

Universidade do Minho
Escola de Engenharia

Francisco Nuno Teixeira Neves

Holistic Performance and Scalability Analysis for Large-Scale Distributed Systems

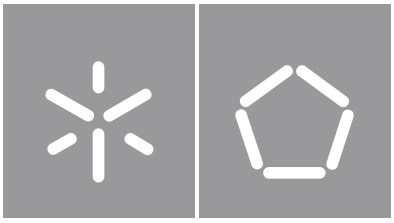
December 2021

Holistic Performance and Scalability Analysis for Large-Scale Distributed Systems

Francisco Neves

UMinho | 2021

FCT Fundação para a Ciência e a Tecnologia
POCH Programa Operacional Capital Humano
UNIÃO EUROPEIA Fundo Social Europeu



Universidade do Minho
Escola de Engenharia

Francisco Nuno Teixeira Neves

Holistic Performance and Scalability Analysis for Large-Scale Distributed Systems

Tese de Doutoramento
Programa Doutoral em Informática - MAP-i

Trabalho efetuado sob a orientação do
Professor Doutor José Orlando Pereira
e do
Doutor Ricardo Pereira Vilaça

December 2021

DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

Este é um trabalho académico que pode ser utilizado por terceiros desde que respeitadas as regras e boas práticas internacionalmente aceites, no que concerne aos direitos de autor e direitos conexos.

Assim, o presente trabalho pode ser utilizado nos termos previstos na licença abaixo indicada.

Caso o utilizador necessite de permissão para poder fazer um uso do trabalho em condições não previstas no licenciamento indicado, deverá contactar o autor, através do RepositóriUM da Universidade do Minho.

Licença concedida aos utilizadores deste trabalho



Atribuição-NãoComercial-CompartilhaIgual
CC BY-NC-SA

https://creativecommons.org/licenses/by-nc-sa/4.0/

Agradecimentos

Ao longo do percurso cujo fim é marcado por este documento, encontrei pessoas que são hoje amigos e companheiros de estrada. Sem eles, este trabalho nunca seria possível. Por isso, quero deixar o meu agradecimento a todos os que contribuíram para o sucesso deste trabalho.

Em especial, aos meus orientadores. Ao Prof. José Pereira, pela sua dedicação, profissionalismo e arte de pensar. Ao Ricardo Vilaça, por estar presente nas dificuldades que surgiram e por ter mantido a confiança no meu trabalho. Todos os momentos de discussão e os conselhos de ambos determinaram o sucesso deste trabalho. Aos meus amigos e companheiros de caminhada: Ana Alonso, Catarina Leones, Cláudia Brito, Daniel Cruz, Filipe Campos, Fábio Coelho, Francisco Cruz, Francisco Maia, João Paulo, Miguel Matos, Nuno Machado, Paula Rodrigues, Ricardo Gonçalves, Ricardo Macedo, Ricardo Vilaça, Rui Gonçalves, Rui Ribeiro, Rogério Pontes, Tânia Araújo e corpo docente; obrigado por me terem proporcionado os melhores momentos destes últimos anos. Diretamente ou indiretamente, todos vocês contribuíram para o sucesso deste trabalho. Ao Prof. Rui Oliveira, pelas conversas e aventuras ao longo desta caminhada e por me ter feito apaixonar por Sistemas Distribuídos, ainda em tempos de licenciatura. Aos meus amigos e Hackathonners: André Pinho, Bruno Ferraz, Daniel Tavares, Diogo Couto, Gabriela Pinto, Hugo Gonçalves, Jaime Leite, José Silva, Marco Couto, Miguel Costa e Miguel Pinto, por me fazerem chegar até ao fim com motivação e persistência.

À minha família, que sempre garantiu as condições necessárias para atingir os meus objetivos. Sem eles, nada disto seria possível. O meu agradecimento especial à Sónia Mendes, que trouxe magia à minha vida e momentos felizes a dois e foi, sem dúvida, um suporte fundamental nos últimos passos deste percurso.

Ao INESC TEC, pelo apoio e financiamento parcial deste doutoramento. À Fundação para a Ciência e a Tecnologia, pelo apoio financeiro durante os últimos três anos através da bolsa de doutoramento SFRH/BD/129771/2017. Por fim, ao Dep. de Informática da Universidade do Minho e ao HASLab pelas condições necessárias ao desenvolvimento deste trabalho.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

University of Minho, 22th December, 2021

Francisco Nuno Teixeira Neves

Francisco Nuno Teixeira Neves

Holistic performance and scalability analysis for large scale distributed systems

Internet services play a critical role day-by-day in personal and professional lives. Monitoring these services is of paramount importance for quick detection and solving anomalies in order to reduce downtime and persistence of erroneous behavior, otherwise they end up causing serious negative impact.

The availability of detailed data about the behavior of the system as a whole is crucial for analysis and troubleshooting anomalies. Although most software components individually provide outputs that developers foresee useful to diagnose performance and behavioral issues, they are not correlated in a causally-consistent fashion. Collecting cross-component outputs, in turn, requires substantial effort in instrumenting the source code of each software component for enabling the generation of causally-correlated outputs, which is not always viable given the heterogeneity architecture's components and unaccessible portions of code. This renders a trade-off between black-box outputs and the detail they provide.

In this dissertation, we explore this trade-off in order to enable effective analysis and troubleshooting of complex distributed systems without requiring application-specific knowledge. This was achieved by operating at the common layer of all software components, through which all interactions are performed. Specifically, we focused on exploring the operating system's kernel as the layer of observation to extract information about the behavior of the system. Our approach relies on events triggered by distributed processes within applicational components and their well-known causal relations to aid diagnosing performance and behavior anomalies. Using three case studies, we validated that our approach represents a step forward towards observable complex distributed architectures without application-specific knowledge.

Keywords: performance analysis; black-box tracing; distributed systems.

Análise holística de desempenho e escalabilidade para sistemas distribuídos de larga escala

Os serviços na Internet desempenham um papel fundamental no dia-a-dia de milhões de pessoas e empresas. A monitorização destes serviços é por isso essencial para a deteção e resolução de anomalias de modo a reduzir indisponibilidade e a persistência de erros.

Para resolver anomalias de forma eficaz, é necessário obter dados detalhados sobre o comportamento do sistema como um todo. Embora a maioria das componentes de software forneça isoladamente dados que os programadores consideram úteis para diagnosticar problemas de desempenho e comportamento, estes não incluem informação suficiente para obter uma visão causalmente coerente do comportamento do sistema como um todo. Para que estes dados forneçam uma visão coerente do comportamento do sistema, é exigido um esforço substancial na instrumentação do código-fonte de cada componente. No entanto, isto nem sempre é viável, dada a natureza heterogénea da arquitetura das componentes e até mesmo o facto de terem porções de código-fonte inacessíveis. Assim, existe um compromisso entre os dados gerados isoladamente pelas componentes, cujo comportamento em detalhe é desconhecido, e o detalhe que estes fornecem sobre o comportamento do sistema como um todo.

Nesta dissertação, explora-se esse compromisso de modo a permitir a análise e resolução de problemas eficazes de sistemas distribuídos complexos, sem necessitar de conhecimento específico do funcionamento interno das suas componentes de software. Para atingir este objetivo, é apresentada uma abordagem que estebelece o núcleo do sistema operativo como o nível de observação para extrair informação sobre o comportamento do sistema. Esta abordagem assenta sobre eventos despoletados por processos distribuídos pertencentes às componentes de software aplicacionais, bem como nas suas relações causais previamente conhecidas. Utilizando três casos de estudo, conclui-se que a abordagem proposta nesta dissertação representa um passo em frente na observação de sistemas distribuídos complexos sem conhecimento específico das suas componentes aplicacionais.

Palavras-chave: análise de desempenho; black-box tracing; sistemas distribuídos.

Contents

1 Introduction **1**

1.1 Problem Statement . . . 3
1.2 Objectives . . . 4
1.3 Contributions . . . 5
1.4 Publications . . . 6
1.5 Document Structure . . . 6

2 Background **8**

2.1 Monitoring Distributed Systems . . . 8
2.1.1 Monitoring at the Infrastructure Level . . . 10
2.1.2 Monitoring at the Application Level . . . 10
2.1.3 Monitoring at the Kernel Level . . . 11
2.2 Log-based Analysis . . . 13
2.2.1 The Need for Causality . . . 14
2.3 Distributed Tracing . . . 17
2.4 Summary . . . 18

3 Monitoring Distributed Interactions **20**

3.1 Monitoring Network Traffic . . . 20
3.2 Building the Communication Graph . . . 21
3.3 Prototype Implementation . . . 22
3.3.1 Agent . . . 23
3.3.2 Data Processor . . . 23
3.4 Evaluation . . . 24
3.4.1 Extended Berkeley Packet Filter (eBPF) Scalability . . . 24
3.4.2 Overhead . . . 25

3.5 Compatibility at Scale . . . 27
3.5.1 Deployment . . . 27
3.5.2 Results . . . 29
3.6 Case Study . . . 30
3.6.1 Application scenario . . . 30
3.6.2 Limitations of resource monitoring . . . 31
3.6.3 Using data exchange monitoring . . . 32
3.6.4 Automatic placement . . . 33
3.6.5 Manual optimization . . . 36
3.7 Summary . . . 39

4 Causal Log-Based Analysis **40**
4.1 Design Requirements . . . 40
4.2 Architecture . . . 41
4.3 Happens-Before Constraint Model . . . 43
4.4 Implementation . . . 46
4.5 Case Study: Apache Zookeeper . . . 46
4.5.1 Establishing Causal Relationships . . . 47
4.5.2 Syscall Tracing Overhead . . . 50
4.5.3 Scalability of Constraint Solving . . . 51
4.6 Summary . . . 51

5 Efficient Causal Log Querying **53**
5.1 Overview . . . 54
5.2 Causal Graph Generation . . . 56
5.2.1 Intra-Process HB Relationships . . . 58
5.2.2 Inter-Process HB Relationships . . . 60
5.3 Efficient Causal Graph Querying . . . 61
5.4 Case Study: TrainTicket . . . 63
5.4.1 Debugging F13 with Elastic Stack . . . 64
5.4.2 Debugging F13 with Horus . . . 66
5.5 Performance Evaluation . . . 70
5.5.1 Event Processing . . . 71
5.5.2 Logical Time Assignment . . . 72
5.5.3 Causal Graph Querying . . . 73

5.6 Summary 75

6 Conclusion **77**

6.1 Future Work 78

Bibliography **79**

Acronyms

API Application Programming Interface.

CPU Central Processing Unit.

eBPF Extended Berkeley Packet Filter.

GCE Google Compute Engine.

HB Happens-Before.

HTTP Hypertext Transfer Protocol.

IP Internet Protocol.

IT Information Technology.

JSON JavaScript Object Notation.

MaxSMT Maximum Satisfiability Modulo Theories.

NoSQL NoSQL.

NUMA Non-Uniform Memory Access.

RAM Random Access Memory.

SMT Satisfiability Modulo Theories.

SQL Structured Query Language.

TCP Transport Communication Protocol.

List of Figures

1 Complete trace resulting from merging the partial outputs of strace on three distinct nodes. [ni-pid] denotes the global identifier of a process pid running on node i. 14

2 Trace resulting from causally reordering the syscalls in Figure 1. [ni-pid] denotes the global identifier of a process pid running on node i. 15

3 Space-time diagram of the trace in Figure 2. 16

4 Architecture of prototype monitoring system. It comprises two main components: Agent and Data Processor. The Agent leverages kernel probes to collect and filter data in kernel space. The collected data is then fetched by the frontend program and sent to the Data Processor component, which is responsible for updating the graph representation accordingly. 22

5 Evolution of loss rate with increasing throughput. 25

6 Throughput and Central Processing Unit (CPU) overhead. 26

7 The obtained architecture of microservice benchmark applications. Each box represents a process running in the deployment, as well as edges represent the amount of exchanged data. Without prior knowledge regarding the organization of each application, our black-box results match the expected application architecture according to their documentation. 28

8 Inter-host traffic with default placement. 32

9 Inter-process traffic with default placement. 33

10 Mapping of execution over hardware nodes with default placement for Q1 (Scan). 34

11 Mapping of execution over hardware nodes with default placement for Q2 (Shuffle). 34

12 Inter-host traffic after manual optimization. 36

13 Inter-process traffic after manual optimization. 37

14 Mapping of execution over hardware nodes after manual placement and configuration for Q1 (Scan). 37
15 Mapping of execution over hardware nodes after manual placement and configuration for Q2 (Shuffle). 38

16 Falcon's pipeline architecture, which comprises modules for trace processing, model generation, and visualization. 42
17 Space-time diagram of two Zookeeper servers connecting to each other and performing a leader election. 48
18 Performance impact on Zookeeper execution caused by tracing with strace. . 50
19 Time spent on solving the generated models for both levels of debugging: INFO and DEBUG. 51

20 Horus architecture and event flow. 55
21 Example causal graph involving three distributed processes. 62
22 The set of log records aggregated by Elastic Stack for the failed payment request. Records are ordered by the timestamp assigned at the moment they were collected. To ease readability, we label each record with the service in which it occurred and a local counter. 65
23 Refinement queries, written in CypherQL, used to debug the F13 failure, along with a snippet of the ShiViz diagram generated for the causal path produced by Horus after processing the queries. 68
24 Horus throughput as the number of clients increases. The incoming event rate reveals the amount of events produced by running clients. 71
25 Comparison between the execution time of Falcon and Horus to assign logical clocks on different graph sizes. 72
26 Comparison between the execution time of the shortest path algorithm and Horus to answer query Q1, for different graph sizes. 74
27 Evolution of the execution time of obtaining the causal graph between two events using traversal-based query and the logical-time-based approach implemented in Horus. 75

List of Tables

1 Resource usage in nodes (CPU in seconds, others in KiB). 31
2 Network traffic with optimizations. 35
3 Manually optimized placement and configuration. 38

4 Number of events per type present in the causal graph under analysis. LOG events are produced by the log4j logging library, while the rest is captured by Horus' kernel-level tracer. 67

Chapter 1

Introduction

Internet services are part of daily personal and professional lives with a large number of relevant examples: social networks, cloud-based collaborative office suites, source-code repositories, continuous integration and delivery pipelines, e-mail services, large-scale data analytics, machine-learning systems, and the public cloud. Millions of users and organizations rely on these services to store their private conversations and documents and even start and run their businesses.

As these services play a critical role in society, any anomaly may cause problems to customers' lives and organizations' businesses. And anomalies are not as scarce as one would like. In 2014, Amazon had a software glitch that changed the price of all products to one penny, leading its selling partners to lose tens of thousands of pounds [13]. Facebook, in 2018, saw users' privacy broken due to a software bug that exposed users' private photos to application developers without permissions [16]. British Airways canceled all flights due to Information Technology (IT) outage, both in 2018 and 2019 [17]. Cloudbleed is the name of a data breach that, in 2017, exposed sensitive data of customers of millions of websites due to a software bug in Cloudflare [18]. Starbucks also lost millions in sales due to a software problem that allowed customers to order free drinks [14]. Apart from these worldwide companies, TechRepublic analyzed 606 software failures from 314 companies and revealed that these affected 3.6 billion people and caused $1.7 trillion in financial losses [15].

Even though most teams adopt high-end testing and validation techniques during the development phase, the truth is that these techniques only minimize the likelihood of errors. Unexpected behaviors will eventually surface while in production in the form of errors and/or performance bottlenecks. To address these, monitoring is of paramount importance to quickly detect and solve anomalies to reduce downtime and the persistence of erroneous behavior.

Otherwise, they end up causing a negative impact.

The architecture of applications is increasing in size and complexity. First, the migration from monolithic systems towards loosely-coupled microservices architectures is breaking one single logical unit into a vast set of independently maintainable and deployable subsystems integrated via well-established Application Programming Interface (API)s [35]. Consequently, the amount of software interdependencies within the systems architecture increases exponentially. Second, part of the integrating software components are black-box open-source, off-the-shelf, and third-party software for distinct purposes of storage, message queuing, stream processing, among others. The source code of these third-party components is either not always available or might exhibit a degree of complexity that turns its analysis into a time-consuming and unpractical task. Alongside, the deployment of software over lightweight virtualized environments (i.e., containers) is also a common practice that introduces additional vertical complexity. The combination of the vast set of custom and third-party software over virtualization layers, therefore, renders modern architectures as complex multi-layer dependency graphs with very distinct specifications and requirements.

The detailed knowledge about the system's behavior as a whole is crucial for analysis and troubleshooting anomalies. System behavior is the system's architecture and organization, as well as the interdependencies and interactions performed by their software components throughout transactions or request executions. The architecture and organization of the system give a big picture of the system's deployment and how software components interconnect. It supports the detection, analysis, and resolution of performance issues from deployments that do not consider the degree of affinity of interdependent software components. The performance issues that arise from the composition of systems mostly remain undetected using traditional monitoring tools [30]. In turn, the detailed view of software component interactions allows inspecting exchanged messages and cross-component call stacks under the context of each request. This level of detail enables effective bug detection and fixing as well as the pinpointing of the root cause of performance anomalies (e.g., high latency) in end-to-end request execution flows [26, 46, 36, 37, 31, 54].

Metrics, logs, and distributed traces are the de facto standard data types that usually capture the behavior of a system.

Metrics are numeric values measured periodically, typically used to monitor the system's health. They often depict the evolution of the status of computational resources (e.g., CPU, Memory and I/O devices) or individual services, from web servers (e.g., Hypertext Transfer Protocol (HTTP) requests per time unit) to stream processing frameworks (e.g., jobs per time

unit). Log messages are unstructured or structured text strings containing abundant contextual information. Judiciously added by software developers to source code, they cater for recording information about the state of the application that developers believe to be the most accessible resource for debugging unexpected behavior [58, 56, **?**]. As they are present in most systems out of the box, logs are the first resource that developers and engineers rely on to pinpoint the root cause of wrong behaviors [60, 28, 19, 57, 42]. Finally, distributed traces are causal sets of distributed events that encode the end-to-end executions through every software component of a distributed system [26]. At strategic points in source code, besides the contextual data required to causally-correlate events, these events also record arbitrary data that developers foresee useful to diagnose unexpected behavior in future analysis.

The combination of these three data types provides the ability to observe the evolution of the system execution and state (i.e., observability). Formally defined in control theory, observability is a measure of the ability to infer the internal state of the system from its external outputs [32].

Under the scope of systems and software, a perfectly observable system provides external outputs (i.e., metrics, logs, and traces) that allow to identifying the reason for any behavior to have occurred. More specifically, these outputs allow developers to accurately pinpoint the precise root cause for a given anomaly, navigating from effect to its cause.

The possibility of having a clear and detailed view of the system's behavior that supports the analysis and troubleshooting of systems is moving efforts, both in academia and industry, to research and develop new techniques towards the dream property — observability — of modern distributed software systems.

1.1 Problem Statement

Operating systems, container managers, and individual software components already expose a set of metrics and logs out of the box.

Metrics disclose the performance and health of infrastructures, virtualized environments, and individual services in isolation. Therefore, they are unsuitable for analysis and troubleshooting performance issues that arise from the composition of software components.

Log messages contain information about the functional behavior of each software component generated as requests arrive at the system. Current tools offer centralization and full-text search to ease the system's behavior analysis within request executions. However, they lack a causally-coherent view of the executions since they centralize and order logs by physical timestamps, which is unsuitable to reason about distributed executions [33].

Distributed traces come from custom source code instrumentation of software components in order to collect contextual information at the system level. In contrast to logs, distributed traces derive from a causal ordering of events of software components. As such, they provide a detailed and causally coherent cross-component view of the system's behavior during the execution of each request. However, architectures comprise open-source, off-the-shelf, and third-party software, which source code is too complex to study and instrument in a timely and error-free manner. This scenario thus renders distributed traces challenging to collect.

In short, on the one hand, we have metrics and logs already exposed by most software components but fall short in being suitable for comprehensive system analysis. On the other hand, we have distributed traces that provide a coherent view of the system's behavior during request executions but are impractical or even impossible to collect from distributed systems that integrate black-box software components.

In this thesis, we thus formulate and address the following research question: How can one obtain useful detailed information about the performance and execution behavior of large-scale and complex distributed systems while treating its components as black boxes?

1.2 Objectives

To address the research question, we focus on exploring a trade-off between black-box and detail that enables to answer queries on the behavior of the system during analysis and debugging tasks.

In this work, we consider as black-box the approaches that operate without source code instrumentation nor application knowledge. Under the application layer, there are distinct layers that approaches could operate as black-box. However, a single operation performed at the application layer is translated to several sub-operations as we go down through the call stack of the lower layers. These operations may result in large amounts of collected data that, when correctly displayed, can unveil hidden behavioral details of the system under analysis.

The main objectives of this work are as follows:

First, identify the appropriate operating layers and collect relevant data for distributed systems analysis. Second, collect and process large amounts of generated data, and construct a solid base for conducting systems analysis.

1.3 Contributions

This thesis introduces three contributions. The first contribution arises from the need to extract information about the architecture and organization of distributed systems to tackle performance bottlenecks associated with software composition. Current monitoring tools fall in assembling an overview of the architecture and organization of the system, as they provide metrics in an isolated fashion for the infrastructure, virtualized, and applicational layers. We achieve that with an efficient black-box monitoring approach for detecting and building a weighted communication graph of collaborating processes in a distributed system that answers queries for a variety of purposes, including performance improvement.

As a second contribution, we address the lack of causality in logs which is key to effectively debugging unexpected behavior. Distributed systems output several log files scattered over dozens of machines. Concatenating them to perform log-based analysis misleads conclusions or even becomes inconclusive at all, enduring the existence of errors. The widely adopted log management tools fail at combining logs in a causally-coherently fashion. We developed Falcon, a practical tool for log-based analysis of distributed executions, as it combines logs generated within a distributed system into a single causally coherent log file.

The third contribution arises from two additional needs associated with complex and heterogeneous distributed architectures. In particular, the sheer amount of log messages the complex systems generate introduce two concerns to deal with: scalability and refinement. The former is related to the challenge in causally combining large amounts of generated logs. The latter concerns the difficulties in debugging unexpected behavior with a singular log file, given the lack of appropriate tools to narrow the search space while looking for the root cause of wrong behaviors. To address these challenges, we developed a software prototype named Horus. It is an online log management tool in which log events are efficiently encoded as a directed acyclic graph and stored in a graph database, thus allowing the usage of rich query languages to reason about runtime behavior.

The source code of Falcon[1] and Horus[2] software prototypes are publicly available.

