

Titre: Distributed Performance Analysis Tools for Large Scale
Title: Computations

Auteur: Pierre-Frédéric Denys
Author:

Date: 2023

Type: Mémoire ou thèse / Dissertation or Thesis

Référence: Denys, P.-F. (2023). Distributed Performance Analysis Tools for Large Scale
Citation: Computations [Thèse de doctorat, Polytechnique Montréal]. PolyPublie.
<https://publications.polymtl.ca/53372/>

 **Document en libre accès dans PolyPublie**
Open Access document in PolyPublie

URL de PolyPublie: <https://publications.polymtl.ca/53372/>
PolyPublie URL:

**Directeurs de
recherche:** Michel Dagenais
Advisors:

Programme: Génie informatique
Program:

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

Distributed Performance Analysis Tools for Large Scale Computations

PIERRE-FRÉDÉRIC DENYS

Département de génie informatique et génie logiciel

Thèse présentée en vue de l'obtention du diplôme de *Philosophiæ Doctor*

Génie informatique

Avril 2023

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

Cette thèse intitulée :

Distributed Performance Analysis Tools for Large Scale Computations

présentée par **Pierre-Frédéric DENYS**

en vue de l'obtention du diplôme de *Philosophiæ Doctor*

a été dûment acceptée par le jury d'examen constitué de :

Alejandro QUINTERO, président

Michel DAGENAIS, membre et directeur de recherche

Foutse KHOMH, membre

Weiyi SHANG, membre externe

DEDICATION

To my family . . .

ACKNOWLEDGEMENTS

First and foremost, Professor Michel Dagenais, I would like to express my deepest thanks for giving me the opportunity to pursue the Ph.D. at Polytechnique Montréal and for the many hours of guidance, teaching, and support.

A special thanks to my parents and grandmother. It is hard to put into words how eternally grateful I am for all the support, encouragement, and sacrifice you made for me.

Geneviève Bastien for all the valuable help on the technical part of my research, Martin Pepin From Ciena for the opportunity to get real industrial usecases for my experiments.

To my labmates, Quentin Fournier, Daniel Capelo, Arnaud Fiorini, Hervé Kabamba, and Benjamin Saint-Cyr, thanks for your help, advice, and shared ideas, as well as all daily conversations and laughs. It was an immense pleasure to work with you.

Richard Nguyen, Moustapha Jadayel, Léon Desgagnés, Juliette Letellier, Laurence Martel for their support.

Maria Darha for facilitating my installation in Canada.

I would also like to thank professors Foutse Khomh, Alejandro Quintero and Weyi Shang for accepting the invitation to serve on my thesis jury.

Finally, I would like to gratefully acknowledge the Natural Sciences and Engineering Research Council of Canada (NSERC), Prompt, Ericsson, Ciena, AMD, and EfficiOS for funding this project.

RÉSUMÉ

Les systèmes de calcul de haute performance et basés sur les conteneurs sont de plus en plus présents et nécessaires dans de nombreux domaines tels que les prévisions météorologiques, le traitement d'images, le séquençage de l'ADN et bien d'autres. La puissance de calcul nécessaire à ces applications a entraîné la transition de systèmes monolithique à des systèmes distribués. L'augmentation inhérente de la complexité de ces systèmes s'accompagne de problèmes de performances plus fréquents et dont la cause est plus difficile à déterminer. L'objectif de ce travail de recherche est d'introduire des outils d'analyse plus performants pour l'analyse des très grands systèmes distribués. L'hypothèse de départ est qu'il est possible de trouver la cause des ralentissements et des problèmes de performance par l'analyse des événements système. Les événements bas niveau du noyau du système d'exploitation sont enregistrés dans des traces pendant le fonctionnement de l'application. Ou encore, à la manière d'une boîte noire d'avion, il peuvent être enregistrés quelques temps avant et après l'apparition du problème. Une fois que ces traces sont collectées, des outils d'analyse sont nécessaires pour en extraire les informations utiles et produire des vues adaptées à ce que recherche l'utilisateur.

Un premier axe de recherche a concerné l'analyse de performance des systèmes basés sur les conteneurs et les intergiciels de messagerie. Une étude approfondie des systèmes de communication actuellement utilisés, ainsi que des différentes architectures de déploiement, a permis d'identifier les problèmes de performance courants et des méthodes pour les identifier. Deux nouvelles méthodes ont été proposées, la première basée sur l'instrumentation du système de communication et la seconde moins intrusive basée sur l'observation des événements du noyau de l'hôte des conteneurs.

Ensuite, une évaluation des outils de traçage présentement disponibles a été effectuée. Parmi les différents enjeux identifiés, l'un des principaux concerne l'extensibilité. Même si la collecte des traces est possible sur un très grand nombre de nœuds, ces outils sont peu adaptés à l'analyse de nombreux fichiers de trace, et encore moins sur des nœuds distants. Il y a donc des enjeux au niveau de la taille des traces collectées, du nombre de fichiers de traces, de la variété de points de traces disponibles et du nombre de nœuds.

