token para depurar
token = str(uuid.uuid4())

# iniciar o trace para um módulo e token
rdb.set_trace(__file__, token)

data = {
    "a": "abc",
}
header = {"DEBUG_TOKEN": token}
response = requests.post("http://127.0.0.1:5000/", headers=header, data=
    data)
```

Código 2 – Script do servidor

```
# Script 2 - Servidor

import rdb
import rpyc
from flask import Flask, request


app = Flask(__name__)


@app.route("/", methods=["GET", "POST"])
def hello_world():
    token = request.headers.get("DEBUG_TOKEN")
    try:
        c = rpyc.connect("localhost", 18861)
    except ConnectionRefusedError:
        pass
    try:
        c.root.set_token(token)
        rdb.set_trace(__file__, token)
        msg = f"<p>{c.root.get_message()}</p>"
    except Exception as e:
        msg = "<p>Hello world without RPC!</p>"
        pass
    return msg


if __name__ == "__main__":
    app.run(debug=True)
```

Código 3 – Script do serviço RPC

```
# Script 3 - Serviço RPC

import rdb
import rpyc


class MyService(rpyc.Service):
    def __init__(self):
        self.token = None

    def exposed_get_message(self):
        if self.token:
            rdb.set_trace(__file__, self.token)
        message = "hello world!"
        return message

    def exposed_set_token(self, token):
        self.token = token


t = rpyc.utils.server.ThreadedServer(
    MyService,
    port=18861,
    protocol_config=dict(sync_request_timeout=100000),
    listener_timeout=100000,
)
t.start()
```

4.1.3 Cenário 2 - Múltiplas Aplicações

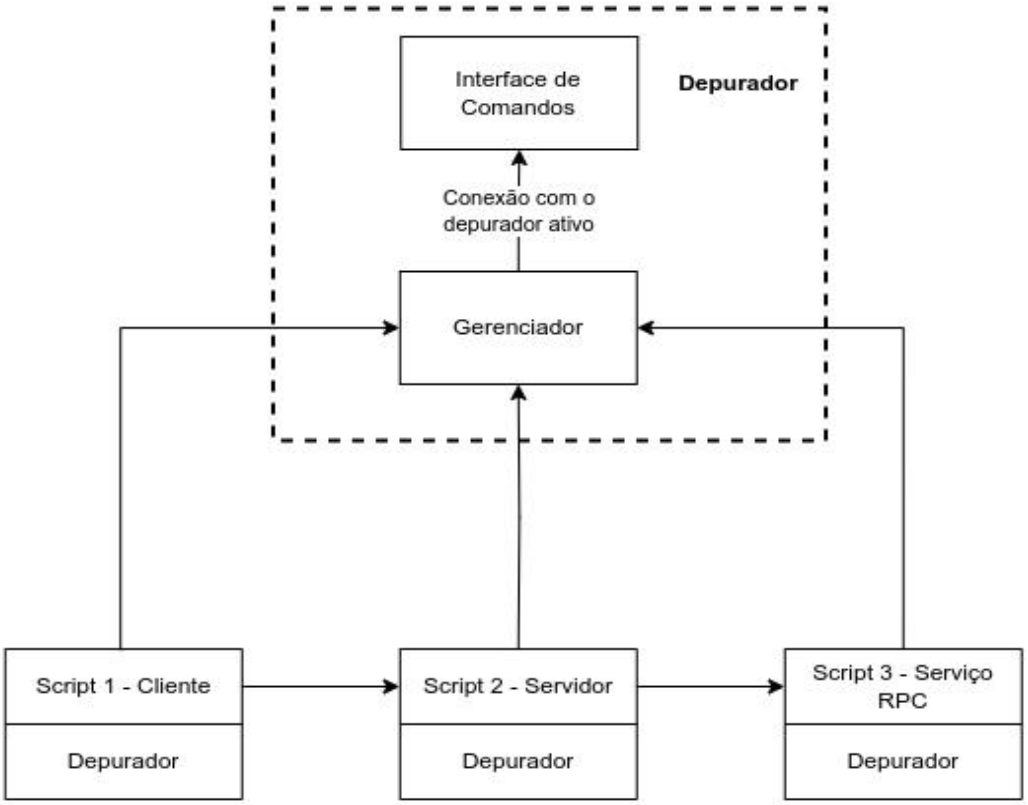
O segundo cenário de teste introduz uma maior complexidade e envolve três aplicações: um serviço web, um serviço RPC e um script que invoca o depurador e utiliza os serviços mencionados na Figura 9. O objetivo desse teste é demonstrar a capacidade do depurador de lidar com a depuração de aplicações distribuídas que se comunicam por meio de diferentes protocolos. No início do teste, o script envia uma requisição para o servidor web, que, por sua vez, precisa chamar uma função do serviço RPC. Em cada etapa desse processo, o depurador é acionado para permitir a análise detalhada do fluxo de execução. O conteúdo do script de teste pode ser visto nos códigos: 1, 2 e 3. Durante a execução desse teste, foram capturadas imagens do depurador, como ilustrado nas Figuras 10 a 13. Nestas figuras, é possível identificar o ponto de entrada do script em cor verde, a depuração do servidor web em cor rosa e do serviço RPC em cor vermelha, assim como os

Figura 8 – Inicialização do depurador e conexão com um script

