UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

EDMUNDO MATHEUS BARBOSA DE SANTANA

RBINDER: uma solução para monitoramento transparente de aplicações baseadas em microsserviços

Recife
2019

EDMUNDO MATHEUS BARBOSA DE SANTANA

RBINDER: uma solução para monitoramento transparente de aplicações baseadas em microsserviços

Trabalho apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Redes de computadores e Sistemas distribuídos
Orientador: Nelson Souto Rosa

Recife
2019

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

S232r Santana, Edmundo Matheus Barbosa de
Rbinder: uma solução para monitoramento transparente de aplicações baseadas em microsserviços / Edmundo Matheus Barbosa de Santana. – 2019.
85 f.: il., fig., tab.

Orientador: Nelson Souto Rosa.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2019.
Inclui referências e apêndices.

1. Redes de computadores. 2. Sistemas distribuídos. 3. Avaliação de desempenho. I. Rosa, Nelson Souto (orientador). II. Título.

004.6 CDD (23. ed.) UFPE - CCEN 2021 – 35

Edmundo Matheus Barbosa Santana

“Rbinder: Uma Solução para Monitoramento Transparente de Aplicações Baseadas em Microsserviços”

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Aprovado em: 01/04/2019.

BANCA EXAMINADORA

_____

Prof. Dr. Vinicius Cardoso Garcia
Centro de Informática/UFPE

_____

Prof. Dr. Fernando Antônio Aires Lins
Departamento de Computação / UFRPE

_____

Prof. Dr. Nelson Souto Rosa
Centro de Informática/UFPE
(**Orientador**)

A todos os brasileiros que se esforçam para a construção de uma realidade social mais justa.

AGRADECIMENTOS

Sou uma pessoa muito privilegiada e não sei a quem posso agradecer por isso se não a Deus e a todas as pessoas a quem eu tive o privilégio de encontrar até agora nesta existência. Agradeço em especial aos meus familiares, que mesmo ausentes estão sempre comigo, aos meus professores, sempre excelentes ainda que não reconhecidos e valorizados, aos meus colegas de estudo e trabalho, que muito me inspiram, aos poucos mas sinceros amigos com que a vida me agraciou e mais especialmente ainda à minha esposa, com quem tenho compartilhado alegrias e perrengues desde a época da graduação.

Sou profundamente grato ao meu orientador, prof. Nelson S. Rosa, por sua paciência na orientação dos meus passos durante o curso, e ao seu orientando Adalberto Sampaio Jr. por sua valorosa contribuição a este trabalho. Agradeço também aos meus eternos amigos do Redu Educational Technologies, à Trustvox, à Interage Software House e ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), que me possibilitaram a concretização do sonho do mestrado.

RESUMO

Tracing tem sido aplicado ao estudo e entendimento do comportamento e desempenho de sistemas distribuídos. Apesar da atenção que o tópico tem recebido, dois importantes aspectos ainda são desafiadores ao contexto das aplicações baseadas em microsserviços: instrumentação de código e degradação de desempenho. Iniciativas de pesquisa tentam contornar a degradação com estratégias de amostragem dos *traces* gerados e coletados, e não contemplam desafios oriundos da grande heterogeneidade tecnológica da arquitetura de microsserviços como a dificuldade de instrumentá-los. Este trabalho apresenta o Rbinder: uma nova abordagem para *tracing* de microsserviços que une utilização de *proxies* e monitoramento de chamadas do sistema operacional. Os *proxies* reúnem todo o código relacionado à geração e coleta de *traces* enquanto o monitoramento de chamadas permite o diagnóstico das relações de causalidade existentes entre as mensagens. Uma avaliação do Rbinder mostra que o prejuízo causado ao desempenho de uma aplicação baseada em microsserviços monitorada é similar ao causado por soluções alternativas. No entanto, ele permite que os desenvolvedores se concentrem na lógica de negócio e não em sua instrumentação. Além disto, contempla a heterogeneidade intrínseca dos microsserviços por depender apenas de modificações na implantação da aplicação e dos mecanismos do sistema operacional em que ela é executada.

Palavras-chaves: Microsserviços. Monitoramento. *Tracing*. Instrumentação. Avaliação de Desempenho.

ABSTRACT

Tracing has been applied to study and understand the behavior and performance of distributed systems. Despite the attention this topic has received, two important aspects are still challenges in the context of microservice-based applications: source code instrumentation and performance overhead. Existing attempts resort on working around overhead (e.g., sampling techniques) and do not address microservices architecture's high technological heterogeneity challenges (e.g., instrumentation hassle). This work presents Rbinder: a novel approach for tracing microservices which joins proxies' usage (for handling tracing concerns) and operating system `syscall`s monitoring (for diagnosing causality between multiple requests). It makes advances on the field by completely separating instrumentation and application code while minimizing performance overhead. Rbinder's performance evaluation shows its impact on the execution of a microservice-based application is similar to the one posed by alternative solutions. Rbinder fosters developers' productivity by allowing them to focus on business logic instead of instrumentation and copes with the intrinsic heterogeneity of microservices by relying on deployment modifications and operating systems mechanisms solely.

Keywords: Microservices. Monitoring. Tracing. Instrumentation. Performance Evaluation.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Este capítulo apresenta o contexto geral e a motivação deste trabalho, que está relacionado à arquitetura de microsserviços e ao monitoramento de aplicações. Contexto e motivação são inicialmente apresentados e seguidos por apresentação e detalhamento do problema. Em seguida, apresentamos um sumário de soluções existentes na indústria de software e na academia e as limitações dessas soluções. A nossa proposta para solução do problema é, então, introduzida e a seção final do capítulo descreve a estrutura da dissertação.

1.1 CONTEXTO E MOTIVAÇÃO

Arquitetura de software é um tópico de pesquisa que ganhou importância com o aumento constante do tamanho e da complexidade de sistemas de software. A partir de então, encontrar maneiras efetivas para conceber e especificar as estruturas que compõem os sistemas se tornou mais importante do que evoluir seus algoritmos e suas estruturas de dados (GARLAN; SHAW, 1993).

Com o advento da Internet e o surgimento das aplicações criadas para a *World Wide Web* (WWW ou simplesmente web), essas estruturas começaram a ser concebidas de forma a se alinhar bem com os protocolos estabelecidos para a comunicação entre as máquinas. A arquitetura das aplicações passou por transformações inspiradas na arquitetura em camadas da Internet e nas facilidades providas pelas redes de computadores que levaram desde a utilização de múltiplas camadas arquiteturais (URGAONKAR et al., 2005) até a concepção de uma arquitetura baseada em serviços (ERL, 2005; PAPAZOGLOU, 2003).

A arquitetura baseada em serviços propunha a utilização de componentes de software que interagem entre si para realizar as funcionalidades do sistema de que fazem parte. Dela emergiu o estilo arquitetural de microsserviços (LEWIS; FOWLER, 2014), que propõe a esses componentes a adição de atributos como autonomia, desacoplamento e diversidade de tecnologia. Além disso, este estilo também prevê a utilização de protocolos leves para a comunicação, que se dá de forma coreografada (NEWMAN; MICROSERVICES, 2015).

A arquitetura de microsserviços tem sido amplamente adotada no enfrentamento a desafios relacionados ao projeto, desenvolvimento, manutenção e implantação de software, tais como: o desenvolvimento colaborativo entre times distribuídos; as restrições de tecnologias que podem ser utilizadas; e a dificuldade de modificação, implantação e mudança de escala de arquiteturas monolíticas. Ela promove a colaboração entre times distribuídos ao facilitar a definição de fronteiras entre os componentes do sistema, que podem ser desenvolvidos e implantados de maneira independente; melhora a resiliência dos sistemas ao permitir que seus componentes sejam mais facilmente replicados e conviver melhor com a ocorrência de falha, investindo em sua tolerância em vez de procurar evitá-la; e também

torna as aplicações mais escaláveis por adotar como princípio a automação de todos os processos operacionais.

Essas facilidades têm seu custo: um ecossistema de serviços pequenos, altamente heterogêneos e replicados impõe novos desafios (LEWIS; FOWLER, 2014). Um deles é a dificuldade de depurar problemas complexos que acontecem em ambientes de produção. Um *bug* em um componente específico de uma arquitetura baseada em microsserviços pode, por exemplo, aumentar a latência experimentada por usuários e não ser detectável através de códigos de erro HTTP nem por outros mecanismos que se baseiam somente em dados fornecidos pelas aplicações e os protocolos que elas implementam.

Outro desafio é alocar os componentes da aplicação baseada em microsserviços de maneira a satisfazer uma dada propriedade do sistema como disponibilidade ou latência. Os componentes poderiam, por exemplo, ser co-alocados para diminuir latência de rede ou espalhados em vários nós para diminuir a competição pelos recursos disponíveis. Quaisquer que sejam os objetivos, é mandatório conhecer como os componentes estão dispostos e quais relações de dependência existem entre eles. Vale perceber que essas relações e alocações podem, na prática, não coincidir com o que foi projetado.

A alocação (SAMPAIO et al., 2017) e a depuração (SAMBASIVAN et al., 2011; SHARMA et al., 2015) de componentes da aplicação são problemas que têm sido abordados com o auxílio de ferramentas para *tracing* distribuído (FONSECA et al., 2007; KALDOR et al., 2017; MACE; ROELKE; FONSECA, 2015; SIGELMAN et al., 2010; TAK et al., 2009; WASSERMANN; EMMERICH, 2011). No entanto, elas geralmente exigem que desenvolvedores modifiquem o código-fonte de suas aplicações e / ou adotem dependências de software para habilitação do monitoramento. Além disso, o monitoramento tende a causar prejuízo ao desempenho do sistema monitorado. Os problemas relacionados à sobrecarga são comumente contornados com técnicas de amostragem enquanto instrumentação nos níveis de bibliotecas e *middleware* é aplicada para mitigar a necessidade de modificação de código. Apesar da popularidade, essas estratégias não são soluções ideais porque podem dificultar a identificação da ocorrência *bugs* e ainda exigem modificações de código.

O grande esforço normalmente necessário para instrumentação de código motivou a proposição de estratégias de instrumentação mais baratas em vários campos de pesquisa como os de sistemas operacionais e sistemas distribuídos (LUK et al., 2005; WANG; SANCHEZ; HERKERSDORF, 2008). No entanto, a sua diminuição em relação ao aumento causado pelos desafios que emergiram da arquitetura de microsserviços ainda é uma questão em aberto. Este trabalho fornece uma resposta a essa questão através da união do que foi proposto nos contextos i) de padronização da observabilidade de aplicações baseadas em microsserviços; e ii) de monitoramento não-intrusivo da execução de processos de sistemas operacionais.

1.2 CARACTERIZAÇÃO DO PROBLEMA

A complexidade inerente às aplicações baseadas em microsserviços faz com que o monitoramento de seu comportamento e a depuração de seus erros seja muito mais difícil que monitoramento e depuração de aplicações monolíticas. Nesse contexto, o monitoramento é importante porque permite a exposição do comportamento da aplicação, facilitando não apenas a depuração de erros como também a obtenção de melhorias de desempenho.

No entanto, a diversidade tecnológica e a alta quantidade de componentes, que estão entre as principais características da arquitetura de microsserviços, dificultam o monitoramento por que tornam a instrumentação mais trabalhosa e aumentam o prejuízo causado ao desempenho da aplicação monitorada. Diante dessas reflexões, a questão de pesquisa que este trabalho considera é:

- Como reduzir o investimento de tempo e dinheiro para habilitação do monitoramento de aplicações baseadas em microsserviços e como, uma vez que ele está habilitado, diminuir o prejuízo ao desempenho das aplicações monitoradas e o aumento do consumo de recursos computacionais devido à sua habilitação?

Além do custo de tempo e dinheiro relacionado à instrumentação de código necessária ao monitoramento, as possíveis respostas à pergunta devem levar em consideração a sobrecarga imposta enquanto o monitoramento está em vigor. Isto é, o custo em questão não é apenas definido em função do investimento necessário para permitir que a aplicação seja monitorada, mas também em função da degradação do desempenho causada pelo monitoramento.

1.3 ESTADO DA ARTE

Microsserviços é um tópico de pesquisa quente na atualidade e muitos dos seus desafios e oportunidades têm sido explorados pela comunidade científica (PAHL; JAMSHIDI, 2016; DRAGONI et al., 2017; SOLDANI; TAMBURRI; HEUVEL, 2018). As iniciativas de pesquisa mais importantes propõem a utilização de microsserviços no contexto de computação em nuvem e destacam como a arquitetura favorece a operação de aplicações nesse contexto (BALALAIE; HEYDARNOORI; JAMSHIDI, 2016).

As fronteiras arquiteturais sugeridas pelos microsserviços e as fronteiras computacionais definidas através de virtualização por *containers* casam muito bem para aplicação no contexto de computação em nuvem. Isso porque, juntas, elas favorecem atributos importantes de aplicações distribuídas em nuvem como tolerância a falhas, escalabilidade e elasticidade. Esses fatores influenciam o sucesso da utilização e a popularidade da adoção de microsserviços como principal estilo arquitetural para projeto de aplicações em nuvem. Na indústria, grandes corporações como Amazon e Netflix têm utilizado microsserviços

para tornar mais eficientes o desenvolvimento de aplicações e a utilização de servidores, provendo uma melhor experiência aos seus usuários finais.

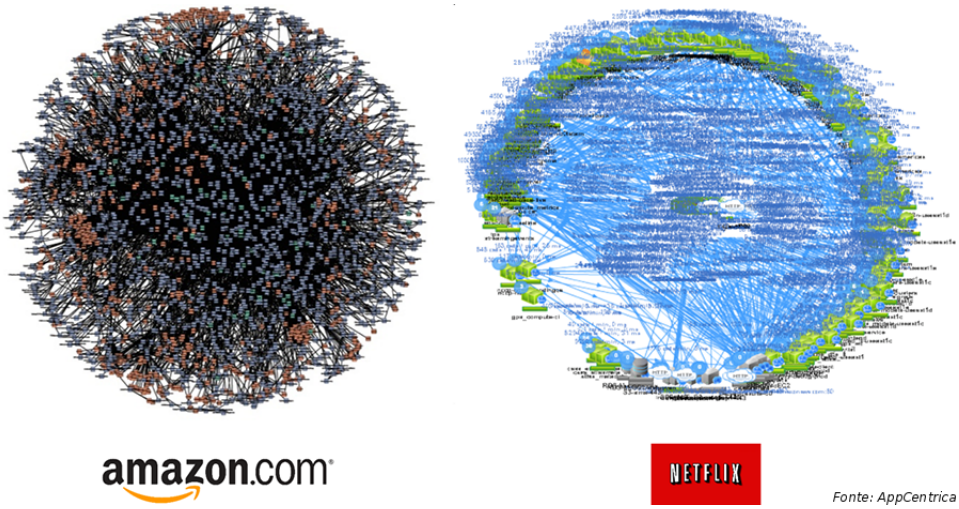


Figura 1 – Utilização de microsserviços nas empresas Amazon e Netflix. Os grafos representam microsserviços em ambientes de produção reais. Os nós dos grafos indicam microsserviços e as arestas indicam as relações de dependência entre eles.

A Figura 1 apresenta dois grafos que representam uma visualização de microsserviços em ambientes de produção reais das empresas Amazon e Netflix (APPCENTRICA, 2016). Nós nos grafos representam microsserviços enquanto as arestas entre os nós representam dependências entre eles. A visualização dá uma dimensão da complexidade de grandes aplicações baseadas em microsserviços. Essa complexidade aumenta a dificuldade de entendimento do comportamento e depuração de problemas dessas aplicações. O aumento da dificuldade ratifica a necessidade de monitorá-las para entendê-las e depurá-las.

Outros trabalhos de pesquisa que se destacam são os que propõem aplicações baseadas em microsserviços para uso da comunidade científica (ADERALDO et al., 2017; BROGI et al., 2017; GAN et al., 2019). A definição dessas aplicações é importante porque possibilita a comparação de soluções sugeridas e a reprodução de experimentos realizados por trabalhos voltados à arquitetura de microsserviços. Para o monitoramento de microsserviços, a possibilidade de comparação de soluções e reprodução de experimentos é fundamental devido ao impacto que o monitoramento tem sobre o desenvolvimento e a operação das aplicações monitoradas. Apesar das proposições já existentes de aplicações para uso em trabalhos acadêmicos, ainda não há consenso sobre as aplicações que devem ser utilizadas e muitas vezes os trabalhos que propõem avanços em microsserviços têm de elaborar suas próprias aplicações ou utilizar alternativas fornecidas pela indústria (SAMPAIO et al., 2017; CAROSI, 2018).

O tópico de monitoramento de sistemas distribuídos, em contraste com o que acontece com microsserviços, tem sido alvo de pesquisa há bastante tempo (JOYCE et al., 1987;

ACETO et al., 2013). Ainda assim, a evolução de hardware, software e comunicação entre os computadores não permite o esgotamento dos problemas a serem pesquisados. Em especial, os avanços na área de redes de computadores para combate das limitações arquiteturais da Internet fazem emergir novos desafios que demandam a evolução de estratégias de monitoramento já propostas e o desenvolvimento de novas estratégias (SUN et al., 2011; CHOWDHURY et al., 2014; LIU et al., 2016).

No contexto da computação em nuvem, uma técnica de monitoramento importante é o *tracing* (BARHAM et al., 2004; FONSECA et al., 2007; SIGELMAN et al., 2010; KALDOR et al., 2017; MACE; ROELKE; FONSECA, 2018). Ela consiste no rastreamento das interações que acontecem entre os componentes que constituem a aplicação distribuída e permite conhecer as relações de dependência entre os componentes e o comportamento de cada um deles. As relações de dependência ficam explícitas pelas interações que acontecem entre os componentes, e.g., um componente faz uma solicitação a outro componente de que ele depende. Ao mesmo tempo, o comportamento de cada microsserviço da aplicação pode ser capturado pelo tempo que ele demora para fornecer uma resposta às solicitações recebidas, por exemplo.

O surgimento da arquitetura de microsserviços impõe novos desafios à utilização de *tracing* para monitoramento de aplicações baseadas nessa arquitetura. Kitajima & Matsuoka (KITAJIMA; MATSUOKA, 2017), por exemplo, propõem uma abordagem imprecisa para diagnóstico de causalidade entre requisições trocadas entre microsserviços. Já Gan et al (GAN et al., 2018) aplica *tracing* em conjunto com *big data* para prever problemas de desempenho em aplicações baseadas em microsserviços.

A maior parte dos trabalhos relacionados tenta contornar o prejuízo causado ao desempenho das aplicações monitoradas através da amostragem de dados, i.e., habilitação de *tracing* de apenas um percentual das interações entre os componentes. Essa abordagem não é ideal porque pode mascarar problemas que não acontecem frequentemente. Já o custo associado à modificação de código necessária para habilitação do *tracing* é comumente reduzido através da instrumentação de *frameworks*, *middleware* e bibliotecas usadas para construir as aplicações. No entanto, essa estratégia ainda impõe custos aos desenvolvedores das aplicações. Esses custos são especialmente altos no contexto de aplicações baseadas em microsserviços devido à grande quantidade de componentes normalmente envolvidos e à diversidade de tecnologias utilizadas na sua implementação.

1.4 RBINDER

Este trabalho apresenta o Rbinder: uma solução para monitoramento transparente de aplicações baseadas em microsserviços. Como uma solução transparente, ele diminui o custo de instrumentação associado ao *tracing* sem causar grande prejuízo ao desempenho das aplicações monitoradas. Desta forma, as principais contribuições são:

- Uma estratégia de monitoramento para habilitação de *tracing* preciso (i.e., o monitoramento aponta tudo o que acontece com acurácia) sem modificação do código-fonte da aplicação monitorada;
- Uma implementação para monitoramento de chamadas do sistema operacional Linux para propagação de informações de *tracing*; e
- Uma avaliação de desempenho de estratégias de monitoramento de aplicações baseadas em microsserviços que apresenta a degradação de desempenho devida ao monitoramento;

A estratégia utiliza a técnica de *tracing* de sistemas distribuídos proposta, implementada e reportada pelo Google em sua ferramenta Dapper (SIGELMAN et al., 2010). Ela é similar a outras utilizadas em vários trabalhos acadêmicos e se tornou padrão para monitoramento de microsserviços na indústria. O mecanismo fornecido por eles se utiliza da propagação de informações de identificação das mensagens e de envio de informações a um servidor de *traces* para rastreamento do comportamento dos sistemas monitorados. Existem, portanto, duas fontes de custo que acarretam esforço extra aos desenvolvedores que querem utilizá-lo: as modificações necessárias para propagação das informações de identificação e as que são necessárias para envio de informações ao servidor de *traces*.

Para reduzir ao máximo o custo da habilitação de *tracing*, a solução proposta utiliza *proxies* que se encarregam do envio de informações ao servidor de *traces*. Dessa maneira, as aplicações monitoradas não precisam implementar código nem adotar dependências (bibliotecas ou *frameworks* de instrumentação, por exemplo) que façam esse envio. Essa já é uma prática proposta por plataformas de gerenciamento de microsserviços em uso na indústria e se alinha bem com o nosso objetivo de minimização de custo para habilitação de *tracing*.

A propagação de informações de identificação das mensagens é um desafio para propostas de *tracing* transparente, i.e., que não dependem de modificação do código da aplicação monitorada. Até mesmo as estratégias que sugerem utilização de *proxies* ainda dependem da modificação do código da aplicação para implementá-la. Propomos o monitoramento de chamadas do sistema operacional para realizar a propagação dessas informações sem modificar o código dos componentes monitorados.

A estratégia sugerida aqui é, portanto, composta por dois blocos básicos: a utilização de *proxies* para envio de informações de *tracing* e o monitoramento de chamadas do sistema operacional para propagação de informações de identificação de mensagens. Já existem *proxies* disponíveis que implementam os requisitos necessários aos nossos propósitos. Assim, quanto à utilização de *proxies*, fornecemos algumas diretrizes sobre a sua implantação no contexto de microsserviços que caracterizam uma contribuição secundária. O monitoramento de chamadas do sistema operacional, por outro lado, é uma contribui-

ção original deste trabalho. Por isso, a estratégia para esse monitoramento é explicada em detalhes e uma implementação dela é fornecida e avaliada.

O impacto do *tracing* no desempenho das aplicações monitoradas também foi avaliado. Foi realizada uma avaliação de desempenho da estratégia proposta. A avaliação teve como principal objetivo a comparação da estratégia com outras abordagens de monitoramento existentes.

1.5 ESTRUTURA DO DOCUMENTO

Este documento está estruturado como segue. O Capítulo 2 explica de maneira mais detalhada os principais conceitos necessários ao entendimento deste trabalho. No Capítulo 3, a nossa estratégia é explicada detalhadamente. O Capítulo 4 apresenta uma avaliação comparativa da estratégia proposta com soluções existentes. O Capítulo 5 resume e discute os principais trabalhos relacionados. O Capítulo 6 apresenta as conclusões, limitações e trabalhos futuros.

2 CONCEITOS BÁSICOS

Este capítulo apresenta os principais conceitos necessários ao entendimento deste trabalho. Ele introduz inicialmente a arquitetura de microsserviços apresentando a motivação para o seu surgimento e seus principais princípios, práticas, oportunidades e desafios. Em seguida, apresentamos estratégias comuns para o monitoramento de sistemas de software. Depois, aprofundamos a apresentação sobre monitoramento no contexto de sistemas distribuídos, dando ênfase à utilização de *tracing* como método para monitoramento de aplicações distribuídas. Por fim, apresentamos duas possibilidades para monitoramento transparente: a interceptação de comunicação em sistemas distribuídos e o monitoramento de chamadas do sistema operacional.

2.1 MICROSSERVIÇOS

O estilo arquitetural de microsserviços é uma derivação da Arquitetura Orientada a Serviços (em inglês, *Service-Oriented Architecture* – SOA) (DRAGONI et al., 2017). Ele propõe a utilização de componentes de software pequenos, independentes e especializados que interagem entre si para prover as funcionalidades do sistema de que fazem parte. O que torna os microsserviços particularmente úteis, e também desafiadores, é que eles levam a granularidade de arquiteturas orientadas a serviços a um nível extremo: cada componente deve resolver um único problema do domínio do negócio e ser desenvolvido, implantado e executado de maneira independente (NEWMAN; MICROSERVICES, 2015).