Les deux axes de recherche suivants se sont concentrés sur l'amélioration d'une méthode d'analyse de traces en particulier, l'analyse du chemin critique. L'analyse du chemin critique permet de déterminer l'origine des goulots d'étranglements par l'analyse du graphe d'exécution du système tracé. Ces périodes d'attente sont souvent causées par des appels

réseaux à d'autres noeuds de la grappe. On doit donc souvent combiner plusieurs fichiers de trace issus de plusieurs noeuds différents afin de réellement comprendre l'origine du blocage. Compte tenu de la taille des fichiers de trace, il est impossible et inefficace de les transférer sur un noeud d'analyse unique. On doit donc répartir les calculs du processus d'analyse entre les différents noeuds tracés. Les outils d'analyse devenant eux-même des systèmes distribués, leur performance est également un enjeu.

Ce travail de recherche a consisté à proposer un algorithme de calcul du chemin critique distribué dans la seconde contribution. Deux architectures optimisées pour l'architecture d'analyse répartie des traces ont été proposées. La troisième contribution a permis d'améliorer les performances de celui-ci en effectuant un pré-traitement des fichiers de traces sur les noeuds distants. Celle-ci est basée sur des fichiers d'indexation de traces et permet d'éviter les communications inutiles entre les noeuds d'analyse lors du calcul du chemin critique.

Une attention particulière a été accordée à la minimisation du surcoût des méthodes développées. Tous les outils et algorithmes développés sont publiquement disponibles dans des dépôts identifiés dans les articles composant cette thèse.

ABSTRACT

High-performance, container-based computing systems are increasingly present and needed in many areas, such as weather forecasting, image processing, DNA sequencing, and many others. The computing power required for these applications has led to the transition from monolithic to distributed systems. The inherent increase in the complexity of these systems is accompanied by more frequent performance problems whose cause is more difficult to determine. This research aims to introduce more efficient analysis tools for analyzing very large distributed systems. The starting hypothesis is that analyzing system events make it possible to find the cause of slowdowns and performance problems. Low-level OS kernel events are recorded in traces during application operation. Or, like an airplane flight recorder, it can be recorded some time before and after the problem appears. Once these traces are collected, analysis tools are needed to extract useful information and produce views that are suitable for what the user is looking for.

The first line of research concerned the performance analysis of systems based on containers and e-mail middleware. An in-depth study of the communication systems currently in use and the different deployment architectures identified common performance issues and methods for identifying them. Two new methods have been proposed, the first based on the instrumentation of the communication system and the second less intrusive based on the observation of the events of the container host.

An evaluation of the currently available tracing tools was then conducted. Among the different issues identified, one of the main concerns is scalability. Even if the collection of traces is possible on a very large number of nodes, these tools are not suitable for analyzing many trace files and even less on remote nodes. There are, therefore, challenges regarding the size of the traces collected, the number of trace files, the variety of available tracepoints, and the number of nodes.

The following two lines of research focused on improving a particular trace analysis method, critical path analysis. Critical Path Analysis is used to determine the origin of bottlenecks through the analysis of the trace system execution graph. These wait times are often caused by network calls to other nodes in the cluster. Therefore, we often have to combine several trace files from several different nodes to understand the origin of the blocking. Given the size of the trace files, it is impossible and inefficient to transfer them to a single analysis node. The calculations of the analysis process must therefore be divided between the various plotted nodes. As analysis tools become distributed systems themselves, their performance

is also an issue.

This research work proposed an algorithm for calculating the distributed critical path in the second contribution. Two architectures optimized for the distributed trace analysis architecture have been proposed. The third contribution made it possible to improve its performance by pre-processing trace files on remote nodes. This is based on trace indexing files and avoids unnecessary communications between analysis nodes when calculating the critical path.

Particular attention has been paid to minimizing the extra cost of the developed methods. All the tools and algorithms developed are publicly available in repositories identified in the articles making up this thesis.

TABLE OF CONTENTS

DEDICATION	iii
ACKNOWLEDGEMENTS	iv
RÉSUMÉ	v
ABSTRACT	vii
TABLE OF CONTENTS	ix
LIST OF TABLES	xii
LIST OF FIGURES	xiii
LIST OF SYMBOLS AND ACRONYMS	xv
CHAPTER 1 INTRODUCTION	1
1.1 Research objectives	2
1.2 Thesis outline	3
1.3 Publications	4
CHAPTER 2 LITERATURE REVIEW	5
2.1 Distributed systems	5
2.2 Container architectures	8
2.2.1 Technologies	8
2.2.2 Orchestration	11
2.2.3 HPC usage	12
2.3 Messaging Systems	13
2.3.1 Message-Oriented Middleware	14
2.3.2 ZeroMQ	17
2.4 Tracing Tools	20
2.4.1 Lttnng	21
2.4.2 Discussion	23
2.5 Tracing methods for distributed systems	25
2.6 Trace viewers	28
2.6.1 Tools for trace visualization	28