```
Terminal open
Waiting connections...
Service request_app.py with id aef8d67f-cf27-4438-8bba-f32a3db1b21c on 127.0.0.1:55370
want to connect, proceed? (y/n)
y
[-] Connected to service request_app.py:aef8d67f-cf27-4438-8bba-f32a3db1b21c on 127.0.0
.1:55370
> /home/joao/dev/TCC/examples/request_app.py(11)<module>()
-> data = {"a": "abc"}
(Pdb)

> help

Documented commands (type help <topic>):
========================================
EOF    cl         display   j         next     run     unalias    where
a      clear      down      jump      p        rv      undisplay
alias  commands   enable    l         pp       s       unt
args   condition  h         list      r        source  until
b      d          help      ll        restart  step    up
break  debug      ignore    longlist  return   tbreak  w
bt     disable    interact  n         retval   u       whatis

Miscellaneous help topics:
==========================
exec  pdb

Undocumented commands:
======================
c  cont  continue  exit  q  quit

(Pdb)


> longlist
  1     import rdb
  2     import requests
  3     import uuid
  4
  5     # cria um token para depurar
  6     token = str(uuid.uuid4())
  7
  8     # iniciar o trace para um módulo e token
  9     rdb.set_trace(__file__, token)
 10
 11  -> data = {"a": "abc"}
```

retornos das funções do depurador e as mudanças de contexto. Vale destacar que o depurador altera a cor do terminal aleatoriamente, evitando repetir cores previamente usadas para representar novas conexões, oferecendo uma maneira visual de distinguir diferentes conexões além do nome fornecido para o depurador, que nos exemplos são os nomes de arquivo gerado pelo interpretador com a palavra reservada do Python `__file__`. Esses exemplos fornecem uma visão mais clara da interação entre as diferentes partes do sistema durante a depuração.

Figura 9 – Organização da arquitetura do cenário de teste 2



4.2 ANÁLISE DETALHADA DOS SCRIPTS DOS CENÁRIOS DE TESTES

Nesta seção, abordamos os scripts apresentados anteriormente, focando nas funcionalidades do depurador (`rdb`).

4.2.1 Script 1 (Cenário 1 e parte do Cenário 2)

O funcionamento do primeiro script é detalhado da seguinte forma:

1. O script começa importando o módulo `rdb`, que é o depurador desenvolvido e que está sendo testado.

2. Em seguida, um token único é gerado usando o módulo `uuid`(Python Software Foundation, 2021d). Este token é essencial para a depuração, pois permitirá que o depurador identifique sessões específicas de depuração.

3. A função `rdb.set_trace(__file__, token)` é invocada para iniciar o rastreamento de depuração para o arquivo atual (`__file__`) usando o token gerado.

4. O script então cria um cabeçalho com o token de depuração e envia uma solicitação POST para um servidor local usando a biblioteca `requests` (REITZ, 2021).

5. Finalmente, a resposta do servidor é impressa, caso haja uma resposta.

4.2.2 Script 2 (Cenário 2)

O segundo script tem o seguinte funcionamento:

1. Cria um servidor Flask (RONACHER, 2021) que possui uma única rota ("/").
2. Ao receber uma requisição, o token de depuração é obtido do cabeçalho da requisição.
3. O script então tenta conectar-se a um serviço `rpyc` (FILIBA, 2021) no `localhost` na porta `18861`.
4. Se a conexão for bem-sucedida, o token é configurado para o serviço `rpyc` e o rastreamento de depuração é iniciado para o arquivo atual.
5. A mensagem recebida do serviço `rpyc` é então retornada.
6. Se qualquer exceção ocorrer, uma mensagem padrão é retornada.

4.2.3 Script 3 (Cenário 2)

Por último, temos o terceiro script:

1. Este é um serviço `rpyc` que possui métodos expostos que podem ser invocados remotamente.
2. O método `exposed_get_message` verifica se um token de depuração está configurado.
3. Se um token estiver configurado, ele inicia o rastreamento de depuração para o arquivo atual.
4. Independentemente da depuração, uma mensagem padrão ("hello world!") é retornada.

4.2.4 Interação entre os Scripts

Os três scripts interagem de forma sequencial para demonstrar o uso do depurador (importado como `rdb` no código). O fluxo pode ser resumido da seguinte maneira:

1. O **Script 1** atua como cliente, gerando um token único e fazendo uma requisição HTTP ao servidor definido pelo **Script 2**, passando o token como cabeçalho.
2. Ao receber a requisição, o **Script 2** tenta estabelecer uma conexão RPC com o serviço definido pelo **Script 3**. Ele transmite o token para o serviço RPC e ativa a depuração remota. Em seguida, solicita uma mensagem ao serviço RPC.

3. O **Script 3**, ao receber o token e o pedido de mensagem, ativa sua depuração e retorna uma mensagem para o **Script 2**.

4. Finalmente, o **Script 2** envia essa mensagem como resposta HTTP para o **Script 1**, que então a imprime.

Este fluxo demonstra a interação e comunicação entre diferentes partes do sistema, bem como a integração do depurador em diferentes pontos da execução.

Figura 10 – Depuração da conexão com um script (em cor verde) e pedido de acesso do servidor HTTP (em cor rosa)