Uma aplicação de comércio eletrônico, por exemplo, pode ser arquitetada com microsserviços como ilustrado na Figura 2. Nela, são exibidos 11 microsserviços que implementam a funcionalidade de realização de uma venda para um usuário do comércio eletrônico e as relações de dependência que existem entre eles.

O componente central da arquitetura é `pedidos`, que recebe a solicitação de realização da compra. Ele interage com `usuários` para obter informações do usuário comprador, com `carrinhos` para recuperar os itens que o usuário deseja comprar, com `pgmtos` para verificar status e condições de pagamento da compra a ser finalizada e com `entregas` para agendar o envio dos itens adquiridos para o endereço do comprador. Os componentes cujos nomes terminam em `bd` são bancos de dados de quem os microsserviços dependem para armazenar informações relacionadas ao domínio de cada um deles. `pedidosmsgr` é um serviço mensageiro que `pedidos` utiliza para notificar a finalização de um pedido.

Os principais benefícios da utilização de uma arquitetura como a da Figura 2 incluem facilidade de modificação, escalabilidade e heterogeneidade da aplicação. Se o código da aplicação estivesse todo compreendido num único componente (arquitetura monolítica) em vez de separado em diversos componentes, a implementação de modificações seria mais

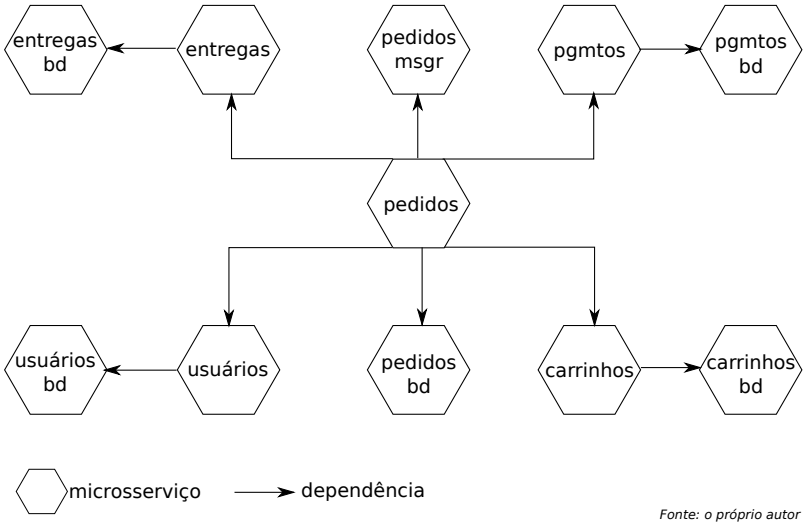


Figura 2 – Exemplo de arquitetura de microsserviços. Cada microsserviço é representado por um hexágono e as setas indicam as dependências entre eles, e.g., o microsserviço `usuários` depende do microsserviço `usuáriosbd`.

custosa. Os desenvolvedores precisariam entender uma porção maior de código para aplicá-la e levar mais tempo executando a suíte de testes de toda a aplicação. A implantação também seria mais demorada porque teria de considerar todo o monolito. Além disso, eventuais erros graves que fossem introduzidos afetariam todo o sistema enquanto na arquitetura de microsserviços eles podem ser isolados nos componentes defeituosos.

A escalabilidade da aplicação também é favorecida porque é possível aumentar a escala através da replicação apenas dos componentes que precisam ser escalados enquanto numa arquitetura monolítica toda a aplicação tem que ser replicada. Isso facilita a operacionalização da aplicação já que os componentes podem ser alocados com mais flexibilidade para respeitar os limites de recursos computacionais disponíveis.

A heterogeneidade da aplicação é favorecida porque é mais fácil utilizar tecnologias diferentes para a implementação de cada um dos componentes. Isso permite que se escolha a melhor linguagem de programação, *framework* ou biblioteca disponível para resolver o problema que o microsserviço foi designado a atacar.

A utilização de uma arquitetura de granularidade tão fina impõe vários desafios que não existem ou são mais fáceis de tratar no contexto de arquiteturas monolíticas. A quantidade de decisões de projeto que precisam ser tomadas aumenta consideravelmente: quais tecnologias utilizar? Como estruturar os times de desenvolvimento? Como decidir a granularidade dos componentes? Além disso, algumas dificuldades estão relacionadas à operação da aplicação: como gerenciar a implantação simultânea de vários componentes? Como alocar os componentes na infraestrutura em que eles serão executados?

O desafio que mais nos interessa está relacionado à dificuldade de rastreamento do comportamento de uma aplicação que normalmente é composta por vários componentes

implementados em diversas tecnologias. O monitoramento de uma aplicação baseada em microsserviços é mais complexo porque demanda a captura do comportamento de suas diversas partes. Além disso, a interação entre os componentes aumenta a dificuldade porque ela pode se dar através de diferentes protocolos de comunicação como o *HyperText Transfer Protocol* (HTTP) ou o *general-purpose Remote Procedure Calls* (gRPC). A estratégia de monitoramento deve, portanto, ser capaz de lidar com a diversidade tanto dos componentes quanto dos protocolos de comunicação que eles utilizam para interagir. Isso restringe a utilização de métodos de monitoramento tradicionais. Geração e coleta de *logs*, por exemplo, tem um custo muito elevado quando o sistema monitorado é composto por centenas de componentes implementados em muitas linguagens de programação, *frameworks* e bibliotecas diferentes que se comunicam através de múltiplos protocolos.

2.2 MONITORAMENTO

O monitoramento de programas de computador é uma prática útil para a garantia do bom funcionamento dos programas monitorados pois permite conhecer seu comportamento em ambientes de produção, i.e., nos ambientes em que eles são de fato utilizados por usuários finais. Por mais criteriosos que sejam os processos e as práticas seguidas para o desenvolvimento de software, eles não asseguram o comportamento livre de falhas das aplicações quando elas são submetidas a cargas de trabalho reais. A variabilidade do tipo dessas cargas em ambientes de produção, por exemplo, pode ocasionar problemas imperceptíveis em outros ambientes (e.g., desenvolvimento e testes) (ARDELEAN; DIWAN; ERDMAN, 2018).

As estratégias mais comuns de monitoramento são *profiling* e *tracing* (SHENDE, 1999). A primeira envolve coleta de estatísticas sobre a execução de sistemas (utilização de recursos computacionais, por exemplo) enquanto a segunda se preocupa com questões temporais (quanto tempo um processo levou para ser executado, por exemplo). Além de permitir a identificação de mau funcionamento, essas estratégias são especialmente úteis para depuração de problemas e implementação de melhorias relacionadas ao desempenho dos sistemas.

A aplicação de estratégias de monitoramento normalmente envolve três etapas:

- Geração de dados de monitoramento;
- Coleta dos dados gerados; e
- Análise e apresentação dos dados coletados;

Uma prática muito comum para a geração de dados de monitoramento é o *logging*. Nela, o próprio programa monitorado ou um agente de monitoramento coleta informações sobre a execução do programa e as escreve em um arquivo de *logs*. A Listagem 2.1

apresenta um exemplo de arquivo de *logs* de *profiling*. Nela, o consumo de CPU em uma determinada instância computacional (e.g., uma máquina virtual) é apresentado. A medição do consumo é feita uma vez a cada segundo e a primeira coluna exibe um rótulo de tempo (`horas:minutos:segundos`) enquanto a segunda mostra o percentual de consumo de CPU naquele instante. Os *logs* permitem observar que houve um pico no consumo às 13:52:53 que perdurou por mais 2 segundos (linhas 6-8). Investigações e análises adicionais teriam que ser feitas para se descobrir o que causou o pico de consumo. As informações de *profiling* poderiam, por exemplo, ser cruzadas com informações de *tracing* para verificar o que estava sendo executado nos instantes em que o consumo de CPU foi maior. O *profiling*, no entanto, permite um primeiro diagnóstico de um eventual problema ou possível melhoria de desempenho.

Listagem 2.1 – Exemplo de arquivo de *logs* de *profiling*. Cada linha apresenta o consumo de CPU num dado instante (horas, minutos e segundos).

```
1  13:52:48 5.4%
   13:52:49 5.4%
3  13:52:50 5.5%
   13:52:51 5.5%
5  13:52:52 5.5%
   13:52:53 98.5%
7  13:52:54 94.5%
   13:52:55 90.5%
9  13:52:56 5.5%
   13:52:57 5.5%
```

Alternativas a essa abordagem de escrita de *logs* em arquivos costumam mudar a forma como os dados são armazenados. As mesmas informações de *profiling* apresentadas na Listagem 2.1, por exemplo, poderiam ser armazenadas num banco de dados relacional em que cada linha do arquivo é registrada como uma linha numa tabela com duas colunas. Como estratégias de monitoramento normalmente geram um grande volume de dados, a decisão sobre como armazená-los é importante porque impacta na latência para sua inserção e recuperação. A importância dessa questão fez surgirem várias propostas de utilização de ferramentas existentes ou desenvolvimento de novas ferramentas para esse contexto. O sistema para *tracing* distribuído Dapper (SIGELMAN et al., 2010), por exemplo, utiliza o banco de dados distribuído Bigtable (CHANG et al., 2008).

A fase de coleta de dados de monitoramento consiste na aquisição de todos os dados gerados por programas monitorados ou por seus agentes de monitoramento. No contexto de computação em nuvem, por exemplo, em que aplicações são implantadas em um *cluster* que contém várias máquinas virtuais, cada uma das máquinas virtuais geraria um arquivo de *logs* parecido com o apresentado na Listagem 2.1. Para que todas as informações de *profiling* sejam consideradas conjuntamente, é necessário agrupá-las de alguma maneira. Uma solução para o contexto do exemplo apresentado seria implementar um *script* que acessa remotamente as máquinas virtuais para recuperar o arquivo de *logs* presente em seus sistemas de arquivos.

Após a aquisição de todos os dados de monitoramento, um passo normalmente útil é o de análise e apresentação dos dados coletados. A análise dos dados consiste na agregação de informações enquanto a apresentação busca formas mais adequadas para apresentar as informações agregadas. Os dados apresentados no exemplo da Listagem 2.1 poderiam, por exemplo, ser agregados para representar o consumo de CPU a cada 5 segundos em vez de a cada 1 segundo. Já uma forma de apresentação que pode ser mais útil a desenvolvedores é a exibição dessas informações através de um gráfico de linhas, por exemplo, que permitiria a identificação de picos de consumo mais facilmente.

Existem vários desafios e questões interessantes relacionados ao monitoramento de programas de computador. Alguns deles têm relação com a problemática da manipulação dos grandes volumes de dados gerados pelo monitoramento. É necessário estabelecer, por exemplo, a precisão da medição (e.g., 100 medições por segundo, 1 medição a cada 5 segundos) e uma política de retenção de dados, i.e., por quanto tempo manter armazenados os dados coletados. Há também desafios relacionados ao desenvolvimento de software como a necessidade de escrever e manter código que é utilizado somente a propósito do monitoramento. Essa necessidade pode tomar tempo considerável dos desenvolvedores de sistemas e o código de monitoramento pode se misturar ao código da lógica de negócio, dificultando a manutenção do código do sistema.

Um desafio que se destaca é o prejuízo do desempenho dos sistemas monitorados. A observação da execução de um programa sempre acarreta alguma degradação do seu desempenho porque ele acumula responsabilidades (no caso do auto-monitoramento) ou compete por recursos computacionais (no caso de ser monitorado por um agente externo). Em qualquer um desses casos, o prejuízo ao desempenho do sistema monitorado é uma preocupação importante e estratégias de monitoramento devem minimizá-lo para cumprir seus objetivos de maneira efetiva.

2.3 *TRACING* DE APLICAÇÕES DISTRIBUÍDAS

O monitoramento de sistemas distribuídos é uma prática utilizada e estudada há bastante tempo para depuração e rastreamento de comportamento desses sistemas (JOYCE et al., 1987). Ela envolve a geração e coleta de dados que refletem eventos de interesse que acontecem durante a execução de um programa. Uma das estratégias de monitoramento mais difundidas é o `logging` (JOHNSON; ZWAENEPOEL, 1987; JOHNSON; ZWAENEPOEL, 1990; TIERNEY et al., 1998; FU et al., 2009; LOU et al., 2010), que consiste no registro de eventos à medida em que a execução do programa avança. Esse registro normalmente é feito pelo próprio programa em arquivo no armazenamento em massa (disco rígido) ou em memória volátil (RAM).

A evolução dos sistemas distribuídos em arquiteturas mais granulares, em que cada sistema é composto por dezenas ou centenas de componentes, impôs novos desafios para as abordagens de monitoramento. O rastreamento da interação entre os processos passou

a ter tanta importância para o monitoramento quanto o detalhamento do comportamento de cada processo. Com isso, a organização de *logs* de eventos em *traces* do sistema, técnica conhecida como *tracing*, ganhou espaço entre as abordagens para monitoramento de sistemas distribuídos.

Tracing é uma estratégia para entender o comportamento de sistemas distribuídos, implementar otimizações de desempenho e remover *bugs* (JOYCE et al., 1987; MATTERN et al., 1989; RAYNAL; SINGHAL, 1996). No contexto de arquiteturas orientadas a serviços, ela habilita a captura e a exposição do comportamento dos componentes através do diagnóstico das relações de causalidade das mensagens trocadas entre eles. Sistemas de *tracing* como X-Trace (FONSECA et al., 2007) e Dapper (SIGELMAN et al., 2010) implementam essa estratégia através da propagação de informação de *tracing* que trafega junto às mensagens trocadas. Dessa maneira é possível fornecer um diagnóstico preciso das relações de causalidade. Na prática, essa estratégia para monitoramento utiliza instrumentação e propagação de meta-informação para modelar e reportar o comportamento de sistemas.

A Figura 3 mostra uma aplicação baseada em microsserviços composta por cinco componentes diferentes que servem duas requisições de usuário. As diferentes texturas dos hexágonos são utilizadas para diferenciar os microsserviços no exemplo de visualização de *traces* que é apresentado mais adiante. Apesar da possibilidade de observação da execução, e.g., através de *logs*, não há como diferenciar as requisições que são causadas por uma ou por outra requisição do usuário.

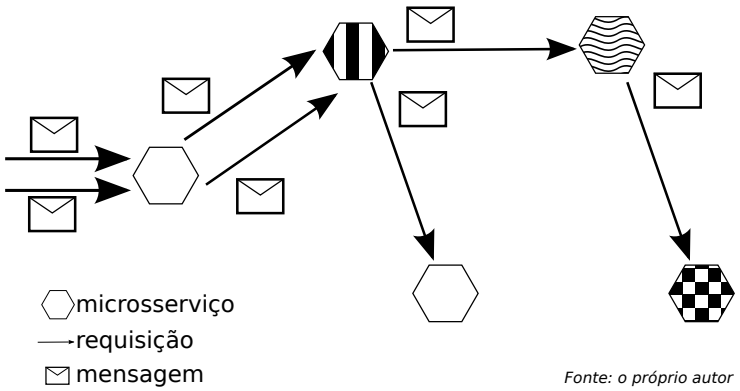
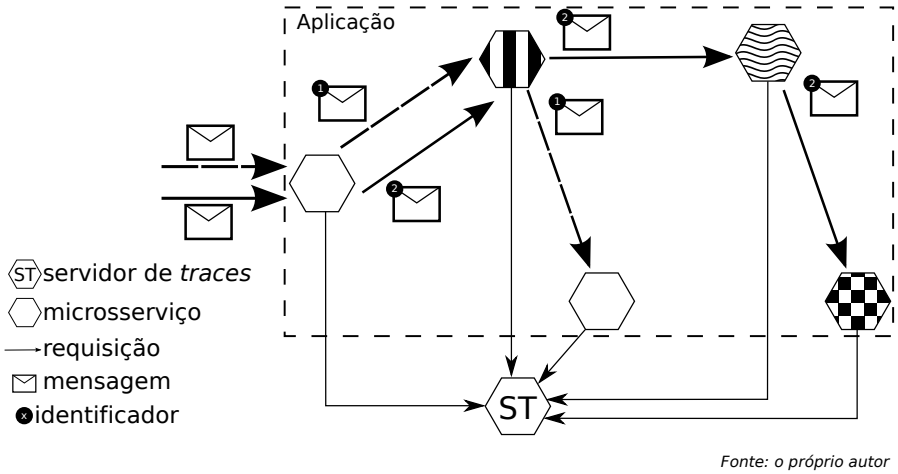


Figura 3 – Fluxo de mensagens entre microsserviços que não propagam identificadores de caminhos de requisições nem enviam informações de *tracing* a um servidor de *traces*. Hexágonos representam microsserviços e as setas representam requisições.

A maneira mais comum para habilitação de *tracing* exigiria que todos os microsserviços fossem instrumentados. O primeiro no caminho das requisições teria de ser alterado para gerar identificadores para cada uma das requisições de usuário e possivelmente outras informações relacionadas ao *tracing* como a decisão de amostragem, i.e., a decisão de reportar ou não um caminho de requisições. Todos os microsserviços teriam de ser instru-

mentados para propagar os identificadores e enviar informações de *tracing* a um servidor de *traces*. A propagação de identificadores é crucial para o diagnóstico de causalidade das mensagens trocadas entre os microsserviços. O servidor de *traces* é uma ferramenta que recebe, consolida e exibe informações de *tracing*. Ele pode ser implantado como um microsserviço.

A Figura 4 exibe os mesmos microsserviços exibidos na Figura 3 modificados para gerar e reportar informações de *tracing*. Ela também conta com a adição do servidor de *traces* para onde as informações de *tracing* são enviadas. Com a habilitação do monitoramento, é possível distinguir caminhos de requisições diferentes: a textura tracejada ou sólida das setas e o número no canto superior esquerdo dos envelopes indicam os diferentes caminhos relacionados a cada uma das requisições de usuário.



Fonte: o próprio autor

Figura 4 – Fluxo de mensagens entre microsserviços que propagam identificadores de caminhos de requisições e enviam informações de *tracing* a um servidor de *traces*. Hexágonos representam microsserviços e as setas representam requisições.

Essa estratégia depende da geração, propagação e envio de informações específicas de *tracing* que dizem respeito a dois conjuntos de informações: um com dados que devem ser propagados junto às mensagens da aplicação e outro com dados que devem ser enviados ao servidor de *traces*.

O primeiro conjunto deveria ser o menor possível porque ele impacta no tamanho das mensagens da aplicação. Ele normalmente inclui os identificadores de requisições e decisões de amostragem e trafegam junto aos dados da aplicação através dos mesmos protocolos que a aplicação utiliza para troca de mensagens entre seus componentes. No caso de HTTP, por exemplo, informações de *tracing* trafegam como cabeçalhos das requisições HTTP como `X-Request-Id: 3`, que indica uma requisição pertencente ao caminho relacionado à requisição de usuário identificada pelo número 3.

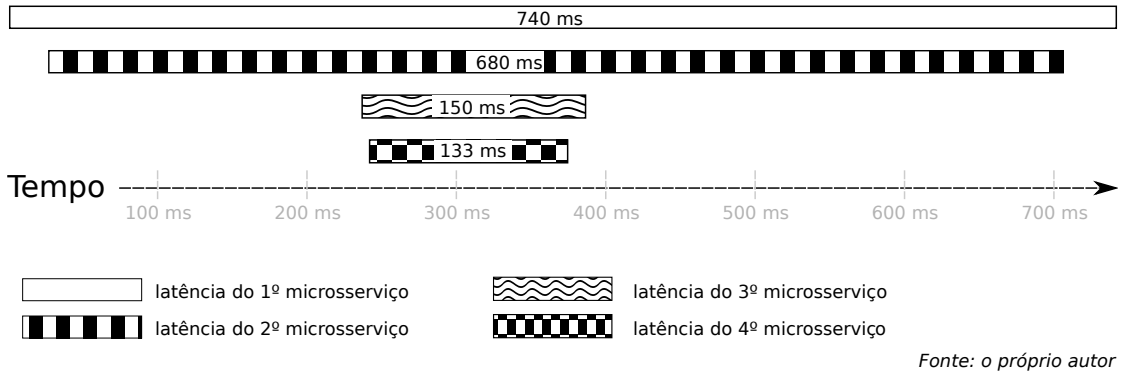
O segundo conjunto também inclui os identificadores de requisições mas podem não incluir outras informações do primeiro conjunto como as decisões de amostragem. Ele nor-

malmente contém informações de tempo sobre as requisições (quando elas foram iniciadas e finalizadas, por exemplo) e pode incluir versão de protocolo, informações específicas da aplicação, atributos da mensagem (tamanho, conteúdo) e assim por diante. Ele é formatado e enviado ao servidor de *traces* de acordo com um protocolo de comunicação que não precisa ser o mesmo usado entre os microsserviços da aplicação. A Listagem 2.2 apresenta um exemplo de JSON que poderia ser enviado via HTTP.

```
1 {
2   "requestId": 3,
3   "traceId": 2,
4   "parentRequestId": 2,
5   "startedAt": 1528921301430,
6   "finishedAt": 1528921518036,
7   "origin": "service2",
8   "destiny": "service3"
9 }
```

Listagem 2.2 – Exemplo da estrutura de informações de *tracing*

Servidores de *traces* usam informações como essas para agregar dados e exibir o comportamento do sistema através de ferramentas que facilitam as atividades de desenvolvedores e operadores de microsserviços. Essas atividades são comumente a depuração e a implementação de melhorias de desempenho do sistema monitorado. Para esse fim, um tipo de visualização que tem se mostrado útil é a disposição de *traces* do sistema em gráficos de árvore (SIGELMAN et al., 2010) como o exibido na Figura 5.



Fonte: o próprio autor

Figura 5 – Árvore de *traces* para a requisição 2 que é exibida na Figura 4. A ordem mencionada nas legendas faz referência à ordem em que cada microsserviço aparece no caminho de requisições.

A Figura 5 mostra o caminho de requisições que foram feitas para servir a requisição de usuário identificada com o rótulo 2 na Figura 4. As barras horizontais representam as latências para cada uma das requisições envolvidas no caminho. A textura delas indica o microsserviço responsável por servi-la. A latência é representada tanto pela largura da barra quanto pelo rótulo presente em sua porção central – que informa a latência absoluta em microssegundos. A latência indicada na barra do topo, que representa o processamento

pelo microsserviço que recebe a requisição inicial, também indica a latência percebida pelo usuário. A análise do gráfico permite concluir que a maior parte do tempo é consumida pelo segundo microsserviço no caminho de requisições. Essa informação indica um possível caminho para depuração de problemas ou implementação de melhoria de desempenho.

Além da dependência de informações específicas, *tracing* também depende da instrumentação do código dos microsserviços para que gerem, propaguem e enviem informações. Existem vários métodos de instrumentação de código para habilitar essa estratégia. Eles incluem desenvolvimento ad-hoc e utilização de bibliotecas. No desenvolvimento ad-hoc, cada microsserviço seria modificado para atender aos requisitos do monitoramento. Mesmo que se considere as possibilidades de reuso de código fornecidas por cada linguagem de programação, esse método é muito custoso porque demanda do desenvolvedor conhecimento sobre o mecanismo de *tracing* para implementar e manter o código de instrumentação. A utilização de bibliotecas também pode ter alto custo por demandar conhecimento sobre como utilizá-las. A diversidade de tecnologias (linguagens de programação, *frameworks* e bibliotecas) utilizadas para implementar microsserviços aumenta ainda mais o custo da implementação desses métodos.

2.4 INTERCEPTAÇÃO DE COMUNICAÇÃO EM SISTEMAS DISTRIBUÍDOS

A interceptação de comunicação é útil quando se quer monitorar o comportamento de componentes de software que se comunicam entre si e a instrumentação de seu código não é uma possibilidade porque, por exemplo, o código pode não estar disponível. A observação da comunicação entre os componentes permite inferir o comportamento geral do sistema. Isso é especialmente válido no contexto de sistemas distribuídos, em que os componentes colaboram para realização das funcionalidades providas pelo sistema, porque a comunicação é intensa. Duas formas para interceptação de comunicação em sistemas distribuídos merecem destaque: *sniffing* de pacotes de rede (ANSARI; RAJEEV; CHANDRASHEKAR, 2002) e *proxies* (CHATEL, 1996; FOX et al., 1998; WEAVER et al., 2014).