[1] https://github.com/fntneves/falcon
[2] https://github.com/DistributedSystemsAnalysis/horus

1.4 Publications

The work presented in this thesis was published and presented in distinct international conferences:

- Francisco Neves, Nuno Machado and José Pereira. Falcon: A practical log-based analysis tool for distributed systems. 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2018. (Tool paper)
- Francisco Neves, Ricardo Vilaça, and José Pereira. Black-box inter-application traffic monitoring for adaptive container placement. Proceedings of the 35th Annual ACM Symposium on Applied Computing, 2020.
- Francisco Neves, Ricardo Vilaça, and José Pereira. Detailed black-box monitoring of distributed systems. SIGAPP Applied Computing Review 21, Volume 1, March 2021.
- Francisco Neves, Nuno Machado and José Pereira. Horus: Non-intrusive Causal Analysis of Distributed Systems Logs. 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2021.

Other collaborations:

- Francisco Maia, João Paulo, Fábio Coelho, Francisco Neves, José Pereira and Rui Oliveira. DDFlasks : IFIP International Conference on Distributed Applications and Interoperable Systems. Springer, Cham, 2017.
- Nuno Machado, Francisco Maia, Francisco Neves, Fábio Coelho and José Pereira. Minha: Large-Scale Distributed Systems Testing Made Practical. 23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
- Tânia Esteves, Francisco Neves, Rui Oliveira, and João Paulo. CAT: content-aware tracing and analysis for distributed systems. In Proceedings of the 22nd International Middleware Conference (Middleware), 2021. (To appear)

1.5 Document Structure

The remainder of this document is structured as follows:

Chapter 2 includes some context around modern architectures and current approaches to extract and analyze systems behavior for purposes of debugging anomalies.

Chapter 3 introduces and validates an approach to achieve efficient monitoring of distributed processes' interactions without requiring application-specific knowledge.

Chapter 4 presents and demonstrates an approach for turning log-based analysis of distributed systems practical and effective. It leverages the operating system's kernel events and the well-known causal relations to construct a causally coherent view of log entries generated by several application software components.

Chapter 5 describes a system that enables the refinement of distributed systems logs in a causal and scalable fashion. We validate this system using an open-source application with 40+ microservices and demonstrate that it surpasses one of the popular log management systems in pinpointing the root cause of an anomaly in a distributed execution.

Finally, Chapter 6 concludes this dissertation, discusses its achievements, and addresses future research paths.

Chapter 2

Background

The main goal of this dissertation is to propose techniques to enable effective analysis and troubleshooting of complex distributed systems without instrumenting source code of their components. More specifically, it aims at extracting organizational and behavioral information of the system from outputs exposed out of the box by software components.

In this chapter, we provide some context around modern architectures and current approaches for system behavior analysis. First, we start by giving an overview of the organization of modern systems architectures and the importance of monitoring the interactions among software components. Regarding system behavior analysis, we proceed by introducing the usage of logs as the primary resource to inspect anomalies. Additionally, we discuss the limitations of carrying out a naïve log-based analysis and compose an example that reveals the importance of tracking causality in logs. Finally, we introduce distributed tracing techniques — the most accurate approach used for analysing systems behavior — and the tradeoffs on implementing it in complex distributed systems.

2.1 Monitoring Distributed Systems

The adoption of microservices over traditional multi-tiered and monolithic architectures split monolith software applications to dozens of loosely-coupled and independently maintainable software services or components [35]. This architectural style enables smaller teams to independently manage, develop and deploy loosely-coupled software services. As there is no single shared code base between teams but APIs, each team is able to carry out faster delivery for each software service, which translates in fast delivery of new versions of the application. Unfortunately, this architectural style comes with great cost in respect to size, heterogeneity and

complexity of systems. First, as it promotes the development of loosely-coupled services within the scope of an application in detriment of single logic units, systems increase in size. Second, teams work independently on each software service and thus each one is able to choose the most suitable ecosystem, including programming language, frameworks, libraries and third-party systems, which contributes to heterogeneity. Finally, the resulting services communicate with each other to perform requests coming to the application. The inherent interdependencies to services cooperation, which did not exist in monolithic architectures, brings unprecedented complexity and challenges to analyse distributed applications.

Additionally, distributed software components are often deployed using containers and managed with container pods and orchestrators. Containers are a lightweight virtualization technology in which the operating system kernel and many resources can be shared between co-located components. Container pods formalize this by making it easy to share specific resources and manage together tightly coupled components. Orchestrators, such as Kubernetes, take care of managing a pool of resources and placing component instances in them to compose complete services and applications.

Summing up, the composition of significant amount of software components deployed over virtualization layers render modern systems architectures as a multi-layered software dependency graph with unprecedented complexity.

An interdependency between two software components exists if they interact by transferring data to each other, being it through shared memory, pipes or network channels. Observing interactions and dependencies provide useful insights of the behavior of the system that permit to answer the question “Which component(s) affect component X?” and to solve performance issues specially originated by misplacement of processes [41]. In fact, this has been exploited in practice at the virtual machine level in cloud-based environments [27, 47] and at the process-level in multi-core Non-Uniform Memory Access (NUMA) architectures [44].

In distributed architectures, the amount of inter-component network traffic determines the degree of dependency of two software components, which is helpful for optimizing the placement of the corresponding containers. For instance, an application with two components with great dependency degree would benefit of having both components closely placed. Otherwise, these components could significantly increase network bandwidth usage and end up, in the worst case, degrading the performance of the whole application. Inter-component network traffic is thus a key issue for performance analysis of an application.

Monitoring network traffic between software components is not trivial though. There are plenty of capable monitoring tools ,from both academia and industry, operating from the infras-

tructure to the application level but their limitations make them unsuitable or even impractical for this purpose.

2.1.1 Monitoring at the Infrastructure Level

Infrastructure monitoring systems are the most common systems for monitoring health of systems. The focus is on gathering metrics about the health of infrastructure, specially CPU usage, memory usage, network and disk I/O metrics, among others. Popular examples of infrastructure monitoring systems are Nagios [51], Ganglia [38], MonALISA [40] and the popular Prometheus [49].

In *nix operating systems, these monitoring systems operate by enabling an agent that periodically reads and parses files from the virtual /proc filesystem. This virtual filesystem exposes the current state of several internal kernel structures, regarding the state of CPU, memory and I/O, which are relevant for the global health of the system and resource consumption of each process. After collecting data, the agent pushes it to a central database, from which dashboards fetch the data and render graphs. The most recent monitoring systems additionally provide powerful query languages to query, aggregate and filter the collected data and to build custom monitoring dashboards.

Although operating systems already expose internal kernel structures for monitoring the health of the instance under monitoring, they do not have any correlation with remote processes or other instances. Consider, for example, the total inbound network metric that represents the amount of traffic received by the instance under monitoring. These monitoring systems would allow detecting when it is a bottleneck and possibly the local network interface where the traffic is coming to, but would fall short in specifying which and how remote processes contribute to such measure. As such, these monitoring systems are unsuitable for monitoring network traffic between application components.

2.1.2 Monitoring at the Application Level

In contrast, monitoring at the application level enables a comprehensive view of interdependencies of application components and performance of the application as a whole [50, 52, 11]. In fact, one of the most relevant features provided by application monitoring tools is the ability to record the execution path of requests. The resulting fine-grained data, often collected at the source code level, is key for effectively pinpointing the root cause of performance bottlenecks in request workflows [37, 31].

The act of recording the cross-component execution path of requests is often characterized as distributed tracing and it is achieved using source code instrumentation. Briefly, it consists of propagating an identifier throughout components involved on a single request that is later used to correlate collected metrics [26, 37, 36]. Later in this chapter, we will explore these techniques in detail.

Application monitoring tools attempt to make collect necessary data through the use of auto-instrumentation agents for different languages, libraries and frameworks. That is, following specific steps for installing and configuring the corresponding instrumentation agents, the dynamic instrumentation of commonly used functions and classes for network-based communication is enabled and the monitoring system is able to record the execution flow of requests.

Developers actual may leverage auto-instrumentation to obtain a basic overview of the behavior of the application without considerable effort, as long as the application is built atop common libraries and frameworks. The same is not true for the cases where developers need a detailed overview or the application resorts to custom libraries and third-party software. In these cases, developers must significantly spend time and effort on manually instrumenting the source code, which might be unachievable in the case of applications integrating black-box software components.

Therefore, although application monitoring tools are suitable to reason about distributed interactions and performance of applications, when compared to infrastructure monitoring tools, there is a trade-off between instrumentation effort and the detail of collected data.

2.1.3 Monitoring at the Kernel Level

The challenges found when monitoring applications' interdependencies and performance associated to instrumentation techniques led to use alternative black-box approaches that operate at the common layer of all software components and virtualization containers: the operating system. In fact, these approaches are desirable specially to cloud providers, as hosted applications are black-boxes in their perspective.

Intercepting and processing all network traffic using network sniffers (e.g., pcap-based tools) is a classical approach. Attached to network interfaces, sniffers scan and parse all incoming and outgoing traffic whether it is related or not to the application. Unfortunately, the procedure of intercepting the network-related execution path, scanning all network packets, parsing several network-layer headers and pushing data to the user space incur non-negligible overhead which can negatively impact the application. Also, this approach is not trivially able to correlate network packets to processes, a key feature for identifying interactions between processes of distributed

systems.

When a process of a given software component sends a message to another process, the execution path crosses functions called in the application's source code, network libraries and, in the end, the corresponding system call of the host's kernel where the data is being sent from. The interactions of an application essentially result in executing network-related system calls on the host's operating system.

The set of system calls (or simply, syscalls), form the public API of operating systems to the applications running in user space. Typically, black-box monitoring approaches resort to tracing tools (e.g., strace) to operate at this level, since the observation of system calls executed by software components unveils details on the behavior of each component useful for diagnosing anomalies [55, 24, 21, 23, 45]. Furthermore, as demonstrated by [48], with adequate assumptions, system calls enable the extraction and analysis the performance of entire request execution paths without application-level knowledge.

Besides monitoring system calls, tracing kernel routines is also a technique for observing interactions. Under the hood, the execution of system calls result in the execution of a set of kernel routines.

Kernel tracing with eBPF The efficient interception of the execution path of system calls in Linux is currently performed through the use of Extended Berkeley Packet Filter (eBPF) [4], an increasingly popular technology for executing programs passed from user space to kernel. The fundamental idea of eBPF is to attach small custom programs to the available kernel tracepoints and to the entry and exit points of kernel routines. The attached programs are compiled and then executed within a virtual machine in an event-driven fashion, namely at every moment a specific kernel routine is called or returns or when a tracepoint event is dispatched by the kernel. These programs are also capable of performing some sort of filtering, keeping state in data structures (e.g., hash maps, arrays, etc.) throughout probe invocation, and send events from kernel to user space using ring buffers that are then collected by a front-end program.

When eBPF programs are installed, they become part of the execution path of instrumented kernel routines. Aiming at avoiding kernel panics, data corruption, unbounded overhead and other dangerous consequences that compromise correctness — as in previous tools like SystemTap [43] —, eBPF imposes several restrictions on what can be done inside the kernel. For instance, loops are not allowed, reading fields of kernel structures requires previously copying them and the stack size is limited to 512 bytes. Such restrictive technology demands planning variables declaration, definition and instantiation of data structures.

This technology is used by WeaveWorks Scope [53], for instance, an open-source tool that resorts to kernel-based tracing to automatically detect processes, virtualized containers, hosts and their established connections as well. Although this tool is able to provide an overview of the system's architecture, from the virtualization level to the processes level, it does not offer any sort of monitoring concerning the amount of data transferred through each connection.

2.2 Log-based Analysis

Developers of distributed systems cater for recording runtime behavior by judiciously adding log statements to source code [**?**, 56]. The number of log statements needed and the detail of the information collected depend on the complexity of the code. In systems that deal with concurrency and faults, such as fault-tolerant consensus protocols, the resulting effort is substantial. However, when an unexpected outcome is noticed, programmers resort to log files as the primary source of information to debug and fix the problem [35].

Diagnosing an anomaly in a distributed execution via log inspection can be a challenging task. Applications tend to produce logs with valuable runtime information that suffices to troubleshoot most failures, even at the default verbosity [**?**]. Still, the inherent heterogeneity of modern distributed architectures lead applications to generate log files spread across several nodes containing messages with very different structures that concern different requests and threads interleaved with each other.

These challenges motivated the adoption of log management tools that aim at easing the analysis based on logs, by means of collecting, parsing and ordering logs shipped by multiple sources. Notable examples of log management systems include Elastic Stack [5], Splunk [12], Logentries [7] and Logzio [9]. Furthermore, their features include powerful text search, grouping and filtering operations that allow developers to narrow down the search space while finding the root cause of an incorrect behavior.

Log management systems establish an ordering of logs based on the timestamps attached. These timestamps are assigned by logging libraries typically based on physical clock (e.g., unix time). Nonetheless, [33] has shown that physical timestamps do not guarantee a causally coherent ordering of events. Interleaved statements or execution on different nodes leads to a wide set of possible event execution flows and intermediate states that have to be considered. Therefore, the timestamp-based ordering of events discloses one possible event execution flow that might not correspond to the actual execution, thus compromising the effectiveness of log-based analysis of distributed executions. This is where causality comes in.

```
    # Format: [pid], syscall(parameters) = return_value
3.1 [n3-894] read(n2, "Go go go", 1023) = 8
3.2 [n3-894] write(n1, "Wait guys", 1023) = 9
3.3 [n3-894] write(n2, "Wait guys", 1023) = 9
3.4 [n3-894] read(n2, "Ok", 1023) = 2
2.1 [n2-782] write(n3, "Go go go", 1023) = 8
2.2 [n2-782] write(n1, "Go go go", 1023) = 8
2.3 [n2-782] read(n3, "Wait guys", 1023) = 9
2.4 [n2-782] write(n1, "Ok", 1023) = 2
2.5 [n2-782] write(n3, "Ok", 1023) = 2
1.1 [n1-675] read(n3, "Wait guys", 1023) = 9
1.2 [n1-675] read(n2, "Ok", 1023) = 2
1.3 [n1-675] read(n2, "Go go go", 1023) = 8
```

Figure 1: Complete trace resulting from merging the partial outputs of strace on three distinct nodes. [ni-pid] denotes the global identifier of a process pid running on node i.

Other tools resort to log parsing for performance analysis[59, 42] and anomaly detection through machine learning[28, 57]. Briefly, Iprof[59] is a profiling request tool that resorts to static analysis to extract information and correlate log entries of each request. LRTrace[42] proposes a keyed structure of unstructured logs through pattern matching to correlate resource usage metrics with logs of each task for performance analysis. Finally, Log3C[28] is an approach that samples, clusters and matches log sequences for identifying problems by correlating clusters of log sequences with relevant key performance indicators. All these approaches rely on the order the logs appear, compromising the analysis of the system's behavior with causality in mind, which importance is explained in the next section.

2.2.1 The Need for Causality

As an example of the importance of causality, consider a simple scenario with three participants of an online multiplayer game, represented by three processes on distinct machines connected through Transport Communication Protocol (TCP) sockets. In this scenario, player 2 (corresponding to process **n2-782**) tells his teammates to advance with a message "Go go go". Player 3 (process **n3-894**) disagrees with the suggestion and asks the team to wait by replying "Wait guys". Player 2 consents and writes "Ok". Player 1 (process **n1-675**), in turn, simply receives the instructions given by the other players. The result of this interaction is that player 1 advanced alone, causing the team to later lose the game. Why did player 1 act against the instructions given by the rest of the team?

In Figure 1, we present a possible log obtained by merging the output of the instructions of

```
    # Format: [pid], syscall(parameters) = return_value
2.1 [n2-782] write(n3, "Go go go", 1023) = 8
2.2 [n2-782] write(n1, "Go go go", 1023) = 8
3.1 [n3-894] read(n2, "Go go go", 1023) = 8
3.2 [n3-894] write(n1, "Wait guys", 1023) = 9
3.3 [n3-894] write(n2, "Wait guys", 1023) = 9
1.1 [n1-675] read(n3, "Wait guys", 1023) = 9
2.3 [n2-782] read(n3, "Wait guys", 1023) = 9
2.4 [n2-782] write(n1, "Ok", 1023) = 2
2.5 [n2-782] write(n3, "Ok", 1023) = 2
1.2 [n1-675] read(n2, "Ok", 1023) = 2
3.4 [n3-894] read(n2, "Ok", 1023) = 2
1.3 [n1-675] read(n2, "Go go go", 1023) = 8
```

Figure 2: Trace resulting from causally reordering the syscalls in Figure 1. [ni-pid] denotes the global identifier of a process pid running on node i.

each process. The log contains the syscalls executed during a chat conversation between the three players, namely the reads and writes on each process' socket. For the sake of readability, each entry is identified by the concatenation of the node and process ids and the actual files descriptors were replaced by the node ids.

In order to correctly reason about the runtime behavior from the trace in Figure 1, one must first establish the happens-before relationship between the syscalls. As defined by Lamport [33], if an event a causally-precedes an event b in a program execution, then a happens-before b (denoted $a \rightarrow b$).

Causality in distributed systems is typically captured by logical clocks[33] or vector clocks[39]. However, low-level syscalls are not able to record logical time. Let us then mimic the procedure of manually inferring the happens-before relations present in Figure 1.

The causal order of syscalls within each process' trace is trivial to define because it respects the program order [33]. As such, the main challenge here is to infer the inter-process happens-before relationships.

Note that the first parameter on each write/read syscall denotes the process that sent/received a message. Considering that a read is always preceded by its corresponding write and that TCP ensures reliable and ordered delivery, one is then able to causally order the syscalls across the three processes.

In Figure 2, we depict a possible trace resulting from causally reordering the syscalls according to the intra- and inter-node happens-before relationships. To further ease the analysis of the execution, we also convert the ordered trace into a space-time diagram, which is shown in Figure 3. In the diagram, vertical lines are the execution timelines of the processes indicated

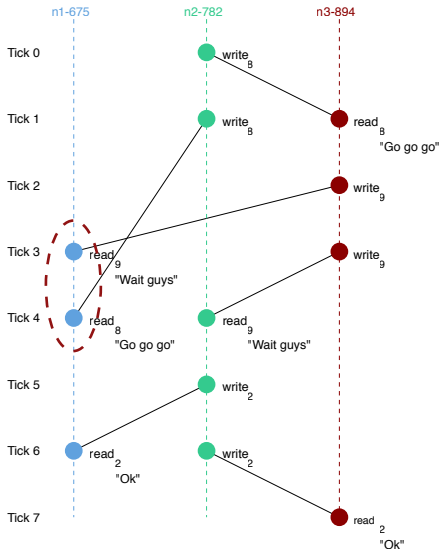


Figure 3: Space-time diagram of the trace in Figure 2.

by the labels, and the circles are the events happening in each process. Each event is associated with a given logical clock "tick". We added the message sizes on each event and the message content on read syscalls. Each pair of connected events indicates a happens-before relationship.

Displaying the messages received by each process, one obtains the following chat logs:

n1-675	**n2-782**	**n3-894**
n3: "Wait guys"	n2: "Go go go"	n2: "Go go go"
n2: "Go go go"	n3: "Wait guys"	n3: "Wait guys"
n2: "Ok"	n2: "Ok"	n2: "Ok"

Note that the chat log of process **n1-675** exhibits an inconsistency (highlighted in red) with respect to the actual message history, which explains the reason behind the reckless move by player 1. The dashed circular area in Figure 3 pinpoints the root cause of this inconsistency: a delay in the arrival of the message "Go go go" sent by process **n2-782** caused an inversion in the expected chat output. Since the inverted messages are (semantically) causally related, this means that there is a message race bug in the system implementation.

This example illustrates a game scenario that simply caused a team to lose one round. However, in complex distributed systems that require coordination, the consequences may be much more severe (e.g. data loss or corruption). It is thus of paramount importance to devise practical and effective tools to aid the analysis of execution logs.

2.3 Distributed Tracing

Distributed tracing is currently the most accurate approach to trace request executions in complex distributed applications, specially those built with microservices. The fine-grained detail in recorded cross-component request execution paths aids developers to precisely pinpoint the root cause of performance bottlenecks and behavioral anomalies [37, 31, 36].

With distributed tracing enabled, application components generate events through requests executions. These events encode the end-to-end view of each request execution throughout every software component of a distributed system [26, 46]. Typically, these events are called spans, which represents a unit of work done in the system. The more spans collected, the more detail is in the view of each request throughout application components.

At its core, distributed tracing consists of generating a unique context (i.e., request or trace identifier) for each incoming request at the entry point of the system that is propagated across call stacks (i.e., intra-component) and attached to messages (i.e., inter-component) [36]. The context is attached to each span along with the identifier of the parent span and other arbitrary key-value data (i.e., baggages). These tags enable the causal correlation of spans together under the context of each request, or trace.

Once centralized and correlated, developers are able to isolate and analyse the execution flow of each request, using cross-cutting tools. These tools depict the complete set of spans collected across application components throughout the end-to-end request execution. From this picture, developers have access to a varied set of data, including the time spent on each span — useful to identify the root cause of high latency requests — and its attached data that might support the analysis.

The downside of analysing requests in an isolated fashion is when the real cause of an anomaly might not be related to the behavior of the request under analysis but, instead, due to the interleaved execution of other requests, the root cause of errors potentially remain hidden. For instance, although cross-cutting tools allow detecting that a given request read value X, they fall short in hinting which requests contributed to the read value [54].

Implementation Challenges Companies are canalizing their efforts in building open-source collection of libraries, agents and components to ease the capture of distributed traces in applications [11]. Ensuring the propagation chain across call stacks and network components is still tough, mostly because all application components must have access to the same request contexts while they execute incoming concurrent requests. To do so, developers must ensure the coverage of all possible execution paths of requests, including custom synchronization

techniques and thread pools, otherwise the propagation chain breaks and the trace ends incomplete. These requirements for implementing distributed tracing demand application knowledge and familiarity with the source code of all integrating software components.

As modern distributed applications integrate several off-the-shelf, third-party and open-source software components in their architectures, obtaining the required knowledge about the application and their components is challenging. For instance, the effort is considerable in applications that incorporate complex third-party distributed storage (e.g., Apache HBase) or coordination systems (e.g., Apache Zookeeper). Given the inherent complexity of these distributed components, analysing and instrumenting these software components to establish a propagation chain is a time-consuming and error-prone task, turning distributed tracing impractical to implement.

Alternative approaches established a sweet spot between tracing precision and effort. For instance, focusing on the instrumentation at the API level rather than instrumenting the private instructions of software components [54]. Or even capturing kernel-level events to infer and reconstruct request traces [45, 55, 48].

2.4 Summary

In this chapter, we presented some context around modern architectures and current approaches to extract and analyse systems behavior for purposes of debugging performance bottlenecks and an incorrect behavior.