```
> longlist
  1     import rdb
  2     import requests
  3     import uuid
  4
  5     # cria um token para depurar
  6     token = str(uuid.uuid4())
  7
  8     # iniciar o trace para um módulo e token
  9     rdb.set_trace(__file__, token)
 10
 11  -> data = {"a": "abc"}
 12     header = {"DEBUG_TOKEN": token}
 13     response = requests.post("http://127.0.0.1:5000/", headers=header, data=data)
 14
 15     print(response.content)
(Pdb)


> n
> /home/joao/dev/TCC/examples/request_app.py(12)<module>()
-> header = {"DEBUG_TOKEN": token}
(Pdb)


> n
> /home/joao/dev/TCC/examples/request_app.py(13)<module>()
-> response = requests.post("http://127.0.0.1:5000/", headers=header, data=data)
(Pdb)


> n
Service /home/joao/dev/TCC/examples/flask_app.py with id dcb342b3-1624-469e-a74f-0ecb92
8a2b56 on 127.0.0.1:58026 want to connect, proceed? (y/n)
y
New service on trace pool id dcb342b3-1624-469e-a74f-0ecb928a2b56:
0 -> request_app.py-dcb342b3-1624-469e-a74f-0ecb928a2b56
1 -> /home/joao/dev/TCC/examples/flask_app.py-dcb342b3-1624-469e-a74f-0ecb928a2b56


[-] Connected to service /home/joao/dev/TCC/examples/flask_app.py:dcb342b3-1624-469e-a7
4f-0ecb928a2b56 on 127.0.0.1:58026
> /home/joao/dev/TCC/examples/flask_app.py(19)hello_world()
-> msg = f"<p>{c.root.get_message()}</p>"
(Pdb)
```

Figura 11 – Depuração da conexão com o serviço RPC (em cor vermelha)

```
[-] Connected to service /home/joao/dev/TCC/examples/flask_app.py:dcb342b3-1624-469e-a7
4f-0ecb928a2b56 on 127.0.0.1:58026
> /home/joao/dev/TCC/examples/flask_app.py(19)hello_world()
-> msg = f"<p>{c.root.get_message()}</p>"
(Pdb)

> longlist
  9     @app.route("/", methods=["GET", "POST"])
 10     def hello_world():
 11         token = request.headers.get("DEBUG_TOKEN")
 12         try:
 13             c = rpyc.connect("localhost", 18861)
 14         except ConnectionRefusedError:
 15             pass
 16         try:
 17             c.root.set_token(token)
 18             rdb.set_trace(__file__, token)
 19  ->         msg = f"<p>{c.root.get_message()}</p>"
 20         except Exception as e:
 21             msg = "<p>Hello world without RPC!</p>"
 22             pass
 23         return msg
(Pdb)


> n
Service rpc_app.py with id dcb342b3-1624-469e-a74f-0ecb928a2b56 on 127.0.0.1:58028 want
 to connect, proceed? (y/n)
y
New service on trace pool id dcb342b3-1624-469e-a74f-0ecb928a2b56:
0 -> request_app.py-dcb342b3-1624-469e-a74f-0ecb928a2b56
1 -> /home/joao/dev/TCC/examples/flask_app.py-dcb342b3-1624-469e-a74f-0ecb928a2b56
2 -> rpc_app.py-dcb342b3-1624-469e-a74f-0ecb928a2b56