2.6.2	General data viewers	30
2.7	Critical path computation and analysis	31
2.8	General discussion	32
CHAPTER 3 OVERVIEW		34
CHAPTER 4 ARTICLE 1: ADVANCED TRACING METHODS FOR CONTAINER MESSAGING SYSTEMS ANALYSIS		36
4.1	Introduction	36
4.2	Related Work	37
4.2.1	Container background	37
4.2.2	Container communication patterns	39
4.2.3	Tracing tools	40
4.3	New instrumentation methods	42
4.3.1	Kernel Level Instrumentation	42
4.3.2	Messaging library level instrumentation	45
4.4	Results	48
4.4.1	Kernel level instrumentation	48
4.4.2	Messaging library level instrumentation	53
4.5	Conclusion	55
CHAPTER 5 ARTICLE 2: DISTRIBUTED COMPUTATION OF THE CRITICAL PATH FROM EXECUTION TRACES		57
5.1	Introduction	57
5.2	Related Work	59
5.2.1	Trace Data Collection	59
5.2.2	Trace Data Storage	59
5.2.3	Communication Between Nodes	60
5.2.4	Critical Path Computation and Analysis	61
5.2.5	Visualization of Large Traces	62
5.3	Methodology to Parallelize the Critical Path Computation	64
5.3.1	Requirements and Challenges	64
5.3.2	Methodology	65
5.3.3	Improved Methodology	65
5.4	Results	69
5.4.1	Representative Use-Case	70
5.4.2	Performance and Overhead	71

5.4.3	Other Use-Cases	73
5.4.4	Discussion	75
5.5	Future work	75
5.6	Conclusion	76
CHAPTER 6 ARTICLE 3: SCALABLE DISTRIBUTED COMPUTATION OF THE CRITICAL PATH FROM EXECUTION TRACES		78
6.1	Introduction	78
6.2	Related Work	80
6.2.1	Trace Collection and Storage	80
6.2.2	Communication Between Nodes	81
6.2.3	Critical Path Computation and Analysis	81
6.2.4	Visualization Tools for Large Traces	82
6.2.5	Index usage in Distributed Systems	84
6.3	Methodology to Optimise Critical Path Computation Using Index	85
6.3.1	Requirements and Challenges	85
6.3.2	Current Methodology	87
6.3.3	Improved Methodology	88
6.4	Results	92
6.4.1	Test Architecture	93
6.4.2	SCP usecase : Performance and Overhead	93
6.4.3	MPI usecase : Performance and Overhead	99
6.4.4	Discussion	102
6.5	Conclusion	103
6.6	Acknowledgements	103
CHAPTER 7 GENERAL DISCUSSION		104
CHAPTER 8 CONCLUSION		106
8.1	Summary of Works	106
8.2	Future work	107
REFERENCES		109

LIST OF TABLES

Table 6.1	The trace file used for the benchmarks : average trace sizes and events number	95
Table 6.2	The external communication index computation overhead : Mean total processing time of each algorithm.	96
Table 6.3	The trace file used for the benchmarks	100
Table 6.4	The overhead of External Communication Index Computation usage .	100

LIST OF FIGURES

Figure 2.1	Comparison of Architecture Layers Between Virtual Machines and Containers	6
Figure 2.2	Docker architecture	9
Figure 2.3	Orchestrators role in container management cycle	11
Figure 2.4	Steps of a RPC call	14
Figure 2.5	AMQP Model	15
Figure 2.6	Example of MQTT publish/subscribe pattern	16
Figure 2.7	Example of Kafka cluster, ZooKeeper is used for managing and coordinating the Kafka broker	17
Figure 2.8	ZeroMQ architecture	18
Figure 2.9	ZeroMQ patterns	19
Figure 2.10	Overview of LTTng tracing tool components	23
Figure 2.11	Example of trace and spans in Jaeger	25
Figure 2.12	Trace Compass architecture	29
Figure 4.1	Docker architecture	38
Figure 4.2	ZeroMQ architecture	40
Figure 4.3	Architecture used for the two methods and instrumentation location .	43
Figure 4.4	Architecture	46
Figure 4.5	Publish subscribe pattern in zeromq	49
Figure 4.6	Impact of kernel tracing on CPU cycles depending of the messaging load (in messages per seconds)	51
Figure 4.7	Impact of kernel tracing on CPU cycles depending of the messaging load (in messages per seconds)	54
Figure 5.1	A time graph in Trace Compass.	62
Figure 5.2	The critical path view of a process in Trace Compass.	63
Figure 5.3	Architecture to retrieve, compute, and display the critical path. . . .	66
Figure 5.4	Loaded critical path view.	66
Figure 5.5	Methodology to identify edges.	67
Figure 5.6	Edges Identification	69
Figure 5.7	Edges Identification	70
Figure 5.8	Processing time of both algorithm versions.	72
Figure 5.9	Processing time of the new algorithm in terms of the number of nodes and trace size.	73

Figure 5.10	Comparison of performance of both algorithms on the viewer node. .	74
Figure 6.1	An example of a time graph displayed in Trace Compass.	82
Figure 6.2	The critical path view of a process as displayed in Trace Compass. . .	83
Figure 6.3	The critical path as displayed in Trace Compass	86
Figure 6.4	The algorithm steps illustrated on the architecture elements.	91
Figure 6.5	The remote node matching with common ID in the payload of events.	92
Figure 6.6	The data gain on transferred data processing	97
Figure 6.7	The gain in terms of global CPU usage of the analysis nodes.	98
Figure 6.8	Example of global critical path view for Soma benchmark	101
Figure 6.9	CPU load reduction with respect to the benchmark used	102