We started by showing that the complexity of modern architectures challenges developers in obtaining a detailed picture of the interactions of the application components, useful to reason about performance improvements. Traditional infrastructure monitoring systems obtain metrics from hosts' operating system in isolation and thus are unable to provide metrics in a comprehensive fashion, turning them unsuitable to monitor distributed interactions. Application monitoring tools leverage distributed tracing techniques to accurately monitor the interactions of distributed applications at great detail. Unless developers have deep application knowledge and are familiar with the source code of every single component, application monitoring tools are just limited to disclosing the behavior at the level of languages, libraries, and frameworks.

In what respects to debugging anomalies, developers primarily seek the root cause in logs. This is due to the fact that they come out of the box in most systems. However, there are challenges in using them while debugging distributed executions. First, logs are spread across several nodes. Second, although logs are timestamped, collecting logs and ordering them by

timestamp is not enough to obtain a causally-coherent view of the behavior of the system. Existing log management systems address the first challenge by centralizing all logs but fall short in ensuring causal ordering log messages, compromising the effectiveness of log-based analysis.

Distributed traces are mainly for pinpointing the root cause of performance bottlenecks. However, they are also suitable to debug executions as an alternative to logs. In addition to the mentioned difficulties in establishing the propagation chain to guarantee the construction of complete and detailed traces from complex distributed applications, distributed traces typically offer a cross-component view of a single request in isolation. It hides bugs caused by the interleaved execution of requests, since they do not causally correlate effects of operations performed by independent requests.

Chapter 3

Monitoring Distributed Interactions

The monitoring of interactions among processes within a distributed system provides insights on the system's behavior to reason about its performance. In distributed systems, processes (software components) interact mostly via network communication channels. An important issue in determining the performance of an application is the amount of data exchanged between different components through those channels. However, accurately monitoring inter-application traffic without instrumenting application or incurring in prohibitive overhead is hard to achieve.

In this chapter, we explain and validate a black-box approach for achieving efficient monitoring of distributed systems at the operating system's kernel layer, which does not require application-specific instrumentation or knowledge. We detail our strategy that resorts to eBPF technology to build a weighted graph representation of the system reflecting the amount of data exchanged with negligible overhead.

3.1 Monitoring Network Traffic

Our proposal is to use eBPF to intercept key system calls, but aggregate information at the operating system kernel, within the confines of the limited ability of kernel probes, to limit the amount of data that has to be copied to user mode to assemble the communication graph.

To use the same approach for measuring traffic we need to consider that, for sending and receiving messages operations, there are several possible syscalls that can be executed by processes. Specifically, write, sendto and sendmsg can be used to write data to a network communication channel whereas read, recvfrom and recvmsg allows processes to read data from the network channel.

In order to measure network traffic, the kernel routines for sending and receiving data need

to be instrumented. Specifically, the immediate kernel routines that are common execution paths of syscalls for writing to and reading from network channels are sock_sendmsg and sock_recvmsg, respectively. Intercepting the execution of such routines enables us to continuously aggregate the total amount of data transferred through each connection.

The aggregation of traffic per connection can be performed in user and kernel space. The former triggers events on every send or receive call which are then collected and processed by a frontend program running in user space. This is easy to implement, as it is the default usage of eBPF programs.

The latter approach is harder, as it requires information to be stored and correlated within the kernel using the limited resources provided by eBPF and decrease the amount of events passed to user space.

In detail, kernel probes are attachable at two moments: at the entry point of a kernel routine, i.e. kprobe, and when it returns, i.e. kretprobe. At the entry point, kprobe is able to inspect the arguments of the associated kernel routine. When the kernel routine returns, a kretprobe can read the returned value, if any. To extract useful data from kernel probes, it is necessary to store temporary state between both points of instrumentation. For instance, to make a correspondence between returned values and arguments of a kernel routine.

In practice, as processes may be calling and returning from this routine interchangeably, it is mandatory to ensure that the returned amount of bytes is associated with the corresponding sock argument. To this end, and leveraging eBPF's map structures, we keep <pid, sock> entries between kprobe and kretprobe calls, where pid is the kernel identifier of the process and sock the socket it is writing to. Therefore, by accessing pid, we are able to recover the related socket at the exit point and aggregate the number of bytes.

Moreover, we use a second map to accumulate data transmitted so far for each connection. For each probe, we update the total and generate new events only if an elapsed time or amount of data threshold is met. Finally, on connection close, the final traffic amount is also sent to user space.

3.2 Building the Communication Graph

Monitoring kernel routines at each endpoint is key for effectively extracting interactions of running processes. However, the data collected on each host only contains the view of one of the two ends in a network channel.

All events are routed to a central location for processing, that correlates them to establish an

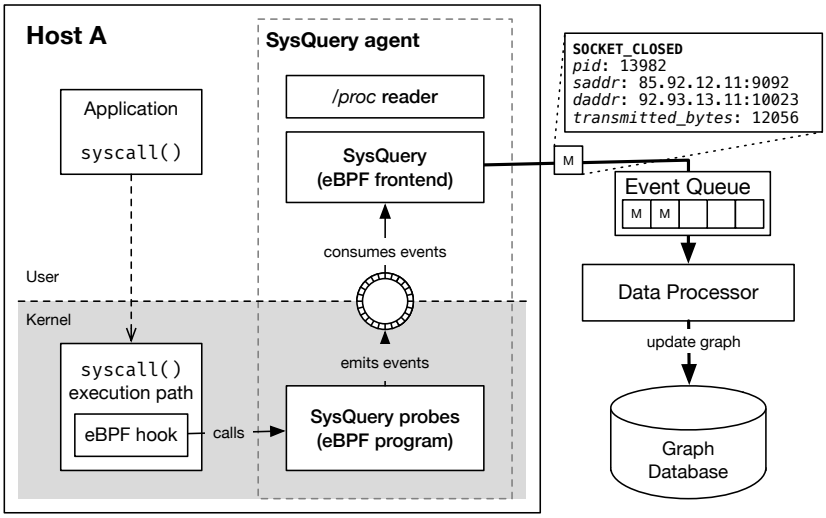


Figure 4: Architecture of prototype monitoring system. It comprises two main components: Agent and Data Processor. The Agent leverages kernel probes to collect and filter data in kernel space. The collected data is then fetched by the frontend program and sent to the Data Processor component, which is responsible for updating the graph representation accordingly.

interaction between pairs of processes. It consists of pairing the local and remote address fields with the remote and local fields of other host and thus, it establishes an inter-process interaction. Established interactions can be stored in a graph database, which nodes represent processes and edges the established connections between a pair of processes. The graph representation of inter-process communication enables operators to query the graph for analyzing system's composition and communication between components. In this work, we chose Apache Kafka for message queuing and Neo4j for storing and querying the communication graph representation.

3.3 Prototype Implementation

In this section, we present the prototype implementation of a monitoring system that monitors resource utilization and inter-process interactions in a black-box fashion.

Figure 4 depicts the architecture of the prototype monitoring system. It comprises two main components: Agent and Data Processor. The Agent module collects resource utilization metrics and observes operations performed over network communication channels. Then, the collected data is sent to the Data Processor module, using an event queue. In turn, the Data Processor component processes the received data and updates the graph representation of the placement of processes as well of their interactions.

The message queue and graph database roles, in this architecture, can be performed by an extensive set of technologies. For this prototype, we chose Apache Kafka for message queueing and Neo4j for storing and querying the graph representation. The discussion of such systems is out of the scope of this paper, since they are easily replaceable by other technologies. In the rest of this section, we describe how each module works in detail.

3.3.1 Agent

The agent module operates at the operating system level. Internally, it consists of a complete eBPF program (i.e., frontend and backend) for probing of kernel routines and a /proc reader for gathering hardware resource utilization metrics.

The included eBPF program enables user-defined probes with purposes of detecting established and closed socket connections as well as collecting the amount of traffic sent and received during the lifetime of those channels. When a client process establishes a new TCP socket connection, the eBPF program creates a new socket kernel structure both in client and server hosts, at the tcp_connect and inet_csk_accept kernel routines, respectively.

As kernel routines execute throughout the lifetime of the processes, the eBPF program aggregates statistics and updates a structure that maps network communication channels to the corresponding aggregated amount of bytes transferred during their lifetime. When a connection is established or closed, the eBPF program prepares an event (i.e., SOCKET_OPEN or SOCKET_CLOSED), attaches the process' information, the corresponding socket information and the current aggregated statistics. Finally, it pushes the event to the ring buffer of the eBPF program. In turn, the frontend program receives the event and stores it in the event queue, to be consumed in the future by Data Processor.

3.3.2 Data Processor

The Data Processor module in this architecture processes the received data from agents to extract the interactions between the processes running in the system under monitoring and update the graph representation accordingly.

After reading the collected data related to a given host A, from the message queue, it seeks in graph storage system whether the corresponding socket combination exists. Determining the socket combination is, in detail, looking for a persisted socket which fields match the expected remote endpoint. When the corresponding socket exists, it establishes a new edge between the involved processes.

When resource utilization metrics are received, the Data Processor annotates each process node with the last measurements. Therefore, having a graph representation of inter-process communication and resource utilization, one is able to leverage operators to query the graph for correlating analyzing system's composition, communication between components and even resource utilization metrics.

Orchestrators and platforms for managing containerized workloads and services, like Kubernetes, usually implement a form of Virtual Internet Protocol (IP) through proxies. A black-box agent acting at the operating system level is not aware of such virtual network layer and thus the provided data is not enough to accurately establish inter-process interactions. In order to support network virtualization layers, Data Processor leverages a map between local and virtual IPs to convert virtual IP:Port addresses to the corresponding local addresses before publishing into the graph database. This map, obtained from those platforms via API call or a command-line interface, re-enables the accuracy of effectively determining inter-process interactions in virtualized environments.

3.4 Evaluation

To evaluate the proposed approach we first show how eBPF instrumentation behaves when generating a large number of events, as would result from directly measuring network traffic without any in-kernel aggregation. Then, we evaluate the overhead introduced by our approach to measuring network traffic.

For both experiments, we use iperf as a worst-case workload. It is a syscall-intensive network I/O tool for measuring the maximum achievable bandwidth on IP networks. We set up a single n1-standard-2 instance in Google Cloud Platform equipped with 2 vRandom Access Memory (RAM)s and 7.5GB RAM. Within the same instance, we instantiated a localhost iperf server and a client that sends, for six minutes, fixed size messages to the server in order to measure the maximum bandwidth.

3.4.1 eBPF Scalability

To push an event from kernel to user space, an eBPF program builds a data structure and writes it in a ring buffer connecting both in-kernel program and a frontend program running in user space. The size of the ring buffer is the same as set by default by Scope [53] — an open-source tool for detecting processes, containers, and hosts in a distributed system based

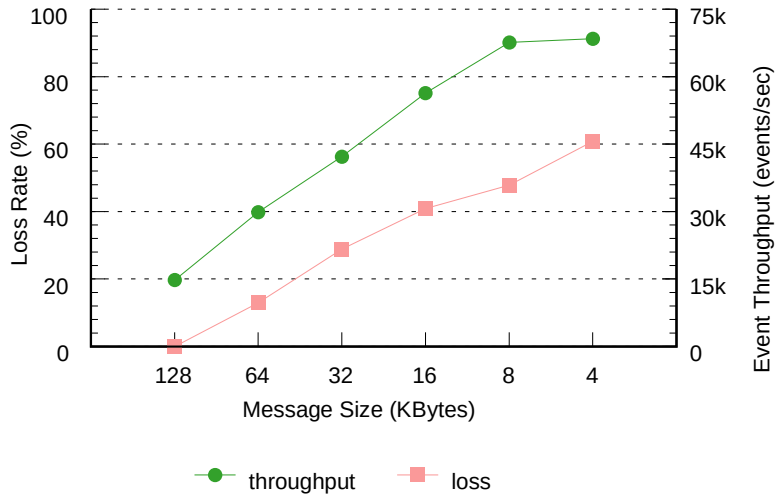


Figure 5: Evolution of loss rate with increasing throughput.

on open connections, i.e., 256 pages of 4KB. When the ring buffer hits the maximum size, new events overwrite the oldest ones, leading to an increased loss rate.

Figure 5 depicts how the loss rate evolves when throughput increases. In iperf, as the size of each message sent from client to server is reduced, the amount of executed syscalls for the same amount of data increases, and so the number of events pushed from kernel to user space.

High loss rates mean the kernel program is producing events faster than the frontend program is consuming them. Even allocating more pages for the ring buffer, which maintains events stored for a longer time and possibly contributing for lower loss rate, the frontend program still needs to process all pushed events, which contributes to resource consumption, as we will show next. This clearly shows why the naive extension of the technique used in Scope [53] to monitor inter-application traffic is not viable.

3.4.2 Overhead

In this second experiment, we aim at measuring the impact of capturing network metrics using eBPF, including on CPU usage. We set iperf's message size to 128KB, which caused zero loss rate in the eBPF scalability evaluation. For monitoring CPU usage, we enabled dstat to collect data second by second and discarded the first and the last thirty seconds of measurements.

We instrumented kernel routines for capturing events related to the opening and closing connections, much the same as in Scope. Then, we extend it to collect network traffic metrics, namely the amount of traffic per connection. This is done in two different ways: First, by pushing

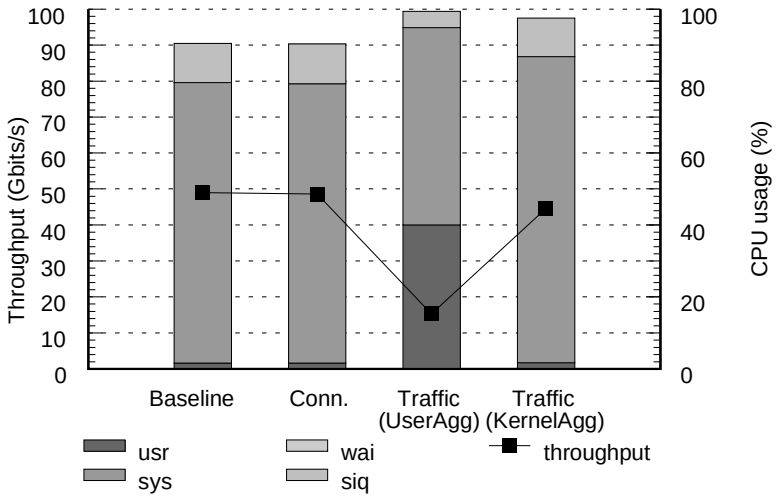


Figure 6: Throughput and CPU overhead.

events for each send/receive operation and aggregating them in user space (i.e., UserAgg). Second, by aggregating measurements in kernel structures and then pushing events of traffic statistics (i.e., KernelAgg).

Figure 6 shows the impact of both alternatives on iperf throughput, as well as on CPU usage. Intercepting only connection establishment (i.e., Conn.) introduces negligible 1% overhead over the baseline, since it only pushes events when a connection opens or closes. In fact, the behavior of iperf supports such results since it only creates a single connection, which is closed in the end of benchmark.

In turn, the Traffic (UserAgg) alternative enables instrumentation on send and receive operations and pushes a single event for each executed operation. The results clearly show a high impact on iperf performance, about 68%, mainly due to CPU usage in user mode. The increased CPU usage in user mode is associated to the significant amount of events pushed from kernel to user space and then consumed by the frontend program, as previously depicted in Figure 5. Again, this shows that a naive extension of Scope is not viable for monitoring inter-application traffic.

Our proposal, the Traffic (KernelAgg) alternative, shows that when moving from pushing an event for each operation to periodically pushing aggregated statistics, the overhead is significantly lower, from 68% to 9%. In detail, and contrarily to user aggregation, the impact on CPU usage is mainly on time spent in sys mode, which is expected since the aggregation is performed in kernel mode.

3.5 Compatibility at Scale

The black-box nature of our approach enables seamless integration of applications built on a wide range of platforms without the need for any modifications to their source code, binaries, or even to their setup and deployment. In this section, we validate the compatibility of our approach with two complex and heterogeneous microservice-oriented applications: Hipster Shop[1] and Sock Shop.[2].

Developed by Google, Hipster Shop is an e-commerce system built atop 10 heterogeneous microservices. The heterogeneity comes from the diversity of used programming languages: 4 services developed in Go (frontend, productcatalog, shipping and checkout), 2 developed in Python (email and recommendation), another 2 in Node.js (currency and payment), and the remaining — cart and ads — in C# and Java, respectively. They all communicate using gRPC and use only a Redis instance to cache data.

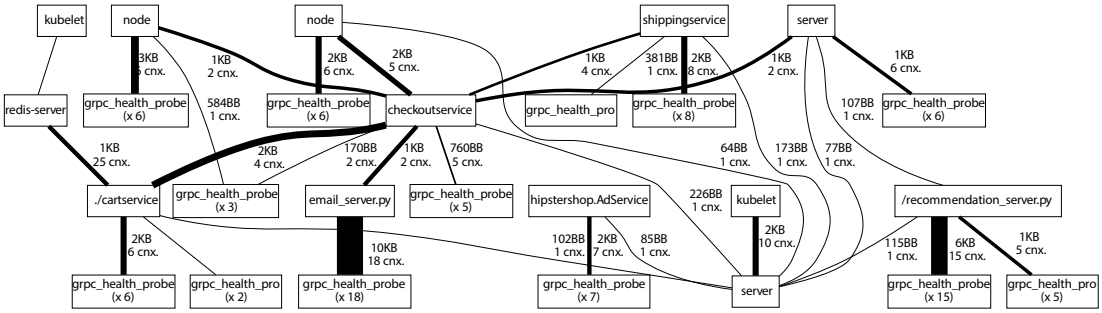
Sock Shop is another web e-commerce system, developed by WeaveWorks, mostly implemented in Java and Go, with some Node.JS and .NET Core components. Besides the applicational services, it adopts MongoDB and MySQL for persistent storage and RabbitMQ, implemented in Erlang, for message passing among microservices during requests executions.

3.5.1 Deployment

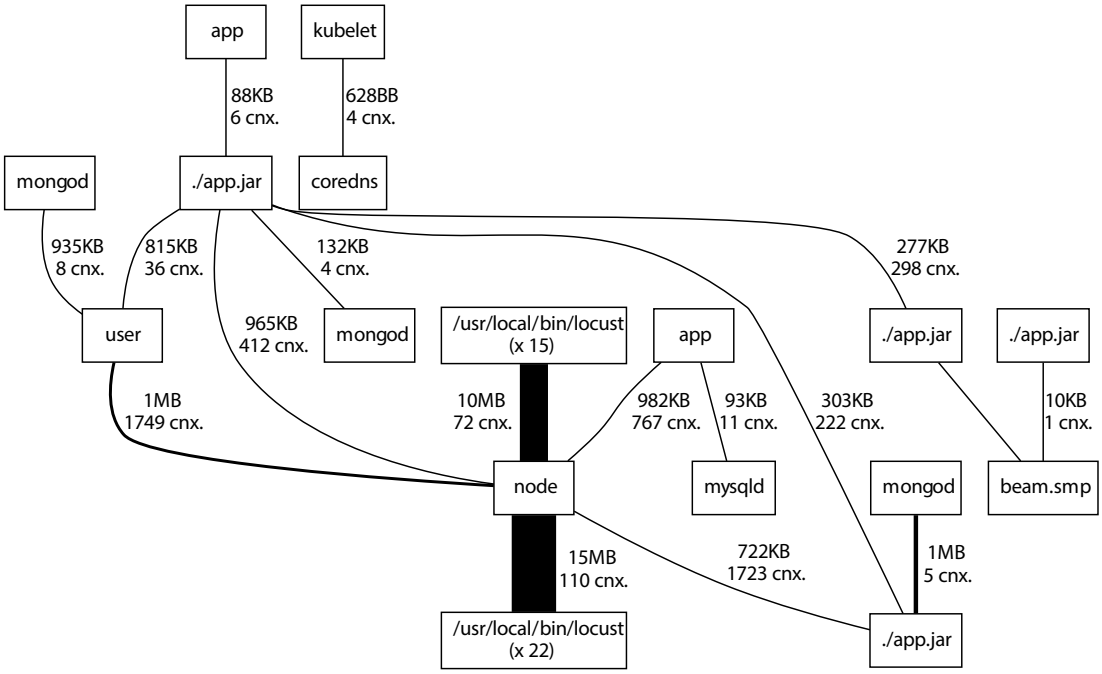
The repositories of both applications provide pre-built Docker containers and Kubernetes manifests that make it possible to deploy them with a single command. Alongside, they also provide synthetic load generators based on Locust, that make it easy to add traffic that exercises all the tiers. Finally, the provided Kubernetes manifests also often make use of virtual IPs in service definitions.

Deployment of all tests uses standard unmodified Docker containers for each of the systems orchestrated with Kubernetes using the originally provided Manifest files. The test environment is composed of five n1-standard-4 Google Compute Engine (GCE) instances (4 vCPUs and 15GB RAM), all in the same region and zone. All instances are running the standard Ubuntu 16.04 LTS (xenial) GCE image. Kubernetes is installed with kubeadm and configured with the Flannel network fabric.[3] Each instance is equipped with a local SSD scratch disk used for all container image and volume storage (i.e., the payload). The prototype that implements our Traffic (Ker-

[1] https://github.com/GoogleCloudPlatform/microservices-demo
[2] https://microservices-demo.github.io/
[3] https://github.com/coreos/flannel



(a) Hipster shop.



(b) Sock shop.

Figure 7: The obtained architecture of microservice benchmark applications. Each box represents a process running in the deployment, as well as edges represent the amount of exchanged data. Without prior knowledge regarding the organization of each application, our black-box results match the expected application architecture according to their documentation.

nelAgg) approach is installed directly in the host operating system and does not make use of the local SSD disk. By using the the standard Kubernetes configuration, one server instance is dedicated to the control plane and the applications are spread-out to the remaining four server instances. Additionally, we use the control plane server to collect and save generated events to disk for off-line processing.

3.5.2 Results

For each microservices-oriented application, we proceed as follows. First we start the application and the load generator. Then, we start the monitoring agent and stop it after a 40-second period, collecting and processing the data in the database. Figure 7 depicts the resulting communication graphs rendered with Graphviz, where each edge's width is proportional to the amount of data exchanged between the connected services. In these figures, short process names have been extracted from the full command line for legibility using an heuristic that depends on the platform. For instance, if the command name is "java" then it looks for the main class name. When directly querying Neo4j, the full command line is available.