[-] Connected to service rpc_app.py:dcb342b3-1624-469e-a74f-0ecb928a2b56 on 127.0.0.1:5
8028
> /home/joao/dev/TCC/examples/rpc_app.py(12)exposed_get_message()
-> message = "hello world!"
(Pdb)

> longlist
  9         def exposed_get_message(self):
 10             if self.token:
 11                 rdb.set_trace(__file__, self.token)
 12  ->         message = "hello world!"
```

4.3 RESULTADOS

Com a implementação do depurador distribuído, foi possível acessar e depurar o código de diferentes tipos de aplicações distribuídas. Em cada caso de teste, o depurador demonstrou estabilidade e permitiu a análise dos ambientes criados durante a depuração. No segundo cenário de teste, em particular, o depurador foi acionado por um script, seguido por um *breakpoint* em um servidor web e, por fim, outro *breakpoint* em um servidor RPC. Durante esse teste, foram enfrentados desafios reais, como a ocorrência de *timeouts*

Figura 12 – Retorno dos contextos do serviço RPC (em cor vermelha) e do servidor HTTP (em cor rosa)

```
> longlist
  9         def exposed_get_message(self):
 10             if self.token:
 11                 rdb.set_trace(__file__, self.token)
 12  ->         message = "hello world!"
 13             return message
(Pdb)


> c
Bye bye from rpc_app.py-dcb342b3-1624-469e-a74f-0ecb928a2b56!

Backward to terminal /home/joao/dev/TCC/examples/flask_app.py-dcb342b3-1624-469e-a74f-0
ecb928a2b56
> /home/joao/dev/TCC/examples/flask_app.py(23)hello_world()
-> return msg
(Pdb)


>  18             rdb.set_trace(__file__, token)
 19             msg = f"<p>{c.root.get_message()}</p>"
 20         except Exception as e:
 21             msg = "<p>Hello world without RPC!</p>"
 22             pass
 23  ->     return msg
 24
 25
 26     if __name__ == "__main__":
 27         app.run(debug=True)
[EOF]
(Pdb)


> msg
'<p>hello world!</p>'
(Pdb)


> c
Bye bye from /home/joao/dev/TCC/examples/flask_app.py-dcb342b3-1624-469e-a74f-0ecb928a2
b56!

Backward to terminal request_app.py-dcb342b3-1624-469e-a74f-0ecb928a2b56
> /home/joao/dev/TCC/examples/request_app.py(15)<module>()
-> print(response.content)
(Pdb)
```

no sistema quando o depurador permanecia ativo por longos períodos. Isso levou à avaliação das limitações impostas pela arquitetura dos sistemas distribuídos, uma vez que, em ambientes reais, as requisições geralmente têm um tempo limite de resposta definido. Ao ajustar as configurações da aplicação para permitir um tempo de vida mais longo das requisições, o problema foi contornado. Por ser um problema difícil de solucionar e que está presente em depuradores interativos, consideramos essa uma limitação frente a depuradores de reprodução/replay.

Figura 13 – Retorno do contexto para o script inicial (em cor verde)

```
> c
Bye bye from /home/joao/dev/TCC/examples/flask_app.py-dcb342b3-1624-469e-a74f-0ecb928a2
b56!

Backward to terminal request_app.py-dcb342b3-1624-469e-a74f-0ecb928a2b56
> /home/joao/dev/TCC/examples/request_app.py(15)<module>()
-> print(response.content)
(Pdb)


>  10
 11     data = {"a": "abc"}
 12     header = {"DEBUG_TOKEN": token}
 13     response = requests.post("http://127.0.0.1:5000/", headers=header, data=data)
 14
 15  -> print(response.content)
[EOF]
(Pdb)


> response.content
b'<p>hello world!</p>'
(Pdb)


> c
Bye bye from request_app.py-dcb342b3-1624-469e-a74f-0ecb928a2b56!