Sniffing de pacotes de rede é uma técnica que permite a captura e inspeção de pacotes que trafegam numa rede local. Ela é comumente utilizada para *logging* de tráfego de rede, análise de gargalos de desempenho da rede e também pode ser usada para fins maliciosos como introspecção do tráfego de rede de outros usuários. Ao capturar e inspecionar pacotes de rede, um software de monitoramento pode observar toda a comunicação que acontece entre componentes de um sistema distribuído com o objetivo de observar o seu comportamento.

A principal vantagem do *sniffing* é que ele pode ser aplicado de maneira completamente transparente. Isto é, a captura de pacotes de rede pode ser feita sem que produtores e consumidores de tráfego (e.g., componentes de um sistema distribuído) sejam modificados ou reconfigurados para isso. De fato, é possível até mesmo fazer com que o *sniffing* passe completamente despercebido pelos agentes do tráfego de rede. Por outro lado, nem sempre

a técnica pode ser utilizada. Há configurações de redes de computadores em que o *sniffing* de pacotes não é permitido. Nesses casos, a utilização da técnica para o monitoramento de sistemas distribuídos não seria possível.

Proxies são componentes de software que se interpõem à comunicação entre programas que interagem através de uma rede de computadores. Diferente de *sniffers* de pacotes, eles podem atuar não somente nas camadas de transporte e rede (protocolos TCP/IP) mas também atuam na camada de aplicação (e.g., HTTP). Isso faz com que a interceptação de comunicação seja mais fácil com a utilização de *proxies* porque eles conseguem capturar mensagens de aplicação (e.g., requisições HTTP). Com *sniffers*, a captura dessas mensagens demanda o tratamento dos pacotes capturados.

Por outro lado, a utilização de *proxies* normalmente demanda que os agentes comunicantes sejam modificados ou configurados para utilizá-los. Isto é, diferente do que acontece com o *sniffing* de pacotes de rede, a interceptação de comunicação através da utilização de *proxies* não pode ser completamente transparente. Felizmente, a configuração para envio e recebimento de tráfego através de um *proxy* é normalmente uma medida direta e fácil de ser aplicada. Num *setup* de computação em nuvem em que endereços são descobertos de maneira dinâmica (e.g., através de um sistema de nomes de domínios – DNS), a habilitação de *proxies* para interceptação de toda comunicação que acontece entre os componentes de uma aplicação pode ser tão simples quanto a reconfiguração dos registros no DNS.

2.5 MONITORAMENTO DE CHAMADAS DO SISTEMA OPERACIONAL

`syscall`s são funções básicas providas pelo *kernel* do Linux para que processos de aplicações consigam executar ações críticas, e.g., interação com hardware (BAGHERZADEH et al., 2018). A Tabela 1 apresenta algumas das `syscall`s disponibilizadas pelo sistema operacional Linux. Nesse sistema, dispositivos de hardware também são tratados como arquivos e referenciados através de descritores de arquivo. Um descritor de arquivo é basicamente um arquivo no sistema de arquivos representado por um número inteiro e associado aos meta-dados da execução de um programa, i.e., cada descritor é único no escopo da execução de um programa.

Dessa maneira, um programa que deseje exibir algo no monitor conectado a um computador gerenciado pelo Linux precisa obter um descritor de arquivo para o monitor (`open`) e escrever o conteúdo que se deseja exibir através de uma chamada `write` nesse descritor. De maneira análoga, para receber informações sobre as teclas pressionadas pelo usuário, um programa precisa de um descritor para o teclado e chamar `read` nesse descritor. Quando um programa não precisa mais de um descritor para um arquivo (no sistema de arquivos) ou dispositivo, ele pode chamar `close` para remover o descritor dos meta-dados de sua execução.

Tabela 1 – Algumas das `syscall`s providas pelo sistema operacional Linux.

`syscall`	**Descrição**
`read`	Faz leitura de um descritor de arquivo.
`write`	Faz escrita em um descritor de arquivo.
`open`	Abre um arquivo e retorna um descritor de arquivo.
`close`	Fecha um descritor de arquivo.
`execve`	Executa um programa.
`clone`	Cria um novo processo.

Outras chamadas interessantes são `execve` e `clone`. `execve` permite que um programa execute outro programa. Através dessa chamada, a execução do programa original é substituída pela execução do novo programa. Alguns meta-dados da execução, e.g., a maior parte dos descritores de arquivos, são mantidos enquanto outros, e.g., descritores de arquivos de filas de mensagens, são descartados. `clone` permite que um programa crie um novo processo. O novo processo é uma cópia do processo que fez a chamada e compartilha alguns meta-dados de execução, e.g., espaço de memória e descritores de arquivos. Além disso, esse tipo de chamada estabelece uma relação de dependência entre os processos, que passam a pertencer a uma mesma família de processos. Essa dependência condiciona, por exemplo, a execução de novos processos (filhos) à execução do processo que os criou (pai), i.e., os filhos são interrompidos quando a execução do pai é finalizada.

O monitoramento das `syscall`s realizadas por um programa permite observar o seu comportamento sem a necessidade de modificação do seu código. Para isso, é necessário que o sistema operacional forneça meios para introspecção da execução dos processos gerenciados por ele. A interface mais completa para monitoramento de chamadas do sistema operacional Linux é disponibilizada através da `syscall` `ptrace`(PADALA, 2002). Ela permite que um processo de monitoramento seja notificado sobre a execução de todas as chamadas ao sistema realizadas por um processo monitorado. Além de observar as chamadas realizadas, seus parâmetros e resultado, o processo monitor também pode interferir na sua execução. É possível, por exemplo, modificar os parâmetros originais da chamada, fornecer um resultado diferente do fornecido pelo sistema operacional ou até mesmo evitar a execução da chamada solicitada pelo processo monitorado.

O monitoramento de chamadas do sistema operacional impõe alguns desafios. O primeiro deles deriva do quase ilimitado poder de gerenciamento que agentes de monitoramento possuem sobre os processos monitorados. Esse poder pode abrir brechas de segurança que permitem, por exemplo, que o processo de monitoramento ganhe privilégios de execução irrestritos no ambiente computacional em que são executados os processos monitorados (GARFINKEL et al., 2003). O desafio de monitorar chamadas de maneira segura fez surgirem soluções como o módulo SECCOMP do Linux, através do qual os próprios processos podem limitar sua capacidade de execução de chamadas (WINTER, 2008).

Outro desafio é o custo da execução do monitoramento de chamadas do sistema operacional. Como as chamadas realizadas são muito numerosas, e.g., a execução de uma função simples numa linguagem de alto nível pode gerar a execução de centenas de chamadas ao sistema, a introspecção de cada uma delas pode causar sério prejuízo ao desempenho das aplicações monitoradas. Isso porque o controle da execução precisa ser constantemente alterado do sistema operacional para o processo monitor a cada chamada ao sistema. No Linux, esse prejuízo é agravado pela necessidade de troca de modo de execução, já que os processos do `kernel` são executados em um modo diferente dos processos de usuários. Uma possibilidade para redução desse prejuízo é a utilização dos filtros providos pelo módulo SECCOMP. Através deles um processo de monitoramento pode limitar a introspecção a apenas algumas das chamadas ao sistema feitas pelos processos monitorados.

As possíveis vulnerabilidades de segurança e a degradação de desempenho relacionadas ao monitoramento de chamadas do sistema operacional não têm impedido a proposição de estratégias e soluções baseadas nele. Além de ser uma ótima fonte de aprendizagem para curiosos e pesquisadores que se dedicam a conhecer melhor a execução e funcionamento de sistemas computacionais, o monitoramento de chamadas do sistema operacional tem sido efetivamente utilizado para fins diversos, e.g., soluções de segurança, monitoramento de computação em nuvem e reprodução da execução de programas (CACERES, 2002; KENISTON et al., 2007; BECK; FESTOR, 2009; O'CALLAHAN et al., 2017).

2.6 CONSIDERAÇÕES FINAIS

O capítulo apresentou os conceitos básicos necessários ao entendimento deste trabalho: o estilo arquitetural de microsserviços, o *tracing* e a interceptação de comunicação em sistemas distribuídos e o monitoramento de chamadas do sistema operacional.

Facilidade de modificação, escalabilidade e heterogeneidade foram relatados como os principais benefícios da utilização de microsserviços. Este último constitui também um desafio, já que a diversidade de tecnologias utilizadas para implementar os vários componentes que constituem uma aplicação baseada em microsserviços dificulta algumas tarefas como a instrumentação de código.

A instrumentação do código da aplicação é comumente uma atividade obrigatória para habilitação do monitoramento de sistemas distribuídos. Mesmo custosa, ela vale o esforço porque o monitoramento permite capturar o comportamento real das aplicações em ambientes de produção. Conhecer como a aplicação se comporta nesses ambientes facilita o diagnóstico e a correção de erros quando eles acontecem e permite a implementação de melhorias de desempenho.

Tracing foi apresentado como uma estratégia útil para monitoramento de aplicações baseadas em microsserviços. O custo de instrumentação de código e do prejuízo ao desempenho imposto pelo monitoramento, que é mais alto no contexto de microsserviços,

demanda a elaboração de abordagens menos custosas para habilitar essa ferramenta tão útil a desenvolvedores e operadores de aplicações.

Por fim, a interceptação de comunicação em sistemas distribuídos e o monitoramento de chamadas do sistema operacional foram apresentados como possíveis abordagens para monitoramento transparente.

3 RBINDER: UMA SOLUÇÃO PARA MONITORAMENTO TRANSPARENTE DE APLICAÇÕES BASEADAS EM MICROSSERVIÇOS

Este capítulo apresenta em detalhes a solução proposta para o problema atacado. Apresentamos primeiro como utilizar *proxies* para delegar a eles as responsabilidades relacionadas ao *tracing* que, de outra maneira, ficariam a cargo da aplicação monitorada. Depois, apresentamos como fazemos monitoramento de chamadas do sistema operacional para diagnosticar as relações de causalidade das mensagens trocadas entre os componentes da aplicação. Por fim, detalhamos a implementação desse monitoramento.

3.1 VISÃO GERAL

Para diminuir ao máximo os custos de tempo, dinheiro e degradação de desempenho associados ao monitoramento de aplicações baseadas em microsserviços, o projeto da solução proposta tem os seguintes objetivos:

- **Transparência**: não se demanda modificação do código-fonte da aplicação monitorada; e
- **Baixa sobrecarga**: o desempenho da aplicação monitorada não é degradado de maneira drástica;

A transparência é o principal requisito da solução porque é através dela que alcançamos o objetivo de minimização dos custos do monitoramento. Esse objetivo implica não apenas no agnosticismo quanto a informações específicas da aplicação (e.g., detalhes de implementação e natureza dos dados processados por ela) mas também quanto aos protocolos sobre os quais ela está implementada. A solução proposta deve, portanto, ser portável aos diferentes protocolos que normalmente dão suporte à implementação de microsserviços (e.g., HTTP e gRPC).

A baixa sobrecarga é um requisito secundário essencial a qualquer estratégia de monitoramento. Como o monitoramento é mais útil em ambientes de produção e esses ambientes têm baixa tolerância ao aumento de latência, o aumento provocado pelo monitoramento pode ser determinante na decisão de monitorar ou não a aplicação. Por isso, para que a solução proposta seja efetiva, é necessário que ela não degrade drasticamente o desempenho da aplicação monitorada.

A Figura 6 mostra a visão geral do Rbinder. Ele se baseia na utilização de *proxies* e no monitoramento de chamadas do sistema operacional. Os *proxies* são utilizados para interceptar a comunicação que acontece entre os microsserviços da aplicação. Dessa maneira eles estão aptos a realizar as atividades relacionadas ao monitoramento: geração, coleta e envio de informações de *tracing* a um servidor de *traces*. Para que essas informações

contenham os identificadores que permitem o diagnóstico de causalidade das mensagens trocadas entre os microsserviços, um processo de monitoramento é instalado em cada um dos microsserviços da aplicação. Esse processo monitora as chamadas do sistema operacional através da API `ptrace` fornecida pelo *kernel* do Linux. Os dados do monitoramento de chamadas são utilizados para manter uma máquina de estados que permite interceptar o envio de mensagens, e.g., `M2` e `M3`, causado pelo recebimento de uma mensagem, e.g., `M1`, e injetar nas mensagens enviadas (`M2` e `M3`) os identificadores presentes na mensagem causal (`M1`). Essa injeção também é feita através da interface `ptrace`.

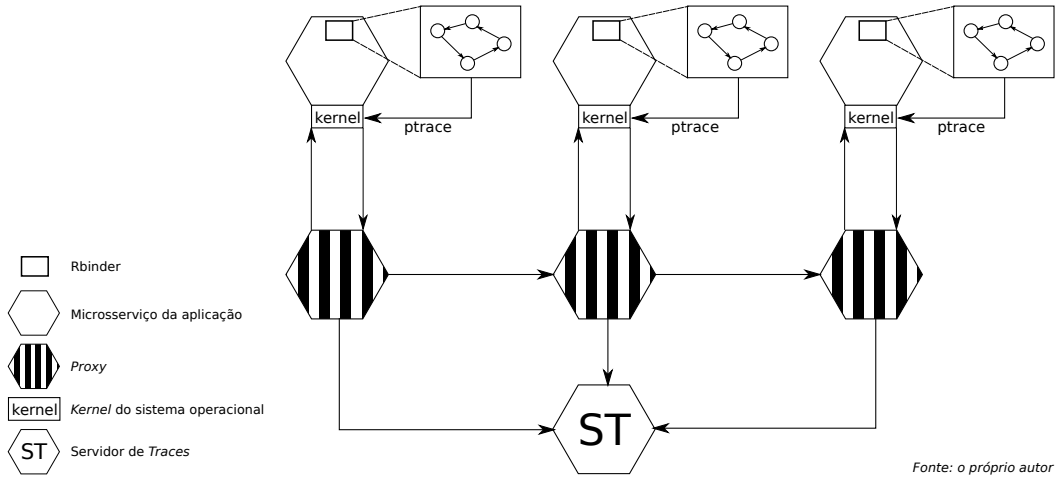


Figura 6 – Visão geral do Rbinder: microsserviços se comunicam entre si através de *proxies* que são responsáveis por gerar, coletar e enviar informações de *tracing* ao servidor de *traces*. Para permitir o diagnóstico de causalidade entre as mensagens trocadas, um processo de monitoramento de chamadas do sistema operacional é instalado em cada um dos microsserviços da aplicação.

3.2 UTILIZAÇÃO DE PROXIES

Microsserviços normalmente são implantados através de tecnologias de conteinerização (Docker[1], por exemplo), em que princípios como transparência de localização e seus benefícios são contemplados: facilidade de escala, movimentação e substituição de componentes. Esse cenário é favorável à adoção de *proxies* para interceptação de toda comunicação, que acontece entre os componentes do sistema, através da modificação da configuração referente ao esquema de implantação dos microsserviços.

Proxies são uma das possibilidades para interceptação de comunicação em sistemas distribuídos que foram apresentadas na Seção 2.4. O principal motivo que nos levou a optar por eles em detrimento das alternativas existentes é que eles são uma alternativa mais fácil de aplicar. Técnicas de *sniffing* de pacotes de rede, por exemplo, impõem dificuldades adicionais como o tratamento de mensagens que trafegam de maneira fragmentada,

[1] https://docker.com

em muitos pacotes. Além disso, a instalação de *proxies* é facilitada pelas estratégias de implantação de aplicações baseadas em microsserviços normalmente utilizadas e pode ser feita mediante modificação de arquivos de definições de implantação. Essa instalação está alinhada com uma noção que tem ganhado espaço entre os desenvolvedores e operadores de microsserviços, que é a de *sidecar*. O conceito sugere a utilização de *containers* auxiliares que são executados em conjunto com um *container* principal – onde o microsserviço da aplicação é executado, por exemplo. Por fim, a existência de soluções prontas para habilitação de *tracing* no contexto de microsserviços também motivou nossa escolha pela utilização de *proxies*.

Nós usamos *proxies* para a intermediação de todas as interações entre os microsserviços através da implantação de um *proxy* por microsserviço. Um *proxy* por microsserviço é suficiente para assegurar a interceptação de toda comunicação. Pode-se argumentar a respeito da possibilidade de alcançar o mesmo objetivo com a utilização de menos unidades de *proxies* mas esta relação 1:1 ajuda a simplificar e a entender o esquema de implantação. Dessa maneira, o esquema de implantação pode ser mais facilmente automatizado – o que está de acordo com o princípio de microsserviços que dita a automatização de tudo que puder ser automatizado (NEWMAN; MICROSERVICES, 2015). A escolha da razão entre número de *proxies* por microsserviço também considera o aumento total do consumo de recursos computacionais devido à demanda de recursos por cada unidade de *proxy*. Apesar de essas unidades serem projetadas e implementadas para consumir a menor quantidade possível de recursos, o aumento total da demanda não pode ser ignorado.

Os *proxies* se valem dessa organização, em que são interpostos entre os microsserviços da aplicação, para tomar conhecimento de todas as mensagens trocadas entre os microsserviços e utiliza-as para geração e envio de informações de *tracing*. As responsabilidades dos *proxies* são as mesmas atribuídas à instrumentação na estratégia apresentada na Seção 2.3.

A Figura 7 exibe um cenário em que três microsserviços, A, B e C, interagem entre si para servir uma dada requisição. Pela aplicação de nossa sugestão de utilização de *proxies*, são interpostos três *proxies* (hexágonos listrados) entre eles e é implantado um servidor de *traces* (`ST`). A numeração indica a ordem em que a comunicação ocorre: A recebe a requisição do usuário e solicita algo de B que, por sua vez, envia uma solicitação para C. Tanto as requisições que chegam (①, ④ e ⑦) quanto as que partem (②, ⑤ e ⑧) são recebidas e enviadas através de *proxies*, que as informam para `ST`. Requisições relacionadas ao *tracing* (ⓧ, ⓨ e ⓩ) não precisam acontecer numa ordem ou momento específicos. As respostas às requisições exibidas foram omitidas para simplificar a ilustração. Vale salientar que os *proxies* também podem injetar meta-informações usadas para determinar a causalidade entre as requisições. Por exemplo, se uma requisição HTTP não possui um cabeçalho `X-Request-Id`, o primeiro *proxy* deve injetar o cabeçalho faltante, que deve ser então propagado pelos demais componentes da aplicação e *proxies*.

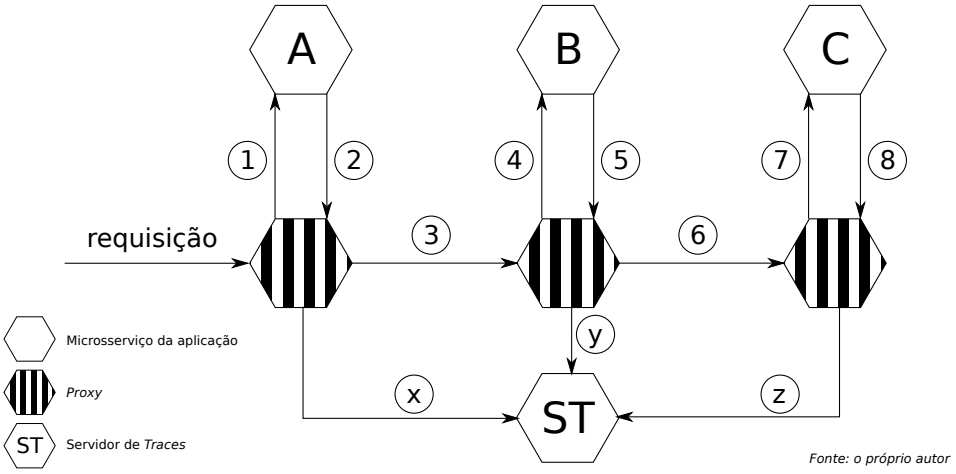


Figura 7 – Visão geral da implantação de *proxies*

Essa abordagem é agnóstica de protocolo já que não depende de nenhuma informação ou estrutura atrelada a um protocolo específico. Utilizamos HTTP sobre TCP/IP, mas outros protocolos podem ser utilizados. A utilização de protocolos diferentes dos apresentados aqui demandará algum esforço adicional: será necessário definir e implementar os meios para extração de informações de *tracing* e diagnóstico de causalidade entre mensagens. No entanto, é provável que isso seja factível através de adaptações simples do que é proposto. Não há prescrição de utilização de nenhum protocolo específico para a comunicação que se dá entre os *proxies* e o servidor de *traces*.

A Figura 8 mostra requisições HTTP que acontecem quando A recebe uma requisição: o *proxy* de A injeta meta-informação de *tracing* antes de encaminhar a requisição para A, envia a requisição que sai de A para B e informa ambas requisições para o servidor de *traces*. A requisição que parte do *proxy* de B foi omitida. Ambos os *proxies* informam a mesma requisição para o servidor de *traces* (a requisição de A para B), mas isso não é problema porque todas as informações de *tracing* são posteriormente conciliadas pelo servidor de *traces*. A linha pontilhada caracteriza uma fronteira virtual entre as requisições operacionais (topo) e as que se devem a propósitos de *tracing* (parte inferior).

Essa estratégia permite que o código das aplicações dê atenção total à lógica de negócio já que as responsabilidades de *tracing* podem ser delegadas para os *proxies*. Vale notar que, de outro modo, todas as atividades relacionadas ao *tracing* teriam de ser executadas por código na pilha de software da aplicação. Assim, a utilização de *proxies* pode também favorecer o desempenho do sistema uma vez que os processos responsáveis pelo *tracing* podem não competir diretamente por recursos com os processos da aplicação, que podem ser instalados em componentes físicos diferentes.

Uma possível desvantagem da estratégia é que ela demanda a implantação e o gerenciamento de mais unidades de processamento já que, adotando-se um *proxy* por microsserviço, o número de *containers* é duplicado. Felizmente, operadores de ambientes de

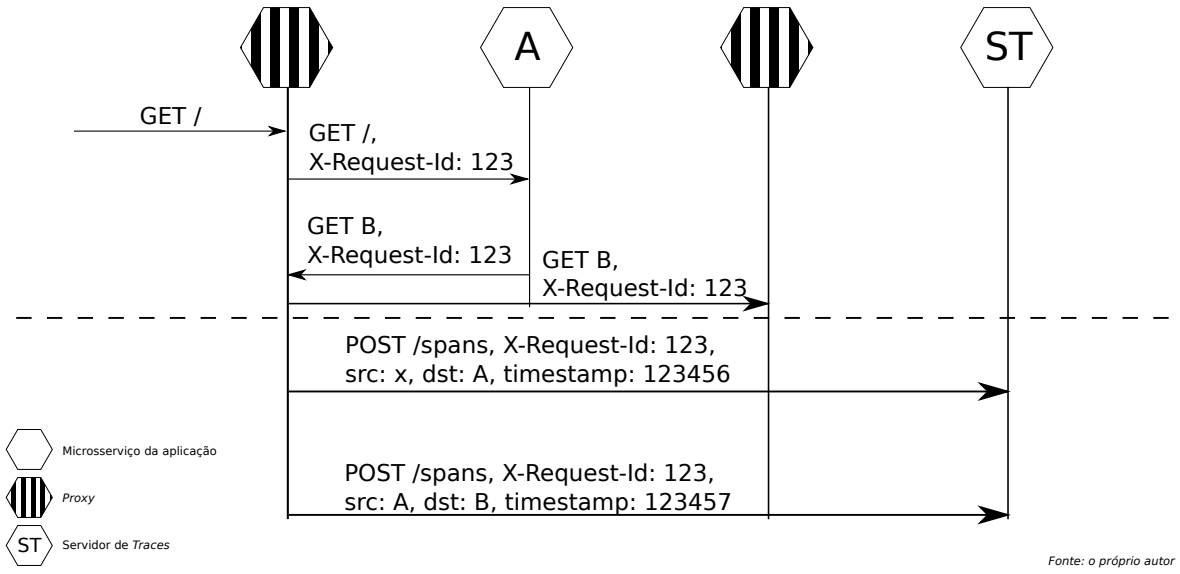


Figura 8 – Sequência de requisições operacionais e de *tracing*

microsserviços normalmente utilizam ferramentas poderosas de orquestração de *containers* para gerenciá-las, e.g., Kubernetes[2]. Outra possível desvantagem é o potencial de aumento de latência devido à comunicação extra entre *containers*. O custo-benefício entre co-alocação de *proxies* e microsserviços (maior disputa por recursos) e a disposição deles em nós distintos (maior latência) deve ser considerado. Essa questão é bastante abrangente e possui suas próprias complicações. Para ela, nós sugerimos uma estratégia autonômica baseada em `Models@Runtime` (SAMPAIO et al., 2017).