The resulting communication graph correctly depicts the expected application architecture as described in their design documents.[4] [5] Even tough the microservices are developed in a wide set of programming languages and have distinct requirements, our monitoring agent was able to detect and collect statistics concerning the amount of data exchanged between services. Additionally, Figure 7a unveils `grpc_health_probe` as used by Kubernetes manifests to check for readiness and liveness of pods.

With these results, one is able to conclude that our black-box approach is compatible with large, complex and heterogeneous architectures, various platforms and also overcomes virtualization network layers (i.e. virtual IPs) without requiring source code instrumentation. Additionally, we validate the advantage of capturing actual data transmission instead of simply connection set up and tear down. First, because some connections remain active throughout the monitoring period and there is no data regarding the utilization of such connections. Other aspect is it provides useful insights for performance improvements that otherwise would remain hidden. For instance, in Sock Shop's communication graph, the large number of connections between the front-end and user services, when compared to the amount of data actually trans-

[4] https://github.com/GoogleCloudPlatform/microservices-demo/blob/master/docs/img/architecture-diagram.png

[5] https://github.com/microservices-demo/microservices-demo.github.io/blob/HEAD/assets/Architecture.png

mitted, shows that they are not persistent, which is inefficient.

3.6 Case Study

We now validate that the information obtained by monitoring data exchange is valuable to determine the best placement of components in physical resources.

3.6.1 Application scenario

We use a layered storage and processing system as the application scenario. For the data storage layer we use the Cassandra NoSQL (NoSQL) data store. For the data processing layer we use Spark through its Structured Query Language (SQL) interface. Spark accesses data in Cassandra using the standard spark-cassandra-connector.

This is an interesting case study as this is a typical architecture for Big Data processing in the cloud and its performance depends on various factors such as: data movement between the storage and the processing layer, as Spark scans Cassandra tables; data movement within Spark, as interim results are shuffled between various processing stages needed; memory available to cache original and interim data; and available CPU time, mainly in the Spark processing layer.

For experiments, we populate the Cassandra database with 2 million randomly generated rows, each containing ten `bigint` and four `text` fields (500 characters each in average). Each row is thus approximately 2 KiB in size. We use two SQL queries (Q1 and Q2) that we describe later, but that we design to minimize computation, which at the same time reduces the time needed to run tests and highlights monitoring overhead.

We use standard unmodified Docker containers for both Spark and Cassandra. The test environment is composed of four n1-standard-4 Google Compute Engine (GCE) instances (4 vCPUs and 15GB RAM), all in the same region and zone. All instances are running the standard Ubuntu 16.04 LTS (xenial) GCE image. Kubernetes is installed with kubeadm and configured with the Flannel network fabric.[6] Each instance is equipped with a local SSD scratch disk used for all container image and volume storage.

We use a `StatefulSet` controller for Cassandra (4 replicas) and `Deployment` controllers for Spark master (1 replica) and worker roles (4 replicas). This results, by default, in having one Spark worker and one Cassandra server in each server instance. The Spark master is

[6] https://github.com/coreos/flannel

	Host	CPU	RAM	Sent	Received
Q1	instance-1	684	7119092	619296	665682
	instance-2	685	7057948	636852	682645
	instance-3	653	7044764	658822	584922
	instance-4	687	7708228	659970	641691
Q2	instance-1	838	7307848	87137	103875
	instance-2	847	7317072	82746	78151
	instance-3	808	7313508	69435	108930
	instance-4	869	8071772	131406	104929

Table 1: Resource usage in nodes (CPU in seconds, others in KiB).

arbitrarily deployed to one of them. SQL jobs are started with spark-submit running in the master container. For each test, we start collecting monitoring data, submit the job and wait for it to finish, and then stop collecting monitoring data.

3.6.2 Limitations of resource monitoring

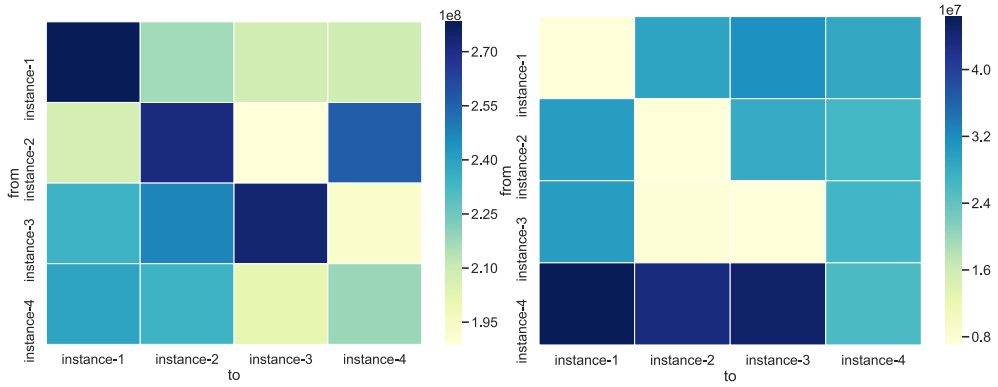
We now establish a baseline by discussing what can be achieved with traditional resource monitoring tools. Table 1 shows resource usage metrics for server instance running each of the queries.

From these results we can observe that query Q2 requires more CPU time than Q1, while transferring less data across the network. Also, data transfer in Q1 is almost balanced while with Q2 instance-4 sends significantly more data than others.

We might thus speculate that workers in Q1 should be concentrated on a smaller number of server instances, to minimize data exchange. Or that, in contrast, Q2 is amenable to using more worker instances to leverage more CPUs as data exchange seems to be much smaller. But we really do not know for sure, as we do not know whether the data exchange observed is done mainly when scanning data from Cassandra or when shuffling interim results within Spark.

Actually, this is such an important issue that the Spark interface exposes this information very clearly to allow the operator or an automated optimizer to make decision. This does however assume that the application is known and that the custom interface can be used. Moreover, it assumes that such custom interface is exposed to the cloud provider and that the cloud provider is able to understand the needs of all potential applications.

The challenge is thus: How to make resource allocation and job placement decisions based



(a) Q1 **(b)** Q2

Figure 8: Inter-host traffic with default placement.

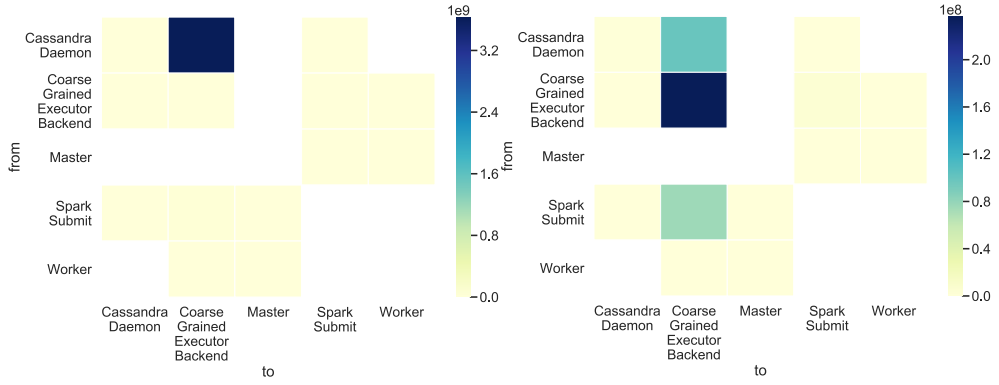
solely on data that can be obtained without knowledge of the application, i.e., while treating the application as a black-box?

3.6.3 Using data exchange monitoring

Taking advantage of information about data exchanged in the system represented as a connection graph as described in Section 3.3, we improve our understanding of host affinity. Figure 8 shows heat-maps of inter-server traffic computed by projecting raw connection graphs on physical server identifiers. In contrast to information collected by cloud provider monitoring in the hypervisor, we also get information about data exchanged within the server, among the various processes. This points out to a clear difference between Q1 and Q2: Data exchange in Q1 is actually higher than Table 1 has shown. The majority of traffic in Q1 is within servers, as shown by the main diagonal. Based on our knowledge of Spark and Cassandra, we can derive this happens because Q1 is focused on scanning data as, with each instance has one Spark worker and one Cassandra server, this is the only way to get intra-host data exchange.

In contrast, Q2 has little intra-host traffic that is mostly uniform except for instance-4 server that sends substantially more data to all other hosts. In fact, instance-4 is where Spark master container, hence also spark-submit, is running. We can speculate that the traffic is related to shuffling in Spark, but we are not really sure, as Cassandra servers also exchange some data. These conclusions are however still not good enough, even if we had to use our knowledge of the application to derive them and they cannot easily be automated for general workloads.

We can get actual evidence to confirm this speculation and avoid making use of application-specific knowledge by projecting the raw connection graphs on process types (i.e., their com-



(a) Q1 **(b)** Q2

Figure 9: Inter-process traffic with default placement.

mand lines), as shown in Figure 9. Although this is achieved with a similar query, it provides substantially different information, as it aggregates information from different server instances as long as they are running processes with the same command line. This information confirms that Q1 is transferring data from Cassandra servers to Spark CoarseGrainedExecutorBackend processes, which are part of the Spark worker container. In addition, it confirms that most data transferred in Q2 is between Spark workers.

Figures 10 and 11 show how processes map to server instances, as rendered by Graphviz. In detail, we set: node height from used RAM; node line width from average CPU used; edge width from amount of bytes exchanged (both directions) between processes; and colors from commands and command-pairs for nodes and edges, respectively. We also remove all edges that correspond to less than 10% of traffic, to improve clarity. From colors, we can easily glance that most of the data exchanged in each query is between different processes. From node line widths, we can see that CPU usage by Cassandra is more relevant in Q1. In short, it is clear that Q1 is performing a scan of a large amount of data and that Q2 is reading little data from Cassandra but performing a computation that requires shuffling. The next challenge is how to take advantage of these conclusions in both an automated application independent way and manually by exploiting all possibilities in the application.

3.6.4 Automatic placement

In this section we show how the presented approach is compatible with techniques for improving the placement of the workload in an automated way without any application-specific knowledge. As this is something that can be done by the cloud provider, we set as the goal to pack the

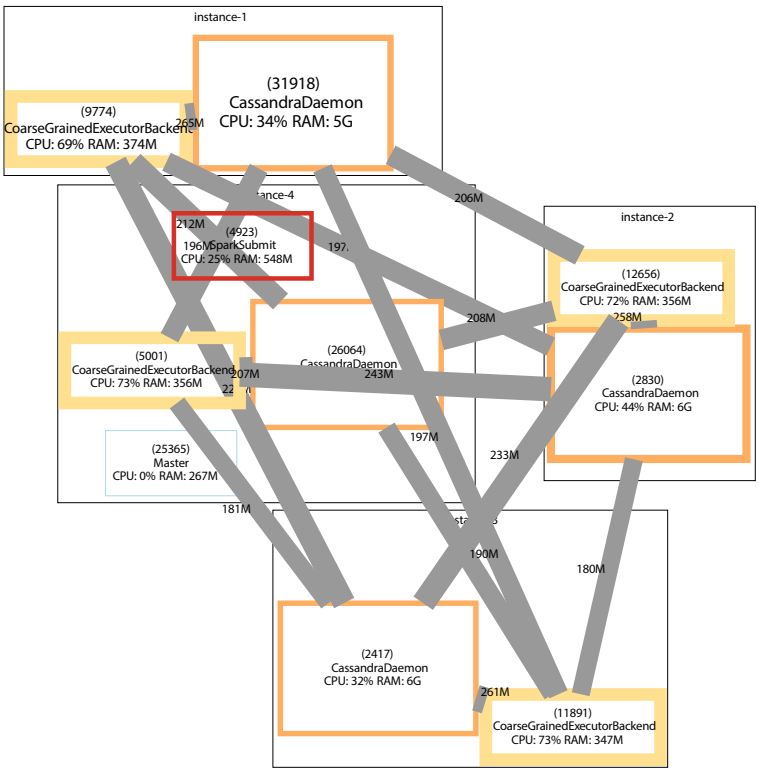


Figure 10: Mapping of execution over hardware nodes with default placement for Q1 (Scan).

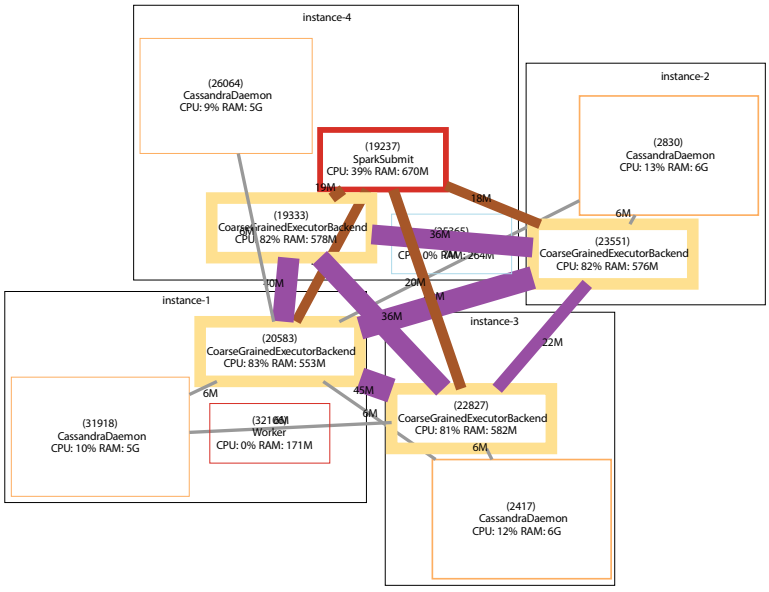


Figure 11: Mapping of execution over hardware nodes with default placement for Q2 (Shuffle).

	Baseline	Scan Opt.	Shuffle Opt.
Q1	2.46 GB	**1.76 GB**	1.96 GB
Q2	355.28 MB	270.42 MB	**214.64 MB**

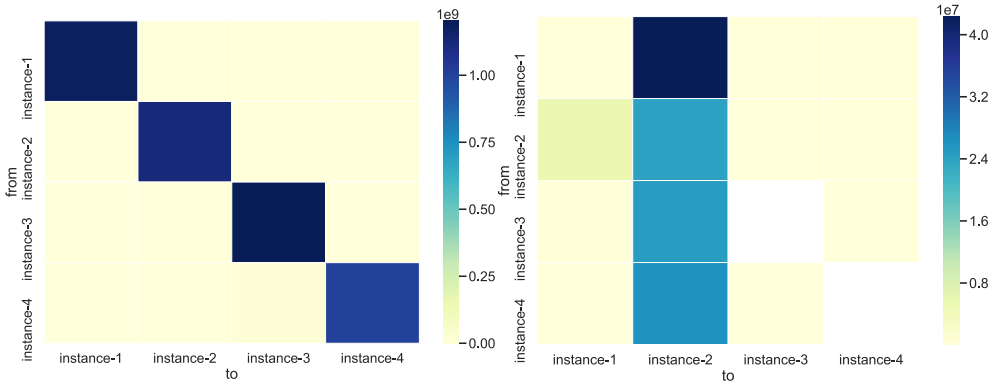
Table 2: Network traffic with optimizations.

system optmized for a known workload in the minimal number of servers and, while doing it, to minimize network traffic with minimal impact in user visible performance. This would allow, for instance, the cloud provider to reduce the number of physical servers that need to be powered on and to reduce a source of congestion.

To this end, we resort to Pyevolve, a genetic algorithm framework written in pure Python, to build a simple optimizer that takes an initial placement of containers in servers, each of them corresponding to a set of processes, and outputs an optimized placement as the end result. The genome is thus a simple vector, with one element for each placeable component (container), and an integer value identifying each possible location (server), which is supported natively by Pyevolve. The fitness function for each individual outputs the product of three factors: optimal result for each server where CPU cores are expected to be fully used, with a penalty for each underused server and a (larger) penalty for each overused server; optimal result for each server where RAM is expected to be fully used, with a penalty for each underused server and a (larger) penalty for each overused server; optimal result for no cross-server communication, with a penalty for all data transferred.

Using this optimization strategy, we produce placements considering each of the benchmark queries. For query Q1 (Scan Opt.), the result is to place two Spark workers and two Cassandra servers in each server instance. For query Q2 (Shuffle Opt.), the result is to place three Cassandra servers in one server instance, and the remaining Cassandra together with all Spark workers in a second instance. In both cases, this corresponds to using only 50% of the resources initially allocated.

We then translate these results into placement constraints in Kubernetes manifests and redeploy and retest both queries with both placements to compare the results with the initial values. The results in terms of exchanged network traffic are shown in Table 2. It can be observed that although both strategies reduce inter-host traffic, which is expected as the number of hosts is reduced, using the strategy that is informed by inter-process data exchanged obtained from instrumentation of read and write operations results in greater reduction. Query runtime is degraded between 9% and 14%, always less with the Scan Opt. strategy based on monitoring of Q1, which results in a more balanced CPU distribution.



(a) Q1 **(b)** Q2

Figure 12: Inter-host traffic after manual optimization.

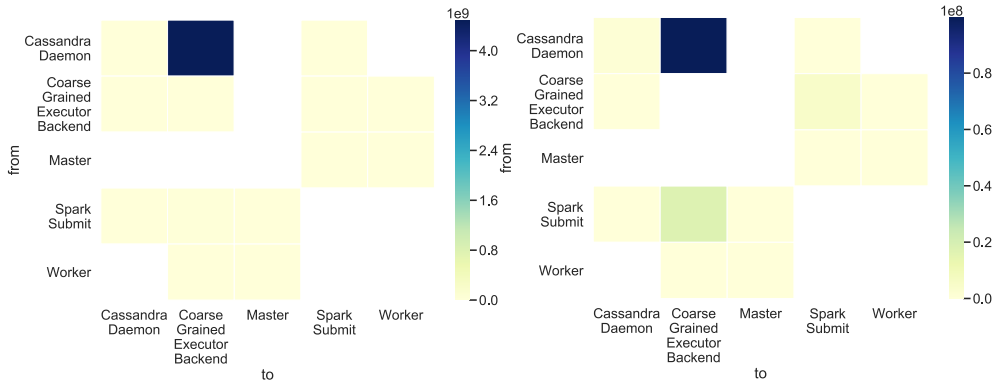
3.6.5 Manual optimization

In this section we focus on improving application-specific configuration but using only results obtained from black-box monitoring. This is something that currently cannot be done easily by a cloud provider and the orchestration system, but is interesting to the application owner and operator. This way the operator can overcome the lack of appropriate application specific monitoring tools and, most interestingly, can use a single monitoring tool for complex systems assembled from a variety of components, as is typical in Big Data storage and processing.

First, aiming at optimizing for the Q1 workload, we modify Kubernetes manifests to deploy each Spark worker together with a Cassandra server within the same pod (Scan Opt.). This makes them have the same IP address and allows Cassandra servers to be recognized as local in the default topology detector. Second, targeting the workload of Q2, we create an alternative deployment configuration that uses only one worker with four times as much resources assigned (Shuffle Opt.) in terms of CPU cores and RAM.

We then run each of the workloads in the corresponding configuration. Figure 12 shows that with Q1 almost all data exchanged is within the same server instance. In contrast, in Q2 data is sent to only a single server instance that is running the worker process. Although in terms of inter-host network traffic this seems so different, Figure 13 shows that actually they both correspond to the same thing: Data is only sent almost exclusively from Cassandra servers to Spark workers.

Figures 14 and 15 show how processes map to server instances as rendered by Graphviz for each query. They directly compare to the original ones Figure 10 and 11, where the same colors are used for the same process types and inter-process links. The consequence of this



(a) Q1 **(b)** Q2

Figure 13: Inter-process traffic after manual optimization.

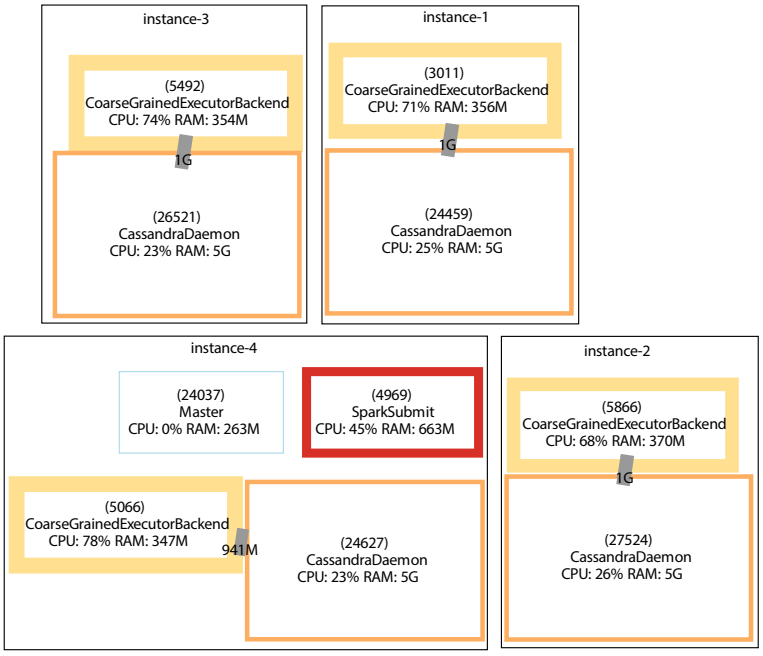


Figure 14: Mapping of execution over hardware nodes after manual placement and configuration for Q1 (Scan).

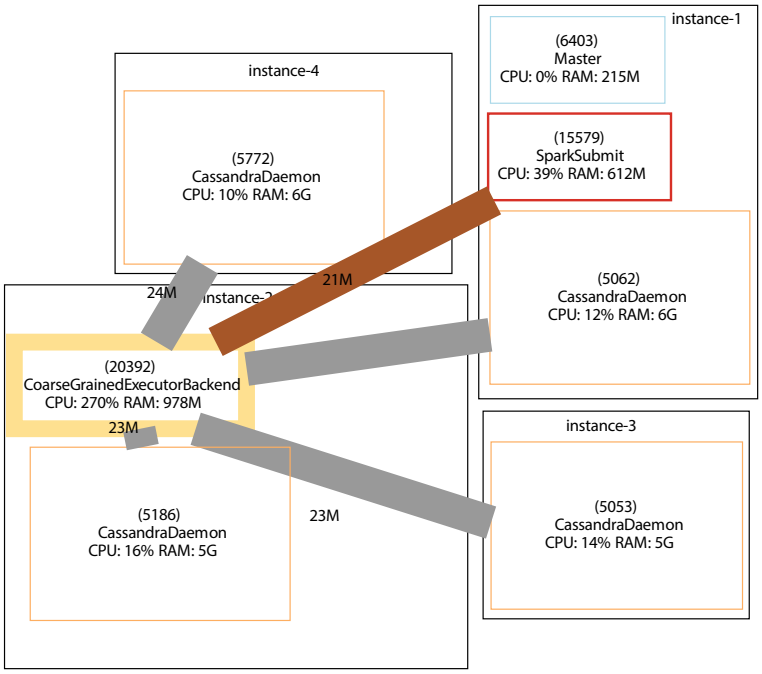


Figure 15: Mapping of execution over hardware nodes after manual placement and configuration for Q2 (Shuffle).