No more terminals to get back...
```

Em suma, o depurador mostrou-se capaz e atendeu aos pré-requisitos estabelecidos na descrição dos casos de teste. Ele se revelou uma ferramenta simples para a depuração de sistemas distribuídos em Python, oferecendo recursos avançados de depuração passo a passo e a capacidade de depurar aplicações distribuídas de forma remota e com poucas mudanças na arquitetura dos sistemas. Por meio dos testes realizados, foi possível confirmar a eficácia do depurador em fornecer detalhes do funcionamento dos serviços e sua estrutura e auxiliar no diagnóstico e possível solução de problemas em sistemas distribuídos complexos. Além disso, a utilização do Docker permitiu a criação de um ambiente de teste encapsulado e apto para interações on-line, contribuindo para a validação dos resultados obtidos.

5 CONCLUSÃO

Ao longo deste trabalho, desenvolvemos um depurador capaz de depurar aplicações em ambientes distribuídos de forma remota. Nos cenários de testes propostos, o depurador atendeu aos objetivos colocados inicialmente. Por meio da utilização de técnicas de tracing e comunicação RPC, o depurador se mostra promissor como uma ferramenta capaz de prestar auxílio ao aprendizado de depuração em sistemas distribuídos. A flexibilidade do depurador em ambiente interpretado permitiu a construção de uma interface adaptativa e versátil para depurar aplicações com propósitos diversos.

A depuração de sistemas distribuídos continua sendo um desafio na engenharia de software, especialmente em ambientes complexos onde múltiplos serviços e componentes estão em execução. O depurador desenvolvido neste trabalho oferece uma abordagem simples, mas que permite aos desenvolvedores obter informações sobre o comportamento das aplicações distribuídas, suas dependências e interações entre os diferentes serviços.

Os cenários de teste demonstraram que o depurador foi capaz de identificar pontos de depuração em diferentes serviços distribuídos e apresentar informações relevantes para que o desenvolvedor pudesse analisar e solucionar problemas. No entanto, nenhum teste além dos apresentados neste trabalho foi realizado para garantir uma avaliação mais abrangente do comportamento do depurador em outros ambientes complexos. Além disso, algumas limitações foram identificadas, e há espaço para futuras melhorias e expansões.

5.1 TRABALHOS FUTUROS

Diversas possibilidades de trabalhos futuros podem ser exploradas para aprimorar o depurador distribuído:

1. **Explorando o uso educativo da ferramenta de depuração**: Uma direção promissora para trabalhos futuros é a investigação do potencial educativo da ferramenta de depuração desenvolvida. Como parte dessa análise, poderia ser conduzido um estudo empírico para avaliar como estudantes de cursos relacionados a sistemas distribuídos percebem e utilizam a ferramenta como uma ferramenta de aprendizado. Esse estudo poderia incluir a coleta de *feedback* qualitativo dos alunos sobre a usabilidade da ferramenta, a eficácia na compreensão de conceitos complexos de sistemas distribuídos e a melhoria da experiência geral de aprendizado.

Além disso, um programa de treinamento ou tutorial poderia ser desenvolvido para orientar os estudantes sobre a aplicação prática da ferramenta em cenários reais de sistemas distribuídos. Isso ajudaria os alunos a adquirir habilidades práticas na depuração de sistemas distribuídos, ao mesmo tempo em que fortaleceria sua compreensão dos conceitos subjacentes.

2. **Suporte a outras linguagens**: O depurador foi desenvolvido especificamente para a linguagem Python. Uma sugestão é adaptar o depurador para suportar outras linguagens de programação populares, como JavaScript, Java, C++ e outras.

3. **Suporte a outros ambientes e plataformas**: Ampliar o suporte do depurador para diferentes ambientes e plataformas de execução, permitindo a depuração em uma variedade maior de cenários. Avaliar uma possível cooperação entre diferentes depuradores.

4. **Integração com outras ferramentas**: Integrar o depurador distribuído com outras ferramentas e ambientes de desenvolvimento, como IDEs, permitindo uma experiência mais completa e integrada para os desenvolvedores.

5. **Aprimoramento da interface do usuário**: Investir em melhorias na interface do depurador, tornando-a mais amigável, intuitiva e visualmente rica, com recursos gráficos que auxiliem na compreensão do fluxo de execução.