LIST OF SYMBOLS AND ACRONYMS

API	Application Programming Interface (interface de programmation applicative)
CTF	Common Trace Format
CPU	Centralized Processing Unit
IETF	Internet Engineering Task Force
JSON	JavaScript Object Notation
LTNg	Linux Trace Toolkit : next generation
OSI	Open Systems Interconnection
PID	Process Identifier
RAM	Random Access Memory
RPC	Remote Procedure Call (appel de procédure à distance)
TSP	Trace Server Protocol

CHAPTER 1 INTRODUCTION

In recent times, more and more applications are deployed as distributed systems. Instead of deploying a monolithic application, these are often separated into modules, which improves failure resilience and scalability. However, the number of modules can reach several hundred, so it is not very efficient to deploy each module on a separate physical machine. To counter this, virtual machines enable to deploy several modules on the same physical machine, and thus optimize resource usage. An even lighter solution is to use containers. All these computing elements can run on machines hosted in several different countries. Thereby, this heterogeneity in architectures involves challenges for performance monitoring.

This research project is about the performance analysis of large computing architectures. The main question is whether current tools can effectively monitor this type of architecture. Thus, the role of the researcher is to find the gaps in current analysis methods and provide new methods and paradigms to try to fill them. The first part will review state of the art for this kind of architecture and the existing tools to analyze their performance. As the components of these architectures need to communicate together, tools for analyzing messaging frameworks will also be reviewed. Tracing is the appropriate method to collect information on the operating state of systems.

Thus, the performance of tracing tools will be measured to find out if these are suitable for the performance analysis of container-based architectures and to identify gaps. After being collected, the information should be displayed for an efficient interpretation by the distributed system developer. The current tools will be compared to see if these allow efficient analysis of the information. Advice will also be given on current methods for analyzing results and how new and improved methods could be developed.

In the second part, the overall planned contribution will be described, as well as the specific contributions for each targeted applied research area. The first step aims to solve the problem of analyzing the communications between containers. Then, some improvements in visualization methods will be studied. These questions will be addressed in the context of a case study provided by an industrial partner. The experiments realized in this research work led to an improvement in current tools. This will also lead to knowledge sharing on new methods for analysis and visualization. It results in a description and evaluation of the proposed new algorithms in research articles and public patches on existing tools.

The core of this work is to improve the quality and versatility of performance analysis tools applied in large-scale environments. After identifying the gaps provided by the literature

review, improvements to current tools and new methods are proposed in the second part to address them.

More generally, this research work attempts to answer the following research questions : (1) whether current tools can effectively monitor large container clusters, including the ability to monitor several systems at once and collect a huge amount of data efficiently? (2) How to improve the efficiency of the current methods to analyze traces collected on large distributed systems? (3) How to take advantage of pre-computation in large traces analysis for faster user requests processing?

1.1 Research objectives

Large computing architectures, with an ever-increasing number of nodes, require efficient tools for their analysis. Tracing remains the best solution to allow a deep understanding of the interactions between computing elements. An exhaustive literature review, and experiences with current tools, allowed us to identify several gaps and the need for better scalability of these tools. This thesis aims to provide solutions for performance analysis of large-scale computation systems by proposing new methods, comparing them with existing methods on real use cases, and providing a critical discussion of the results before identifying future improvements.

1. **Comprehensive study and tracing tools review for container-based architectures messaging system** : Container usage for application deployment implies a larger number of nodes and communication links between them. Therefore, performance issues and bottlenecks are harder to understand. The focus of this first research axis is to characterize the patterns used for inter-container messaging systems and propose new methods to analyze their performance. The first method focuses on direct instrumentation of the messaging library and led to a publically available patch of the ZeroMQ library, as well as a good description of the method used to similarly instrument other messaging libraries. A second method, based on kernel instrumentation, has been proposed to overcome the frequent inability to modify the source code of deployed applications. This method has been tested with benchmarks representative of architectures currently used in the industry. The goal is to provide more universal methods applicable to a large majority of architectures, even when there are concerns with scalability, overhead, or security. The instrumentation of messaging systems allows the production of traces that can be analyzed to understand slowdowns or bottlenecks in container clusters. The main contribution of this first part is the new instrumentation

and analysis methods for large container cluster messaging systems.

2. **Distributed Computation of the Critical Path from Execution Traces :** This second research project focuses on improving tools to understand bottlenecks in large distributed systems caused by network latencies, storage contention, or physical issues. Critical path analysis on collected traces is used to uncover the root cause of bottlenecks in distributed systems and other smaller-scale systems. However, the amount of data collected during tracing, and necessary for critical path processing, is challenging with current trace analysis tools. Rather than transferring all trace files to a single node for analysis, the goal of this research project is to develop a method to distribute the computation of trace file analysis and, more specifically, critical path analysis. Avoiding the transfer of all trace files to a single analysis node, and harnessing the computing power of several parallel analysis nodes, can greatly increase the analysis performance and reduce the associated computing time. This faster processing allows the collection of a larger number of tracepoints and a better understanding of observed bottlenecks. The main contribution of this second part is the new distributed trace analysis algorithm and its implementation in a trace analysis tool.
3. **Execution Traces Preprocessing for Distributed Computation of Critical Path :** This last research project introduces a new algorithm to compute critical paths based on pre-computed indexes for inter-analysis node communications. The goal is to improve performance and efficiency compared to the previous algorithm proposed in the second research project. The essential concern is the communication between analysis nodes. A pre-computation of the links between the traces located on the different parallel trace analysis nodes can avoid useless communications and data transfers. This research work will focus on this improvement and will test it with extended benchmarks. The main contribution of this last part is the improved distributed trace analysis algorithm, using pre-computations and its implementation in a trace analysis tool.