A utilização de *proxies* para *tracing* de microsserviços é recente e suportada por ferramentas como a plataforma Istio[3]. Mas ela falha em possibilitar *tracing* de maneira transparente porque não consegue diagnosticar a causalidade entre requisições sem intervenção no código da aplicação. Por exemplo, o microsserviço `A` na Figura 8 teria que ser modificado para adicionar o cabeçalho `X-Request-Id` nas requisições realizadas por ele usando como valor o mesmo recebido na requisição que motivou a realização. Nós propomos o monitoramento de chamadas ao sistema (`syscall`s) do Linux para sanar essa falta.

3.3 MONITORAMENTO DE SYSCALLS

O monitoramento das `syscall`s executadas por um processo (ver Seção 2.5) permite rastrear comportamento relacionado à realização de caminhos de requisições. Também é possível obter e injetar meta-informação de *tracing*. É dessa maneira que conseguimos

[2] https://kubernetes.io
[3] https://istio.io

alcançar o objetivo de propagar meta-informação sem exigir a modificação de código-fonte das aplicações monitoradas.

A Figura 9 mostra os elementos envolvidos na estratégia de monitoramento de `syscall`s e as principais interações entre eles. O `Tracer` (nosso processo de monitoramento) é responsável pela inicialização do `Tracee` (processo do microsserviço) e pelo *setup* do monitoramento de `syscall`s. Esse monitoramento é realizado através da interface `ptrace` (apresentada na Seção 2.5). Depois do *setup*, o *kernel* do Linux informa ao `Tracer` sempre que o `Tracee` faz uma chamada ao sistema. O `Tracer` pode conferir e modificar os parâmetros da chamada, o que também é feito através do `ptrace`.

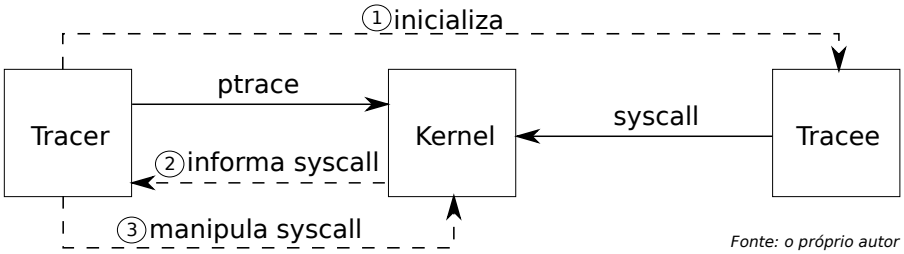


Figura 9 – Visão geral do monitoramento de `syscall`s. As setas sólidas indicam a utilização. O rótulo dessas setas indica a API utilizada para utilização. As setas tracejadas indicam as ações específicas básicas que acontecem durante o monitoramento.

Nós identificamos as `syscall`s chamadas por *threads* para recebimento e envio de requisições HTTP. A execução de `syscall`s sempre é observada no contexto de uma *thread* executora porque nós usamos essa associação para o diagnóstico de causalidade entre requisições: assumimos que qualquer requisição realizada por uma *thread* que está servindo a uma requisição foi causada pela requisição que está sendo servida. Na prática, nós interceptamos a `syscall` que indica o recebimento de uma requisição HTTP (como um `read` em um *socket* TCP) para extrair cabeçalhos HTTP referentes às informações de *tracing* e qualquer `syscall` relacionada ao envio de requisições (`sendto`, por exemplo) pela mesma *thread* (ou clones dela) para injetar informações de *tracing*. As relações derivadas da criação de *threads* (e.g., uma *thread* cria outras, que passam a ser suas filhas) precisam ser consideradas porque é comum que *threads* iniciem novas *threads* para realizar requisições de maneira assíncrona e também que servidores de aplicação iniciem várias *threads* para servir requisições.

A Figura 10 apresenta as `syscall`s que nós manipulamos e as ações realizadas durante o monitoramento. As ações do processo de monitoramento são ilustradas por quadrados enquanto os círculos representam possíveis estados assumidos pelas *threads* monitoradas. As `syscall`s que disparam ações e mudanças de estado são indicadas nos rótulos das arestas.

Toda nova *thread* (inclusive a inicial, que é criada pelo nosso processo monitor) inicia

no estado ociosa por que ainda não está servindo requisições. Nesse estado, uma chamada `accept` indica a abertura de um *socket* que permite o recebimento de requisições. Nesse caso, o descritor do *socket* aberto é registrado e a *thread* passa para o estado de espera por requisições. Uma chamada *clone* indica a criação de uma *thread* filha. Nesse caso, o identificador da *thread* filha é registrado e ambas as *threads* (mãe e filha) permanecem no estado de ociosidade.

A partir do estado de espera por requisições, uma *thread* pode: criar novas *threads* (`clone`); fechar o *socket* em que está esperando por requisições (`close`); ou receber requisições (`read`). A criação de *threads* surte o mesmo efeito observado no estado de ociosidade. O fechamento de *socket* causa o registro do fechamento e a atualização do estado da *thread*, que passa a estar ociosa ou, se ainda houver algum *socket* aberto, permanece à espera de requisições.

O recebimento de requisições causa a extração e o registro dos cabeçalhos de *tracing* da requisição e a mudança do estado da *thread*, que passa a servir uma requisição. Enquanto está servindo uma requisição, uma *thread* pode fechar *sockets*, criar novas *threads* ou fazer novas requisições. Fechamento de *sockets* e criação de *threads* funcionam como nos outros dois estados (ocioso e à espera de requisições). Quando uma *thread* que está servindo uma requisição faz outra requisição (`sendto`), os cabeçalhos de *tracing* extraídos da requisição que está sendo servida são injetados na requisição feita.

Vale notar que uma família de *threads* (mãe e filhas) pode manter vários *sockets* abertos ao mesmo tempo. Por isso, o fechamento de *sockets* só leva a *thread* ao estado de ociosidade quando não há mais nenhum *socket* aberto.

O monitoramento de `syscall`s foi implementado com `ptrace` em conjunto com filtros do SECCOMP (ver Seção 2.5) para reduzir a sobrecarga devida à interrupção das `syscall`s. Essas interrupções podem ser numerosas e caras por conta da troca de contexto que acontece entre espaço do *kernel* e espaço do usuário. `ptrace` permite ao `Tracer` inspecionar e controlar a execução de chamadas feitas pelo `Tracee` por notificá-lo sempre que elas são invocadas. O `Tracer` pode inspecionar os parâmetros atribuídos pelos `Tracee`s e também modificá-los antes de solicitar ao *kernel* que dê prosseguimento à execução da `syscall` solicitada.

Para por em prática o modelo representado pela máquina de estados da Figura 10, três atividades são críticas: a inicialização do monitoramento, a extração de cabeçalhos de requisições recebidas e a injeção de cabeçalhos em requisições feitas. Por isso e para permitir melhor entendimento da solução proposta, os próximos parágrafos detalham a inicialização de um processo `Tracee` e a interceptação de chamadas `read` e `sendto` para extração e injeção de cabeçalhos. Vale notar que alguns parâmetros e retornos de `syscall`s foram omitidos para facilitar o entendimento.

A Figura 11 apresenta o *setup* de um `Tracee`, que é a forma como o processo dos microsserviços da aplicação monitorada é inicializado com monitoramento de suas chamadas

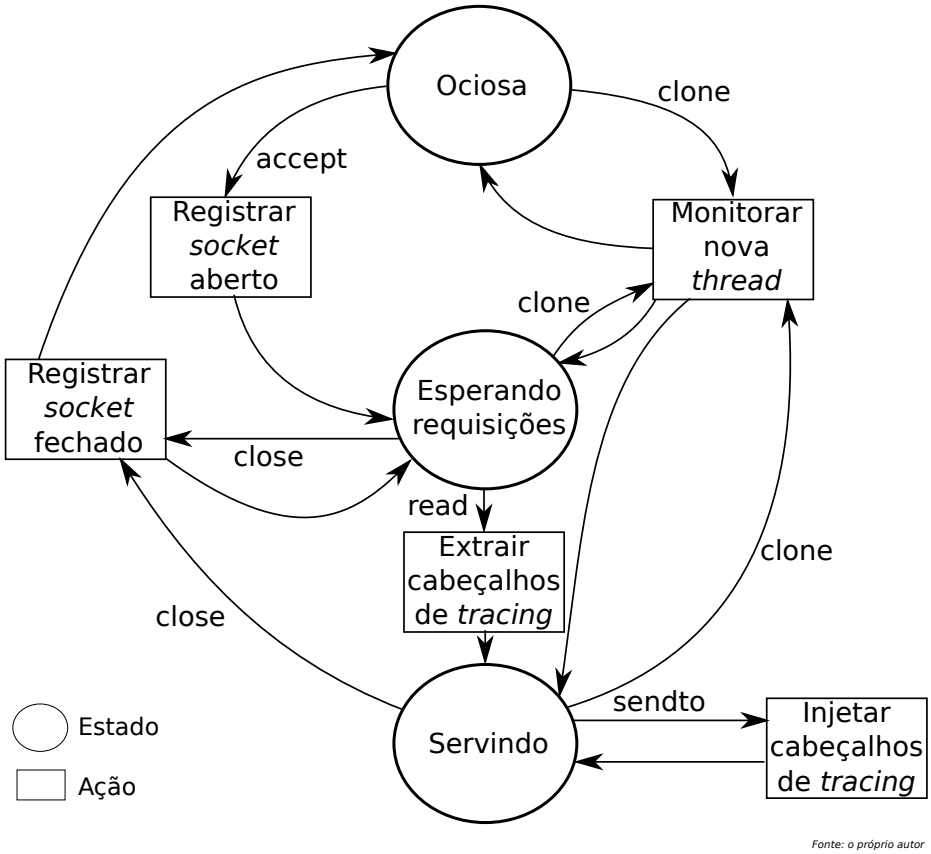
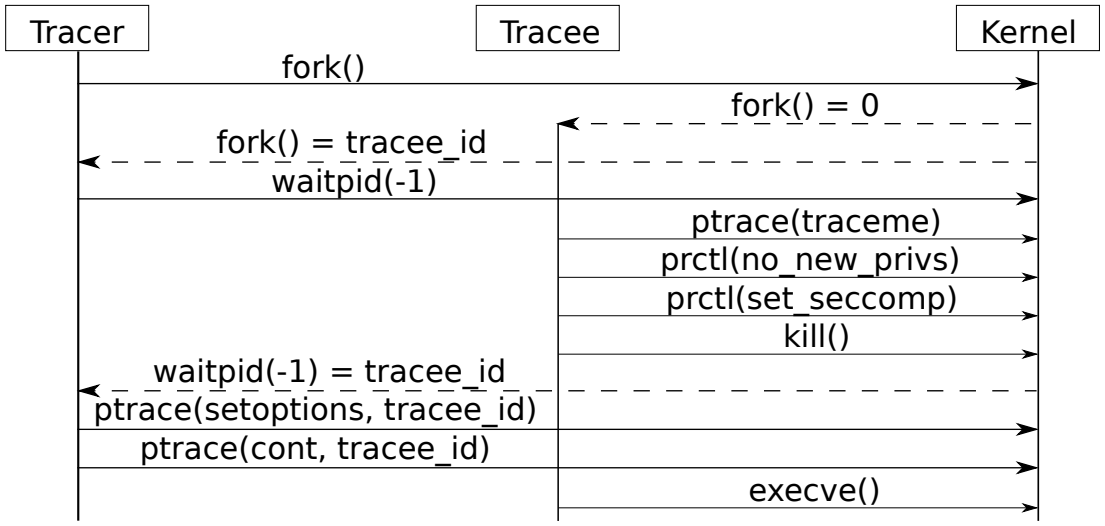


Figura 10 – Ações de monitoramento e estados de *thread* para cada `syscall` monitorada. As ações do processo de monitoramento são ilustradas por quadrados enquanto os círculos representam possíveis estados assumidos pelas *threads* monitoradas.

ao sistema operacional. Inicialmente, o `Tracer` faz *fork* de um processo filho para executar o código do microsserviço. Depois disso, ele espera (`waitpid`) que o `Tracee` realize algumas operações essenciais: informar ao *kernel* que ele deve ser monitorado por seu pai (`ptrace(traceme)`); proibir que ele, em qualquer momento de sua execução, ganhe mais privilégios que seu pai (`prctl(no_new_privs)`), o que permite a execução do nosso monitoramento sem privilégios de administrador do sistema; e instalar os filtros SECCOMP (`prctl(set_seccomp)`). O `Tracee` então interrompe sua execução (`kill`) para permitir que o `Tracer` configure opções necessárias (`ptrace(setoptions)`). Essas opções incluem *flags* que instruem o `ptrace` a monitorar automaticamente processos filhos do `Tracee` e refinam a forma como as interrupções relacionadas aos filtros do SECCOMP são tratadas. A seguir, o `Tracer` permite a continuidade da execução do `Tracee` (`ptrace(cont)`), que executa o código do microsserviço (`execve`).

Após a inicialização do `Tracee`, o `Tracer` mantém um *loop* de controle infinito estabelecido pela chamada a `waitpid`. Essa chamada interrompe a execução do `Tracer` até que algum de seus `Tracee`s faça uma chamada do sistema que dispare uma notificação. As notificações só são disparadas se a chamada satisfizer as regras estabelecidas pelos filtros



Fonte: o próprio autor

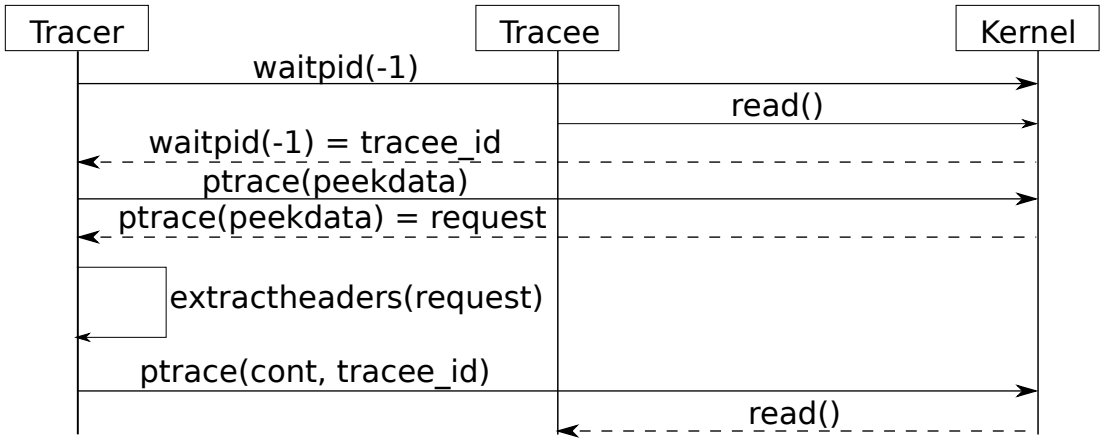
Figura 11 – Inicialização do `Tracee`. As linhas verticais representam cada um dos componentes envolvidos no processo, que são especificados pelos retângulos. As setas sólidas indicam chamadas entre os componentes. As setas tracejadas indicam o retorno das chamadas realizadas.

SECCOMP, que são detalhadas mais adiante na apresentação dos detalhes da implementação.

A Figura 12 mostra a interceptação da chamada `read` para extração de cabeçalhos de requisições HTTP que chegam em um microsserviço. Por simplicidade, a figura omite a verificação preliminar que condiciona a execução de `extractheaders`: a chamada `read` que causou a interrupção deve estar relacionada a uma leitura de um *socket* TCP aberto e `requisição` deve ser uma requisição HTTP. `ptrace(peekdata)` tem de ser executado após a execução da chamada do sistema porque, caso contrário, `requisição` não estaria disponível. Além disso, a chamada do sistema original é executada sem qualquer modificação de seus parâmetros ou resultado.

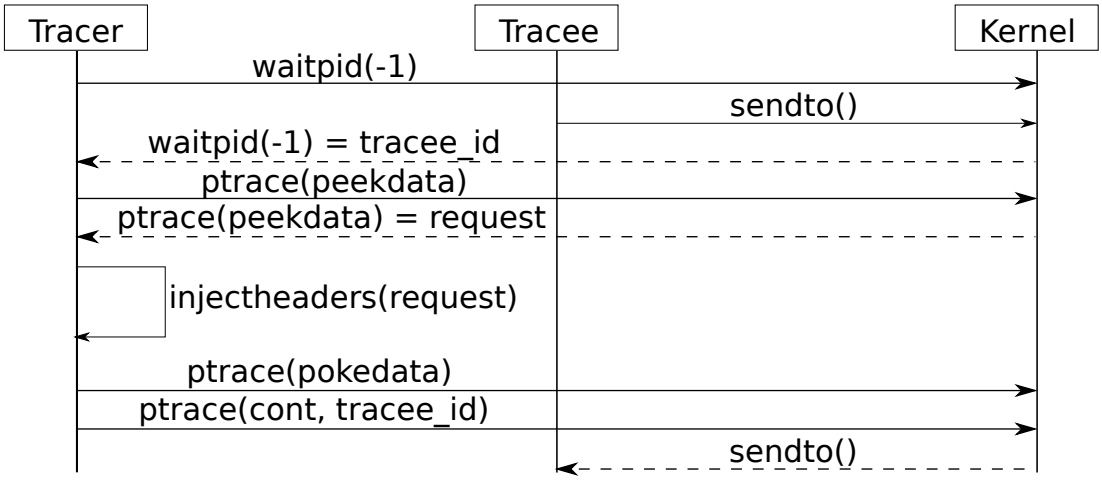
A Figura 13 mostra a interceptação da chamada `sendto`, que é similar à interceptação de `read`, exceto por: `requisição` é uma requisição que parte do microsserviço; e `pokedata` é usado para modificar os parâmetros da chamada do sistema, substituindo a requisição original por uma que possui os cabeçalhos de *tracing*. Portanto, ao contrário do que acontece na interceptação de `read`, a manipulação de parâmetros ocorre antes da execução da chamada do sistema.

Linux é a plataforma hegemônica sobre a qual as tecnologias de *containers* operam. Por isso, o monitoramento de chamadas do sistema tem a vantagem de poder ser adotado de imediato por aplicações baseadas em microsserviços já existentes e também por aplicações que ainda estão para ser criadas. Nós usamos HTTP sobre TCP/IP como um caso específico para implementação da solução proposta. No entanto, o monitoramento de `syscall`s também pode ser usado para diagnosticar a causalidade de mensagens de ou-



Fonte: o próprio autor

Figura 12 – Extração de cabeçalhos na interceptação da chamada `read`. As linhas verticais representam cada um dos componentes envolvidos no processo, que são especificados pelos retângulos. As setas sólidas indicam chamadas entre os componentes. As setas tracejadas indicam o retorno das chamadas realizadas.



Fonte: o próprio autor

Figura 13 – Injeção de cabeçalhos na interceptação da chamada `sendto`. As linhas verticais representam cada um dos componentes envolvidos no processo, que são especificados pelos retângulos. As setas sólidas indicam chamadas entre os componentes. As setas tracejadas indicam o retorno das chamadas realizadas.

tros protocolos implementados pelo *kernel* e adotados por aplicações. De maneira análoga ao que acontece com a utilização de *proxies*, o monitoramento de chamadas do sistema também envolve a identificação e adequada interceptação de chamadas específicas aos protocolos adotados pelas aplicações monitoradas.

Apesar da vantagem de operar muito próximo ao *kernel*, a grande quantidade de

chamadas do sistema pode fazer com que o monitoramento degrade o desempenho dos processos monitorados. É comum que um processo simples execute milhares de chamadas do sistema em um breve período, o que torna imprescindível extrema cautela para a realização de seu monitoramento. No entanto, o prejuízo ao desempenho pode ser minimizado pela filtragem das chamadas que são interrompidas – optando por interromper apenas as que são úteis ao propósito do monitoramento da aplicação.

3.4 DETALHES DE IMPLEMENTAÇÃO

Esta seção apresenta detalhes da implementação da solução proposta para monitoramento transparente de aplicações baseadas em microsserviços com o intuito de possibilitar a reprodução do que está sendo reportado.

3.4.1 Utilização de Proxies

Utilizamos *proxies* Envoy[4] para a intermediação da comunicação entre os *containers* da aplicação monitorada. Estes *proxies* foram escolhidos porque eles permitem a interceptação de atividades tanto em níveis mais baixos das camadas de rede (camadas de rede e de transporte) quanto na camada de aplicação e dispõem de facilidades para envio de informações de *tracing* a múltiplos servidores de *traces*. Como nosso objetivo é utilizá-los para gerar e reportar *traces*, eles foram configurados como apresentado na Listagem 3.1.

A Listagem 3.1 apresenta a configuração de *proxy* utilizada para o microsserviço `carts`, que representa bem todas as configurações utilizadas para a instalação de *proxies* para cada um dos microsserviços da aplicação. O *proxy* é instalado como um recurso estático (linha 1) porque não há necessidade de modificação dos parâmetros de configuração em tempo de execução.

A listagem de ouvintes (linhas 3 a 28) configuram um único ouvinte para a porta 80 na interface de rede 0.0.0.0 (linhas 4 a 6). É esse o endereço que os outros microsserviços utilizam para acessar este serviço. A configuração de filtros do ouvinte (linhas 7 a 28) indicam a utilização de um filtro fornecido pelo Envoy que é o gerente de conexões HTTP (linha 9) e o configuram para reportar *traces* (linhas 11 e 12) e encaminhar todas as requisições recebidas para um *cluster* (linha 25) que é posteriormente especificado (linhas 29 a 37). As linhas 26 a 28 são úteis para remoção de algumas configurações padrão do Envoy que impedem o bom funcionamento da aplicação utilizada para avaliação da nossa solução, e.g., a utilização da versão 2 do protocolo HTTP.

A configuração de *clusters* para encaminhamento de requisições (linhas 29 a 37) especifica o endereço de um único *host* (linhas 34 a 37) e uma política de balanceamento de carga (linha 33). As demais linhas (38 a 43) apenas configuram a interface web de administração do *proxy* e não são relevantes para os nossos propósitos.

[4] https://www.envoyproxy.io

Listagem 3.1 – Arquivo de configuração do *proxy* Envoy

```
static_resources:
  listeners:
  - address:
      socket_address:
        address: 0.0.0.0
        port_value: 80
    filter_chains:
    - filters:
      - name: envoy.http_connection_manager
        config:
          tracing:
            operation_name: ingress
          codec_type: auto
          stat_prefix: ingress_http
          route_config:
            name: service1_route
            virtual_hosts:
            - name: service1
              domains:
              - "*"
              routes:
              - match:
                  prefix: "/"
                route:
                  cluster: carts_service
          http_filters:
          - name: envoy.router
            config: {}
  clusters:
  - name: carts_service
    connect_timeout: 0.250s
    type: strict_dns
    lb_policy: round_robin
    hosts:
    - socket_address:
        address: carts
        port_value: 8080
admin:
  access_log_path: "/dev/null"
  address:
    socket_address:
      address: 0.0.0.0
      port_value: 8001
```

Todas as configurações de *proxies* para os microsserviços da aplicação seguem esse mesmo padrão, sendo a única exceção a configuração do *proxy* que é o primeiro a receber a requisição do usuário. Este *proxy* tem a responsabilidade extra de gerar o cabeçalho de identificação da requisição. Sua configuração é, no entanto, extremamente similar às demais e conta praticamente com a adição de uma única linha `generate_request_id: true` na configuração do gerente de conexões HTTP do Envoy.