6. **Análise de desempenho e análise comparativa**: Realizar testes de desempenho e escalabilidade do depurador distribuído em cenários com um grande número de aplicações e usuários simultâneos é essencial para identificar possíveis gargalos e limitações. Além disso, é importante realizar uma análise comparativa entre o depurador proposto e os outros depuradores distribuídos existentes.

7. **Suporte a outros protocolos de comunicação**: Expandir o suporte a outros protocolos de comunicação além do RPC, permitindo a depuração de aplicações que utilizam diferentes métodos de comunicação distribuída.

8. **Depuração em ambientes em nuvem**: Investigar a aplicação do depurador distribuído em ambientes de computação em nuvem, onde as aplicações são implantadas em máquinas virtuais ou contêineres distribuídos em vários servidores. E diferente dos testes casos de testes, deve se avaliar se existem desafios e/ou problemas quando os serviços e depurador estão por trás de um DNS.

9. **Segurança e privacidade**: Desenvolver métodos que garantam a segurança e a privacidade de dados sensíveis durante o processo de depuração, especialmente em ambientes distribuídos onde os dados são compartilhados entre vários pontos.

10. **Depuração em tempo real**: Investigar como o depurador pode lidar com aplicações distribuídas em tempo real, onde respostas rápidas e confiáveis são essenciais.

11. **Sistema de gerenciamento independente**: Dividir o componente de interface do usuário e o gerenciador de depuração em sistemas independentes, tornando o gerenciador mais adaptável, escalável e menos acoplado à interface de usuário.

Essas são apenas algumas sugestões de trabalhos futuros. A área de depuração de sistemas distribuídos é vasta e oferece muitas oportunidades para pesquisa e desenvolvimento. É importante considerar as necessidades específicas dos ambientes de aplicação para direcionar o desenvolvimento e aprimoramento do depurador de forma a atender as demandas do mundo real. Por meio do constante aprimoramento do depurador distribuído, esperamos fornecer aos desenvolvedores uma ferramenta poderosa para solucionar

problemas e depurar sistemas distribuídos de maneira prática.

REFERÊNCIAS

AGANS, D. **Debugging: The 9 Indispensable Rules for Finding Even the Most Elusive Software and Hardware Problems**. [S.l.]: AMACOM, 2002. ISBN 9780814426784.

ALEXANDER, S. **RFC 1572, Telnet Environment Option**. USA: RFC Editor, 1994. Disponível em: https://www.rfc-editor.org/rfc/rfc1572.html.

CLOUD NATIVE COMPUTING FOUNDATION. **Jaeger**. USA, 2017. Disponível em: https://www.jaegertracing.io/.

CLOUD NATIVE COMPUTING FOUNDATION. **OpenTelemetry**. 2019. Disponível em: https://opentelemetry.io/.

DAVIDSON, T.; WALL, E.; MACE, J. A qualitative interview study of distributed tracing visualisation: A characterisation of challenges and opportunities. **IEEE Transactions on Visualization and Computer Graphics**, p. 1–12, 2023.

DUMAIS, S. et al. Understanding user behavior through log data and analysis. In: OLSON, J. S.; KELLOGG, W. A. (Ed.). **Ways of Knowing in HCI**. New York, NY: Springer New York, 2014. p. 349–372. ISBN 978-1-4939-0378-8. Disponível em: https://www.microsoft.com/en-us/research/publication/understanding-user-behavior-through-log-data-and-analysis/.

FIELDING, R. et al. **RFC 2616, Hypertext Transfer Protocol – HTTP/1.1**. USA: RFC Editor, 1999. Disponível em: https://www.rfc-editor.org/rfc/rfc2616.html.

FILIBA, T. **RPyC: Remote Python Call**. 2021. Disponível em: https://rpyc.readthedocs.io/en/latest/.

FREIRES, J. **Depurador em Python**. 2023. Disponível em: https://gitlab.com/joaofreires/tcc-sd/.

GEELS, D. et al. Friday: Global comprehension for distributed replay. In: **Proceedings of the 4th USENIX Conference on Networked Systems Design Implementation**. USA: USENIX Association, 2007. (NSDI'07), p. 21.

GNU Operating System. **GDB: The GNU Project Debugger**. 1986. Acesso em: 10 de Janeiro de 2022. Disponível em: https://www.sourceware.org/gdb/.

JIA, Z. et al. Big-data analysis of multi-source logs for anomaly detection on network-based system. In: **13th IEEE Conference on Automation Science and Engineering (CASE)**. Xi'an, China: IEEE, 2017. p. 1136–1141.

KING, J. C. **Symbolic Execution for Program Testing and Analysis**. [S.l.]: MIT Press, 2009.

LAKSHMAN, A.; MALIK, P. Cassandra: A decentralized structured storage system. **SIGOPS Oper. Syst. Rev.**, Association for Computing Machinery, New York, NY, USA, v. 44, n. 2, p. 35–40, apr 2010. ISSN 0163-5980. Disponível em: https://doi.org/10.1145/1773912.1773922.

LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 21, n. 7, p. 558–565, jul. 1978. ISSN 0001-0782. Disponível em: https://doi.org/10.1145/359545.359563.

LENCEVICIUS, R. **Advanced Debugging Methods**. Springer US, 2012. (The Springer International Series in Engineering and Computer Science). ISBN 9781441987747. Disponível em: https://books.google.com.br/books?id=tAXpBwAAQBAJ.

MA, J. et al. An overview of a load balancer architecture for vnf chains horizontal scaling. In: **2018 14th International Conference on Network and Service Management (CNSM)**. [S.l.: s.n.], 2018. p. 323–327.

MACE, J.; ROELKE, R.; FONSECA, R. Pivot tracing: Dynamic causal monitoring for distributed systems. In: **Proceedings of the 25th Symposium on Operating Systems Principles**. New York, NY, USA: Association for Computing Machinery, 2015. (SOSP '15), p. 378–393. ISBN 9781450338349. Disponível em: https://doi.org/10.1145/2815400.2815415.

MAZIERO, C. **Sistemas Operacionais: Conceitos e Mecanismos**. Curitiba: UFPR, 2019. ISBN 978-85-7335-340-2.

MERKEL, D. Docker: lightweight linux containers for consistent development and deployment. **Linux journal**, v. 2014, n. 239, p. 2, 2014.

MICROSOFT. **Visual Studio Code - Debugging**. 2015. https://code.visualstudio.com/docs/editor/debugging.

MILLS, D. **RFC 1129, Internet Time Synchronization: The Network Time Protocol**. USA: RFC Editor, 1989. Disponível em: https://www.rfc-editor.org/rfc/rfc1129.pdf.

Mozilla Foundation. **SpiderMonkey - Mozilla's JavaScript and WebAssembly Engine**. 1996. Acesso em: 7 de Setembro de 2023. Disponível em: https://spidermonkey.dev/.

NAIK, N. Migrating from virtualization to dockerization in the cloud: Simulation and evaluation of distributed systems. In: **10th IEEE International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based Environments (MESOCA)**. Raleigh, NC, USA: IEEE, 2016. p. 1–8.

OPENCENSUS. 2018. https://opencensus.io/. Acessado em 22 de agosto de 2023.

OPENTRACING. 2016. https://opentracing.io/. Acessado em 22 de agosto de 2023.

OPENZIPKIN. **Zipkin**. 2021. Disponível em: https://zipkin.io/.

PARKER, A. et al. **Distributed Tracing in Practice: Instrumenting, Analyzing, and Debugging Microservices**. [S.l.]: O'Reilly Media, Incorporated, 2020. ISBN 9781492056638.

PINHEIRO, P.; APARICIO, M.; COSTA, C. Adoption of cloud computing systems. In: **Proceedings of the International Conference on Information Systems and Design of Communication**. New York, NY, USA: Association for Computing Machinery, 2014. (ISDOC '14), p. 127–131. ISBN 9781450327138. Disponível em: https://doi.org/10.1145/2618168.2618188.

Python Software Foundation. **pdb - The Python Debugger**. 1996. Acesso em: 10 de Dezembro de 2021. Disponível em: https://docs.python.org/3/library/pdb.html.

Python Software Foundation. **HTTP servers**. 2021. Acesso em: 10 de Dezembro de 2021. Disponível em: https://docs.python.org/3/library/http.server.html.

Python Software Foundation. **select - Waiting for I/O completion**. 2021. Acesso em: 10 de Dezembro de 2021. Disponível em: https://docs.python.org/3/library/select.html.

Python Software Foundation. **socket - Low-level networking interface**. 2021. Acesso em: 10 de Dezembro de 2021. Disponível em: https://docs.python.org/3/library/socket.html.

Python Software Foundation. **uuid - UUID objects according to RFC 4122**. 2021. Python Standard Library. Disponível em: https://docs.python.org/3/library/uuid.html.

REITZ, K. **Requests: HTTP for Humans**. 2021. Disponível em: https://docs.python-requests.org/en/master/.

RONACHER, A. **Flask: The Python Micro Framework for Web**. 2021. Disponível em: https://flask.palletsprojects.com/.

RONSSE, M.; BOSSCHERE, K. D.; KERGOMMEAUX, J. Execution replay and debugging. 08 2000.

SAGIROGLU, S.; SINANC, D. Big data: A review. In: **2013 International Conference on Collaboration Technologies and Systems (CTS)**. San Diego, CA, USA: IEEE, 2013. p. 42–47.

SEREBRYANY, K. et al. Addresssanitizer: A fast address sanity checker. In: **Proceedings of the 2012 USENIX Conference on Annual Technical Conference**. USA: USENIX Association, 2012. (USENIX ATC'12), p. 28.

SEWARD, J. **Valgrind**. 2002. http://valgrind.org/.

SIGELMAN, B. H. et al. **Dapper, a Large-Scale Distributed Systems Tracing Infrastructure**. Mountain View, CA, USA, 2010. Disponível em: https://research.google.com/archive/papers/dapper-2010-1.pdf.

STEEN, M. v.; TANENBAUM, A. S. **Distributed Systems**. 3. ed. Enschede: Edição do autor, 2017. Disponível em: https://www.distributed-systems.net/.