1.2 Thesis outline

The remainder of this document is organized as follows. Chapter 2 presents some background information and provides a critical literature review of distributed system tracing and analysis studies. Chapter 3 describes an overview of this thesis's contributions. In the subsequent chapters, three research articles are presented. Chapter 4 propose some methods for container messaging systems analysis. Chapter 5 introduces a new distributed algorithm for critical path processing from execution traces collected on large computing clusters. Chapter

6 presents a new method that takes advantage of External communication index usage or an optimized critical path computation. Finally, Chapter 7 summarizes this thesis's contributions and discusses limitations and promising research avenues.

1.3 Publications

Chapters 4, 5 and 6 of this thesis include, respectively, the following submitted articles :

1. Pierre-Frédéric Denys, Martin Pepin and Michel R. Dagenais, "Advanced tracing methods for container messaging systems analysis", July 15th, 2021, Journal of Cloud Computing - Springer
2. Pierre-Frédéric Denys, Quentin Fournier and Michel R. Dagenais, "Distributed Computation of the Critical Path from Execution Traces", July 27th, 2022, Software: Practice and experience
3. Pierre-Frédéric Denys, Quentin Fournier and Michel R. Dagenais, "Scalable Distributed Computation of the Critical Path from Execution Traces", January 13th, 2023, Concurrency and Computation: Practice and Experience

CHAPTER 2 LITERATURE REVIEW

Large Distributed systems are becoming common, and more suitable performance analysis tools are needed. These systems were historically deployed on clusters composed of physical machines with dedicated operating systems on each one. Nodes were connected with network sockets and the amount of data exchanged between them was relatively small. Messages were exchanged using RPC or other methods. Two changes have gradually been introduced: virtual machines and containers. These new architectures use a large number of computing nodes. As a result, their usage implied the advent of new communications paradigms like Message Oriented Middlewares, suitable for large networks of communicating modules. Consequently, performance analysis tools must evolve to handle these new systems.

This chapter will begin with an overview of the key characteristics of distributed systems. A focus will then be given to systems based on containers, and the specificities of High Performance Computing (HPC) applications running on such systems. It will be followed by a review of communication methods, in large distributed systems, especially those based on containers. Then, methods used to analyse and collect performance data will be reviewed, as well as analysis methods for large traces issued from these systems.

To sum up, this chapter will be divided into three parts : (1) characteristics of distributed systems and their messaging systems (2) tracing methods for distributed applications, and finally (3) large traces analysis and visualization.

2.1 Distributed systems

The use of IT resources is increasingly *on demand* and *in the cloud*. In order to be able to quickly adjust the capacity of resources to demand, distributed systems are used.

"A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system" [1]. The word autonomous in this definition refers to the independence of computing elements that can be achieved by isolation. The goal of distributed systems is to ensure the isolation of applications, optimize hardware usage, as well as provide scalability.

To ensure the isolation of these systems, two methods can be used. Virtual machines and containers.

A virtual machine is the emulation [2] of a full-computer system. Hardware is emulated, and software is independent between virtual machines. The goal is to optimize physical

machine resource usage, running multiple services on the same machine, while preserving isolation.

On the opposite, a **container** is a way to provide software isolation, on the host operating system kernel. Thus, multiple workloads can run on a single operating system instance. However, a container runs within the same operating system as the host[3], and thus is bound to that version.