3.4.2 Monitoramento de syscalls

O monitoramento de chamadas do sistema é habilitado através de um programa escrito em C cujo código-fonte está disponível no Apêndice B. A lógica implementada pode ser dividida em duas partes: o *setup* do monitoramento e o laço infinito para controle das *threads* monitoradas.

A Listagem 3.2 apresenta os principais passos da inicialização do processo a ser monitorado (chamado `Tracee`). A linha 1 checa se o processo que está executando esse código é o `Tracee`. Se for, as demais linhas são executadas: as linhas 3 a 20 definem uma estrutura de dados para instalação de filtros do SECCOMP que interrompem o processo monitor no caso de o processo monitorado executar uma das chamadas de interesse (`read`, `close`, `accept`, `sendto` e `clone`). A instalação desses filtros é efetivada na linha 25 com a chamada a `prctl`. A linha 21 usa a mesma operação do *kernel* (`prctl`) para estabelecer que o `Tracee` não pode ganhar mais privilégios do que aqueles concedidos no momento de sua inicialização. Isso é interessante para permitir que a solução de monitoramento seja habilitada sem a necessidade de privilégios de administrador do sistema.

As linhas 29 e 30 são essenciais para o *setup* do monitoramento. A chamada `kill` por parte do `Tracee` permite que o processo de monitoramento (`Tracer`) configure algumas opções necessárias à realização do monitoramento através da interface do `ptrace`. Quando a linha 29 é executada, a execução do `Tracee` é interrompida e o código exibido na Listagem 3.3 é executado no processo do `Tracer`. Só depois que ele permite a continuação da execução do `Tracee` (através da chamada a `ptrace(PTRACE_CONT)`) é que a linha 30 é executada. Ela é responsável por executar o código do microsserviço, que teria sido executado inicialmente pelo sistema operacional caso o monitoramento não estivesse habilitado.

Listagem 3.2 – *Setup* do `Tracee`

```
1  if(child == 0) {
     ptrace(PTRACE_TRACEME, NULL, NULL, NULL);
3    struct sock_filter filter[] = {
       BPF_STMT(BPF_LD+BPF_W+BPF_ABS, offsetof(struct seccomp_data, nr)),
5      BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, SYS_read, 0, 1),
       BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_TRACE),
7      BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, SYS_close, 0, 1),
       BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_TRACE),
9      BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, SYS_accept, 0, 1),
       BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_TRACE),
11     BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, SYS_sendto, 0, 1),
       BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_TRACE),
13     BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, SYS_clone, 0, 1),
       BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_TRACE),
15     BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_ALLOW),
     };
17   struct sock_fprog prog = {
       .filter = filter,
19     .len = (unsigned short) (sizeof(filter)/sizeof(filter[0])),
     };
```

```
21    if (prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0) == -1) {
        perror("prctl(PR_SET_NO_NEW_PRIVS)");
23      return 1;
      }
25    if (prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, &prog) == -1) {
        perror("prctl(PR_SET_SECCOMP)");
27      return 1;
      }
29    kill(getpid(), SIGSTOP);
      execv(argv[1], argv + 1);
31  }
```

A Listagem 3.3 apresenta o código de *setup* executado no processo do `Tracer`. O primeiro passo (linha 1) é interromper a própria execução até que algum processo monitorado interrompa sua execução (o que acontece na linha 29 da Listagem 3.2). Quando um `Tracee` retorna, o seu monitoramento é configurado com `ptrace(PTRACE_SETOPTIONS)` (linha 2): as *flags* `PTRACE_O_TRACEVFORK`, `PTRACE_O_TRACEFORK` e `PTRACE_O_TRACECLONE` configuram o monitoramento automático de *threads* criadas pelo `Tracee`; `PTRACE_O_EXITKILL` faz com que o `Tracee` seja finalizado se o `Tracer` também for; `PTRACE_O_TRACESECCOMP` faz com que as interrupções devidas a filtros SECCOMP instalados possam ser tratadas da maneira como o restante do código trata; e, por fim, `PTRACE_O_TRACEEXEC` faz com que o `Tracer` seja notificado e o `Tracee` seja interrompido quando executar uma chamada a `execv`, que é a chamada do sistema operacional que o `Tracee` utiliza para executar o código do microsserviço.

Depois que o `ptrace` é devidamente configurado, o `Tracer` dá continuidade à execução do processo do `Tracee` na linha 8.

Listagem 3.3 – *Setup* do `Tracer`

```
1  cid = waitpid(-1, &status, __WALL);
   if(ptrace(PTRACE_SETOPTIONS, cid, 0, PTRACE_O_TRACEEXEC|PTRACE_O_EXITKILL|\
3        PTRACE_O_TRACEVFORK|PTRACE_O_TRACECLONE|PTRACE_O_TRACEFORK|\
         PTRACE_O_TRACESECCOMP) < 0) {
5    perror("ptrace(PTRACE_SETOPTIONS)");
     exit(1);
7  }
   if(ptrace(PTRACE_CONT, cid, NULL, WSTOPSIG(status)) < 0) {
9    perror("ptrace(PTRACE_CONT)");
     exit(1);
11 }
```

A Listagem 3.4 apresenta o *loop* de controle do `Tracee`, executado pelo `Tracer`. Dentro do *loop* infinito (linhas 1-14), a execução do `Tracer` é interrompida enquanto alguma das *threads* monitoradas não retorna (chamada a `waitpid` na linha 2). Quando algum `Tracee` executa algum código que dispara a notificação do `Tracer`, o `Tracer` primeiro verifica se essa não é uma notificação de término da execução do `Tracee` (linha 3) e remove o `Tracee` de seus registros (linhas 4 a 8) se for esse o caso. Se o `Tracee` não foi terminado, então o `Tracer` verifica se essa é uma interrupção devida aos filtros do SECCOMP que foram instalados (linha 11). Essa verificação é útil porque o `ptrace` pode interromper o `Tracer`

em outras situações como em passagens de sinais para os processos monitorados. Isto é possível devido à configuração da *flag* `PTRACE_O_SECCOMP` exposta na Listagem 3.3.

Listagem 3.4 – *Loop* de controle do `Tracee`

```
1  while(1) {
     cid = waitpid(-1, &status, __WALL);
3    if(WIFEXITED(status)) {
       tracee = find_tracee(cid);
5      if(tracee) {
         rmtracee(tracee);
7      }
       continue;
9    }

11   if (status >> 8 == (SIGTRAP | (PTRACE_EVENT_SECCOMP << 8))) {
       // ...
13   }
   }
```

Nas linhas omitidas que constam entre as linhas 11 e 13 acontecem as verificações de qual foi a chamada do sistema executada pelo `Tracee` e, a depender dela, as ações aplicáveis. O código completo do monitoramento está disponível no Apêndice B.

3.5 CONSIDERAÇÕES FINAIS

Este capítulo apresentou a solução para monitoramento transparente de aplicações baseadas em microsserviços. Ela está baseada em dois pontos centrais: a utilização de *proxies* para intermediação da comunicação entre os componentes da aplicação e o monitoramento de chamadas do sistema operacional, que permite diagnosticar a causalidade das requisições sem demandar modificação do código-fonte da aplicação monitorada.

4 AVALIAÇÃO

Este capítulo expõe uma avaliação da solução proposta. Inicialmente, apresentamos o objetivo da avaliação. Depois, descrevemos o *setup* experimental em que os testes foram realizados. Por fim, apresentamos e discutimos os resultados da avaliação.

4.1 OBJETIVOS

A degradação do desempenho dos sistemas monitorados é um dos principais argumentos contra o monitoramento de sistemas distribuídos devido ao impacto que ele pode causar em serviços críticos, comprometendo o sucesso dos negócios que deles dependem. Nós realizamos uma avaliação comparativa de desempenho para medir o quanto a solução proposta impacta os sistemas monitorados e como esse impacto se relaciona com o impacto causado por soluções alternativas.

4.2 DESCRIÇÃO DOS EXPERIMENTOS

Nós usamos a aplicação Sock Shop[1] como carga de trabalho (BROGI et al., 2017) por ela ser suficientemente complexa para os propósitos desta avaliação e estar acessível para fins de pesquisa experimental. Essa aplicação provê funcionalidades típicas de lojas de comércio eletrônico e é composta por cerca de 15 microsserviços implementados nas linguagens de programação Java e Go. A métrica escolhida para a avaliação é o *tempo de resposta* porque ela representa bem a experiência do usuário ao utilizar a aplicação. Esse tempo é sempre medido sob a perspectiva do usuário com relação à sua interação com o sistema. Isto é, o tempo que ele leva para receber uma resposta para as suas requisições. A Tabela 2 sumariza os parâmetros utilizados nos experimentos.

A carga de trabalho contou com a execução de 1.000 requisições HTTP `POST` direcionadas à operação de *checkout* de pedido, que é a que envolve o maior número de

[1] https://microservices-demo.github.io

Tabela 2 – Parâmetros experimentais

Parâmetro	**Valor**
Operação	*Checkout* de pedido
Número de requisições	1.000
Tempo entre requisições	Aleatório seguindo distribuição gaussiana (média: 5, desvio padrão: 2)
Cenários de monitoramento	Desabilitado (*Sem Monitoramento*) Habilitado por *Código Instrumentado* Habilitado pelo *Rbinder*

microsserviços (ver Figura 14). A quantidade de requisições foi escolhida para permitir a coleta de dados suficientes para análise estatística. Para tentar reproduzir um contexto real de utilização de uma aplicação web num momento de baixa carga, introduzimos intervalos aleatórios entre a execução de cada uma das requisições. Os experimentos foram realizados em três cenários diferentes:

1. Aplicação sem monitoramento (*Sem Monitoramento*);
2. Aplicação monitorada através da instrumentação do seu código (*Código Instrumentado*); e
3. Aplicação monitorada pelo *Rbinder*;

A aplicação utilizada provê uma solução de *tracing* habilitada através de configurações. O cenário sem monitoramento consiste da implantação dos microsserviços com configuração para desabilitação de *tracing*. Nele, o código da aplicação é executado sem qualquer modificação e as requisições feitas à aplicação são servidas pelos microsserviços sem a interposição de *proxies* para intermediação de comunicação ou o envio de informações de *tracing* a um servidor de *traces*. Portanto, nesse caso, apenas os microsserviços da aplicação são implantados – mas não um servidor de *traces* nem *proxies*.

A instrumentação no segundo cenário é habilitada através da biblioteca Sleuth do Spring[2] nos microsserviços Java. Nos microsserviços Go, além da utilização de bibliotecas de instrumentação, também foi necessária a modificação do código-fonte da aplicação. As versões de código de todos os projetos que tiveram de ser modificados estão disponíveis no GitHub[3]. Nesse cenário, não há *proxies* intermediando a comunicação: os próprios microsserviços da aplicação são responsáveis pela geração e envio de informações de *tracing* a um servidor de *traces*. Portanto, nesse caso, além da implantação dos microsserviços da aplicação também foi implantado um servidor de *traces*.

Para o terceiro cenário, *proxies* Envoy[4] foram usados para intermediar a comunicação entre os microsserviços da aplicação e realizar atividades relacionadas ao *tracing*. Além dos *proxies* dedicados a cada um dos microsserviços da aplicação, também foi implantado um para a adição do cabeçalho de identificação em requisições que vêm do usuário (`orders front` na Figura 14). Nesse cenário, o código dos microsserviços da aplicação não sofreu nenhuma modificação. Em vez disso, o diagnóstico da causalidade entre as requisições é possibilitado através do monitoramento de chamadas do sistema. Esse monitoramento foi habilitado através da modificação das imagens Docker dos microsserviços para que o processo principal da imagem fosse iniciado sob a supervisão do Rbinder. Portanto, nesse cenário os microsserviços da aplicação são implantados de uma maneira diferente dos outros e além deles também são implantados *proxies* e um servidor de *traces*.

[2] https://spring.io
[3] https://github.com/gfads
[4] https://envoyproxy.io

A Figura 14 apresenta uma visão da implantação dos microsserviços que participam da realização da operação de *checkout* de pedido no terceiro cenário e a dependência entre eles. O primeiro componente (`router`) a receber a requisição do usuário é um microsserviço de roteamento de requisições que não participa ativamente da realização da operação (da perspectiva da lógica de negócio), da mesma maneira que o microsserviço `front-end`, responsável pela interface do usuário. O componente `orders front` é responsável pela adição de um cabeçalho de identificação da requisição e por seu encaminhamento para o próximo microsserviço: `orders`. Este microsserviço é o primeiro, dos destinados a servir requisições de *checkout* de pedido, a receber a requisição nos outros cenários. `orders` faz requisições aos demais microsserviços que participam da operação (`user`, `carts`, `payment` e `shipping`) para por em prática a lógica necessária à conclusão da compra do usuário. Esta lógica inclui a verificação da disponibilidade dos itens solicitados, a confirmação do pagamento e o envio dos produtos comprados até o endereço do usuário. Todas as requisições feitas aos microsserviços de que `orders` depende são intermediadas por seu próprio *proxy* e pelos *proxies* dos microsserviços requisitados.

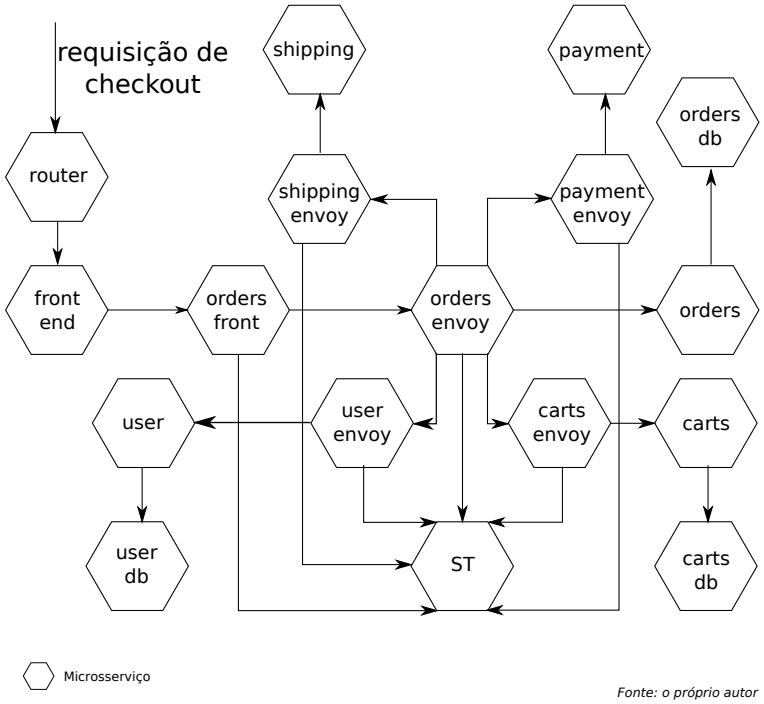


Figura 14 – Microsserviços envolvidos na operação de *checkout* de pedido, seus serviços de banco de dados (sufixo `db`), *proxies* para *tracing* (sufixo `envoy`) e servidor de *traces* (`ST`).

Ambos os cenários com monitoramento habilitado contaram com a implantação do servidor de *traces* Zipkin[5]. Todos os microsserviços foram implantados com Docker Compose. Nós usamos uma máquina virtual gerenciada pelo KVM para melhor isolamento do

[5] https://zipkin.io

ambiente de experimentação. A Tabela 3 apresenta as especificações das máquinas física e virtual usadas no experimento.

Tabela 3 – Especificações de máquinas física e virtual

Descrição	**Distribuição Linux**	**Versão do *Kernel***	**# cpus**	**RAM**
Máquina física	Debian 8	3.16	4 * 988 MHz	6 GB
Máquina virtual	Alpine 3.7	4.9	1	6.3 GB

A modificação da implantação da aplicação no terceiro cenário pode ser dividida em duas partes principais: as mudanças aplicadas aos arquivos de manifesto utilizados para implantar os componentes e as mudanças aplicadas aos arquivos utilizados para construir as imagens dos componentes. Os manifestos de implantação e arquivos de definições para criação de imagens estão relacionados de maneira muito próxima e é possível obter o mesmo resultado através da implementação de algumas configurações tanto em um quanto no outro. O comando de inicialização do microsserviço, por exemplo, pode ser configurado no arquivo de definição para construção da imagem ou (sobrescrito) no manifesto de implantação da aplicação.

Para a execução dos experimentos, optamos por concentrar as modificações nos arquivos de definições da imagem, o que fez com que as modificações aplicadas ao manifesto de implantação fossem tão simples quanto apenas a renomeação das imagens usadas originalmente pelo *label* das imagens construídas com a habilitação do Rbinder.

A Listagem 4.1 apresenta um trecho do manifesto referente à configuração da implantação do microsserviço `orders`. Nele, os *containers* que executam o microsserviço são configurados para executar a imagem `weaveworksdemos/orders:0.4.7` (linha 5), ser identificados através do *hostname* `orders` (linha 6), reiniciar sempre que o processo principal for terminado (linha 7), habilitar apenas a capacidade `NET_BIND_SERVICE` (linhas 8 a 11), não permitir escrita no sistema de arquivos (linha 12) e montar um sistema de arquivos temporário (linhas 13 e 14). Além disso, são configuradas algumas opções para execução da máquina virtual Java (linhas 15 e 16), que é a linguagem de programação em que o microsserviço está escrito. Para habilitação do Rbinder, basta, portanto, a modificação da linha 5. Para a execução dos experimentos, por exemplo, o valor da chave `image` foi modificado para `gfads/orders:rbinder`, que é o *label* da imagem construída.

Listagem 4.1 – Trecho do manifesto de implantação da aplicação

```
version: '2'

services:
  orders:
    image: weaveworksdemos/orders:0.4.7
    hostname: orders
    restart: always
    cap_drop:
      - all
```

```
10      cap_add:
          - NET_BIND_SERVICE
12      read_only: true
        tmpfs:
14        - /tmp:rw,noexec,nosuid
        environment:
16        - JAVA_OPTS=-Xms64m -Xmx128m -XX:+UseG1GC -Djava.security.egd=file:/dev/
              urandom -Dspring.zipkin.enabled=false
```

A Listagem 4.2 apresenta o arquivo de definições para construção da imagem do Docker do serviço `orders`. Essas definições são muito parecidas com as dos demais serviços e podem ser usadas como exemplo para as modificações necessárias a todos os arquivos usados para a execução dos experimentos. A primeira linha indica a imagem base para construção da imagem do microsserviço. A linha 3 configura o diretório de trabalho padrão como `/usr/src/app`. A linha 4 copia o JAR da aplicação para a imagem. A linha 6 modifica as permissões de acesso do JAR para que ele possa ser executado por um usuário que não seja administrador. A linha 8 configura o usuário padrão da imagem. A última linha define o comando a ser executado quando o *container* responsável por executar a imagem é iniciado.

Listagem 4.2 – Arquivo de definição da imagem do microsserviço `orders`

```
FROM weaveworksdemos/msd-java:jre-latest
2
WORKDIR /usr/src/app
4 COPY *.jar ./app.jar

6 RUN chown -R \${SERVICE_USER}:\${SERVICE_GROUP} ./app.jar

8 USER \${SERVICE_USER}

10 ENTRYPOINT ["/usr/local/bin/java.sh","-jar","./app.jar", "--port=80"]
```

A Listagem 4.3 apresenta o arquivo de definições para construção da imagem do microsserviço `orders` com o Rbinder habilitado. A imagem base (linha 1), a definição de diretório de trabalho (linha 5) e a cópia do binário da aplicação (linha 6) mantêm-se inalteradas. As principais alterações em relação às definições originais (Listagem 4.2) dizem respeito à compilação do Rbinder (linhas 3, 10, 11 e 12) e à modificação do comando de inicialização do *container* (linha 14). Para a compilação, é necessária a instalação de algumas dependências (linha 3), a cópia do código-fonte (linhas 10 e 11) e a execução do comando de compilação (linha 12). O comando de inicialização (linha 14) é quase idêntico ao original e a principal mudança é a adição do `rbinder` ao início do comando. Além disso, utiliza-se o caminho absoluto para o local do binário da aplicação e a porta 8080 em lugar da 80.

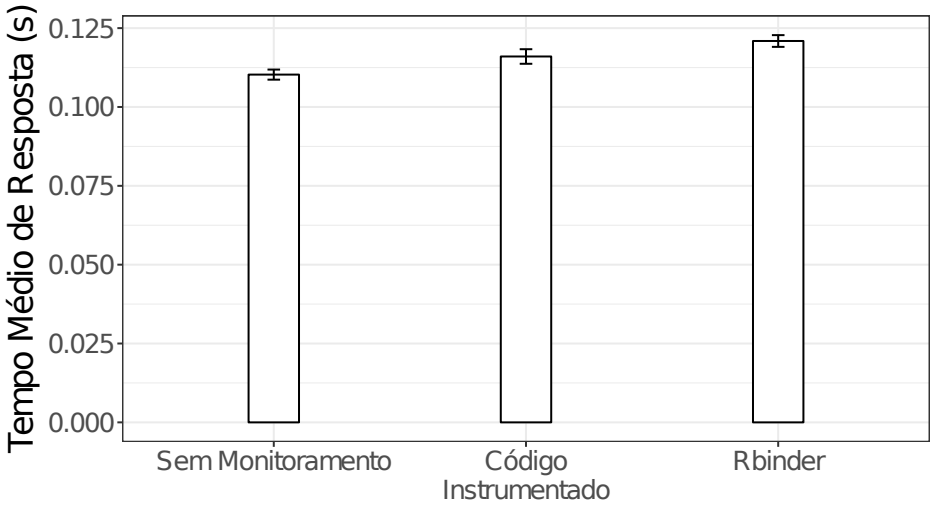
Listagem 4.3 – Arquivo modificado da definição da imagem do microsserviço `orders`

```
FROM weaveworksdemos/msd-java:jre-latest
2
RUN apk add --update gcc musl-dev linux-headers strace gdb
```

```
4
   WORKDIR /usr/src/app
6  COPY *.jar ./app.jar

8  EXPOSE 8080

10 COPY uthash.h uthash.h
   COPY rbinder.c rbinder.c
12 RUN gcc -g -o rbinder rbinder.c

14 ENTRYPOINT ["/usr/src/app/rbinder", "/usr/local/bin/java.sh", "-jar", "/usr/src/app/
       app.jar", "--port=8080"]
```

4.3 RESULTADOS

A Figura 15 mostra os resultados dos experimentos. As barras de erro indicam um intervalo de confiança de 95%. Os tempos de resposta dos cenários com monitoramento habilitado são aproximados. O tempo médio de resposta quando a aplicação não está sendo monitorada é de 0.110s contra 0.116s quando o código é instrumentado e 0.121s quando o Rbinder está habilitado. Isso significa que a instrumentação do código aumenta o tempo médio de resposta da aplicação em 5,45% enquanto o Rbinder aumenta esse tempo em 10%. Portanto, considerando essa métrica, ambas as estratégias para habilitação de *tracing* têm impacto considerável sobre o desempenho da aplicação.



Fonte: o próprio autor

Figura 15 – Tempo médio de resposta para os três cenários avaliados.

A Figura 16 apresenta a utilização de CPU na máquina virtual em que os experimentos foram executados. Apesar de o padrão de utilização de CPU ser parecido nos três cenários avaliados, é possível perceber um sutil aumento da utilização nos cenários com *tracing* habilitado. A utilização média de CPU quando a aplicação não está sendo monitorada é de 17.44%. Para a instrumentação tradicional, essa média aumenta para 18.58% e com o

Rbinder ela aumenta para 21.52%. Isso pode ser explicado pela quantidade de instruções extras necessárias para o monitoramento. No caso do Rbinder, a utilização de CPU é ainda maior devido ao monitoramento de chamadas do sistema operacional: esse monitoramento requer a execução de instruções para copiar *buffers* do espaço do *kernel* para o espaço do usuário e vice-versa, por exemplo.