	Baseline	Scan Opt.	Shuffle Opt.
Q1	34.33 (0.31)	**30.19** (0.64)	31.87 (0.20)
Q2	52.96 (1.34)	51.95 (0.67)	**37.30** (0.41)

(a) Runtime (in seconds, average of 10 runs, standard deviation in parenthesis).

	Baseline	Scan Opt.	Shuffle Opt.
Q1	2.46 GB	**17.12 MB**	2.67 GB
Q2	355.28 MB	279.56 MB	**95MB**

(b) Network traffic.

Table 3: Manually optimized placement and configuration.

change is shown in Table 3a, where it is noticeable that the first optimization reduces runtime of Q1 by 12% and the second reduces the runtime of Q2 by 29%. Table 3b shows the impact in network traffic, which is particularly dramatic with the scan optimization and the Q1 workload.

The difference between these results and those of Section 3.6.4 lead to an interesting conclusion. As long as one needs application specific tools to monitor data exchange within distributed data storage and processing systems, it makes sense that configuration such as needed for the optimization of Q2, by trading workers processes for additional resources, are also performed in application specific ways. In fact, this configuration step for Spark needs to be done twice: One by setting the number of cores for each worker in Spark configuration and the other in Kubernetes manifests to make resources available.

However, now that we can infer the need for this optimization in an application-independent way, it would be interesting to have more standard and automatic ways to do this configuration. This would allow, for example, a Kubernetes controller that can control such trade-off when informed by a monitoring system capable of collecting connection traffic network metrics. This would nicely complement the ability of Spark to detect and adapt to data locality.

3.7 Summary

Existing distributed monitoring systems either collect resource usage at the operating system level in an application independent way and with low overhead, or perform application-level tracing, that can depict and quantify interactions between components in distributed systems in great detail.

Our approach focuses on precisely quantifying the amount of data exchanged between processes and thus provides a new trade-off in between traditional monitoring approaches. In this chapter, we described how this can be achieved with very low overhead, compatible with usage on production systems, by taking advantage of kernel probes in judiciously chosen operating system primitives and by deferring the bulk of processing to an external graph database.

We validated the proposed approach with data-exchange intensive micro-benchmark, showing less that 9% overhead. The usefulness of the approach for drawing a variety of conclusions is shown with a case study of Spark and Cassandra, that includes demonstration of manual and automated configuration actions.

Chapter 4

Causal Log-Based Analysis

Programmers and operators typically rely on log data to narrow down the root cause of unexpected behaviors in distributed systems. The fact that logs are provided by most systems, with or without access to source code, motivates developers to use logs as a primary resource for debugging anomalies.

Unfortunately, the inherently distributed nature and complexity of distributed executions often lead to multiple independent logs, scattered across different physical machines, with thousands or millions of entries poorly correlated in terms of event causality. This renders log-based debugging a tedious, time-consuming, and potentially inconclusive task.

In this chapter, we present an approach that uses events captured at the kernel layer to make log-based analysis a practical task. Taking advantage of the well-known causal relation among kernel events, it consists of constructing a causally coherent view of the log entries generated several application software components without requiring application-specific knowledge. To validate this approach, we implemented a log-based analysis tool — Falcon — for distributed systems, capable of generating a global execution schedule from multiple independent log files while preserving event causality. We demonstrate its applicability by inspecting the leader election protocol behavior in Apache Zookeeper [29], an open-source project that enables highly reliable distributed coordination.

4.1 Design Requirements

Time spent at post-mortem software debugging is directly affected by the availability of useful information captured during production runs. Since logging is an expensive operation, there is a trade-off between the log's verbosity level, the performance, and space overhead imposed

at runtime [58]. For that reason, different tracing tools opt for focusing on different aspects and provide distinct features (e.g. printing log statements, sniffing network packets, profiling performance, etc.). Nevertheless, one should be able to leverage all those features in order to ease the burden of debugging complex distributed systems. A practical and effective log-analysis tool should thus meet the following design requirements:

- **Support several log sources.** The tool should be able to extract useful knowledge about the execution from multiple data sources, such as logging libraries, network sniffers (e.g. libpcap-based tools), and low-level tracing tools (e.g. ptrace-based tools).
- **Combine data in a causally consistent way.** The tool should be able to combine all logged events in a seamless and coherent fashion, even if they were captured at different physical machines with unsynchronized clocks. In practice, this corresponds to ensuring that the happened-before relationship between events is established across all log files regardless of their source.
- **Provide a visual representation of the execution.** To obviate complexity due to log verbosity and further help developers to reason about the execution, the tool should be able to display events in a "human-friendly" way. In the particular context of distributed systems, space-time diagrams depicting the inter-process causal dependencies have long been used to aid the understanding of distributed protocols over multiple processes [20].

In the next section, we describe how Falcon meets the mentioned requirements.

4.2 Architecture

Falcon is designed with a modular architecture, whose components operate together as a pipeline. In a nutshell, Falcon receives as input log files from multiple data sources and outputs a space-time diagram that preserves event causality. Figure 16 depicts the architecture of Falcon, composed by three main modules: the trace processor, the happens-before model generator, and the visualizer. Each module is described in detail as follows.

Trace Processor. Since the events logged by the different tools can vary in both format and content, Falcon needs to first normalize and merge the collected data into a global event trace with a common scheme. This procedure is done by the trace processor module. The trace processor is equipped with a dedicated driver for each type of log, responsible for translating the library-specific entries into events that can be processed by Falcon. As such, drivers may range

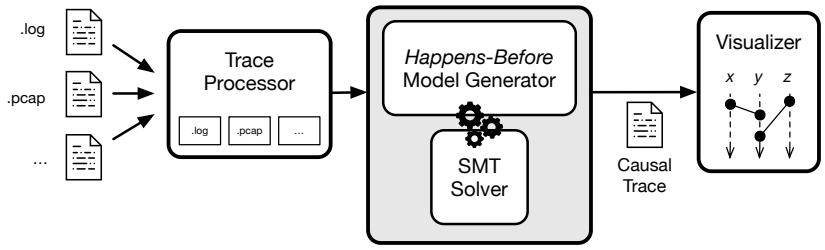


Figure 16: Falcon's pipeline architecture, which comprises modules for trace processing, model generation, and visualization.

from simple parsers for textual logs (e.g. for log4j) to packet unpackers for network sniffers (e.g. tshark). In some cases, the trace processor generates events that are the result of merging data from different logs. For example, an event representing the sending of a message can be built by augmenting the information of a write syscall with the message payload captured by a network sniffer, which operates at the kernel layer. The events resulting from Falcon's log normalization and merging are the following:

- **START**$(process)$: a process starting event;
- **END**$(process)$: a process finishing event;
- **FORK**$(parent, child)$: a process creation event, where $child$ denotes the process spawned by process $parent$;
- **JOIN**$(parent, child)$: represents a join event, where process $parent$ waits until the $child$ process finishes;
- **CONNECT**$(process, src, dst)$: represents a new connection, where src and dst denote the addresses (IP and port) of the local and remote processes, respectively;
- **ACCEPT**$(process, src, dst)$: event indicating that a connection was established, where src and dst also denote the local and remote addresses, respectively;
- **SND**$(process, src, dst, msg)$: a message sending event, where msg is the identifier of message being sent from the src address to the dst address;
- **RCV**$(process, src, dst, msg)$: a message receiving event, where msg denotes the identifier of the message sent by the src address and received by the dst address.
- **LOG**$(process, msg)$: a log entry event, where msg is the content of the message logged by process $process$.

The trace processor module also exposes a public API to ease the development of drivers and the integration of additional logging libraries into Falcon.

Happens-Before Model Generator. The complete, normalized event trace is then fed into the Happens-Before (HB) model generator. This module is responsible for combining all events into a single causally-consistent schedule. To this end, the HB model generator builds a symbolic constraint formulation encoding the happens-before relations between events. For instance, the model encodes a constraint stating that the send event of a message must happen-before the corresponding receive event. The HB constraints are further described in Section 4.3.

Solving the model with an off-the-shelf constraint solver yields a causally-ordered event schedule.

Visualizer. The visualizer finishes the Falcon's pipeline by providing a graphical representation of the causal trace generated in the previous step. In detail, the visualizer generates a "space-time diagram", as introduced by Lamport [33], depicting both the events executed by each process and the inter-process causal relationships between them.

4.3 Happens-Before Constraint Model

As defined by Lamport [33], there exists a happens-before relationship between two events a and b, denoted $a \rightarrow b$, if:

- a and b belong to the same process[1] and a precedes b in the execution.
- a and b belong to different processes and a represents the sending of a message m and b represents the reception of m.

Distributed executions often comprise other causal relations that should be considered, namely $a \rightarrow b$ also holds if:

- a is the fork event of a process q by a process p and b is the first event of q.
- a is the last event of a process q and b the join event of q by a process p.
- a is the connect event issued by a process p to a process q and b is the accept event in q.

[1] We use the term process to denote both processes and threads.

Note that the happens-before relation is transitive, irreflexive and antisymmetric. Also, when $a \nrightarrow b$ and $b \nrightarrow a$ holds, then a and b are considered to be concurrent.

Falcon casts the problem of combining the events from independent logs into a global, causally-ordered execution schedule as a Maximum Satisfiability Modulo Theories (MaxSMT) problem. The MaxSMT problem can be seen as an optimization version of the satisfiability problem (for types of variables other than boolean ones) and has the goal of finding a total assignment to variables of a formula that maximizes the number of satisfied clauses. Among the variants of the MaxSMT problem, in this work we assume a partial MaxSMT problem where some clauses are considered as hard and others are considered as soft. The goal is thus to find an assignment to the variables such that all hard constraints are satisfied and the amount of satisfied soft constraints is maximized.

Falcon's causality model comprises i) integer symbolic variables that represent the logical clocks [33] of the events supported by Falcon (see Section 4.2) and ii) hard constraints over those variables stating the causal relations between the events. A solution to this model thus assigns a value to each variable such that all happens-before rules are satisfied. In practice, this corresponds to inferring a causally-consistent execution schedule by computing a logical clock per event.

More formally, the constraint model, denoted Φ_{HB}, consists of a MaxSMT formulation defined as the following conjunction of sub-formulae:

$$\Phi_{HB} = \underbrace{\phi_{inter} \wedge \phi_{intra}}_{hard} \wedge \underbrace{GOAL}_{soft} \tag{4.1}$$

where ϕ_{inter} encodes the inter-process causality constraints, ϕ_{intra} encodes the intra-process happens-before rules due to program order, and $GOAL$ states the soft constraints that allow steering the solving procedure towards a given goal. Falcon currently provides support for generating logical clocks that: i) follow the original timestamp order as much as possible (ϕ_{ts}), and ii) expose concurrency issues by minimizing the logical time intervals between events (ϕ_{min}). We now describe each sub-set of constraints in more detail.

a) Inter-process HB Constraints (ϕ_{inter}) these constraints represent the causal dependencies due to message exchanges and inter-process synchronization. Following the happens-before rules presented at the beginning of this section, the inter-process HB constraints ϕ_{inter} are written as follows:

$$\text{fork}_{p,q} < \text{start}_q$$
$$\text{end}_q < \text{join}_{p,q}$$
$$\text{connect}_p < \text{accept}_q$$
$$\text{snd}_{p,m} < \text{rcv}_q$$

where p and q are distinct processes, m represents a given message, and the variable names correspond to the events described in Section 4.2. For instance, for a message m sent by p to q, the constraints encodes that the logical clock of the corresponding event **SND**(p, p, q, m) in the trace must be smaller than that of the event **RCV**(q, p, q, m).

b) Intra-process HB Constraints (ϕ_{intra}) these constraints state that events in the same process execute sequentially according to the program order. Let Γ_p denote the event trace of a process p, and let c_i and c_j be the symbolic variables representing the logical clocks of events i and j. The intra-process HB constraints ϕ_{intra} are given by:

$$\forall i, j \in \Gamma_p : (i < j \implies c_i < c_j)$$

c) Timestamp Constraints (ϕ_{ts}) timestamp constraints are soft constraints (see $GOAL$ in Equation 4.1) that aim at approximating the schedule produced by the solver to the actual event ordering observed during the production run. These constraints state that events should be given logical clocks that follow the order given by timestamps in the log files. However, since two causally-ordered events logged on different machines may exhibit physical timestamps conflicting with their HB relationship, timestamp constraints may be violated in order to satisfy causality.

d) Clock Minimization Constraint (ϕ_{min}) clock minimization constraints are also encoded as $GOAL$ soft clauses and strive to minimize the values assigned to the symbolic variables. The goal is to produce a compact schedule capable of exposing event concurrency. For instance, if two distinct **SND** events exhibit the same logical order in the schedule yielded by the solver and their corresponding **RCV** events belong to the same process, then there is a message race.

Let $e \in \Gamma$ be an event in the complete execution trace and c_e the symbolic variable representing its logical clock. The clock minimization constraint ϕ_{min} is written as:

$$\phi_{min} = min \sum_{\forall e \in \Gamma} c_e$$

Solving the Φ_{HB} model generated by Falcon using an off-the-shelf SMT solver yields an execution schedule in which events are guaranteed to be causally ordered.

4.4 Implementation

This section discusses some relevant implementation details of our prototype[2] of Falcon.

The trace processor module is implemented as an extensible Python program that allows the integration of custom drivers for normalizing log files into a pre-defined JSON format. Currently, the trace processor provides three out-of-the-box drivers. The first is a ptrace-based tool that collects syscall traces. The second driver handles logs generated by log4j, a logging library for Java programs. The third uses tshark to extract message payloads from pcap files and add them to the corresponding send and receive events.

When tracing syscalls for pairs of events causally related, we intercept the syscall of the first event solely at its entry point and the syscall of the second event only at its exit point. Since ptrace-based tracing utilities do not guarantee that the two interception points of a syscall appear contiguously in the trace, this approach is crucial to correctly infer causality.

The happens-before model generator is written in Java and uses the Z3 solver [22] to solve the model. The causal trace produced by the solver is then output in JSON format.

Falcon's current visualizer is implemented as a JavaScript program that consumes the causal trace and generates a space-time diagram using the SVG.js library. We are currently extending Falcon to use ShiViz [20] as the visualization module, as it provides interactive analysis features.

4.5 Case Study: Apache Zookeeper

Apache Zookeeper [29] is an open-source, scalable and reliable service that enables distributed coordination. Zookeeper poses a good case study for Falcon as its implements complex algorithms and protocols for leader election and atomic broadcast, which are hard to analyze and understand in detail.

[2] Available at `https://github.com/fntneves/falcon/`

In a distributed deployment, Zookeeper runs with several servers, of which one is leader and the others are followers. Both roles are distinguishable in the sense that read requests can be served by the followers while write requests are handled only by the leader. For this case study, using Zookeeper 3.5.0, we analyze a setup containing two Zookeeper nodes that communicate with each other to elect the leader. In particular, our execution scenario consisted of setting up a standalone Zookeeper server and, then, adding a new node to the server quorum.

During the execution, we collected Zookeeper's built-in log file produced with the log4j logging library and used our ptrace-based tracer tool to record syscalls regarding thread synchronization events, connections, and messages exchanges.

As the output layout generated by log4j is configurable, we set the layout parameter to a custom Java class. In order to correctly identify the thread responsible for logging a given message, we augment each log entry with a unique identifier consisting of the concatenation of both the thread and process ids. However, since the Java Virtual Machine does not allow accessing the native thread identifier from a high-level API, we rely on the Java Native Access to execute the gettid() system call and retrieve the thread id directly from the native operating system. The result of the syscall is thus introduced as a parameter in the output layout of log4j.

In the following, we show how Falcon can be used to analyze the execution of Zookeeper and evaluate the performance and scalability of the constraint solving procedure.

4.5.1 Establishing Causal Relationships

Figure 17 depicts the space-time diagram generated by Falcon for the logs collected during our Zookeeper execution scenario. The causal trace was obtained by solving the model with the timestamp soft constraints. Vertical lines are thread timelines, and circles denote the executed events on each thread timeline. The dashed rectangle highlights the leader election algorithm execution. The diagram shows that there are two main processes (5598 and 5670) that spawn several threads while running. For brevity, we include just the thread timelines relevant for this example. In other words, we discarded the threads that have only **START**, **END** and **LOG** events. However, they can be useful for conducting a more thorough behavior analysis.

The **LOG** events correspond to the timestamped entries logged by log4j, while the remaining events were collected by the syscall tracer. To ease the understanding of the diagram, we highlight certain events with a circled number and display their message content (for **LOG** events) or payload (for **SND** and **RCV** events).

Figure 17 shows that, after booting, the server joining the quorum starts by connecting to the existing peer. In particular, the **LOG** events identified by ① and ③ reveal that the process

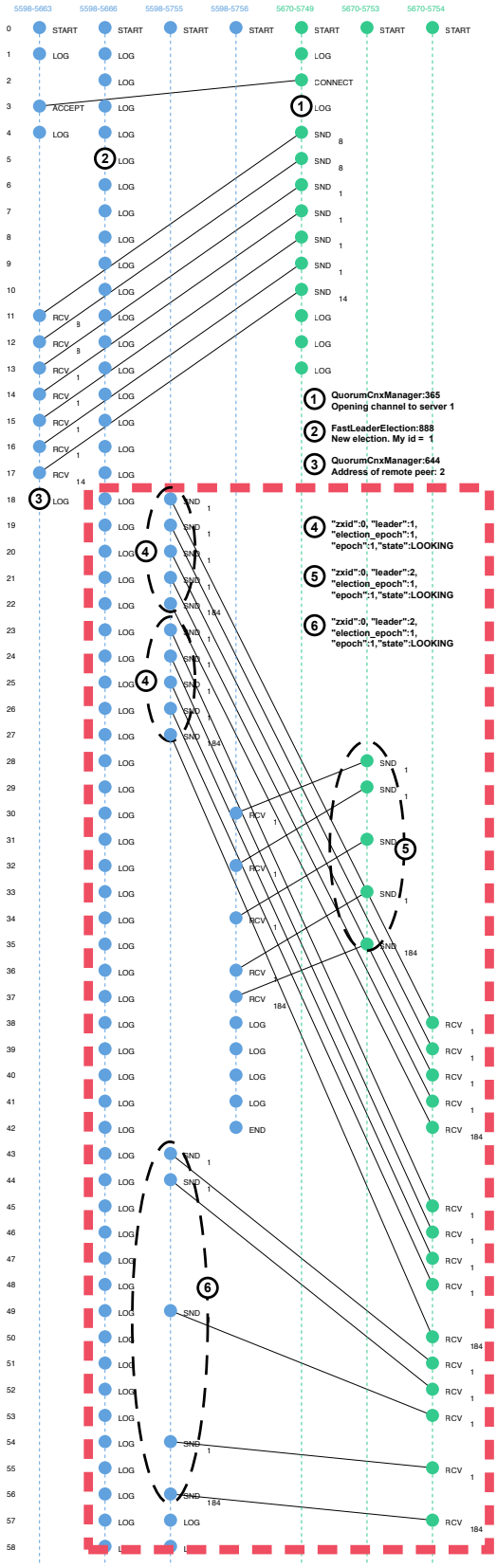


Figure 17: Space-time diagram of two Zookeeper servers connecting to each other and performing a leader election.

5598 is the node with the server identifier (sid) 1 while process 5670 has $sid = 2$. Note that the events also reveal the lines of code at which the messages were logged, namely lines 365 and 644 of the QuorumCnxManager class.

The most interesting part of the diagram is arguably the leader election procedure though, since it is one of the major applications of fault-tolerant consensus protocols. When the two quorum peers are connected, server 1 triggers a new leader election (see event ②). In Zookeeper, each server can be in one of the following states: LOOKING, FOLLOWING, LEADING and OBSERVING. At the beginning of the execution, all servers are in the LOOKING state and vote in themselves to be the leader by sending notifications to the other servers with the $leader$ field set to their sid (see messages ④, ⑤ and ⑥). If a server receives a notification with a sid higher than its own, then it updates its vote proposal to the higher sid and broadcasts the new vote proposal to the rest of the quorum. Note that, since there are no client requests during this execution, the last seen transaction identifier $zxid$ remains unchanged for both servers. Otherwise the servers would vote for the peer with higher $zxid$.

In a nutshell, the diagram of Figure 17 reveals the following leader election protocol in Zookeeper:

- Server 1 sends the vote message ④ with payload $\{``leader'' : 1\}$ to server 2;
- Server 2 sends the vote message ⑤ with payload $\{``leader'' : 2\}$ to server 1;
- Server 1 receives the vote message sent by server 2, updates its vote proposal for the latter because it has higher sid, and sends back the updated vote – message ⑥ with new payload $\{``leader'' : 2\}$ – to server 2;
- Server 1 and server 2 update their state from LOOKING to FOLLOWING[3] and LEADING[4], respectively, as indicated by the log messages.

A further analysis of this space-time diagram also allows drawing some conclusions regarding the Zookeeper's behavior:

Notification timeout The message ④ is sent twice due to a timeout that occurs when a server does not receive enough notifications within a given time frame.

[3] FastLeaderElection:636 - leader=2, ..., my id=1, my state=FOLLOWING
[4] FastLeaderElection:636 - leader=2, ..., my id=2, my state=LEADING

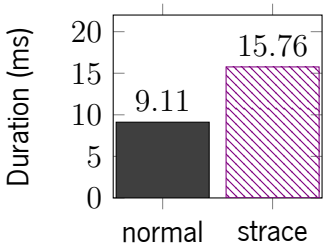


Figure 18: Performance impact on Zookeeper execution caused by tracing with strace.

Message partitioning The sending of a message is actually partitioned into several **SND** events, i.e., to write syscall executions. Inspecting the Zookeeper's source code[5], we noticed that the messages sent during the leader election protocol are composed by an integer and a buffer. The integer is sent by executing the write syscall four times, while sending the buffer requires a single write invocation.

Causality The diagram shows that messages ④ and ⑤ are not causally related, because the sender and receiver threads execute concurrently. In contrast, there is a causal relationship between messages ⑤ and ⑥ based on an state change. Upon reception, message ⑤ is added to a queue by the receiver thread. Afterwards, the sender thread dequeues the message, processes it (i.e., updates the vote proposal of server 1), and sends message ⑥ to server 2. Therefore, ⑤ $\rightarrow$ receiver thread enqueues message $\rightarrow$ sender thread dequeues message $\rightarrow$ change of vote proposal $\rightarrow$ ⑥.