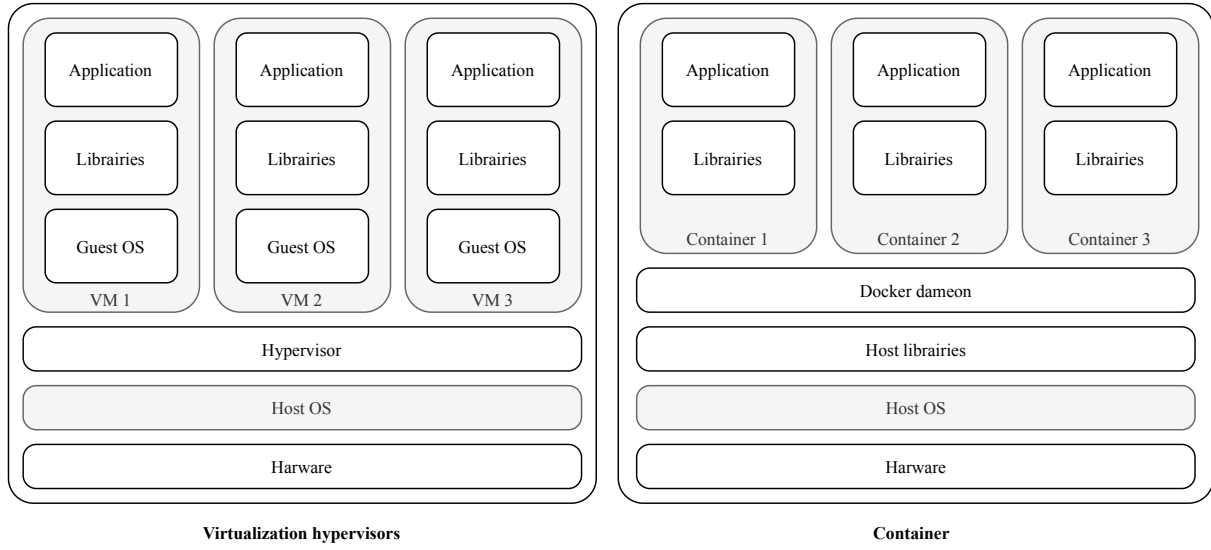


Figure 2.1 Comparison of Architecture Layers Between Virtual Machines and Containers

The speed with which a system is put into service is often an important factor. According to [3], containers outperform virtual machines, being twenty-two times faster in startup time during scaling. The application performance is also better in containers according to the same article.

This article [2] quantifies the performance overhead of virtual machines and containers compared to bare metal. It also shows the benefits of the combination of virtual machines and containers. But the experiences of this article were only made with one use case (MySQL database). Compared to that, this article [4] ran different benchmarks for different applications. In these two articles, there was no HPC concern.

Many other articles [5] [6] [7] [8] [9] [10] present some benchmarks and comparisons between virtual machines and containers, but are mainly focused on a specific application and on a relatively small architecture, not representative of a datacenter environment, for example.

To summarize, virtual machines provide isolation and flexibility, compared to conventional

systems, but containers allow better reactivity for variations in computing capacity. The difference in performance observed is explained by the fact that containers do not require hardware virtualization or an independent operating system for each entity, unlike virtual machines.

The reasons which led to the invention of the containers will now be presented. Initially, **Chroot** (*for change root*) has been developed to restrict the view of the file system for a process. Indeed, it is not possible for such a process to access files and binaries that reside outside the restricted environment [11]. This was the beginning of process isolation.

Then, a generalization was made with the ability to isolate other resources such as network interfaces. Two mechanisms have been added to the Linux kernel to allow greater isolation granularity[12] :

Control Groups (**cgroups**) provide a mechanism to limit the amount of resource a process can use. It involves resource metering and limiting, for memory, CPU, I/O blocks and network. **namespace** limits the scope of a process, and what the process is able to see in the operating system. Each process inside a **namespace** only sees its own isolated instance of the global resource.

The combination of **namespaces** and **cgroups** is a container. From this, a more sophisticated isolation system, **LXC**, has been implemented in 2008, and was one of the first implementations of Linux container managers before Docker. This system is still used nowadays, and it will be compared to Docker in the next part.

Finally, **Docker** arrived in 2013 and offers an entire ecosystem for container management ¹.

In conclusion, an hypervisor-based architecture (virtual machines) is preferable when applications require different operating systems or OS versions. While virtual machines excel in isolation, they add a significant overhead compared to containers, and the system management and monitoring are more difficult. Also, the scaling process with virtual machines is slower, as compared to containers. By opposition, in a container-based architecture, applications share the same operating system, so the deployment of applications requires less memory and needs fewer resources.

But the combination of the two virtualization technologies, as described in this article [2], reaps the benefits of both systems, the strong isolation of the virtual machines, and the scalability and smaller resource needs of container-based architectures.

¹<https://blog.aquasec.com/a-brief-history-of-containers-from-1970s-chroot-to-docker-2016>

2.2 Container architectures

This section will focus on container architectures, isolation, and multi-tenancy technology. With a container, the applications share the operating system and binaries. The deployment can be smaller than hypervisor deployments (on virtual machines), as shown in the previous part [13]. Also, some container technologies are portable across many operating systems and are independent of physical hardware.

2.2.1 Technologies

The three most popular container systems are now **LXC**, **Docker**, and **Singularity**. This part will allow us to specify the similarities and advantages of each of these three solutions.

To support the point, a benchmark [14] between the three technologies shows that the performance is equivalent for a small system. However, it does not explain how these experiments can be representative of higher loads.

The security concern is also essential in many applications, and this article [15] benchmarks different container technologies and lists vulnerabilities affecting Docker uses.

A more technical performance comparison will now be presented for these three technologies.

LXC

LXC (Linux Containers) is a lightweight virtualization system that allows running a full Linux operating system without emulating physical hardware, and the overhead that goes with this [16]. Contrary to **Docker** containers, which are restricted to a single application by design, there is a proper init inside the container, with services, cron, daemons, and log management².

This virtualization system uses the cgroups feature and provides more segregation than **chroot**, without having the overhead of a full virtualization solution. To conclude, **LXC** is now less popular than **Docker**, but **LXC** remains a good choice for HPC, compared to virtual machines.

Docker

Docker is a lightweight technology to package and deploy some applications and their dependencies on heterogeneous systems. The article[17] defines a docker container as a lightweight

²<https://archives.flockport.com/lxc-vs-docker/>

virtual machine. Docker is an application platform that goes beyond LXC for decoupling with both a kernel and application-level API, as shown in this figure.