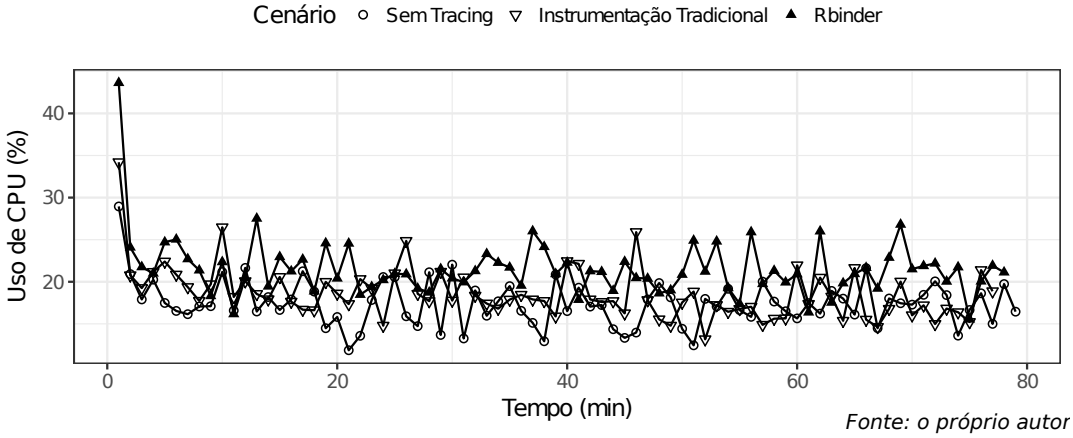


Figura 16 – Utilização de CPU

A Figura 17 apresenta a utilização de memória RAM na máquina virtual em que os experimentos foram executados. Novamente, o consumo é maior nos cenários com *tracing* habilitado. Considerando a média do consumo, ela é igual a 4.441MB quando a aplicação não é monitorada, 5.102MB quando o monitoramento se dá através da instrumentação tradicional e 5.596MB com o Rbinder. O aumento no consumo de memória nos cenários em que a aplicação é monitorada pode ser explicado pela alocação de memória relacionada ao monitoramento. O servidor de *traces*, por exemplo, demanda uma boa porção de memória. Além disso, a utilização do Rbinder para o monitoramento da aplicação demanda mais memória devido à implantação dos *proxies* que interceptam a comunicação entre os microsserviços da aplicação.

A utilização de bibliotecas desenvolvidas por terceiros para instrumentação de aplicações pode levar a consequências indesejáveis como um considerável aumento no tamanho dos binários dos componentes e o tempo necessário para sua inicialização. A Figura 18 apresenta o tamanho dos arquivos utilizados para inicializar os microsserviços da Sock Shop. Para os serviços em Go e Java, esses arquivos são os binários resultantes da compilação do código e os arquivos JAR, respectivamente. Os resultados mostram que a instrumentação resultou em binários maiores para todos os microsserviços, especialmente os escritos em Java. Em comparação com esses binários, o uso do Rbinder faz com que seu tamanho reduza em média 25%.

Nós também medimos o tempo que os microsserviços levam para se tornar operacionais do ponto de vista do usuário da aplicação. Esse tempo é relevante porque ele impacta

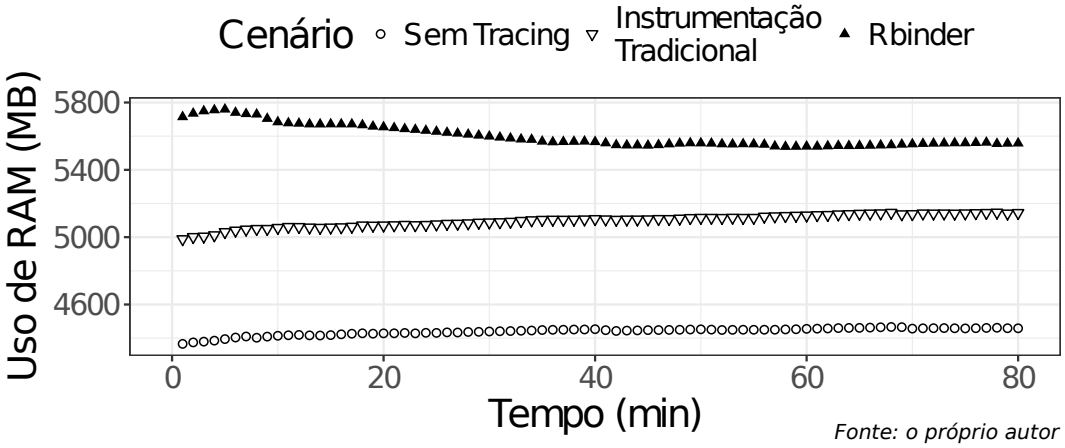


Figura 17 – Utilização de memória RAM

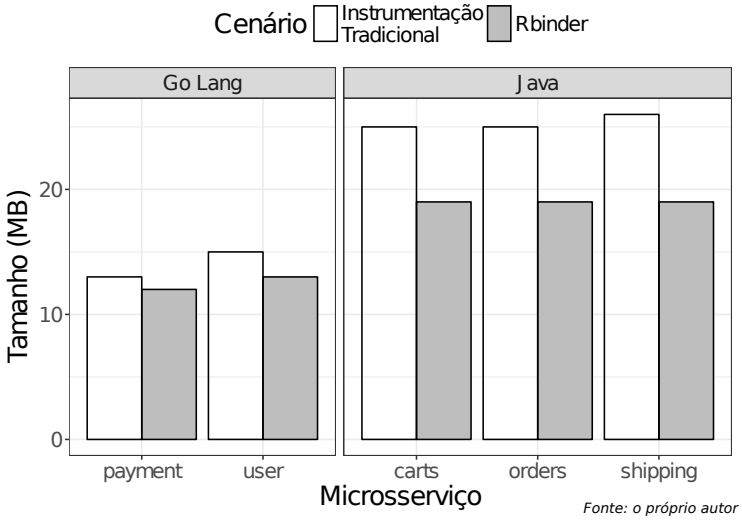


Figura 18 – Tamanho dos binários dos microsserviços.

diretamente na produtividade dos desenvolvedores e operadores dos microsserviços. Por exemplo, os desenvolvedores irão precisar de muito mais tempo se eles tiverem que esperar por períodos de inicialização mais longos em seus ambientes de desenvolvimento. Além disso, atividades de operação como replicação de instâncias para escalabilidade dos microsserviços são prejudicadas se eles levam tempo demais para iniciar. A Figura 19 apresenta os resultados. Não há grande diferença para o tempo de inicialização quando os microsserviços escritos em Go (`user` e `payment`) são considerados. O microsserviço `user` apresentou uma média de tempo de inicialização de 2.21s (desvio padrão de 0.27s) na instrumentação tradicional e 2.22s (desvio padrão de 0.17s) para Rbinder. Já `payment` levou 2.49s em média (desvio padrão de 0.10s) para inicializar com o código instrumentado e 2.22s (desvio padrão de 0.09s) para Rbinder. No entanto, há grande diferença no tempo de inicialização quando os microsserviços Java são considerados. `carts` levou 46.50s (desvio

padrão de 1.20s) para inicializar com o código instrumentado e 30.33s (desvio padrão de 0.78s) com Rbinder, `orders` demorou 46.11s (desvio padrão de 0.98s) com o código instrumentado e 32.07s (desvio padrão de 1.90s) quando monitorado com Rbinder e `shipping` levou 44.09s (desvio padrão de 0.99s) com o código instrumentado contra 28.20s (desvio padrão de 0.33s) quando Rbinder foi usado.

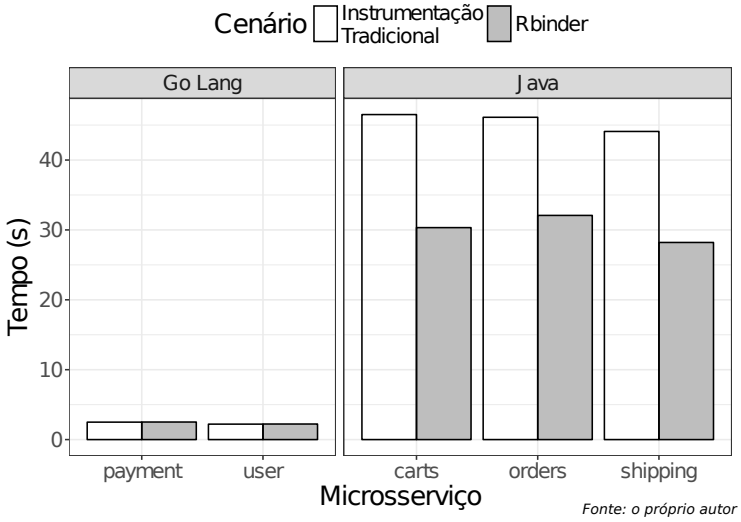


Figura 19 – Tempo de inicialização dos microsserviços

A análise dos resultados indica que o Rbinder impõe maior sobrecarga de desempenho e usa mais memória RAM. No entanto, a abordagem permite habilitar monitoramento sem qualquer intervenção no código da aplicação. Além disso, o monitoramento com o Rbinder produz binários menores, que levam menos tempo para iniciar, o que é muito proveitoso para o desenvolvimento e a operação de microsserviços porque permite implantação mais rápida em ambientes de produção e ciclos mais ágeis de desenvolvimento.

A utilização do Rbinder, mesmo diante do aumento da sobrecarga e do uso de recursos, deve ser considerada no contexto do monitoramento de aplicações baseadas em microsserviços devido aos desafios impostos pela arquitetura. A grande heterogeneidade, comum aos componentes dessa arquitetura, torna a instrumentação de código para habilitação de monitoramento muito custosa. Além disso, há situações em que o código dos microsserviços não está disponível. Nesses casos, a instrumentação de código não seria uma possibilidade e o Rbinder pode ser uma boa alternativa.

Algumas medidas podem ser adotadas para redução da degradação de desempenho e do uso de recursos apontados nos resultados da avaliação. Como o uso do Rbinder produz binários menores e de inicialização mais rápida que o monitoramento com instrumentação de código, a habilitação e a desabilitação do monitoramento é menos custosa com o Rbinder. Assim, a prática comum de habilitar o monitoramento apenas quando necessário pode ser facilmente implementada com o Rbinder. Além disso, a utilização de uma infraestrutura computacional dedicada à execução dos serviços de monitoramento (e.g., *proxies*

e servidor de *traces*) pode diminuir a degradação de desempenho e o aumento na utilização de recursos (da infraestrutura compartilhada) devido à diminuição na competição por recursos. Por fim, é possível configurar os *proxies* e expandir a implementação do Rbinder para permitir a amostragem das requisições, de maneira que apenas um percentual delas é amostrado, para reduzir degradação de desempenho e uso de recursos.

4.4 CONSIDERAÇÕES FINAIS

Este capítulo apresentou uma avaliação de desempenho do Rbinder. A avaliação teve como objetivo a comparação com outras abordagens para monitoramento de aplicações baseadas em microsserviços e os resultados mostraram que a estratégia proposta possui desempenho equiparável à alternativa para habilitação de *tracing* distribuído como meio para monitoramento. Além disso, os experimentos realizados também mostraram que a adoção da estratégia proposta acarreta benefícios para desenvolvedores e operadores de aplicações baseadas em microsserviços porque através dela é possível diminuir o tamanho dos binários dos microsserviços e reduzir o tempo que eles levam para iniciar.

5 TRABALHOS RELACIONADOS

Este capítulo apresenta os trabalhos relacionados ao Rbinder. Eles estão organizados em seções que iniciam explorando os trabalhos de monitoramento de maneira mais ampla, passam pelo monitoramento de sistemas distribuídos até chegar aos trabalhos mais similares ao Rbinder. Vale notar que alguns trabalhos relacionados pertencem a interseções dessas categorias.

5.1 MONITORAMENTO

Shende (SHENDE, 1999) apresenta uma visão geral de ferramentas de *tracing* e *profiling* no contexto do sistema operacional Linux. O tipo de instrumentação de cada ferramenta é classificado em três categorias, de acordo com o nível em que a instrumentação é realizada: linguagem de programação, biblioteca e plataforma. As que instrumentam linguagens de programação operam no código-fonte, pré-processador, compilador ou código-objeto (e.g., *bytecodes* Java). As que atuam no nível de biblioteca operam nos *linkers* (i.e., os programas que unem o código-fonte às bibliotecas de que ele depende) ou nos próprios executáveis. As que atuam no nível de plataforma operam na execução do programa monitorado. O autor destaca as relações de custo-benefício existentes entre as ferramentas que atuam nos diferentes níveis: instrumentação de mais alto nível (i.e., mais próxima do código-fonte) é mais fácil de implementar mas limita-se à linguagem alvo; já instrumentação de mais baixo nível (i.e., mais próxima da execução) é mais difícil de implementar mas limita-se apenas à plataforma alvo (contemplando todas as linguagens aplicáveis à plataforma).

Técnicas de monitoramento têm sido amplamente adotadas no desenvolvimento de sistemas adaptativos. Oreizy et al. (OREIZY et al., 1999), por exemplo, propõem uma abordagem para implementação de software adaptativo baseada na arquitetura dos sistemas que são alvos da adaptação. Os autores argumentam sobre a importância da arquitetura de software do sistema a ser adaptado para a definição de estratégias de adaptação e elencam ferramentas que podem ser usadas para implementação de sistemas adaptativos. Monitoramento é destacado como uma atividade fundamental tanto para coletar informações usadas para decidir se uma adaptação é necessária quanto para garantir que o comportamento do sistema esteja de acordo com dadas propriedades após a execução de adaptações. Como nós, os autores também chamam atenção para os custos (de instrumentação e degradação de desempenho) associados ao monitoramento e argumentam que eles devem ser levados em consideração durante a avaliação dos benefícios de uma adaptação.

Delgado et al. (DELGADO; GATES; ROACH, 2004) apresentam uma taxonomia para classificação de sistemas de monitoramento que visam a detecção de falhas. Para isso, são consideradas as características que, segundo os autores, são essenciais para a cons-

trução desses sistemas: a linguagem de especificação; o mecanismo de monitoramento; e o gerente de eventos. Além da taxonomia, os autores também fornecem uma classificação interessante de 19 sistemas. As características de sistemas de monitoramento e a abordagem utilizada para classificá-los podem ser parcialmente aplicadas para classificação da abordagem proposta nesta dissertação e em trabalhos parecidos. No entanto, a dependência da utilização de uma linguagem de especificação contrasta muito com o que tem sido praticado no contexto de microsserviços e de outras aplicações que são alvos das propostas de *tracing* distribuído – que normalmente não utilizam formalismos. Essa distinção é explicada pelo fato de os autores visarem a detecção de falhas através de abordagens que demandam a especificação dos sistemas monitorados, e.g., para geração e checagem de modelos.

PayLess (CHOWDHURY et al., 2014) é um *framework* para minimização da sobrecarga devida ao monitoramento de redes definidas por software (SDNs). A ferramenta permite coletar estatísticas de fluxo de rede através de uma API REST. A coleção de estatísticas é feita por um algoritmo adaptativo que maximiza a precisão das informações coletadas e minimiza a sobrecarga de rede imposta pelo monitoramento. Como nós, os autores têm como preocupação central a redução de custos do monitoramento. No entanto, nossos trabalhos diferem quanto aos alvos do monitoramento: Chowdhury ataca o problema no contexto de SDNs enquanto nós visamos a arquitetura de microsserviços. Adicionalmente, os autores utilizam como infraestrutura um protocolo de comunicação que já prevê a disponibilização de informações de monitoramento enquanto nós não exigimos das aplicações monitoradas nenhum suporte à atividade de monitoramento.

5.2 MONITORAMENTO DE SISTEMAS DISTRIBUÍDOS

Joyce et al. (JOYCE et al., 1987) fornecem uma definição para a atividade de monitoramento de sistemas distribuídos e apresentam seus principais desafios e aplicações. Segundo eles, esse monitoramento é mais difícil do que o de outros sistemas e envolve extração dinâmica, coleta e exibição de informações sobre interações entre processos. Além de apresentarem os aspectos básicos do monitoramento de sistemas distribuídos, os autores também fornecem uma arquitetura genérica para monitoramento desses sistemas. A arquitetura é constituída de uma metodologia para extração e coleta de informações e três diferentes possibilidades para sua exibição. Além de não contemplar os desafios específicos da utilização da arquitetura de microsserviços, a arquitetura proposta difere da nossa estratégia porque se aplica a um ambiente de programação que provê os mecanismos de monitoramento necessários à observação da aplicação. Nossa proposta, por outro lado, tenta impor o mínimo de exigências possível às aplicações monitoradas e depende basicamente do esquema de implantação e do sistema operacional utilizados.

Aguilera et al. (AGUILERA et al., 2003) publicaram um dos primeiros trabalhos a considerar caminhos de requisições – ou, em outras palavras, as relações de causalidade que

existem entre as requisições – de sistemas distribuídos. A intenção dos autores é utilizar essas informações para depurar problemas de desempenho de aplicações distribuídas sem demandar modificação de código-fonte. Eles fornecem dois algoritmos para diagnóstico de relações de causalidade que geram resultados imprecisos – isto é, nem sempre os *traces* resultantes refletem exatamente o comportamento do sistema monitorado. Além disso, o trabalho apenas argumenta sobre a possibilidade de aplicar a abordagem proposta sem demandar qualquer instrumentação mas seus experimentos contam com *logs* gerados por plataformas de *middleware* específicas. Assim, além de não contemplar os desafios específicos da arquitetura de microsserviços, a estratégia dos autores, diferente da nossa, é imprecisa e demanda instrumentação da aplicação monitorada.

Monere (WASSERMANN; EMMERICH, 2011) e Pinpoint (CHEN et al., 2002) usam instrumentação de código para coletar informações com o objetivo de realizar descoberta de serviço e detecção de falhas, respectivamente. Monere extrai estruturas de dependência a partir da documentação enquanto Pinpoint depende de instrumentação no nível de *middleware* para logar as requisições servidas por cada componente. Nossa abordagem é mais genérica no sentido de que permite capturar o comportamento mais amplo dos sistemas. Além disso, nós dependemos somente da interceptação das chamadas ao sistema operacional em vez de documentação ou instrumentação de *middleware*. Isso porque nossa proposta é direcionada a qualquer aplicação baseada em microsserviços – mesmo que ela não utilize plataformas de *middleware* específicas ou seus componentes não estejam documentados.

A estratégia de *tracing* que utilizamos é idêntica à sugerida por X-Trace (FONSECA et al., 2007) e Dapper (SIGELMAN et al., 2010). Esses trabalhos objetivam o entendimento do comportamento de sistemas distrbuídos para auxiliar na depuração de problemas de desempenho. Diferente do que sugerimos, eles dependem de instrumentação no nível de aplicação, biblioteca ou *middleware* para propagação de informações de *tracing* utilizadas para diagnosticar a causalidade das requisições. Nós também dependemos da propagação dessas informações mas usamos interceptação de chamadas do sistema e *proxies* entre os componentes para evitar a instrumentação de código da aplicação.

Mais recentemente, o Facebook apresentou a sua infraestrutura para *tracing* de desempenho ponta-a-ponta Canopy (KALDOR et al., 2017). O funcionamento é muito parecido com o de outras estratégias de *tracing*: dados de *tracing* são gerados e coletados dos componentes da aplicação e posteriormente agregados, processados e exibidos pela infraestrutura de *tracing* (CHEN et al., 2002; FONSECA et al., 2007; SIGELMAN et al., 2010). A estratégia proposta pelos autores dá ênfase à necessidade de modelar as informações de *tracing* de maneira genérica para que elas possam ser utilizadas para gerar diferentes visualizações – a depender do caso de uso de cada desenvolvedor ou operador que for depurar problemas ou implementar melhorias de desempenho. Os desenvolvedores das aplicações monitoradas precisam utilizar as APIs de instrumentação do Canopy para ha-

bilitar seu monitoramento. Por isso, diferente da nossa, essa estratégia não contempla uma possibilidade transparente para monitoramento de aplicações.

Uma abordagem diferente para captura do comportamento de sistemas distribuídos é a utilização de técnicas estatísticas para inferência de causalidade entre mensagens relacionadas. Essas técnicas têm a vantagem de exigir menos conhecimento acerca dos componentes das aplicações monitoradas, podendo até mesmo tratá-los como "caixas pretas" – isto é, sem demandar instrumentação de código. Em contrapartida, a precisão delas é apenas parcial: pode ser que os resultados obtidos não reflitam exatamente o comportamento do sistema monitorado. Xu et al. (XU et al., 2016), por exemplo, propõem uma estratégia para inferência de caminhos de processamento de requisições baseada em aprendizagem de máquina. A estratégia proposta pelos autores combina conhecimento do domínio da aplicação monitorada e o modelo gerado a partir da aprendizagem para inferir esses caminhos. O modelo é gerado a partir de *traces* do sistema operacional. Eles também utilizam chamadas do sistema operacional mas de uma maneira mais passiva: *traces* de eventos do sistema operacional são usados para alimentar o algoritmo de aprendizagem. Como nós, os autores não utilizam a modificação do código do sistema monitorado. No entanto, a proposta deles, diferente da nossa, possui um erro associado aos caminhos de requisição que são inferidos pelo modelo de inteligência artificial, i.e., esses caminhos podem não refletir com precisão o comportamento da aplicação monitorada.

5.3 MONITORAMENTO DE MICROSSERVIÇOS

A arquitetura de microsserviços é recente e, por conta disso, é difícil encontrar *benchmarks* ou aplicações de referência que possam ser utilizados. Os trabalhos na área de monitoramento, por exemplo, dependem de aplicações que suportem sua experimentação. Aderaldo et al. (ADERALDO et al., 2017) propõem um conjunto de requisitos necessários para aplicações que podem ser utilizadas para esse fim. Eles são agrupados em requisitos arquiteturais, operacionais e genéricos e incluem atributos como facilidade de acesso e popularidade. Além de sugerir parâmetros para a escolha de um *benchmark*, os autores apresentam um estudo que caracteriza 5 aplicações baseadas em microsserviços de acordo com os parâmetros sugeridos.

Da mesma forma, Brogi et al. (BROGI et al., 2017) propõem um conjunto de aplicações de referência para dar suporte à experimentação da arquitetura de microsserviços. O *benchmark* é formado por um conjunto de aplicações baseadas em microsserviços que permitem avaliar propostas tanto quantitativa quanto qualitativamente. Os autores argumentam que o conjunto de aplicações possibilita a comparação sistemática de soluções existente e também o desenvolvimento de experimentos repetíveis que podem ser utilizados para avaliar novas soluções. Os trabalhos de Aderaldo et al. (ADERALDO et al., 2017) e Brogi et al. (BROGI et al., 2017) se relacionam com o nosso no sentido de que as aplicações fornecidas poderiam ser utilizadas para a avaliação de desempenho da nossa

estratégia para monitoramento transparente. De fato, avaliação de desempenho que apresentamos utiliza como carga de trabalho uma das aplicações sugeridas por Aderaldo et al. (ADERALDO et al., 2017).

Soldani et al. (SOLDANI; TAMBURRI; HEUVEL, 2018) fazem um estudo sobre vantagens e desvantagens atreladas à utilização da arquitetura de microsserviços. Os autores identificam o monitoramento como uma das maiores dificuldades de desenvolvedores e operadores de microsserviços. Junto a ela são listadas o *design* da arquitetura, as distribuição e heterogeneidade das ferramentas de armazenamento utilizadas e o gerenciamento de componentes. A dificuldade do monitoramento é atribuída à grande quantidade e heterogeneidade dos componentes que compõem as aplicações baseadas em microsserviços.

Na mesma linha do trabalho realizado por Soldani, Dragoni et al. (DRAGONI et al., 2017) investigam os desafios associados à utilização de microsserviços. Após introduzirem o contexto e as motivações para o surgimento da arquitetura, os autores apresentam suas vantagens e desvantagens. Eles caracterizam a arquitetura de acordo com atributos de QoS: disponibilidade, confiabilidade, manutenibilidade, desempenho, segurança e testabilidade. O monitoramento de microsserviços é apontado como atividade mandatória para garantia da confiança e segurança das aplicações. Os autores também destacam a maior dificuldade de monitoramento de aplicações baseadas em microsserviços (em comparação com as demais aplicações distribuídas) devida à intrínseca complexidade de comunicação em rede que acontece entre os componentes e à sua alta heterogeneidade.