4.5.2 Syscall Tracing Overhead

We developed a micro-benchmark to evaluate the performance overhead imposed on Zookeeper due to tracing syscalls with strace. The micro-benchmark consists of a client issuing requests to two Zookeeper servers. Concretely, the client performs 10K iterations of four operations: i) check whether a znode exists, ii) create a new znode, iii) check again whether the znode exists, and iv) delete the created znode.

Figure 18 compares the average duration of an iteration (in milliseconds) between a vanilla execution of Zookeeper and an execution with strace. As expected, enabling the tracing affects negatively the runtime performance, causing Zookeeper to be $1.7\times$ slower comparing to the baseline.

[5]See class QuorumCnxManager at line 698.

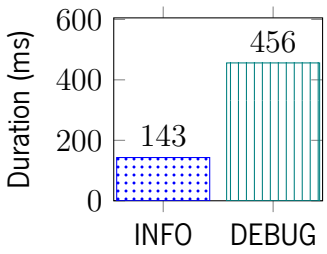


Figure 19: Time spent on solving the generated models for both levels of debugging: INFO and DEBUG.

4.5.3 Scalability of Constraint Solving

Depending on the debugging level, message logs may contain hundreds or thousands of entries. In order to better understand how the constraint solving time varies with an increasing amount of log entries, we ran the same configuration of Zookeeper for INFO and DEBUG logging levels. To further increase the log size, we duplicated the number of entries on both cases log. The resulting logs contained 568 events for the DEBUG level and 342 events for the INFO level.

The time required by Z3 to solve the constraint models for both log files is depicted in Figure 19. The results show that adding the DEBUG-level log events to the model caused the solver to take $3.18\times$ more time to find a solution than with an INFO-level log. Although a trade-off must be made between the duration of the constraint solving and the amount of information logged, we believe that Falcon is useful in practice and much more scalable than manual log analysis.

4.6 Summary

In this chapter we introduced a black-box approach in the form of a log-based analysis tool — Falcon — for causally combining and visualizing log data from several data sources. The key contribution of this approach is the ability to merge log data from multiple sources and establish happens-before relationships between events, providing a coherent diagram for a visual analysis.

The tool is applied to Zookeeper and demonstrated with a syscall trace and log4j log files. The resulting diagram helps in understanding how remote threads interact among themselves and what messages were exchanged to execute a given task. Clearly, the combination of hundreds of events that is ordered by the Satisfiability Modulo Theories (SMT) solver to produce a trace would clearly be unfeasible manually. Additionally, we assess the performance impact on syscall-level tracing in Zookeeper, showing that the impact is tolerable, in line with what is

expected from a more detailed log level configuration in log4j.

Chapter 5

Efficient Causal Log Querying

Modern distributed architectures have dozens of several integrated software components generating tons of log entries. The sheer amount of generated log entries unveils some limitations both in scalability and refinement in our presented approach — Falcon. First, using current state-of-the-art solvers [22], Falcon does not scale to executions with more than a few thousands of events and is not practical to address log volumes generated by production systems [58]. Second, for millions of generated log entries, Falcon does not provide the ability to practically filter nor query the resulting execution trace, which further hinders its ability to troubleshoot real-world bugs.

Alternatively, log management tools, such as the popular Elastic Stack [5], handle considerable amounts of log entries and more importantly, offer powerful filtering, querying and full text search features that aid developers and supporters during the analysis. The downside, when compared to Falcon, is however their lack of causality while ordering log entries. This limitation results in failing at providing a causally-coherent view of the combined log entries, which might turn the analysis into an inconclusive task.

In this chapter, we focus on overcoming the shortcomings of the previous presented approach and building a system that enables the refinement of distributed system logs in a causally-consistent and scalable fashion. Horus is the name of the system we introduce in this chapter. Instead of using state-of-the-art solvers, it assembles a directed acyclic graph which is stored in a graph database, thus allowing the use of rich query languages to reason about runtime behavior.

The case study presented in this chapter with TrainTicket – an open-source ticket booking application with 40+ microservices – shows that Horus surpasses the log management system Elastic Stack in pinpointing the root cause of an anomaly in a distributed execution. Additionally,

we show that Horus builds a causally-consistent log of a distributed execution with much higher performance (up to 3 orders of magnitude) and scalability than our prior approach, Falcon.

5.1 Overview

Horus is a system that addresses the limitations of prior log analysis solutions to further ease the burden of debugging distributed executions. To this end, Horus provides the following key features:

- **Lightweight and non-intrusive causality tracking.** Horus does not require the developer to manually instrument the code with tracing points with context metadata in order to track causality. Instead, it uses kernel probes (e.g. eBPF) to efficiently capture low-level operations that can later be used to define a partial order of log messages that respects causality. For example, eBPF probes allow recording socket send and receive operations, which are connected by a happens-before relation.

- **Causally-consistent aggregation of distributed events.** Horus combines both kernel events and logging events from different sources into a single directed, acyclic graph of the production run. In this graph, nodes represent the execution events and edges represent the happens-before (HB) relations between them. Intra-node HB relations are encoded using the original event timestamps, whereas inter-node HB relations are encoded using the kernel-level event causality rules.

- **Execution analysis and refinement via rich querying.** Casting the causal aggregation of multiple logs of a distributed execution as a graph generation problem gives Horus the ability to leverage off-the-shelf third-party graph databases[10, 6], which not only scale to executions with millions of events, but also provide support for rich query languages.

To better understand how Horus operates, we depict its architecture and execution flow in Figure 20. As shown in the figure, Horus comprises four main components: Event Sources, Event Queue, Event Processor, and Graph Database. Each component can be described as follows.

Event Sources This component represents the set of heterogeneous and independent sources that produce events at runtime. To handle different types of event sources, Horus requires the

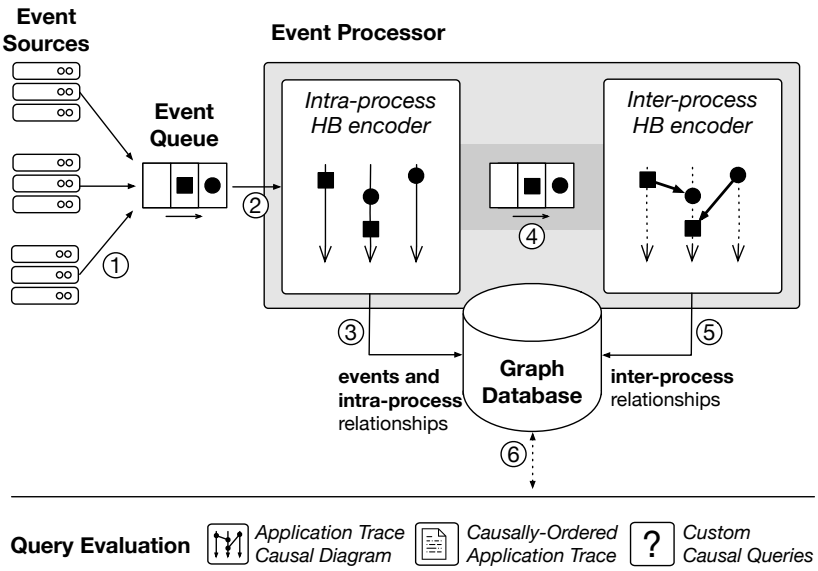


Figure 20: Horus architecture and event flow.

existence of adapters, which are responsible for collecting the data, normalizing it into a Horus-compatible format, and shipping it to the Horus backend for further processing (see ① in Figure 20). Our current prototype provides adapters to automatically collect events from:

- Log4j [2]. The adapter for Log4j consists of a simple formatter which outputs log messages as JavaScript Object Notation (JSON) objects indicating the timestamp, the name of the process/thread, and the textual message written in the source code. Each log message is thus considered an independent event.

- I/O and Process System Calls. The adapter for I/O and process system calls reuses eBPF probes from Falcon to trace events regarding: the start, end, fork, join of a process or thread; the request and accept of a network connection between two processes; and the sending and receiving of a message. eBPF allows implementing efficient and safe programs that run inside the kernel for multiple purposes [3], namely I/O analysis, performance, monitoring, security and tracing.

We plan to extend Horus with adapters for additional logging libraries in the future, such as Logrus [8].

Event Queue This component abstracts a set of persistent, distributed, and replicated queues within Horus that keep events waiting for being processed. In detail, Horus maintains two types

of event queues: one responsible for storing the stream of events coming from the sources (see ①), and another responsible for linking the different stages for building the causal graph (see ④). The current prototype uses Apache Kafka [1] to manage the event queues.

Event Processor This component is the processing core unit of Horus and aims at generating the execution causal graph. To this end, the Event Processor comprises two sub-components that operate as a two-stage pipeline. In the first stage, the Intra-Process HB Encoder establishes the HB relations between events of the same process, which are then periodically persisted in the graph database (see ③) and forwarded to the next processing stage (see ④).

In the second stage, the Inter-Process HB Encoder encodes the HB dependences between events of distinct processes, which are then flushed to the database in periodic batches like in the previous step (see ⑤). Section 5.2 describes the causal graph generation procedure in more detail.

Graph Database This component is responsible for storing the execution causal graph and computing the developer's queries at debugging time (see ⑥). The current prototype of Horus uses Neo4j [10], a graph database that provides a rich query language named Cypher for filtering, refining, and visualizing a graph.

Although Neo4j is capable of handling graphs with millions of events, our experiments have shown that the time it takes to process causality queries on an execution graph does not scale well with the size of the graph. In Section 5.3, we show how Horus addresses this issue with a novel technique that leverages logical time to prune irrelevant portions of graphs during query computation.

5.2 Causal Graph Generation

This section details how Horus builds the causal graph of a distributed execution. As mentioned before, the nodes of this graph denote execution events (i.e., log messages and kernel operations) and the edges indicate the happens-before relationships between them. More concretely, there are two types of edges: intra-process and inter-process.

Algorithm 1 Infer intra-process causal relationships of incoming events

```
1:  processesEvents : set of (processId, set of events ordered by timestamp)
2:  latestProcessesEvent : set of (processId, event)
3:  causalGraph : set of (eventNodes, hbEdges)
4:  processedEvents : set of event
5:
6:  procedure processEvent(event)
7:      processId = getProcessId(event)
8:      processesEvents.insert(processId, event)
9:
10:     ensure flushIntraProcessData runs periodically.
11: end procedure
12:
13: procedure flushIntraProcessData
14:     for each (processId, events) ∈ processesEvents do
15:         latestProcessEvent = latestProcessesEvent.getEvent(processId)
16:
17:         for each event ∈ events do
18:             causalGraph.insertNode(event)
19:
20:             if latestProcessEvent is not nil then
21:                 causalGraph.insertEdge(latestProcessEvent, event)
22:             end if
23:
24:             latestProcessEvent = event
25:             processedEvents.insert(event)
26:         end for
27:
28:         processesEvents.put(processId, {})
29:         latestProcessesEvent.put(event, latestProcessEvent)
30:     end for
31:
32:     merge causalGraph with existing graph in graph storage
33:     send processEvents to the next stage
34:     processedEvents.clear()
35: end procedure
```

5.2.1 Intra-Process HB Relationships

In the first stage of Horus' event processing pipeline, the Intra-Process HB Encoder computes the happens-before relationships between the subsets of events belonging to the same process. As stated by the first property of the definition of happens-before, these relationships can be derived directly from the program order.

In practice, logging libraries already assign timestamps to the log messages with the purpose of later easing their analysis. This means that, for the same process, it suffices to order the messages by timestamp in order to obtain the causal ordering of events at runtime.

However, in scenarios with multiple independent loggers and tracers, the same process may trigger those tools without any kind of synchronization, thus permitting the resulting events to arrive out of order at the Event Queue component. For that reason, Horus requires the timestamp source adopted by those tools (e.g., physical clock) to be the same and accurate enough to define a total order of events across them. Otherwise, the program order property ceases to hold.

The pseudo-code of the task performed in this stage is presented in Algorithm 1. To handle events coming from several sources, the Intra-Process HB Encoder maintains process timelines, in which it maps each process' identifier with its set of generated events (see $processesEvents$). Each event set is a sequence of events, ordered by timestamp, that were executed by the same process. For each incoming $event$, the Intra-Process HB Encoder is responsible for inserting it into the corresponding process timeline in the correct position based on its timestamp. This procedure guarantees that, for a given process, events enqueued out of order will still produce a causally-consistent timeline.

After the first event it processes, the Intra-Process HB Encoder starts to regularly flushing the process timelines to the graph database (see $flushIntraProcessData$. For each timeline, the Intra-Process HB Encoder creates a new node per event and encodes intra-process edges between pairs of nodes that correspond to contiguous events in the timeline. As an example, let us consider a timeline $T : \{A, B, C, D\}$ with four events sorted in ascending order of their timestamp. Upon persisting T into the database, the Intra-Process HB Encoder creates four nodes – A, B, C, D – and three edges – $(A, B), (B, C), (C, D)$.

Alongside, the Intra-Process HB Encoder sends the timeline events to the queue of the next processing stage.

Algorithm 2 Find event matches of incoming SND event

```
1:  pendingSnd : map of (channelId, queue of pending SND events)
2:  pendingRcv : map of (channelId, queue of pending RCV events)
3:
4:  procedure findRcvMatches(sndEvent)
5:      matchesFound : set of RCV events ordered by timestamp
6:      rcvCandidates = pendingRcv.getEvents(sndEvent.channelId)
7:
8:      while sndEvent.remainingBytes > 0 && rcvCandidates.size > 0 do
9:          currentRcv = rcvCandidates.peek()
10:         sndRemainingBytes = sndEvent.remainingBytes
11:         rcvRemainingBytes = currentRcv.remainingBytes
12:         matchedBytes = min(sndRemainingBytes, rcvRemainingBytes)
13:
14:         sndEvent.remainingBytes− = matchedBytes
15:         currentRcv.remainingBytes− = matchedBytes
16:
17:         if currentRcv.remainingBytes == 0 then
18:             matchCandidates.dequeue()   ▷ remove the event from the pending set.
19:         end if
20:
21:         matchesFound.add(currentRcv)
22:     end while
23:
24:     if sndEvent.remainingBytes > 0 then
25:         pendingSnd.enqueue(channelId, sndEvent)      ▷ add to the pending set.
26:     end if
27:
28:     return matchesFound
29: end procedure
```

5.2.2 Inter-Process HB Relationships

In the second processing stage, the Inter-Process HB Encoder computes the happens-before relationships between events of two different processes.

Inter-process causality stems from the second property of the happens-before definition and, in contrast to intra-process causality, does not rely on timestamps. Instead, it relies on a message m that unequivocally identifies that its departure from a process causally-precedes its arrival at another process. In practice, m is usually either a unique message identifier or a piece of context data that indicates the causal relation.

In the current version of Horus, the Inter-Process HB Encoder determines inter-process causal dependencies based on the attributes of the events captured by eBPF kernel probes — the events captured are the same as in Section 4.2. For instance, in a TCP connection scenario, events of sending (SND) and receiving (RCV) bytes include attributes concerning the source and destination IP addresses and ports. These attributes are suitable for generating a stream channel identifier (i.e., channelId). Considering the TCP delivery and ordering guarantees, one can establish $SND \rightarrow RCV$ causal pairs for each connection channel. One can also define the following causal pairs of events between a parent process p and a child process c: $FORK_p \rightarrow START_c$ and $END_c \rightarrow JOIN_p$. The same can be applied to the other causal pairs as mentioned in Section 4.3.

The Algorithm 2 describes how Horus finds the event matches for an incoming SND event. Specifically, the procedure $findRcvMatches$ is responsible to return the event(s) that match the incoming SND event, considering its channel stream and the remaining unmatched bytes. The logic used in this algorithm also applies to find RCV matches. In this algorithm, we resort to event's attributes with the source and destination IP addresses and ports to generate the $channelId$, as well as the $size$ of the message, which is useful to keep track of the unmatched bytes and whether an event is fully matched or not.

The Inter-Process HB Encoder integrates procedures similar to the aforementioned to process other causal pairs. It processes the incoming events processed in a stream fashion, building inter-process causal dependencies according to the nature of each event. To improve performance, incomplete causal pairs are kept in memory until the corresponding pairing event is consumed from the queue, e.g., $pendingSnd$ and $pendingRcv$ in Algorithm 2.

Similarly to the Intra-Process HB Encoder, this processor also periodically flushes the complete causal pairs by inserting a new edge per pair into the graph database. Once the causal pairs are persisted, the events exit the processing pipeline and become available for analysis.

We note that the Inter-Process HB Encoder can be extended with new causality rules based

on event attributes. This feature enriches Horus with the ability to accommodate arbitrary types of events and happens-before dependencies into the execution causal graph.

5.3 Efficient Causal Graph Querying

Once the causal graph is built and stored in the graph database, the developer can start focusing on the error's root cause. In Horus, this is achieved by refining the causal graph using the database's built-in query language, as it provides filtering and grouping operations for graph algorithms.

In general, the analysis of the causal execution graph comprises two main types of queries: Q1) May event a causally affect event b?; and Q2) What is the causal path between a and b?

Q1 is the fundamental query for evaluating the happens-before relation between any two events a and b. In turn, Q2 aims at extracting the sub-graph of causal events occurring in-between them, which is particularly useful to reconstruct the causal trace of a given request.

These queries can be expressed with path discovery and traversal operations that are provided by the graph database. More concretely, one can answer Q1 with a shortest path algorithm, and answer Q2 by finding all paths between the two events. Unfortunately, as these built-in traversals are oblivious to the semantics of distributed systems, they are inefficient for causal queries and cannot scale for large graphs. To understand why this is the case, let us consider the causal graph depicted in Figure 21 with three process timelines ($P1$, $P2$, and $P3$). Nodes denote events along their execution, and edges their causal dependencies. The logical time assigned to each event is depicted on the left and right sides, according to logical clocks and vector clocks. Thick-edged nodes (respectively, shaded) indicate the nodes explored by a default graph database traversal (respectively, Horus) to compute the causal path between C and F.

Assume that we want to obtain the events that happened between C and F, which is a query of type Q2. A graph database's default traversal would correctly yield the result set $\{C, D, E, F\}$, although at the expense of visiting the nodes $\{C, D, E, G, F, H, I, L, M, K\}$ (represented in Figure 21 with a thick border). Observing the figure, one can notice that this approach is far from optimal, as, for example, nodes I and J are visited despite being clearly irrelevant to the query because they occur after F. The reason behind this inefficiency is that the traversal performed by the database does not take into account the notion of time nor the directed acyclic properties of the causal graph.

To address this limitation and speed up query processing, Horus augments the causal graph

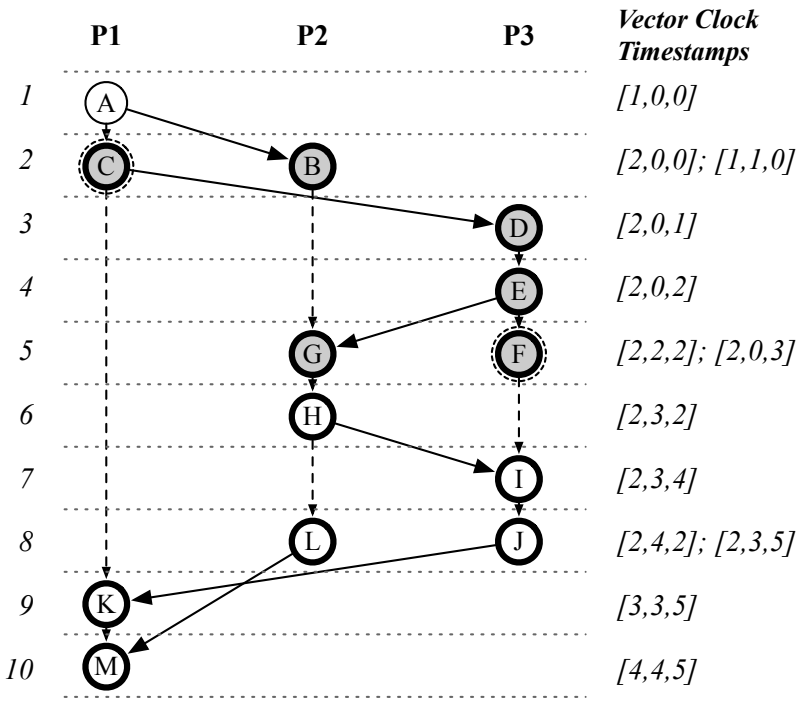


Figure 21: Example causal graph involving three distributed processes.

with logical timestamps, namely logical clocks (LC) [33] and vector clocks (VC) [25].

Logical clocks are scalar values assigned to each event such that, for every two events a and b, the following condition holds: $a \rightarrow b \implies LC(a) < LC(b)$. In Figure 21, logical clocks are depicted as the timestamps on the left. For example, $B \rightarrow G \implies LC(B) = 2 < LC(G) = 5$. Note, however, that the reverse implication is not guaranteed: $LC(C) = 2 < LC(G) = 5$, but C does not happen before G. In fact, logical clocks alone are not enough to reason about event causality.

In contrast, vector clocks suffice to reason about the causal ordering of events in a distributed execution by providing the property: $a \rightarrow b \iff VC(a) < VC(b)$. We represent vector clocks in Figure 21 next to each event: for example, $C \rightarrow D \iff VC(C) = [2,0,0] < VC(D) = [2,0,1]$. In turn, for events C and G, since neither $VC(C) < VC(G)$ nor $VC(G) < VC(C)$ holds, we can conclude that they are concurrent.

Horus performs a graph traversal similar to topological sorting to assign both logical and vector clocks to each event in the graph. These logical timestamps are then used to speed up querying in a twofold fashion, as described below.

Let $G = (V, E)$ be the causal graph under analysis and $a, b \in V$ the start and end event

of a query, respectively.

First, Horus leverages the logical clocks of a and b to quickly bound the portion of the graph that is relevant to the query. In practice, this corresponds to using the logical clocks as database indexes and compute the following over-approximation V' of the nodes in the result set: $V' = \{v \in V \mid LC(a) <= LC(v) <= LC(b)\}$. For the example of computing the causal path between C and F considered earlier in this section, this step produces the following subset of nodes $\{C, B, D, E, G, F\}$ (shaded nodes in Figure 21). Note that VCs would also allow computing V', however they are inappropriate for graph database indexing due to their non-scalar nature.

Second, Horus uses vector clocks to prune out the events in V' that are concurrent to a and b. This operation yields a sub-set V'' containing only the events in the causal path between a and b: $V'' = \{v \in V' \mid VC(a) < VC(v) < VC(b)\}$. For the previous example, this step will discard events B and G and leave only the sub-set $\{C, D, E, F\}$, which is the expected one.