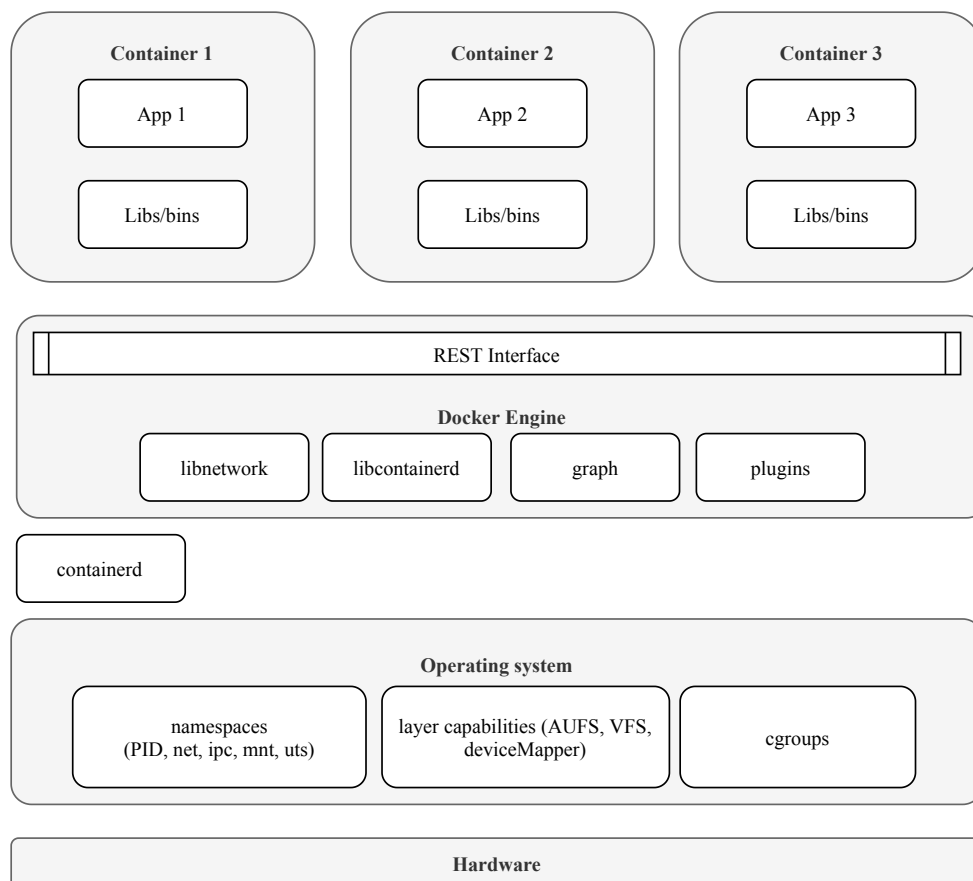


Figure 2.2 Docker architecture

Thanks to the Linux kernel features called cgroups and namespaces, detailed in the previous section, **Docker** provides a high level of isolation between containers. This includes the network system, the storage, the file system, and the resource management. Containers share the same kernel and have no distinct operating system[18].

The storage management of Docker is based on **AUFS** storage drivers[19]. It allows layered storage, reduces the size of shared images, and speeds up the migration of containers if needed.

Measuring the performance of a container-based architecture and comparing it to other virtualization technologies is not straightforward [20]. The measurement methodology is really different from one article to another, and there are no particular studies on measurement best practices.

Singularity

Singularity is the most recent container platform. Singularity is described as a more secure approach, suitable for HPC [14]. The difference from a security point of view is that Docker requires superuser privileges for container usage and management. **Singularity** effectively runs as the current user and does not require elevated access privileges [21]. **Singularity** runs container processes without a daemon. They just run as child processes. So a daemon is not shared between users, unlike with **Docker**. This reduces the possibilities of security vulnerabilities but may not be as flexible in terms of sharing resources among containers.

The performance of Singularity for HPC is compared to other types of virtualization in [22]. On the compatibility side, **Docker** images can be used to build singularity containers. As this container platform is really new, very few scientific papers present benchmarks comparing it with other techniques.

Singularity can achieve near-native performance in MPI and GPU parallel applications, according to [14]. This is important for HPC, because typical HPC applications heavily rely on parallel CPU and GPU usage.

Discussion

Given the significant similarities shared by the two technologies described above, the strengths and weaknesses must be compared. **Singularity** is described as the winner in most benchmarks, thanks to its lower overhead and simpler resource management.

The lower overhead of **Singularity** is possible thanks to the fact that the processes are executed without a daemon but directly as a child process.

For **Singularity**, the security paradigm is similar to Docker, but the fact that the container runtime does not execute as root reduces the attack surface.

The main drawback of **Singularity** is the independence from the Docker ecosystem. This could lead to difficulties in ensuring portability and interoperability. Also, **Singularity** does not provide any dedicated container orchestration tool at the moment [23].

Finally, **Singularity** is a better choice when limited user privileges are mandatory, native support for GPU is needed, and a particular concern for performance but not for a complex orchestrated environment.

2.2.2 Orchestration

Container orchestration involves organizing the work of individual components and application levels. These tools allow users to control the startup and shutdown of containers, group them into clusters, and coordinate all the processes involved in a distributed application.