Toffetti et al. (TOFFETTI et al., 2017) propõem uma arquitetura para aplicações em nuvem escaláveis, resilientes e auto-gerenciáveis. A proposta dos autores utiliza informações de monitoramento para gerenciamento automático de escala, estado e ciclo de vida dos componentes de aplicações baseadas em microsserviços. Ela depende que cada um dos componentes seja instrumentado para agregar as informações necessárias e enviá-las a um serviço de armazenamento. Desse modo, o custo de instrumentação da aplicação monitorada não é uma preocupação dos autores. Eles também não fazem considerações sobre a degradação de desempenho devida ao monitoramento. Os autores compartilham sua experiência na aplicação prática da arquitetura proposta a uma aplicação desenvolvida e utilizada na indústria e realizam experimentos para avaliar escalabilidade e tolerância a falhas da aplicação modificada.

Sampaio et al. (SAMPAIO et al., 2017) aplicam o monitoramento para auxiliar a alocação de componentes de aplicações baseadas em microsserviços. Os componentes são alocados de acordo com regras de afinidade calculadas através de informações do monitoramento. O monitoramento acontece dentro de um ciclo de controle de computação autonômica que permite que, de acordo com as regras definidas dinamicamente, os componentes sejam realocados de maneira automática. A dificuldade de instrumentação do código para habilitação de *tracing* durante o desenvolvimento do trabalho de Sampaio et al. motivou a elaboração da proposta de *tracing* transparente apresentada nesta dissertação.

Mace e Fonseca (MACE; FONSECA, 2018) propõem uma estratégia para propagação de informações de *tracing* com o objetivo de separar lógica de instrumentação e lógica de negócio da aplicação. Para isso, os autores definem um conjunto de informações de *tracing* genérico suficiente para ser utilizado por várias ferramentas que dependem dessas informações (para permitir depuração de problemas e implementação de melhorias de desempenho, por exemplo). Ao aderir ao conjunto proposto, o código de instrumentação aplicado a cada um dos componentes da aplicação não precisa atender a demandas específicas de cada uma das ferramentas utilizadas para depuração e implementação de melhorias. É suficiente que ele propague as informações especificadas no conjunto. Como o nosso, o objetivo principal dos autores é a diminuição do custo de instrumentação para monitoramento de aplicações baseadas em microsserviços. No entanto, diferente do Rbinder, a solução proposta depende de instrumentação do código dos sistemas monitorados.

Carosi (CAROSI, 2018) correlaciona informações de *tracing* e de *profiling* para auxiliar a depuração dos sistemas monitorados. Como nossa abordagem, esse trabalho usa *proxies* para geração e envio de informações de *tracing*. Carosi, no entanto, aplica modificações a servidores web para propagação dessas informações enquanto nós utilizamos interceptação de chamadas do sistema operacional.

Os proponentes de Seer (GAN et al., 2018) utilizam *traces* para tentar prever violações à qualidade de serviço de aplicações baseadas em microsserviços. Uma rede neural utiliza dados do *tracing* para o cálculo de previsões. Apesar de a proposta ser direcionada à arquitetura de microsserviços, os autores não consideram os custos associados à modificação de código e demandam que as aplicações monitoradas sejam instrumentadas – em contraste com o que é proposto neste trabalho. Além disso, a utilização de uma técnica de inteligência artificial implica em aceitação de um percentual de erro em relação aos resultados obtidos. A abordagem que propomos, por outro lado, é precisa em relação aos *traces* que expressam o comportamento da aplicação monitorada.

Gan et al. (GAN et al., 2019) fornecem um *benchmark* para a arquitetura de microsserviços composto de aplicações baseadas em microsserviços em 5 diferentes domínios: uma rede social, um serviço de mídia, um site de de comércio eletrônico, um sistema bancário e aplicações de Internet das coisas (IoT). Os autores utilizam o *benchmark* para realizar experimentos com o objetivo de estudar características e desafios da arquitetura de microsserviços. O trabalho considera a instrumentação dos microsserviços do *benchmark* para habilitação de *tracing* distribuído com o objetivo de apontar componentes defeituosos quando acontecem problemas de desempenho. Os autores implementam sua própria estratégia para geração, coleta e exibição de *traces*, que é baseada no mesmo mecanismo do Rbinder, e instrumentam o código das aplicações. Diferente da avaliação apresentada nesta dissertação, os autores consideram que a sobrecarga devida ao *tracing* é irrisória, i.e., latência adicional menor que 0.1%. Além da divergência na avaliação de resultados, o trabalho ignora o custo de *tracing* associado à necessidade de modificação do código

das aplicações monitoradas. Nós, por outro lado, tentamos anular esse custo através da utilização de uma abordagem de *tracing* transparente.

Kitajima e Matsuoka (KITAJIMA; MATSUOKA, 2017) propõem uma maneira para diagnosticar relações de causalidade numa aplicação baseada em microsserviços. Eles sugerem a utilização de uma heurística que diagnostica essas relações de acordo com os intervalos de tempo que compreendem cada uma das requisições. Esse tipo de abordagem tem a vantagem de ser menos exigente em relação à instrumentação porque depende de menos informações sobre as mensagens trocadas. No caso da sugestão dos autores, por exemplo, as únicas informações necessárias são os endereços de origem e de destino e o momento em que cada mensagem aconteceu. No entanto, as relações de causalidade reportadas pela abordagem não são precisas. Isto é, elas podem não refletir com exatidão o comportamento da aplicação.

5.4 MONITORAMENTO DE CHAMADAS DO SISTEMA OPERACIONAL

Yaghmour e Dagenais (YAGHMOUR; DAGENAIS, 2000) fornecem um *kit* de ferramentas para captura e análise de comportamento de sistemas de software complexos. As ferramentas são baseadas no sistema operacional Linux e dependem das facilidades fornecidas pelo *kernel* do sistema operacional e de instrumentação do *kernel* para geração de *logs* de eventos relacionados à execução de chamadas do sistema. Assim como o nosso, o trabalho dos autores utiliza monitoramento de chamadas do sistema operacional para permitir o rastreamento do comportamento de aplicações mas eles não consideram o contexto de aplicações distribuídas.

Os idealizadores de Magpie (BARHAM et al., 2003) argumentam sobre a necessidade de transformar modelagem de desempenho numa preocupação holística através da associação de *traces* do desempenho de cada componente dos sistemas monitorados. Eles apresentam sua criação como uma ferramenta de modelagem *online* capaz de instrumentar componentes "caixa-preta" (sem modificação de código) e reportar caminhos da execução do sistema (isto é, expondo causalidade entre as requisições). A ferramenta coleta *traces* de todos os componentes, combina-os, destaca a requisição inicial, constrói um modelo probabilístico de comportamento através de aprendizagem de máquina e detecta anomalias através da comparação de requisições individuais com o modelo probabilístico. A instrumentação considera atividades do *kernel*, RPCs, chamadas do sistema e comunicação de rede. Diferente do Rbinder, a proposta de Barham depende de instrumentação no nível de linguagem de programação.

Ardelean et al. (ARDELEAN; DIWAN; ERDMAN, 2018) usam uma estratégia de *tracing* para coleta de informações num dado período com o objetivo de entender o comportamento do sistema diante de variações de carga que são comuns em ambientes de produção. Eles também usam chamadas do sistema para diagnosticar causalidade entre mensagens trocadas por componentes de diferentes camadas da arquitetura. Diferente da

nossa proposa, que considera apenas as chamadas que são feitas pela aplicação, eles injetam chamadas ao sistema operacional para permitir o diagnóstico de causalidade entre mensagens.

Callahan et al. (O'CALLAHAN et al., 2017) usam chamadas do sistema operacional não para monitorar sistemas distribuídos mas para auxiliar na depuração de programas. Eles propõem uma técnica que utiliza monitoramento de chamadas do sistema para rastrear e registrar tudo que um programa faz. A partir dos registros, a execução do programa pode ser reproduzida de maneira fiel ao que aconteceu no ambiente de produção e isso auxilia os desenvolvedores a depurar erros que não acontecem nos ambientes de desenvolvimento e testes.

5.5 CONSIDERAÇÕES FINAIS

O capítulo apresentou os principais trabalhos relacionados à nossa estratégia transparente para *tracing* de aplicações baseadas em microsserviços. Eles foram agrupados em quatro grupos: monitoramento, monitoramento de sistemas distribuídos, monitoramento de microsserviços e monitoramento de chamadas do sistema operacional. Os trabalhos de destaque pertencem à categoria dos que sugerem abordagens para monitoramento de aplicações baseadas em microsserviços. Eles sempre demandam algum tipo de instrumentação ou fornecem resultados imprecisos. A principal vantagem do Rbinder é que ele fornece resultados precisos ao mesmo tempo em que não demanda instrumentação do código da aplicação ou das bibliotecas e plataformas de *middleware* sobre as quais ela está implementada.

6 CONCLUSÕES E TRABALHOS FUTUROS

Este capítulo apresenta as principais contribuições do trabalho, suas limitações e possibilidades para trabalhos futuros.

6.1 CONTRIBUIÇÕES

O monitoramento de sistemas computacionais é uma ferramenta útil para auxílio à depuração de problemas e implementação de melhorias de desempenho. No entanto, sua utilização implica em custos de desenvolvimento e operacionais. O desenvolvimento é afetado porque a maior parte das estratégias existentes depende de modificações de software enquanto as operações sofrem porque a observação do sistema degrada o seu desempenho.

No contexto da arquitetura de microsserviços, esses custos são agravados pela numerosa quantidade de componentes que compõem esse tipo de arquitetura e a heterogeneidade de tecnologias com que eles são implementados. Este trabalho propôs uma estratégia para monitoramento de aplicações baseadas em microsserviços que visa minimizar custos de desenvolvimento e operacionais. Dentre as principais contribuições, estão:

- A proposta de uma estratégia de monitoramento para habilitação de *tracing* preciso sem modificação do código-fonte da aplicação monitorada;
- A disponibilização de uma implementação para monitoramento de chamadas do sistema operacional Linux para propagação de informações de *tracing*; e
- Uma avaliação de desempenho de estratégias de monitoramento de aplicações baseadas em microsserviços que mostra a degradação de desempenho devida ao monitoramento.

Microsserviços são um tópico de pesquisa que tem ganhado atenção recentemente e parece promissor para o desenvolvimento de vários outros trabalhos de pesquisa. Essa intuição é corroborada pela publicação recente de iniciativas para elaboração e coleção de sistemas de referência que possam ser utilizados para execução de experimentos. A maior parte desses experimentos depende do monitoramento desses sistemas para avaliação das estratégias propostas. Daí decorre a relevância do tópico em que este trabalho se insere.

Assim, além das contribuições diretas a pesquisadores, desenvolvedores e operadores de microsserviços, este trabalho também contribui ao chamar atenção para a possibilidade de alternativas transparentes de monitoramento que muito facilitariam as atividades desses profissionais.

6.2 LIMITAÇÕES

A razão da existência deste trabalho é a diminuição dos custos associados ao monitoramento de aplicações baseadas em microsserviços. Essa diminuição se volta principalmente ao custo relacionado à necessidade de instrumentação de código da aplicação. No entanto, apesar de argumentarmos sobre os benefícios da extinção dessa necessidade, seu impacto na produtividade de desenvolvedores e operadores de microsserviços não foi avaliado. Essa avaliação demandaria um estudo empírico da utilização da nossa estratégia em ambientes reais de desenvolvimento e operação. Diante das dificuldades associadas a um estudo desse tipo, restringimos a avaliação da nossa estratégia a uma avaliação de desempenho frente a estratégias alternativas.

A avaliação de desempenho realizada, apesar de expressiva, também tem limitações. A carga de trabalho utilizada, por exemplo, conta com uma única aplicação baseada em microsserviços fictícia que foi implantada com o propósito específico de ser avaliada. Além disso, apesar de termos avaliado os principais aspectos associado à operação de aplicações web, a avaliação não contou com averiguação do impacto do monitoramento sobre utilização de rede (quantidade de pacotes trafegados e banda utilizada, por exemplo). Essa avaliação é especialmente interessante no contexto de microsserviços porque a arquitetura estimula a utilização de rede devido a granularidade dos componentes, que acabam dependendo muito de comunicação através de mensagens.

Os resultados mostraram que a estratégia proposta acarreta degradação de desempenho similar à alternativa com que ela foi comparada. Como nosso propósito é fornecer a desenvolvedores e operadores de microsserviços uma maneira para habilitar *tracing* preciso e contínuo (sempre ligado) de suas aplicações, este resultado também é uma limitação do trabalho. Essa limitação é agravada pelo fato de a degradação de desempenho ser um fator preponderante para a não utilização de monitoramento.

O monitoramento de chamadas do sistema operacional implementado pode ter impacto crucial sobre o comportamento das aplicações monitoradas porque inclui a modificação de parâmetros de algumas chamadas. Além disso, a API do Linux utilizada para esse monitoramento permite total controle sobre os processos monitorados, o que possibilita, por exemplo, sua interrupção ou a decisão sobre execução ou não das chamadas feitas por eles. Isso demanda que a estratégia seja melhor avaliada e refinada antes de ser aplicada em ambientes de produção.

Por fim, a estratégia, pelo menos até o ponto em que ela foi explorada no escopo deste trabalho, possui algumas limitações que parecem facilmente contornáveis. Uma delas é o suporte a outros protocolos além do HTTP. Foi argumentado que a extensão de sua aplicação a outros protocolos é factível e facilmente implementável mas seria interessante experimentar essa extensão na prática. Essa extensão é muito interessante no contexto de microsserviços, que comumente conta com pluralidade de protocolos utilizados.

Uma outra extensão que seria útil a desenvolvedores ao depurar suas aplicações é o

suporte a informações específicas de aplicação. A estratégia apresentada permite conhecer o comportamento genérico ao expor tempo de resposta dos componentes e as dependências estruturais entre eles. Mas respostas a perguntas específicas do domínio da aplicação como "qual o identificador do usuário quando o microsserviço X demora a responder?" podem ser muito úteis ao depurar problemas em ambientes de produção. Essa limitação parece facilmente contornável e pode ter relação com a limitação aos protocolos suportados já que, por exemplo, cabeçalhos HTTP podem ser usados para trafegar essas informações específicas de aplicação.

6.3 TRABALHOS FUTUROS

Muitas possibilidades de trabalhos futuros derivam das limitações apresentadas. Algumas delas dizem respeito à avaliação de estratégias para monitoramento de aplicações baseadas em microsserviços. Seria interessante realizar um estudo empírico do impacto da transparência da estratégia de monitoramento na produtividade de desenvolvedores e operadores dessas aplicações. Também seria importante avaliar o desempenho em ambientes de produção reais, incluindo a averiguação de como o monitoramento impacta a utilização de recursos de rede no contexto de microsserviços.

Outra possibilidade que deriva das limitações apresentadas é a melhoria de desempenho da estratégia para monitoramento transparente. Da maneira como vemos, essa possibilidade poderia se dar através de duas maneiras: refinamento da estratégia proposta neste trabalho ou elaboração de uma estratégia diferente que seguisse os princípios de transparência e baixo impacto ao desempenho do sistema monitorado.

Refinamentos na abordagem de utilização de *proxies* poderiam reduzir o impacto sobre desempenho. A utilização de menos *proxies*, por exemplo, pode reduzir o consumo de recursos computacionais e influenciar positivamente o tempo de resposta dos microsserviços monitorados. Da mesma maneira, refinamentos da implementação de monitoramento de chamadas do sistema operacional podem reduzir a degradação do desempenho. O monitoramento dessas chamadas tem alto impacto devido às trocas de contexto que precisam acontecer a cada interrupção. Tentou-se mitigar esse impacto através da utilização de filtros que só interrompem as chamadas de interesse mas talvez existam formas mais eficientes para realização desse monitoramento.

Em relação à elaboração de uma estratégia diferente que tivesse menos impacto sobre o desempenho da aplicação monitorada, durante a execução do trabalho esboçamos algumas abordagens que parecem interessantes. A mais promissora delas utiliza *logs* do *kernel* do Linux para armazenamento de informações de identificação das mensagens em um serviço de armazenamento que é consultado pelos *proxies*. Isso permite que as mensagens tenham suas relações de causalidade diagnosticadas sem a interceptação direta de chamadas do sistema – suas informações são extraídas a partir dos *logs* do *kernel*. A intenção é que a

alternativa à interceptação de chamadas consuma menos tempo e, assim, a degradação do desempenho da aplicação monitorada seja reduzida.

REFERÊNCIAS

ACETO, G.; BOTTA, A.; DONATO, W. D.; PESCAPÈ, A. Cloud monitoring: A survey. *Computer Networks*, Elsevier, v. 57, n. 9, p. 2093–2115, 2013.

ADERALDO, C. M.; MENDONÇA, N. C.; PAHL, C.; JAMSHIDI, P. Benchmark requirements for microservices architecture research. In: IEEE PRESS. *Proceedings of the 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering*. [S.l.], 2017. p. 8–13.

AGUILERA, M. K.; MOGUL, J. C.; WIENER, J. L.; REYNOLDS, P.; MUTHITACHAROEN, A. Performance debugging for distributed systems of black boxes. *ACM SIGOPS Operating Systems Review*, ACM, v. 37, n. 5, p. 74–89, 2003.

ANSARI, S.; RAJEEV, S.; CHANDRASHEKAR, H. Packet sniffing: a brief introduction. *IEEE potentials*, IEEE, v. 21, n. 5, p. 17–19, 2002.

APPCENTRICA. *The Rise of Microservices*. 2016. Disponível em: <https://www.appcentrica.com/the-rise-of-microservices/>.

ARDELEAN, D.; DIWAN, A.; ERDMAN, C. Performance analysis of cloud applications. In: USENIX ASSOCIATION. *15th USENIX Symposium on Networked Systems Design and Implementation NSDI 18)*. [S.l.], 2018.

BAGHERZADEH, M.; KAHANI, N.; BEZEMER, C.-P.; HASSAN, A. E.; DINGEL, J.; CORDY, J. R. Analyzing a decade of linux system calls. *Empirical Software Engineering*, Springer, v. 23, n. 3, p. 1519–1551, 2018.

BALALAIE, A.; HEYDARNOORI, A.; JAMSHIDI, P. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, IEEE, v. 33, n. 3, p. 42–52, 2016.

BARHAM, P.; DONNELLY, A.; ISAACS, R.; MORTIER, R. Using magpie for request extraction and workload modelling. In: *OSDI*. [S.l.: s.n.], 2004. v. 4, p. 18–18.

BARHAM, P.; ISAACS, R.; MORTIER, R.; NARAYANAN, D. Magpie: Online modelling and performance-aware systems. In: *HotOS*. [S.l.: s.n.], 2003. p. 85–90.

BECK, F.; FESTOR, O. Syscall interception in xen hypervisor. 2009.

BROGI, A.; CANCIANI, A.; NERI, D.; RINALDI, L.; SOLDANI, J. Towards a reference dataset of microservice-based applications. In: *Proceedings of the 1th Workshop on Microservices: Science and Engineering (MSE 2017). Springer*. [S.l.: s.n.], 2017.

CACERES, M. Syscall proxying-simulating remote execution. *Core Security Technologies*, 2002.

CAROSI, R. *Protractor: Leveraging distributed tracing in service meshes for application profiling at scale*. 2018.

CHANG, F.; DEAN, J.; GHEMAWAT, S.; HSIEH, W. C.; WALLACH, D. A.; BURROWS, M.; CHANDRA, T.; FIKES, A.; GRUBER, R. E. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, ACM, v. 26, n. 2, p. 4, 2008.

CHATEL, M. *Classical versus transparent IP proxies*. [S.l.], 1996.

CHEN, M. Y.; KICIMAN, E.; FRATKIN, E.; FOX, A.; BREWER, E. Pinpoint: Problem determination in large, dynamic internet services. In: IEEE. *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*. [S.l.], 2002. p. 595–604.

CHOWDHURY, S. R.; BARI, M. F.; AHMED, R.; BOUTABA, R. Payless: A low cost network monitoring framework for software defined networks. In: IEEE. *Network Operations and Management Symposium (NOMS), 2014 IEEE*. [S.l.], 2014. p. 1–9.

DELGADO, N.; GATES, A. Q.; ROACH, S. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on software Engineering*, IEEE, v. 30, n. 12, p. 859–872, 2004.

DRAGONI, N.; GIALLORENZO, S.; LAFUENTE, A. L.; MAZZARA, M.; MONTESI, F.; MUSTAFIN, R.; SAFINA, L. Microservices: yesterday, today, and tomorrow. In: *Present and Ulterior Software Engineering*. [S.l.]: Springer, 2017. p. 195–216.

ERL, T. *Service-oriented architecture*. [S.l.]: Prentice hall New York, 2005. v. 8.

FONSECA, R.; PORTER, G.; KATZ, R. H.; SHENKER, S.; STOICA, I. X-trace: A pervasive network tracing framework. In: USENIX ASSOCIATION. *Proceedings of the 4th USENIX conference on Networked systems design & implementation*. [S.l.], 2007. p. 20–20.

FOX, A.; GRIBBLE, S. D.; CHAWATHE, Y.; BREWER, E. A. Adapting to network and client variation using infrastructural proxies: Lessons and perspectives. *IEEE Personal Communications*, IEEE, v. 5, n. 4, p. 10–19, 1998.

FU, Q.; LOU, J.-G.; WANG, Y.; LI, J. Execution anomaly detection in distributed systems through unstructured log analysis. In: IEEE. *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*. [S.l.], 2009. p. 149–158.

GAN, Y.; PANCHOLI, M.; CHENG, D.; HU, S.; HE, Y.; DELIMITROU, C. Seer: leveraging big data to navigate the complexity of cloud debugging. In: USENIX ASSOCIATION. *Proceedings of the 10th USENIX Conference on Hot Topics in Cloud Computing*. [S.l.], 2018. p. 13–13.

GAN, Y.; ZHANG, Y.; CHENG, D.; SHETTY, A.; RATHI, P.; KATARKI, N.; BRUNO, A.; HU, J.; RITCHKEN, B.; JACKSON, B. et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud and edge systems. In: *Proceedings of the Twenty Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. [S.l.: s.n.], 2019.

GARFINKEL, T. et al. Traps and pitfalls: Practical problems in system call interposition based security tools. In: *NDSS*. [S.l.: s.n.], 2003. v. 3, p. 163–176.

GARLAN, D.; SHAW, M. An introduction to software architecture. In: *Advances in software engineering and knowledge engineering*. [S.l.]: World Scientific, 1993. p. 1–39.

JOHNSON, D. B.; ZWAENEPOEL, W. *Sender-based message logging*. [S.l.]: Rice University, Department of Computer Science, 1987.

JOHNSON, D. B.; ZWAENEPOEL, W. Recovery in distributed systems using optimistic message logging and checkpointing. *Journal of algorithms*, Elsevier, v. 11, n. 3, p. 462–491, 1990.

JOYCE, J.; LOMOW, G.; SLIND, K.; UNGER, B. Monitoring distributed systems. *ACM Transactions on Computer Systems (TOCS)*, ACM, v. 5, n. 2, p. 121–150, 1987.

KALDOR, J.; MACE, J.; BEJDA, M.; GAO, E.; KUROPATWA, W.; O'NEILL, J.; ONG, K. W.; SCHALLER, B.; SHAN, P.; VISCOMI, B. et al. Canopy: An end-to-end performance tracing and analysis system. In: ACM. *Proceedings of the 26th Symposium on Operating Systems Principles*. [S.l.], 2017. p. 34–50.

KENISTON, J.; MAVINAKAYANAHALLI, A.; PANCHAMUKHI, P.; PRASAD, V. Ptrace, utrace, uprobes: Lightweight, dynamic tracing of user apps. In: *Proceedings of the 2007 Linux symposium*. [S.l.: s.n.], 2007. p. 215–224.

KITAJIMA, S.; MATSUOKA, N. Inferring calling relationship based on external observation for microservice architecture. In: SPRINGER. *International Conference on Service-Oriented Computing*. [S.l.], 2017. p. 229–237.