Finally, Horus computes the result set of the query by collecting the edge set E' with the connections between the nodes in V'': $E' = \{e_{x \to y} \in E \mid x \in V'' \wedge y \in V''\}$.

In summary, Horus leverages logical time to answer the two types of refinement queries as follows.

Q1. $VC(a) < VC(b)$;

Q2. $CausalPath_{a \to b} = (V'', E')$, where:

$$V' = \{v \in V \mid LC(a) <= LC(v) <= LC(b)\}$$
$$V'' = \{v \in V' \mid VC(a) < VC(v) < VC(b)\}$$
$$E' = \{e_{x \to y} \in E \mid x \in V'' \wedge y \in V''\}$$

5.4 Case Study: TrainTicket

TrainTicket [60] is an open-source benchmark developed to foster research on fault analysis and debugging of distributed systems, namely those based on microservices. It consists of a ticket booking application with multiple functionalities: ticket inquiry, reservation, payment, order updates, and user notifications. TrainTicket's architecture comprises 40+ microservices written in different programming languages (e.g., Java, Nodejs, Python, Go), along with a user interface and third-party components, namely MongoDB for data storage and RabbitMQ for message queueing.

We use the TrainTicket in this work because it mimics a real-world, complex distributed system and already contains 22 representative faults collected from a recent industrial survey on microservice applications [60]. Each fault was injected into an independent source code snapshot, available at a public repository.[1] For this motivating example, we use the one labeled as F13, as it represents an order violation caused by a message race, which is a common type of distributed concurrency bug [34].

The F13 fault The F13 fault results from the interleaved processing of two messages arriving concurrently at the system. In TrainTicket, each order has a property representing its current state. When an order is created, it has state `UNPAID`. The order state can then migrate to either `PAID` or `CANCELED`, according to whether the client issues a Payment Order or a Cancel Order, respectively.

The Payment Order succeeds when both the following conditions hold: i) the client has enough funds, and ii) the order state transition from `UNPAID` to `PAID` is valid. Otherwise, the request fails, and the order state remains unchanged. In turn, the Cancel Order succeeds if the state can be set to `CANCELED`.

Since both requests change the state of an order and the TrainTicket application processes them without synchronization guarantees, their concurrent execution may non-deterministically lead to a payment failure.

The test driver for replicating this error in TrainTicket consists in a client application, Launcher, that first books a train trip with a new client account and, then, issues concurrent Payment Order and Cancel Order requests. When the payment failure occurs, TrainTicket renders a page outputting an error message.

5.4.1 Debugging F13 with Elastic Stack

The increasing amount of applications based on microservices has motivated the industry to adopt toolsets tailored for distributed log monitoring. Among those toolsets, the Elastic Stack [5] is arguably one of the most popular and widely used. It provides support for collecting large sets of log data from multiple sources (via Logstash), analyze them using queries and filters (via Elasticsearch), and creating user-friendly visualization dashboards (via Kibana).

However, the Elastic Stack does not guarantee that log data is causally ordered, which hinders its effectiveness to diagnose unexpected behavior in distributed systems. In this section, we demonstrate such limitation by using it to diagnose the F13 fault.

[1] `https://fudanselab.github.io/research/MSFaultEmpiricalStudy`

```
1  [Launcher-1] [Reservation Result] Success
2  [Payment-1.1] - [URI:/pay][Request: {"orderId":"652aaf9b"}]
3  [Order-1.1] - [URI:/getById][Request: {"orderId":"652aaf9b"}]
4  [Order-1.2] - Response: {"status":true, order":{"id":"652aaf9b", "
       status":"UNPAID"}}
5  [Payment-1.2] - Response: "false"
6  [Cancel-1.1] - [URI:/cancelOrder][Request: {"orderId":"652aaf9b"}]
7  [Order-2.1] - [URI:/getById][Request: {"orderId":"652aaf9b"}]
8  [Order-2.2] - Response: {"status":true, order":{"id":"652aaf9b", "
       status":"CANCELED"}}
9  [Payment-2.1] - [URI:/drawBack][Request: {"userId":"c01d7008"}]
10 [Payment-2.2] - Response: "true"
11 [Cancel-1.2] - Response: {"status":true, "message":"Success."}
12 [Launcher-2] java.lang.RuntimeException: [Error Queue]
```

Figure 22: The set of log records aggregated by Elastic Stack for the failed payment request. Records are ordered by the timestamp assigned at the moment they were collected. To ease readability, we label each record with the service in which it occurred and a local counter.

For this experiment, we deployed the TrainTicket benchmark on a cluster of three n1-standard-8 Google Cloud Engine instances managed by Docker Swarm. In addition, we set up a Docker container running the Elastic Stack toolset to collect and aggregate the logs produced by the TrainTicket microservices. More concretely, we placed a Filebeat daemon on each instance to continuously send container log messages (application logs augmented with container monitoring data) to Logstash, which is the event processing component of the Elastic Stack. Finally, we ran the F13 test driver until observing a failing execution and collected the corresponding logs captured by the toolset.

In the following, we describe how a developer would typically use Elastic Stack for debugging the error observed in the experiment, using an fictional character named Steve.

Steve is a software engineer at the company that owns TrainTicket and has been assigned to fix an issue ticket reporting an intermittent error related to payment requests.

Steve starts by inspecting the logs produced by the Launcher service, as it was the service in which the unexpected behavior surfaced. Using the container information present in the logs, Steve is able to isolate the services that processed the request and find the events that delimit the payment request. Having identified the portion of the logs relevant to the analysis, he finally gathers the log messages that will hopefully reveal the root cause of the failure.

Figure 22 unveils the subset of log messages, ordered by timestamp, that Steve relies on to reason about the execution path that led to the error at line 12. Each log statement begins with

the corresponding service's name where it was collected. The events logged by the Launcher service (lines 1 and 12) delimit the portion of the execution that comprises the failure. The log messages in-between (lines 2-11) were recorded by three other services, specifically Payment, Order, and Cancel, and provide applicational context such as the identifier and the status of the order at any given moment.

By inspecting the logs, Steve realizes that, after completing the reservation step, the client sent a payment request (line 2), followed by a cancellation request for the same order (line 6).

Throughout the processing of the payment request, the logs show that the Payment service fetched the details of the order from the Order service (lines 3 and 4). At that moment, the order's state was `UNPAID` – a valid state for transitioning to `PAID` (line 4). Yet, the request later returned `false` (line 5), thus indicating that the payment had failed to complete.

In turn, the cancel request completed successfully (line 11), as it managed to set the order's state to `CANCELED` (lines 7 and 8) and issue a refund through the Payment service (lines 9 and 10).

The fact that the order's state was valid for the payment request (i.e., the response in line 4 had status `UNPAID`) leads Steve to believe that the reason for the payment failure was insufficient money. However, after carefully reviewing the account balance, he verifies that there were enough funds to pay for the reservation order.

Steve thus concludes that there is surely some other factor causing the error, although from the logs it is not clear what that factor might be. Moreover, as the failure is non-deterministic, Steve is unable to obtain additional details about the problem simply by re-executing the application. Consequently, he marks the ticket as "cannot reproduce" and the issue remains unresolved. Hopefully, it will not surface in a near future, he thinks.

This cautionary tale aims at showing that the lack of causality in toolsets like Elastic Stack often renders the analysis of distributed system logs an inconclusive task, which further contributes to the lingering of harmful bugs in production.

5.4.2 Debugging F13 with Horus

In this section, we demonstrate how Horus' approach overcomes the limitations of toolsets like Elastic Stack and allows troubleshooting the F13 error.

For collecting TrainTicket's events at runtime, we set up Horus with two event sources, as follows. First, we configured Horus' kernel-level tracer to attach details about the Docker containers in which TrainTicket's services run. The goal was to later allow uniquely identifying

Event Type	Occurrences (approx. %)
LOG	4,531 (22.52%)
RCV	4,339 (21.57%)
CREATE	3,618 (17.99%)
START	3,340 (16.60%)
SND	2,689 (13.37%)
END	660 (3.28%)
JOIN	357 (1.77%)
CONNECT	260 (1.11%)
FSYNC	173 (0.86%)
ACCEPT	149 (0.74%)
Total Events	**20,116**

Table 4: Number of events per type present in the causal graph under analysis. LOG events are produced by the log4j logging library, while the rest is captured by Horus' kernel-level tracer.

each individual host in the system. Second, we enabled a new log4j appender for attaching useful process information to TrainTicket's log messages before forwarding them to Horus. We note that, on each host, both event tracers rely on the same monotonically increasing physical clock to later ensure the correct encoding of the intra-process causality.

Table 4 reports the amount of events, grouped by type, captured by Horus when building the causal graph for TrainTicket's F13 failing execution. Overall, the causal graph available for analysis contains 20,116 events, spread across 96 process timelines, and 27,859 causal relationships, from which 4,593 (~16.49%) encode inter-process causality. Despite LOG being the most common event type, SNDs and RCVs together, which are essential for encoding inter-process causality, account for around 35% of the graph. The discrepancy between the percentage of SND and RCV events is explained by different buffer sizes on the hosts, which can cause a single message to be read by multiple partial RCVs.

In the following, we revisit the cautionary tale from Section 5.4.1 and describe how Steve, our fictional TrainTicket's software engineer, would use Horus to analyze and debug the payment failure report in the issue tracker.

Steve starts the day with an issue ticket on his hands, reporting that an error popped up when a user performed the payment request for order 652aaf9b. Fortunately, the user has also

```
// Find events that denote the beginning of the payment request and the noticed error.
MATCH
  (reqSnd:SND {host: 'Launcher'})-->(:RCV {host: 'Payment'}),
  (reqError:LOG {host: 'Launcher'})
WHERE
  reqError.message CONTAINS 'java.lang.RuntimeException: [Error Queue]'
  reqError.lamportLogicalTime > reqSnd.lamportLogicalTime
WITH
  reqSnd.lamportLogicalTime as reqSndTime,
  min(reqError.lamportLogicalTime) as reqErrorTime

MATCH
  (reqSnd:EVENT {host: 'Launcher', lamportLogicalTime: reqSndTime}),
  (reqError:EVENT {host: 'Launcher', lamportLogicalTime: reqErrorTime})

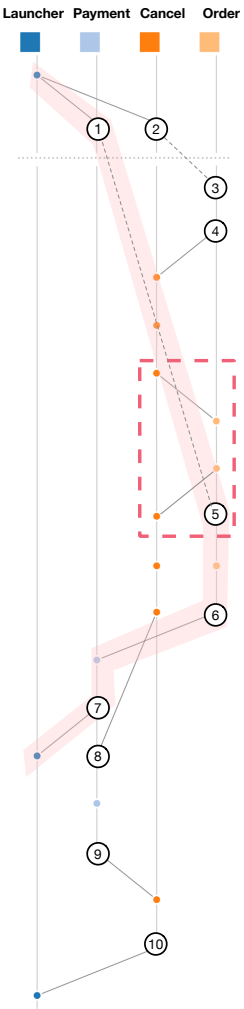
// getCausalGraph(startNode, endNode, onlyLogs)
CALL horus.getCausalGraph(reqSnd, reqError, true) yield node
WITH reqSnd, reqError, node ORDER BY node.lamportLogicalTime ASC
WITH
  reqSnd.eventId as startEventId,
  reqError.eventId as endEventId,
  collect(node) as logs

// Return:
// 1. startEventId: the event that denotes the start of the payment request
// 2. endEventId: the event that denotes the noticed error
// 3. logs: log messages containing the order identifier.
UNWIND logs as log
WITH startEventId, endEventId, log
WHERE log.message CONTAINS '652aaf9b'
RETURN startEventId, endEventId, collect(log.message) as logs
```

a) Query to retrieve the events that denote the beginning of the payment request concerning the order '652aaf9b' and the noticed error. The returned result contains the events' identifiers and the logs containing the order identifier.

```
    // startEventId: hipster-1.europe-west2-c.c.horus-262311.internal64523
    // endEventId: ts-launcher-LOG-584277764938
(1) [Payment-1.1] - [URI:/pay][Request: {"orderId":"652aaf9b"}]
(2) [Cancel-1.1] - [URI:/cancelOrder][Request: {"orderId":"652aaf9b"}]
(3) [Order-1.1] - [URI:/getById][Request: {"orderId":"652aaf9b"}]
(4) [Order-1.2] - Response: {"status":true, "order":{"id":"652aaf9b", "status":"UNPAID"}}
(5) [Order-2.1] - [URI:/getById][Request: {"orderId":"652aaf9b"}]
(6) [Order-2.2] - Response: {"status":true, "order":{"id":"652aaf9b", "status":"CANCELED"}}
(7) [Payment-1.2] - Response: "false"
(8) [Payment-2.1] - [URI:/drawBack][Request: {"userId":"c01d7008"}]
(9) [Payment-2.2] - Response: "true"
(10) [Cancel-1.2] - Response: {"status":true, "message":"Success."}
```

b) Causally-ordered log messages from the moment the request starts to the moment the error is noticed for order 652aaf9b. Each log statement is prefixed with its originating service's name and its thread's local counter.



c) Space-time diagram rendered by ShiViz for the failed payment request.

Figure 23: Refinement queries, written in CypherQL, used to debug the F13 failure, along with a snippet of the ShiViz diagram generated for the causal path produced by Horus after processing the queries.

provided the error message in the ticket description: `java.lang.RuntimeException: [Error Queue]`.

Steve decides to inspect the logs recorded from the moment the payment request started until the moment in which the error message appeared. To this end, he has to first identify the events in the causal graph that delimit that relevant portion of the execution.

The beginning of the payment is defined by a message sent from the Launcher service to the Payment service. Steve knows that, in Horus, this message is represented by the first

$SND_{Launcher} \rightarrow RCV_{Payment}$ causal pair. As such, he is able to obtain the logs concerning the failing request by executing a query to compute the causal path from that $SND_{Launcher}$ event to the $LOG_{Launcher}$ event containing the message `java.lang.RuntimeException: [Error Queue]`. Also, to ensure that he is focusing on the right request, Steve augments the query with an additional clause stating that the START event of the payment request must happen before (in terms of logical time) the error event.

Figure 23a details the aforementioned CypherQL query to refine the causal graph for the failing request under analysis. The query is composed of three parts, identified by the three comment blocks in Figure 23a. The first part filters the events that either belong to the beginning of a payment request or represent an error message. In other words, this part aims at finding the potential boundaries of the failing request.

The second part computes the causal paths that connect the events returned in the previous step and collects the log messages for each one of them. Note how Horus extends Neo4j with the procedure `getCausalGraph()` for efficiently extracting the causal path between two events in the graph.

Finally, the third part of the query aims at filtering the events belonging solely to the order 652aaf9b. The final output of the query is depicted in Figure 23b. Note that the log statements in the figure are the same as those in Figure 22, but this time the events are causally ordered.

After executing the query, Steve proceeds with the analysis by inspecting the log messages. At this point, he notices that the order status changed from `UNPAID` (line 4) to `CANCELED` (line 6). The only valid state transition after a payment request is from `UNPAID` to `PAID`. However, the logs show that the order state transits unexpectedly from `UNPAID` to `CANCELED`, which is an invalid final state for a payment request. On the other hand, Steve realizes that, in addition to the payment request (line 1), the logs also report a cancellation request (line 2) for the same order. Consequently, he suspects that the failure might be related to the concurrent execution of both requests.

To clarify the suspicion that the cancellation request was the culprit of the invalid state change, Steve decides to render the execution causal graph in ShiViz [20], a popular space-time diagram visualizer compatible with Horus.

Figure 23c illustrates the diagram rendered by ShiViz. It comprises four process timelines regarding the Launcher, Payment, Cancel and Order services, respectively. Each thick-bordered number identifies the corresponding LOG event in Figure 23b. The execution path of the payment request under analysis is highlighted in red.

In the diagram, Steve observes that log messages ① and ② appear at the same level, which

confirms that both payment and cancellation requests started concurrently. Since both requests depend on the order's status, the diagram shows each one issuing a `getById` request to the Order service (③ and ⑤). However, while the response sent to the Cancel service indicated the status `UNPAID` (④), the one sent to the Payment service revealed that the order status was `CANCELED` (⑥). Steve thus confirms his suspicion: somehow the order state changed between events ④ and ⑤.

From a closer look at the diagram, he identifies that, after receiving the order details, the Cancel service issued another request back to the Order service, this time to update the order state to `CANCELED` (see the red dashed box). Since this request executed right before the arrival of the Payment's request, the order was already canceled at the time of the fetching. However, the application requirements state that users cannot pay orders that are already canceled.

Steve was finally able to explain the root cause of the failure: as the order status was changed after the payment had started, the payment request received the response false (⑦), which then raised an exception in the Launcher service. The fact that the TrainTicket permits a payment request to be performed after an order cancellation represents an order violation, a common type of distributed concurrency bugs.

To fix this incorrect behavior once for all, Steve implements a synchronization mechanism to enforce the correct execution between payment and cancellation requests. It took a full day of work, but he successfully managed to mark the issue ticket as "resolved" and accomplish his goal.

5.5 Performance Evaluation

In this section, we conduct an experimental evaluation of Horus aimed at assessing the benefits and limitations of our system with respect to: i) the scalability of the event processing pipeline; ii) the performance of the logical time assignment algorithm, and iii) the performance impact of leveraging logical time during query processing.

To enable experiments with causal graphs of different sizes, we implemented a synthetic event generator that mimics an arbitrary number of rounds of a synchronous client-server scenario. In detail, it generates request-reply interactions between two processes P1 and P2 by creating the causal pairs $\text{SND}_{\text{P1}} \rightarrow \text{RCV}_{\text{P2}}$ and $\text{SND}_{\text{P2}} \rightarrow \text{RCV}_{\text{P1}}$. The output of this micro-benchmark is thus an execution causal graph with N events and $3N/2 - 2$ edges in total (comprising both intra- and inter-process edges).

In the rest of this section, we describe the environment setup and discuss the results for

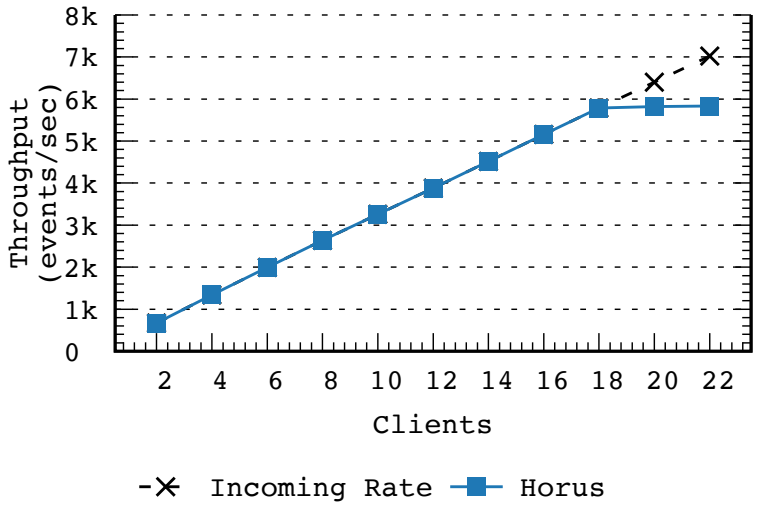


Figure 24: Horus throughput as the number of clients increases. The incoming event rate reveals the amount of events produced by running clients.

each experimental scenario.

5.5.1 Event Processing

Event processing consists of determining intra- and inter-process causality for each event and storing the event and its corresponding causal relationships in the graph database.

In this experiment, we evaluate the scalability of this process in terms of the number of events per unit of time that Horus is able to process.

We configured two instances in Google Cloud. The first instance is of type n1-standard-16 (16 vCPUs and 60GB RAM) and hosts the Horus' pipeline. The other instance, n1-standard-8 (8 vCPUs and 30GB RAM), hosts event generator clients. Each experiment ran for 15 minutes, with clients generating and submitting events to Horus.

Figure 24 illustrates the evolution of the event processing throughput as the amount of clients increases. The dashed line indicates the incoming event rate measured in Apache Kafka. Horus follows the incoming rate until the 18 clients scenario, which produce close to 6,000 events/second.

When Horus reaches its maximum throughput, pending events still remain in the event queues. Therefore, Horus will still be able to process all events generated during workload peaks and make them available for analysis, even with delay.

Horus architecture, however, allows for scale-out of causality encoders within the pipeline.

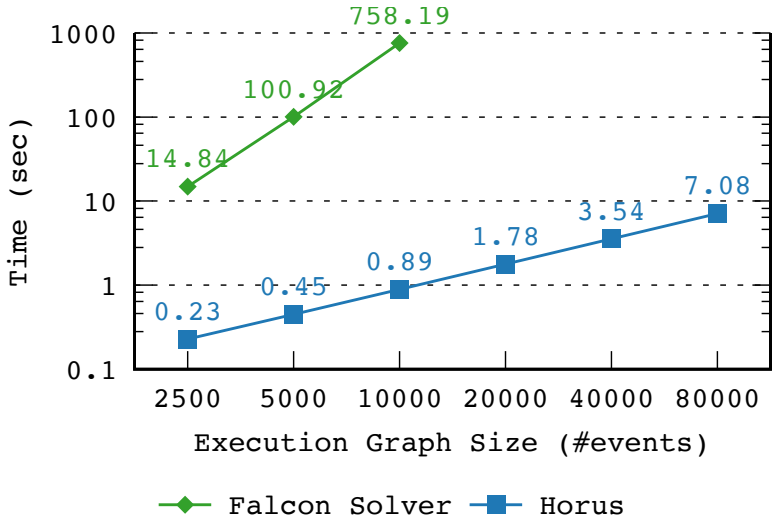


Figure 25: Comparison between the execution time of Falcon and Horus to assign logical clocks on different graph sizes.

To achieve this, one must configure Horus to ensure the following: i) all events in a process are processed by the same intra-process causality encoder, thus preserving the program order; ii) the events belonging to a causal pair are delivered to the same inter-process causality encoder; and iii) the program order is preserved from the intra-process encoder to the inter-process encoder. This allows distributing event processing among several encoders without requiring synchronization and stills guarantee correctness in constructing the causal execution graph.

In short, this experiment shows how many events can be handled in real-time with a single event-processing server, while knowing that, even when this rate is exceeded momentarily, that no events are lost.

5.5.2 Logical Time Assignment

Horus introduces an algorithm based on graph traversal to assign logical time to events in the stored causal graph.

In this experiment, we compare the performance of the Horus' algorithm with the Falcon's approach that, in turn, resorts to a state-of-the-art SMT constraint solver to causally order logs. We first populated the Horus database using the synthetic event generator. Then, we exported the unordered events in the format compatible with the Falcon's solver.