The Ruby Programming Language. **debug.rb - Debugging functionality for Ruby**. 2006. Acesso em: 10 de Dezembro de 2021. Disponível em: https://github.com/ruby/debug.

THURLOW, R. **RFC 5531, RPC: Remote Procedure Call Protocol Specification Version 2**. USA: RFC Editor, 2009. Disponível em: https://www.rfc-editor.org/rfc/rfc5531.html.

TIERNEY, B. et al. The netlogger methodology for high performance distributed systems performance analysis. In: **Proceedings of the Seventh International Symposium on High Performance Distributed Computing**. Chicago, IL, USA: IEEE, 1998. p. 260–267.

GLOSSÁRIO

Big Data Processo que envolve a análise e interpretação de grandes volumes de dados variados.

breakpoint Um breakpoint é uma marcação estabelecida no código-fonte por um programador ou depurador, indicando um ponto onde a execução do programa deve ser interrompida. Quando um programa em execução alcança um breakpoint, ele é suspenso, permitindo ao programador inspecionar, modificar e controlar a execução do programa. Breakpoints são uma ferramenta fundamental em depuração, pois permitem que os programadores identifiquem e corrijam erros em seu código de forma interativa..

bug Falha ou erro num programa informático, dispositivo, site etc., que impede o seu funcionamento correto ou causa um mau desempenho.

hardware Conjunto dos equipamentos físicos que compõem um computador (dispositivos eletrônicos, monitor, placas, teclado etc.), juntamente com seus equipamentos periféricos (impressora, scanner etc.).

log Arquivo informático que, num computador, armazena todas as operações ou registros relevantes nele efetuadas; log de dados: um log pode ser usado para comprovar um crime cibernético.

offline/off-line Sem acesso à Internet; desconectado ou temporariamente inativo.

script Arquivos de script geralmente são apenas documentos de texto que contêm instruções escritas em uma determinada linguagem de script. Isso significa que a maioria dos scripts pode ser aberta e editada usando um editor de texto básico. No entanto, quando aberto pelo mecanismo de script apropriado, os comandos dentro do script são executados.

socket/soquete de rede Um ponto final de um fluxo de comunicação entre processos através de uma rede de computadores.

software Programa; reunião dos procedimentos e/ou instruções que determinam o funcionamento de um computador.

timeout Uma situação em que um programa de computador para de funcionar porque uma atividade demorou muito tempo para finalizar sua execução ou informar que ainda está processando.

token Na rede, um token é uma série de bits que circulam em uma rede token-ring. Quando um dos sistemas da rede possui o "token", ele pode enviar informações para os outros computadores. Como há apenas um token para cada rede token-ring, apenas um computador pode enviar dados por vez..

trace Encontrar a origem de algo. Um sinal de que algo aconteceu ou existiu.

watchpoint Um watchpoint é um tipo especial de breakpoint que causa a interrupção da execução do programa quando o valor de uma variável ou expressão específica muda ou atende a uma determinada condição. Em vez de se basear em uma localização específica no código (como um breakpoint tradicional), um watchpoint é ativado com base nas alterações de estado de um programa. Ele é particularmente útil quando um programador deseja rastrear mudanças inesperadas em variáveis ou quando é necessário analisar o comportamento de uma variável ao longo do tempo..

web Rede mundial de computadores; designação através da qual a Internet se tornou mundialmente conhecida.

APÊNDICE A – COMANDOS DE DEPURAÇÃO DO PDB

a (arguments): Mostra os argumentos que foram passados para a função atual.

b (breakpoint): Cria um *breakpoint* em uma determinada linha ou método.

c (continue): Avança o depurador até o próximo *breakpoint* ou até ocorrer uma exceção.

l (list): Lista algumas linhas do código que estão em volta da linha atual. Por padrão, serão apresentadas 11 linhas (5 acima e 5 abaixo).

n (next): Avança para a próxima linha do script.

p (print): Executa o comando print do python.

q (quit): Sai da execução do script.

r (return): Libera a execução do script até sair da função atual.

s (step into): Ao realizar a navegação através do comando **n**, o depurador não irá entrar em métodos que possivelmente forem invocados. Para que o depurador entre no método que está sendo invocado na linha corrente, basta trocar o comando **n**, pelo comando **s**.

ENTER: Ao pressionar a tecla ENTER sem nenhum comando no pdb, ele irá repetir o último comando executado.