LEWIS, J.; FOWLER, M. Microservices: a definition of this new architectural term. *MartinFowler. com*, v. 25, Mar 2014.

LIU, G.; TROTTER, M.; REN, Y.; WOOD, T. Netalytics: Cloud-scale application performance monitoring with sdn and nfv. In: ACM. *Proceedings of the 17th International Middleware Conference*. [S.l.], 2016. p. 8.

LOU, J.-G.; FU, Q.; WANG, Y.; LI, J. Mining dependency in distributed systems through unstructured logs analysis. *ACM SIGOPS Operating Systems Review*, ACM, v. 44, n. 1, p. 91–96, 2010.

LUK, C.-K.; COHN, R.; MUTH, R.; PATIL, H.; KLAUSER, A.; LOWNEY, G.; WALLACE, S.; REDDI, V. J.; HAZELWOOD, K. Pin: building customized program analysis tools with dynamic instrumentation. In: ACM. *Acm sigplan notices*. [S.l.], 2005. v. 40, n. 6, p. 190–200.

MACE, J.; FONSECA, R. Universal context propagation for distributed system instrumentation. In: ACM. *Proceedings of the Thirteenth EuroSys Conference*. [S.l.], 2018. p. 8.

MACE, J.; ROELKE, R.; FONSECA, R. Pivot tracing: Dynamic causal monitoring for distributed systems. In: ACM. *Proceedings of the 25th Symposium on Operating Systems Principles*. [S.l.], 2015. p. 378–393.

MACE, J.; ROELKE, R.; FONSECA, R. Pivot tracing: Dynamic causal monitoring for distributed systems. *ACM Transactions on Computer Systems (TOCS)*, ACM, v. 35, n. 4, p. 11, 2018.

MATTERN, F. et al. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, Citeseer, v. 1, n. 23, p. 215–226, 1989.

NEWMAN, S.; MICROSERVICES, B. O'reilly media inc. *Building Microservices*, 2015.

OREIZY, P.; GORLICK, M. M.; TAYLOR, R. N.; HEIMHIGNER, D.; JOHNSON, G.; MEDVIDOVIC, N.; QUILICI, A.; ROSENBLUM, D. S.; WOLF, A. L. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems and Their Applications*, IEEE, v. 14, n. 3, p. 54–62, 1999.

O'CALLAHAN, R.; JONES, C.; FROYD, N.; HUEY, K.; NOLL, A.; PARTUSH, N. Engineering record and replay for deployability. In: *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC'17). USENIX Association, Berkeley, CA, USA*. [S.l.: s.n.], 2017. p. 377–389.

PADALA, P. Playing with ptrace, part i. *Linux Journal*, Belltown Media, v. 2002, n. 103, p. 5, 2002.

PAHL, C.; JAMSHIDI, P. Microservices: A systematic mapping study. In: *CLOSER (1)*. [S.l.: s.n.], 2016. p. 137–146.

PAPAZOGLOU, M. P. Service-oriented computing: Concepts, characteristics and directions. In: IEEE. *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*. [S.l.], 2003. p. 3–12.

RAYNAL, M.; SINGHAL, M. Logical time: Capturing causality in distributed systems. *Computer*, IEEE, v. 29, n. 2, p. 49–56, 1996.

SAMBASIVAN, R. R.; ZHENG, A. X.; ROSA, M. D.; KREVAT, E.; WHITMAN, S.; STROUCKEN, M.; WANG, W.; XU, L.; GANGER, G. R. Diagnosing performance changes by comparing request flows. In: *NSDI*. [S.l.: s.n.], 2011. v. 5, p. 1–1.

SAMPAIO, A. R.; KADIYALA, H.; HU, B.; STEINBACHER, J.; ERWIN, T.; ROSA, N.; BESCHASTNIKH, I.; RUBIN, J. Supporting microservice evolution. In: IEEE. *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. [S.l.], 2017. p. 539–543.

SHARMA, D.; PODDAR, R.; MAHAJAN, K.; DHAWAN, M.; MANN, V. H ansel: diagnosing faults in openstack. In: ACM. *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*. [S.l.], 2015. p. 23.

SHENDE, S. Profiling and tracing in linux. In: CITESEER. *Proceedings of the Extreme Linux Workshop*. [S.l.], 1999. v. 2.

SIGELMAN, B. H.; BARROSO, L. A.; BURROWS, M.; STEPHENSON, P.; PLAKAL, M.; BEAVER, D.; JASPAN, S.; SHANBHAG, C. *Dapper, a large-scale distributed systems tracing infrastructure*. [S.l.], 2010.

SOLDANI, J.; TAMBURRI, D. A.; HEUVEL, W.-J. V. D. The pains and gains of microservices: A systematic grey literature. 2018.

SUN, P.; YU, M.; FREEDMAN, M. J.; REXFORD, J. Identifying performance bottlenecks in cdns through tcp-level monitoring. In: ACM. *Proceedings of the first ACM SIGCOMM workshop on Measurements up the stack*. [S.l.], 2011. p. 49–54.

TAK, B.-C.; TANG, C.; ZHANG, C.; GOVINDAN, S.; URGAONKAR, B.; CHANG, R. N. vpath: Precise discovery of request processing paths from black-box observations of thread and network activities. In: *USENIX Annual technical conference*. [S.l.: s.n.], 2009.

TIERNEY, B.; JOHNSTON, W.; CROWLEY, B.; HOO, G.; BROOKS, C.; GUNTER, D. The netlogger methodology for high performance distributed systems performance analysis. In: IEEE. *High Performance Distributed Computing, 1998. Proceedings. The Seventh International Symposium on*. [S.l.], 1998. p. 260–267.

TOFFETTI, G.; BRUNNER, S.; BLÖCHLINGER, M.; SPILLNER, J.; BOHNERT, T. M. Self-managing cloud-native applications: Design, implementation, and experience. *Future Generation Computer Systems*, Elsevier, v. 72, p. 165–179, 2017.

URGAONKAR, B.; SHENOY, P.; CHANDRA, A.; GOYAL, P. Dynamic provisioning of multi-tier internet applications. In: IEEE. *Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on*. [S.l.], 2005. p. 217–228.

WANG, Z.; SANCHEZ, A.; HERKERSDORF, A. Scisim: a software performance estimation framework using source code instrumentation. In: ACM. *Proceedings of the 7th international workshop on Software and performance*. [S.l.], 2008. p. 33–42.

WASSERMANN, B.; EMMERICH, W. Monere: Monitoring of service compositions for failure diagnosis. In: SPRINGER. *International Conference on Service-Oriented Computing*. [S.l.], 2011. p. 344–358.

WEAVER, N.; KREIBICH, C.; DAM, M.; PAXSON, V. Here be web proxies. In: SPRINGER. *International Conference on Passive and Active Network Measurement*. [S.l.], 2014. p. 183–192.

WINTER, J. Trusted computing building blocks for embedded linux-based arm trustzone platforms. In: ACM. *Proceedings of the 3rd ACM workshop on Scalable trusted computing*. [S.l.], 2008. p. 21–30.

XU, H.; NING, X.; ZHANG, H.; RHEE, J.; JIANG, G. Pinfer: Learning to infer concurrent request paths from system kernel events. In: IEEE. *Autonomic Computing (ICAC), 2016 IEEE International Conference on*. [S.l.], 2016. p. 199–208.

YAGHMOUR, K.; DAGENAIS, M. R. Measuring and characterizing system behavior using kernel-level event logging. In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference. Berkeley, CA, USA*. [S.l.: s.n.], 2000. v. 2, p. 2.

GLOSSÁRIO

API	Application Programming Interface
CPU	Central Processing Unit
DNS	Domain Name System
gRPC	gRPC Remote Procedure Call
JAR	Java ARchive
JSON	JavaScript Object Notation
KVM	Kernel-based Virtual Machine
HTTP	HyperText Transfer Protocol
IoT	Internet of Things
IP	Internet Protocol
QoS	Quality of Service
RAM	Random Access Memory
REST	Representational State Transfer
RPC	Remote Procedure Call
TCP	Transmission Control Protocol
SDN	Software-Defined Network
SECCOMP	SECure COMPuting
SOA	Service-Oriented Architecture
WWW	World Wide Web

APÊNDICE A – CONFIGURAÇÃO DO PROXY `orders-front`

Listagem A.1 – Arquivo de configuração do proxy `orders-front`

```
static_resources:
  listeners:
  - address:
      socket_address:
        address: 0.0.0.0
        port_value: 80
    filter_chains:
    - filters:
      - name: envoy.http_connection_manager
        config:
          generate_request_id: true
          tracing:
            operation_name: egress
          codec_type: auto
          stat_prefix: ingress_http
          route_config:
            name: local_route
            virtual_hosts:
            - name: backend
              domains:
              - "*"
              routes:
              - match:
                  prefix: "/"
                route:
                  cluster: orders
                decorator:
                  operation: checkAvailability
          http_filters:
          - name: envoy.router
            config: {}
  clusters:
  - name: orders
    connect_timeout: 0.250s
    type: strict_dns
    lb_policy: round_robin
    hosts:
    - socket_address:
        address: orders-envoy
        port_value: 80
admin:
  access_log_path: "/dev/null"
  address:
    socket_address:
      address: 0.0.0.0
      port_value: 8001
```

APÊNDICE B – CÓDIGO C DO RBINDER

Listagem B.1 – Código em linguagem de programação C do Rbinder

```c
#include <ctype.h>
#include <errno.h>
#include <stdio.h>

#include <linux/filter.h>
#include <linux/seccomp.h>

#include <sys/prctl.h>
#include <sys/ptrace.h>
#include <sys/reg.h>
#include <sys/syscall.h>
#include <sys/wait.h>

#include "uthash.h"

#define SCREAD(number)     (number == SYS_read)
#define SCSENDTO(number)   (number == SYS_sendto)
#define SCRECVFROM(number) (number == SYS_recvfrom)

#define SCENTRY(code) (code == -ENOSYS)

#ifdef __x86_64__
#define WORD_LENGTH 8
#else
#define WORD_LENGTH 4
#endif

#define REG_SC_NUMBER  (WORD_LENGTH * ORIG_RAX)
#define REG_SC_RETCODE (WORD_LENGTH * RAX)
#define REG_SC_FRSTARG (WORD_LENGTH * RDI)
#define REG_SC_SCNDARG (WORD_LENGTH * RSI)
#define REG_SC_THRDARG (WORD_LENGTH * RDX)

#define ARG_SCRW_BUFF     1
#define ARG_SCRW_BUFFSIZE 2

const int long_size = sizeof(long);

const char *fine_headers[] = {
  "x-ot-span-context",
  "x-request-id",
  "x-b3-traceid",
  "x-b3-spanid",
  "x-b3-parentspanid",
  "x-b3-sampled",
  "x-b3-flags"
};

/*
 * ptrace helper functions.
```

```
 */
void peekdata(pid_t child, long addr, char *str, int len) {
  char *laddr;
  int i, j;
  union u {
    long val;
    char chars[long_size];
  }data;
  i = 0;
  j = len / long_size;
  laddr = str;
  while(i < j) {
    data.val = ptrace(PTRACE_PEEKDATA, child, addr + i * 8, NULL);
    memcpy(laddr, data.chars, long_size);
    ++i;
    laddr += long_size;
  }
  j = len % long_size;
  if(j != 0) {
    data.val = ptrace(PTRACE_PEEKDATA, child, addr + i * 8, NULL);
    memcpy(laddr, data.chars, j);
  }
  str[len] = '\0';
}

void pokedata(pid_t child, long addr, char *str, int len) {
  char *laddr;
  int i, j;
  union u {
          long val;
          char chars[long_size];
  }data;
  i = 0;
  j = len / long_size;
  laddr = str;
  while(i < j) {
    memcpy(data.chars, laddr, long_size);
    ptrace(PTRACE_POKEDATA, child, addr + i * 8, data.val);
    ++i;
    laddr += long_size;
  }
  j = len % long_size;
  if(j != 0) {
      memcpy(data.chars, laddr, j);
      ptrace(PTRACE_POKEDATA, child, addr + i * 8, data.val);
  }
  ptrace(PTRACE_POKEUSER, child, 8 * RDX, len);
}

long peekuser(pid_t cid, unsigned int reg) {
  long ret = ptrace(PTRACE_PEEKUSER, cid, reg, NULL);
  if(errno < 0) {
    perror("ptrace(PTRACE_PEEKUSER)");
    exit(1);
  }
  return ret;
}
```

```
108
void peek_syscall_thrargs(pid_t cid, long *params) {
110   params[0] = peekuser(cid, REG_SC_FRSTARG);
      params[1] = peekuser(cid, REG_SC_SCNDARG);
112   params[2] = peekuser(cid, REG_SC_THRDARG);
}
114
/*
116  * Helper functions for handling requests.
 */
118 void extract_headers(char *str, char *headers) {
      int chidx, hdidx, matchidx, i;
120   chidx = 0;
      matchidx = 0;
122   const char *cheader = NULL;
      char elected[1024] = {'\0'};
124   int electedidx = 0;
      char cchar = '\0';
126
      // Try matching each tracing header.
128   for(hdidx = 0; hdidx < 7; hdidx++) {
        cheader = fine_headers[hdidx];
130     matchidx = 0;

132     // Check each char.
        for(chidx = 0; chidx < strlen(str); chidx++) {
134       cchar = str[chidx];

136       // Get out before reaching HTTP data section.
          if(chidx > 0 && cchar == '\r' && str[chidx-1] == '\n' && str[chidx+1] == '\n')
               {
138         continue;
          }
140
          // Don't care about this char.
142       if(cchar == '\r') {
            continue;
144       }

146       // Line break: restart matching info.
          if(cchar == '\n') {
148         matchidx = 0;
            continue;
150       }

152       // Matching already failed for current line.
          if(matchidx == -1) {
154         continue;
          }
156
          // Still didn't match entire header.
158       if(matchidx < strlen(cheader)) {
            if(tolower(cchar) == cheader[matchidx]) {
160           ++matchidx;
            } else {
162           matchidx = -1;
            }
```

```
164       }

166       // Matched entire header.
          else {
168         // Copy header key.
            if(matchidx == strlen(cheader)) {
170           for(i = 0; i < matchidx; i++) {
                headers[electedidx] = cheader[i];
172             ++electedidx;
              }
174         }

176         // Copy header value (including ": ").
            headers[electedidx] = cchar;
178         ++electedidx;
            ++matchidx;
180         if(str[chidx+1] == '\r') {
              headers[electedidx] = '\r';
182           headers[electedidx+1] = '\n';
              electedidx = electedidx + 2;
184         }
          }
186     }
    }
188   // Fill headers with \0.
    for(i = electedidx; i < 1024; i++) {
190   headers[i] = '\0';
    }
192 }

194 void inject_headers(char *str, char *headers, char *newstr, int newstrsize) {
    int i, j;
196   int stridx = 0;
    int injected = 0;
198
    for(i = 0; i < newstrsize; i++) {
200   newstr[i] = str[stridx];
      if(str[stridx] == '\n' && str[stridx+1] == '\r' && injected == 0) {
202     for(j = 0; j < strlen(headers); j++) {
          newstr[i+1+j] = headers[j];
204     }
        i += strlen(headers);
206     injected = 1;
      }
208   ++stridx;
    }
210   newstr[newstrsize] = '\0';
  }
212
  int is_http_request(char *str) {
214   char *httpmeths[9] = {
      "GET", "HEAD", "POST", "PUT", "DELETE", "CONNECT", "OPTIONS", "TRACE", "PATCH"
216   };
    int i;
218   for(i = 0; i < 9; i++) {
      if(strncmp(str, httpmeths[i], strlen(httpmeths[i])) == 0) {
220     return 1;
```

```
    }
  }
  return 0;
}

/*
 * tracee_t struct types & functions.
 */
struct tracee_t {
  pid_t id;
  char headers[1024];
  UT_hash_handle hh;
};

struct tracee_t *tracees = NULL;

void add_tracee(struct tracee_t *s) {
  s->headers[0] = '\0';
  HASH_ADD_INT(tracees, id, s);
}

struct tracee_t *find_tracee(int tracee_id) {
  struct tracee_t *t;
  HASH_FIND_INT(tracees, &tracee_id, t);
  return t;
}

void rmtracee(struct tracee_t *tracee) {
  HASH_DEL(tracees, tracee);
  free(tracee);
}

/*
 * rbinder main function.  Call with cmd line args:
 *
 *     $ ./rbinder /usr/bin/python server.py
 */
int main(int argc, char **argv) {
  pid_t child, cid;
  int status, fd, i;
  void *buf;
  size_t len;
  long syscall_number, syscall_return;
  long params[3];
  char *str;
  struct tracee_t *tracee;
  unsigned int open_socks[1024];

  for(i = 0; i < 1024; i++) {
    open_socks[i] = 0;
  }

  child = fork();

  // Start server within traced thread (just like a gdb inferior).
  if(child == 0) {
    ptrace(PTRACE_TRACEME, NULL, NULL, NULL);
```

```
278     /* Filters for the syscalls we want to trace */
        struct sock_filter filter[] = {
280       BPF_STMT(BPF_LD+BPF_W+BPF_ABS, offsetof(struct seccomp_data, nr)),
          BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, SYS_read, 0, 1),
282       BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_TRACE),
          BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, SYS_close, 0, 1),
284       BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_TRACE),
          BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, SYS_accept, 0, 1),
286       BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_TRACE),
          BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, SYS_sendto, 0, 1),
288       BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_TRACE),
          BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, SYS_clone, 0, 1),
290       BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_TRACE),
          BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_ALLOW),
292     };
        struct sock_fprog prog = {
294       .filter = filter,
          .len = (unsigned short) (sizeof(filter)/sizeof(filter[0])),
296     };
        /* To avoid the need for CAP_SYS_ADMIN */
298     if (prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0) == -1) {
          perror("prctl(PR_SET_NO_NEW_PRIVS)");
300       return 1;
        }
302     if (prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, &prog) == -1) {
          perror("prctl(PR_SET_SECCOMP)");
304       return 1;
        }
306     kill(getpid(), SIGSTOP);
        execv(argv[1], argv + 1);
308   }
      // Watch server syscalls for extracting incoming request tracing headers and
310   // injecting them into outgoing requests performed while request is being
      // serviced.
312   else {
        // Setup ptrace for tracing further children threads.
314     cid = waitpid(-1, &status, __WALL);
        if(ptrace(PTRACE_SETOPTIONS, cid, 0, PTRACE_O_TRACEEXEC|PTRACE_O_EXITKILL|\
316           PTRACE_O_TRACEVFORK|PTRACE_O_TRACECLONE|PTRACE_O_TRACEFORK|\
              PTRACE_O_TRACESECCOMP) < 0) {
318       perror("ptrace(PTRACE_SETOPTIONS)");
          exit(1);
320     }
        if(ptrace(PTRACE_CONT, cid, NULL, WSTOPSIG(status)) < 0) {
322       perror("ptrace(PTRACE_CONT)");
          exit(1);
324     }

326     while(1) {
          // Wait for tracees' activity.
328       cid = waitpid(-1, &status, __WALL);
          if(WIFEXITED(status)) {
330         tracee = find_tracee(cid);
            if(tracee) {
332           rmtracee(tracee);
            }
334         continue;
```

```
    }

    syscall_number = peekuser(cid, REG_SC_NUMBER);
    syscall_return = peekuser(cid, REG_SC_RETCODE);

    if (status >> 8 == (SIGTRAP | (PTRACE_EVENT_SECCOMP << 8))) {

    //// Extract headers from incoming request.
    if(SCREAD(syscall_number)) {
      peek_syscall_thrargs(cid, params);

      // Inject a new trap if this is a read on an open socket so we can
      // examine syscall results.
      if(open_socks[params[0]] == 1) {
        if(ptrace(PTRACE_SYSCALL, cid, NULL, WSTOPSIG(status)) < 0) {
          perror("ptrace(PTRACE_SYSCALL)");
          exit(1);
        }
        continue;
      }
    } // SYS_read
    else if(syscall_number == SYS_close) {
      peek_syscall_thrargs(cid, params);

      // Mark socket as not open if it was open and is being closed.
      if(open_socks[params[0]] == 1) {
        open_socks[params[0]] = 0;
      }
    } // SYS_close
    else if(syscall_number == SYS_accept) {
      if(ptrace(PTRACE_SYSCALL, cid, NULL, WSTOPSIG(status)) < 0) {
        perror("ptrace(PTRACE_SYSCALL)");
        exit(1);
      }
      continue;
    } // SYS_accept

    //// Inject headers into outgoing requests.
    else if(SCSENDTO(syscall_number)) {
      tracee = find_tracee(cid);
      peek_syscall_thrargs(cid, params);
      str = (char *)calloc(1, (params[ARG_SCRW_BUFFSIZE]+1) * sizeof(char));
      peekdata(cid, params[ARG_SCRW_BUFF], str, params[ARG_SCRW_BUFFSIZE]);

      // Check if HTTP request.
      if(is_http_request(str)) {
        if(!tracee) {
          perror("Tracee not found when injecting headers into outgoing request");
          exit(1);
        }

        int newstrsize = strlen(str) + strlen(tracee->headers);
        char newstr[newstrsize];
        inject_headers(str, tracee->headers, newstr, newstrsize);
        pokedata(cid, params[1], newstr, newstrsize);
      }
```

```
392           free(str);
            } // SYS_sendto
394         else if(syscall_number == SYS_clone) {
              if(ptrace(PTRACE_SYSCALL, cid, NULL, WSTOPSIG(status)) < 0) {
396             perror("ptrace(PTRACE_SYSCALL)");
                exit(1);
398           }
              continue;
400         } // SYS_clone

402         //} // PTRACE_EVENT_SECCOMP
            } else {
404           if(SCREAD(syscall_number)) {
                peek_syscall_thrargs(cid, params);
406
                if(syscall_return != -38) {
408               str = (char *)calloc(1, (params[ARG_SCRW_BUFFSIZE]+1) * sizeof(char));
                  peekdata(cid, params[ARG_SCRW_BUFF], str, params[ARG_SCRW_BUFFSIZE]);
410               if(is_http_request(str)) {
                    tracee = malloc(sizeof(struct tracee_t));
412                 tracee->id = cid;
                    add_tracee(tracee);
414                 extract_headers(str, tracee->headers);
                  }
416               free(str);
                  if(ptrace(PTRACE_CONT, cid, NULL, 0) < 0) {
418                 perror("ptrace(PTRACE_CONT)");
                    exit(1);
420               }
                  continue;
422             }
              } // SYS_read
424           else if(syscall_number == SYS_accept) {
                if(syscall_return != -38) {
426               if(syscall_return > 0) { // Note: 0 is a valid file descriptor.
                    open_socks[syscall_return] = 1;
428               }
                  if(ptrace(PTRACE_CONT, cid, NULL, 0) < 0) {
430                 perror("ptrace(PTRACE_CONT)");
                    exit(1);
432               }
                  continue;
434             }
              } // SYS_accept
436           else if(syscall_number == SYS_clone) {
                if(syscall_return != -38) {
438               if(syscall_return > 0) {
                    tracee = find_tracee(cid);
440                 if(tracee) {
                      struct tracee_t *cloned;
442                   cloned = malloc(sizeof(struct tracee_t));
                      cloned->id = syscall_return;
444                   add_tracee(cloned);
                      for(i = 0; i < 1024; i++) {
446                     cloned->headers[i] = tracee->headers[i];
                      }
448                 }
```

```
            }
            if(ptrace(PTRACE_CONT, cid, NULL, 0) < 0) {
              perror("ptrace(PTRACE_CONT)");
              exit(1);
            }
            continue;
          } else {
            if(ptrace(PTRACE_SYSCALL, cid, NULL, 0) < 0) {
              perror("ptrace(PTRACE_SYSCALL)");
              exit(1);
            }
            continue;
          }
        } // SYS_clone
      }

      if(ptrace(PTRACE_CONT, cid, NULL, WSTOPSIG(status)) < 0) {
        perror("ptrace(PTRACE_CONT)");
        exit(1);
      }
    }
  }

  return 0;
}
```