Figure 25 illustrates the evolution of the execution time of the Falcon's solver and the Horus' algorithm for execution graphs with different sizes. The Falcon's solver depicts exponential

behavior as graph size increases whereas Horus scales with the graph size.

Falcon is thus unable to assign logical time to more than few thousands of events in a timely manner. For instance, it spends more than 12 minutes for 10,000+ events, while the Horus' algorithm spends 7 seconds to assign the logical time.

We note that this algorithm can perform with low execution time, even when the graph size increases over time. This is it because, running in a periodic basis, the algorithm is able to resume the procedure from the most recent event of each process timeline already with logical timestamps and proceeding to the recently added events. Thus, the execution time does not depend on the total amount of events in the graph but the amount of unprocessed events.

In summary, the execution time of the proposed logical time assignment algorithm scales with the amount of events and causal relationships in the execution graph and thus is suitable for real use cases.

5.5.3 Causal Graph Querying

Horus enables developers to refine the causal graph using the database's built-in query language, as it provides filtering and grouping operations for graph algorithms. As discussed in Section 5.3, the built-in algorithms are not capable of efficiently answer to refinement queries as follows: Q1 - May event a causally affect event b?, and Q2 - What is the causal path between a and b?

To address this limitation, Horus annotates events with logical time to enable efficient causal graph querying.

In this section, we conduct separate evaluations for Q1 and Q2 query types. For each type, we compose the corresponding Cypher queries on each approach, one resorting to built-in path traversal algorithms and the other leveraging logical time.

We setup a n1-standard-16 instance that hosts the Horus pipeline, alongside the Neo4j database. For each experiment, we populated the database using the synthetic event generator.

Evaluation for Q1. The Cypher query that follows a path-traversal-based approach makes use of the shortest path algorithm to determine whether a causal path between the two events exist. For the Horus' approach, in turn, the query just compares the logical timestamps of the two events.

We selected ten event pairs, each whose causal graph contains 10% of the graph's total events, and then executed the queries for each pair. For instance, for a graph with 100 events,

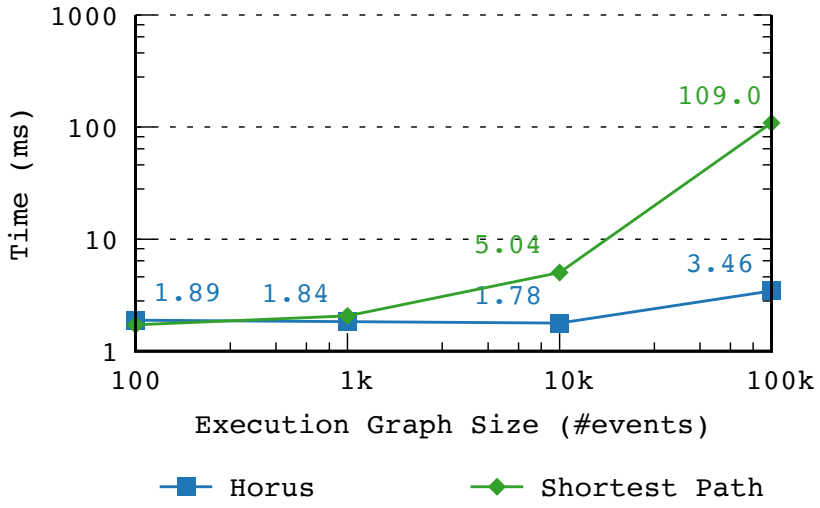


Figure 26: Comparison between the execution time of the shortest path algorithm and Horus to answer query Q1, for different graph sizes.

each causal graph contains 10 events. Clearly, the dimension of each pair's causal graph also increases with the graph size.

Figure 26 unveils the execution time of each query for different graph sizes. Observe that both axes are represented using logarithmic scale. One important note, derived from the negligible std. dev., is that both queries are insensitive to pair location. This is, each query shows similar performance either for a pair positioned at the top of the graph or for a pair positioned at the middle or at the bottom of it.

Using the shortest path algorithm, however, results in an exponential behavior as graph size increases. For 100,000+ nodes, it becomes ~30 times slower than comparing logical timestamps. This is due to the fact that comparing the logical time of any two events does not require any sort of path traversal. Instead, it compares logical timestamps according to the properties of vector clocks.

Evaluation for Q2. For this experiment, the Cypher query that relies on built-in algorithms simply aims at finding all paths between the two events. In this case, the position of the pair and the direction of the internal traversal algorithm influence the query performance in the traversal-based approach. For instance, a pair positioned at the top would lead the query to be less performant if the traversal's direction from the top the the bottom of the graph and vice-versa.

To perform a rigorous and fair comparison, we evaluate the traversal-based query with the pair positioned in the middle of the graph, which causal graph contains 10 nodes. We selected

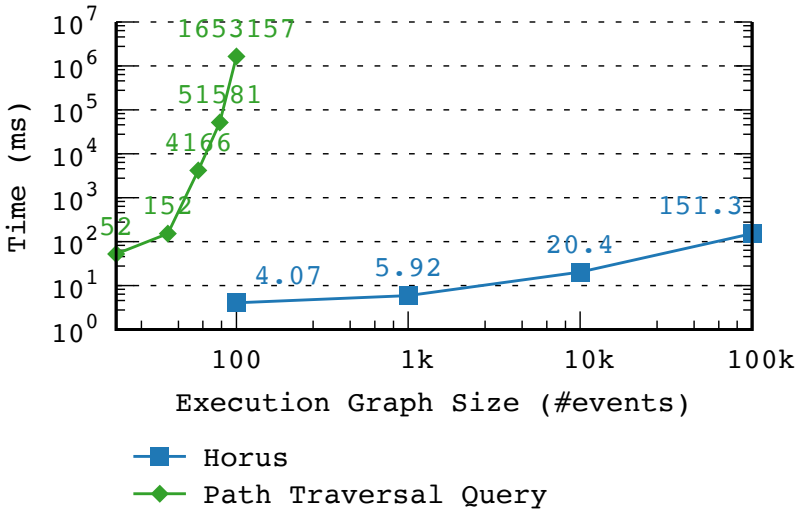


Figure 27: Evolution of the execution time of obtaining the causal graph between two events using traversal-based query and the logical-time-based approach implemented in Horus.

this pair for two reasons: 1) we assume that most analysis do not focus on the beginning nor on the bottom of the causal graph; and 2) the performance of the path traversal is the same, independently of the direction it takes.

In turn, for evaluating the Horus' approach, we adopt the same procedure as in for Q1, that is, we choose ten event pairs, each of which causal graph contains 10% of the total events.

Figure 27 confirms the expected exponential behavior on using the traversal-based query to extract causal paths on usually small graphs (from 10 to 100 nodes).

In contrast to the traversal-based query, our proposed logical-time-based approach is insensitive to pair location in the graph. Therefore, we can compare the performance of both approaches in the scenario of a graph with 100 events.

For the scenario that compares both approaches, i.e., graph with 100 events, using logical time decreases the execution time from ~1653 seconds to ~4 milliseconds.

In summary, the Horus' logical-time-based approach scales much better than traversal-based query for causal queries.

5.6 Summary

In this chapter, we introduced Horus, a system that enables causally-consistent refinement of distributed system logs in a non-intrusive and scalable fashion. Horus combines log messages and kernel-level operations to produce a graph of a distributed execution in which events are

causally ordered. Additionally, Horus leverages logical clocks to dramatically reduce the time to run queries over the execution graph.

Our case study with TrainTicket, a ticket booking application with 40+ microservices, demonstrates the higher effectiveness of Horus in pinpointing the root cause of anomalous behavior compared to state-of-the-art solutions.

Finally, we assess its performance and conclude that Horus i) is designed to scale with the number of events, ii) stands out from the state-of-the-art in respect with performance in generating a causal trace and iii) enables efficient causal queries for analysis refinement of causal traces.

Chapter 6

Conclusion

In this work, we demonstrated that it is possible to improve and leverage black-box techniques in order to obtain useful detailed information about the performance and execution behavior of distributed systems. Without application-specific knowledge, we were able to discover the architecture of the system, pinpoint performance issues, and inspect distributed executions. Particularly in the latter, our results show that black-box techniques can outperform currently widely-adopted techniques in unveiling the root cause of unexpected behavior.

Answering to the research question that opened this thesis — How can one obtain detailed useful information about the performance and execution behavior of large-scale and complex distributed systems while treating its components as black boxes? — we demonstrated that this is achieved by observing the operations at the operating system layer, the ultimately common layer of all software components. While these operations execute, we were able to access data and trigger events that, by leveraging their well-known causal relations, aid on diagnosing performance and behavior anomalies.

At first place, we developed an approach focused on monitoring distributed interactions for discovering the architecture and organization of the system under monitoring. As interactions mostly use network channels, tracing network-related kernel routines to monitor the amount of inter-application traffic resulted in an accurate and system-wide view of interdependencies of distributed processes. The outcome obtained from such monitoring is a weighted communication graph of distributed processes that revealed itself useful for improving the performance in the adaptive placement case study using two real systems.

In respect with debugging systems, logs are the first chosen resource by developers to pinpoint the root cause of bugs. Nevertheless, we demonstrated that carrying out log-based analysis using the popular log management system Elastic Stack has limitations that hinder the

effectiveness of log-based analysis. The effectiveness of using logs for debugging anomalous behavior stops when it lacks the required causality property for analysing distributed executions. We faced this while debugging a fault in TrainTicket, an application with 40+ microservices.

To address the limitations of log-based analysis, we leveraged the well-known causal relations of collected kernel events to assemble a causal execution graph of distributed executions. Once causally correlated, these events serve as guides to establish causality between log entries independently generated by software components of a complex architecture. We first validated it with Falcon using kernel events to causally order log entries, and then we achieved high scalability and refinement capabilities, these in a more capable system named Horus. At this point, Horus provides not only the same capabilities of log management systems but also a query-enabled causally coherent view of distributed executions.

In conclusion, our proposed approaches represent a step forward towards observable complex distributed architectures without application-specific knowledge.

6.1 Future Work

From the presented work and solutions, it is possible to derive a new set of challenges. In the following we briefly discuss such possibilities toward future research directions.

Fist, we primarily designed Horus to improve log-based analysis. From this process, Horus became able to query request flows for debugging purposes. Yet, it would be worth exploring its capabilities in terms of performance management. More specifically, exploring the ability of Horus to provide performance insights trough the generation of flame graphs[1].

Second, the black-box approaches we presented here rely on operating system's kernel. In the case of debugging, even though causality between processes operations can be inferred using communication-related kernel events, there is gap to fill concerning data flows. Specially, in the case of threads using shared memory for synchronization and/or passing data to each other. Exploring a black-box approach towards causality tracking of memory accesses would complement Horus by establishing causality concerning data flows. Also, that would open the possibility of integrating mechanisms of data race detection from the collected data.

[1] http://www.brendangregg.com/flamegraphs.html

Bibliography

[1] Apache Kafka: A Distributed Streaming Platform. https://kafka.apache.org/. Accessed: April 2020. - **Cited** on page 56.

[2] Apache Log4j. https://logging.apache.org/log4j/2.x/. Accessed: April 2020. - **Cited** on page 55.

[3] BCC - Tools for BPF-based Linux IO analysis, networking, monitoring, and more. https://github.com/iovisor/bcc. Accessed: May 2020. - **Cited** on page 55.

[4] eBPF. https://ebpf.io/. Accessed: May 2020. - **Cited** on page 12.

[5] Elastic Stack: Elasticsearch, Logstash and Kibana. https://www.elastic.co/pt/products/. Accessed: April 2020. - **Cited** on pages 13, 53 and 64.

[6] JanusGraph - Distributed, open source, massively scalable graph database. https://janusgraph.org/. Accessed: April 2020. - **Cited** on page 54.

[7] Log Management and Analysis Software Made Easy | Logentries. https://logentries.com. Accessed: April 2020. - **Cited** on page 13.

[8] Logrus: Structured logger for Go. https://github.com/sirupsen/logrus. Accessed: May 2020. - **Cited** on page 55.

[9] Logz.io: Cloud Observability for Engineers. https://logz.io. Accessed: April 2020. - **Cited** on page 13.

[10] Neo4j Graph Platform – The Leader in Graph Databases. https://neo4j.com/. Accessed: April 2020. - **Cited** on pages 54 and 56.

[11] OpenTelemetry. https://opentelemetry.io/. Accessed: April 2020. - **Cited** on pages 10 and 17.

[12] SIEM, AIOps, Application Management, Log Management, Machine Learning, and Compliance | Splunk. https://splunk.com. Accessed: April 2020. - **Cited** on page 13.

[13] Amazon refuses to compensate sellers for 1p price glitch. https://www.theguardian.com/technology/2014/dec/16/amazon-compensate-sellers-1p-price, 2014. - **Cited** on page 1.

[14] Starbucks lost millions in sales because of a 'system refresh' computer problem. https://www.geekwire.com/2015/starbucks-lost-millions-in-sales-because-of-a-system-refresh-computer-problem/, 2015. - **Cited** on page 1.

[15] Starbucks lost millions in sales because of a 'system refresh' computer problem. https://www.geekwire.com/2015/starbucks-lost-millions-in-sales-because-of-a-system-refresh-computer-problem/, 2017. - **Cited** on page 1.

[16] Another Facebook bug may have exposed millions of users' private photos to app developers. https://www.vox.com/2018/12/14/18140862/facebook-software-bug-privacy-issue-private-photos, 2018. - **Cited** on page 1.

[17] British Airways Flights disrupted by IT failures, again. https://www.forbes.com/sites/marisagarcia/2019/08/07/british-airways-flights-disrupted-by-it-failures-again/, 2018. - **Cited** on page 1.

[18] Here's what you need to know about the massive 'Cloudbleed' data breach. https://www.sciencealert.com/here-s-what-you-need-to-know-about-the-massive-cloudbleed-data-breach, 2018. - **Cited** on page 1.

[19] Jenny Abrahamson, Ivan Beschastnikh, Yuriy Brun, and Michael D Ernst. Shedding light on distributed system executions. In Companion Proceedings of the 36th International Conference on Software Engineering, pages 598–599, 2014. - **Cited** on page 3.

[20] Ivan Beschastnikh, Patty Wang, Yuriy Brun, and Michael D. Ernst. Debugging distributed systems: Challenges and options for validation and debugging. Communications of the ACM, 59(8):32–37, August 2016. - **Cited** on pages 41, 46 and 69.

[21] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATEC '04, page 2, USA, 2004. USENIX Association. - **Cited** on page 12.

[22] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and Jakob Rehof, editors, Tools and Algorithms for the Construction and Analysis of Systems, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. - **Cited** on pages 46 and 53.

[23] Mathieu Desnoyers and Michel R Dagenais. The lttng tracer: A low impact performance and behavior monitor for GNU/Linux. In OLS (Ottawa Linux Symposium), volume 2006, pages 209–224. Citeseer, 2006. - **Cited** on page 12.

[24] Úlfar Erlingsson, Marcus Peinado, Simon Peter, Mihai Budiu, and Gloria Mainar-Ruiz. Fay: Extensible distributed tracing from kernels to clusters. ACM Transactions on Computer Systems (TOCS), 30(4):1–35, 2012. - **Cited** on page 12.

[25] Colin J Fidge. Timestamps in message-passing systems that preserve the partial ordering. Australian Computer Science Communications, 10(1):55–56, 1988. - **Cited** on page 62.

[26] Rodrigo Fonseca, George Porter, Randy H Katz, Scott Shenker, and Ion Stoica. X-trace: A pervasive network tracing framework. In NSDI '07. USENIX Association, 2007. - **Cited** on pages 2, 3, 11 and 17.

[27] Ioana Giurgiu, Claris Castillo, Asser Tantawi, and Malgorzata Steinder. Enabling efficient placement of virtual infrastructures in the cloud. In ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing, pages 332–353. Springer, 2012. - **Cited** on page 9.

[28] Shilin He, Qingwei Lin, Jian-Guang Lou, Hongyu Zhang, Michael R Lyu, and Dongmei Zhang. Identifying impactful service system problems via log analysis. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 60–70, 2018. - **Cited** on pages 3 and 14.

[29] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In 2010 USENIX Annual Technical Conference (USENIX ATC 10). USENIX Association, June 2010. - **Cited** on pages 40 and 46.

[30] Olumuyiwa Ibidunmoye, Francisco Hernández-Rodriguez, and Erik Elmroth. Performance anomaly detection and bottleneck identification. ACM Computing Surveys (CSUR), 48(1):4, 2015. - **Cited** on page 2.

[31] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, et al. Canopy: An end-to-end performance tracing and analysis system. In Proceedings of the 26th Symposium on Operating Systems Principles, pages 34–50, 2017. - **Cited** on pages 2, 10 and 17.

[32] Rudolf E Kalman. On the general theory of control systems. In Proceedings First International Conference on Automatic Control, Moscow, USSR, pages 481–492, 1960. - **Cited** on page 3.

[33] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 21(7), 1978. - **Cited** on pages 3, 13, 15, 43, 44 and 62.

[34] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. Taxdc: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, page 517–530, New York, NY, USA, 2016. Association for Computing Machinery. - **Cited** on page 64.

[35] James Lewis and Martin Fowler. Microservices a definition of this new architectural term. http://martinfowler.com/articles/microservices.html, 2004. Accessed: April 2020. - **Cited** on pages 2, 8 and 13.

[36] Jonathan Mace and Rodrigo Fonseca. Universal context propagation for distributed system instrumentation. In Proceedings of the Thirteenth EuroSys Conference, pages 1–18, 2018. - **Cited** on pages 2, 11 and 17.

[37] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In SOSP '15. ACM, 2015. - **Cited** on pages 2, 10, 11 and 17.

[38] Matthew L Massie, Brent N Chun, and David E Culler. The ganglia distributed monitoring system: design, implementation, and experience. Parallel Computing, 30(7):817–840, 2004. - **Cited** on page 10.

[39] Friedemann Mattern. Virtual Time and Global States of Distributed Systems. In Parallel and Distributed Algorithms, pages 215–226. North-Holland, 1988. - **Cited** on page 15.

[40] Harvey Newman, Iosif Legrand, Ramiro Voicu, and Catalin Cirstoiu. Monalisa: A distributed monitoring service architecture. CoRR, cs.DC/0306096, 06 2003. - **Cited** on page 10.

[41] Adam J. Oliner and Alex Aiken. Online detection of multi-component interactions in production systems. In 2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN), pages 49–60, 2011. - **Cited** on page 9.

[42] Aidi Pi, Wei Chen, Xiaobo Zhou, and Mike Ji. Profiling distributed systems in lightweight virtualized environments with logs and resource metrics. In Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing, pages 168–179, 2018. - **Cited** on pages 3 and 14.

[43] Vara Prasad, William Cohen, FC Eigler, Martin Hunt, Jim Keniston, and Brad Chen. Locating system problems using dynamic instrumentation. In Proceedings of the Linux Symposium, volume 2, page 16, 2005. - **Cited** on page 12.

[44] Isaac Sánchez Barrera, Marc Casas, Miquel Moretó, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. Graph partitioning applied to dag scheduling to reduce numa effects. In Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '18, page 419–420, New York, NY, USA, 2018. Association for Computing Machinery. - **Cited** on page 9.

[45] Bo Sang, Jianfeng Zhan, Gang Lu, Haining Wang, Dongyan Xu, Lei Wang, Zhihong Zhang, and Zhen Jia. Precise, scalable, and online request tracing for multitier services of black boxes. IEEE Transactions on Parallel and Distributed Systems, 23(6):1159–1167, 2011. - **Cited** on pages 12 and 18.

[46] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Technical report, Google, 2010. - **Cited** on pages 2 and 17.

[47] Jason Sonnek, James Greensky, Robert Reutiman, and Abhishek Chandra. Starling: Minimizing communication overhead in virtualized computing platforms using decentralized affinity-aware migration. In 2010 39th International Conference on Parallel Processing, pages 228–237. IEEE, 2010. - **Cited** on page 9.

[48] Byung-Chul Tak, Chunqiang Tang, Chun Zhang, Sriram Govindan, Bhuvan Urgaonkar, and Rong N Chang. vPath: Precise Discovery of Request Processing Paths from Black-Box Observations of Thread and Network Activities. In USENIX Annual technical conference, 2009. - **Cited** on pages 12 and 18.

[49] https://prometheus.io/. Prometheus - Monitoring system and time series database. - **Cited** on page 10.

[50] https://www.dynatrace.com/. Dynatrace: Digital Performance and Application Performance Monitoring. - **Cited** on page 10.

[51] https://www.nagios.org/. The Industry Standard In IT Infrastructure Monitoring. - **Cited** on page 10.

[52] https://www.newrelic.com/. New Relic | Deliver more perfect software. - **Cited** on page 10.

[53] https://www.weave.works/oss/scope/. Weave scope. - **Cited** on pages 13, 24 and 25.

[54] Michael Whittaker, Cristina Teodoropol, Peter Alvaro, and Joseph M Hellerstein. Debugging Distributed Systems with Why-Across-Time Provenance. In Proceedings of the ACM Symposium on Cloud Computing, pages 333–346. ACM, 2018. - **Cited** on pages 2, 17 and 18.

[55] H. Xu, X. Ning, H. Zhang, J. Rhee, and G. Jiang. PInfer: Learning to Infer Concurrent Request Paths from System Kernel Events. In 2016 IEEE International Conference on Autonomic Computing (ICAC), pages 199–208, 2016. - **Cited** on pages 12 and 18.

[56] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. Sherlog: error diagnosis by connecting clues from run-time logs. In ACM SIGARCH Comp. Arch. news, volume 38. ACM, 2010. - **Cited** on pages 3 and 13.

[57] Xu Zhang, Yong Xu, Qingwei Lin, Bo Qiao, Hongyu Zhang, Yingnong Dang, Chunyu Xie, Xinsheng Yang, Qian Cheng, Ze Li, et al. Robust log-based anomaly detection on unstable log data. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 807–817, 2019. - **Cited** on pages 3 and 14.

[58] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. Log20: Fully Automated Optimal Placement of Log Printing Statements Under Specified Overhead Threshold. In SOSP '17, New York, NY, USA, 2017. ACM. - **Cited** on pages 3, 41 and 53.

[59] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm. Lprof: A Non-Intrusive Request Flow Profiler for Distributed Systems. In Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, OSDI'14, pages 629–644, USA, 2014. USENIX Association. - **Cited** on page 14.

[60] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding. Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study. IEEE Transactions on Software Engineering, pages 1–1, 2018. - **Cited** on pages 3, 63 and 64.