The orchestrator allows going from a distributed application to scalable cloud computing. Indeed, containers allow resources to be shared from a software point of view, and orchestrators allow resources to be shared from a hardware point of view.

To illustrate this, **Kubernetes**, a popular orchestrator with resource abstractions pods, and nodes, is used. A **pod** is a **Kubernetes** abstraction that represents a group of several application containers ³.

A **node** in **Kubernetes** is a virtual or a physical worker machine, depending on the cluster. For example, with a **Kubernetes** cluster, an application is deployed in a pod, which is a group of **nodes** (containers). The **nodes** can be hosted on the same physical machine or not. A cloud system is an assembly of raw resources, mainly linked by the network. The orchestrator is in charge of scheduling containers on these resources.

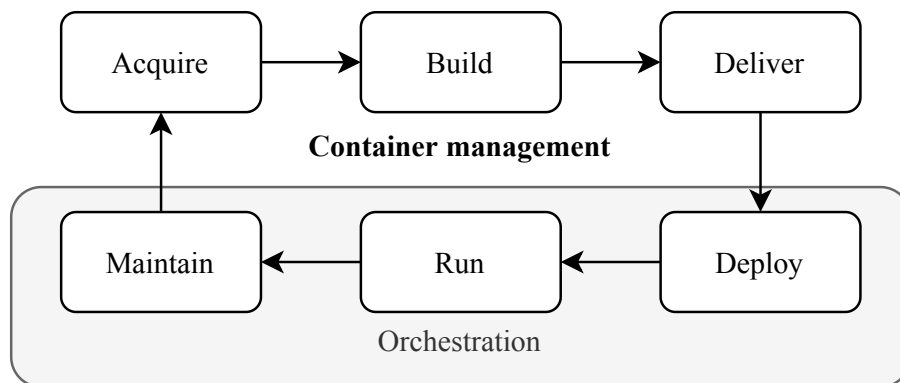


Figure 2.3 Orchestrators role in container management cycle

Some key characteristics of an orchestration system are given in this article [24], like the cluster state management, the scheduling of containers, the high availability of applications deployed, and the monitoring of the whole system.

The orchestration is the management and scheduling of hardware resources for applications. Container orchestration is focused on managing the life cycle of containers, but it does not monitor communications between containers.

³<https://kubernetes.io/docs/tutorials/kubernetes-basics/explore/explore-intro/>

Many orchestrators are available, like `Kubernetes`, `Apache Mesos`, `docker swarm`, `rancher OS` and `nomad`. There are also some proprietary systems that are used internally, like Amazon ECS and Microsoft Azure. However, according to these articles [25], [26], companies can also build their own container manager on top of `Kubernetes` or `Apache Mesos`, for example. This is the case of Netflix, which developed `Titus`, their container management system. [27].

This article [28] presents use cases of container orchestrators. Depending on the application, the needs may differ. For example, one main concern in big data analytics is the scalability of stream processing, whereas the deployment of parallel applications on CPUs and GPUs (using `MPI` and `CUDA` libraries, for example, [29]) is the challenge for HPC applications.

In a different area, business applications are generally built on the micro-service pattern, according to [30], and the challenge is to ensure a rapid response time for the systems with a lot of interactions between containers. Scalability is the main concern in all these applications.

2.2.3 HPC usage

In this part, the specificities of HPC environments will be examined, as well as the importance of instrumenting the messaging frameworks within these environments.

High Performance Computing environments are typically multi-user systems where users should only have access to their own data. This implies applications that run mainly on the CPU and GPU. This kind of environment also implies running some experimental programs with memory leaks or other problems. Therefore, the isolation between user-defined spaces is mandatory for scientific computing.

Currently, physical machines are mainly used with a pure application resource allocation. Indeed, the overhead caused by virtual machines is the main obstacle to using virtualization techniques. However, containers, which are still very little used in this area, can provide a relevant response. They bring a negligible overhead while maintaining an acceptable level of insulation. Indeed, the only resource that can be successfully isolated is the CPU. The isolation for memory, disk, and network is not as strong, as shown in [31]. As discussed in article[28], containers are used in HPC, in order to deploy very large applications [32] [33] [34].

An important fact to also consider in this type of environment is security. The processing can be done on confidential (e.g. medical) data, and some malicious actions can be attempted from inside the container, like privilege escalation, denial of service, and, of course, software vulnerabilities, on parts that can be exposed on the web. This article [35] describes the need for scientific experiments in containers.

This usage also implies many challenges in scaling and monitoring, this is described in [36], and this issue can be resolved by using an orchestrator, like those listed in the previous part. And last but not least, all the components of these huge systems need to communicate. The means of communication will be described in the next part.

2.3 Messaging Systems

The origin of current messaging systems is the request-response communication protocol used for process synchronization in multiprogramming systems. Then, with the creation of network features, these programs had to communicate across the network. This gave rise to distributed computing consisting of procedure calls between processes. The creation of Network File Systems and emails has reinforced this need, and remote procedure calls (RPC) have become essential. RPC is a synchronous method of requesting remote service execution, so consumers must suspend the service execution until they receive a reply from the provider. This blocking system creates latency and can affect application performance[1]. To counter this, Message-oriented middlewares (MOM) were created. With this, the consumers of a service may be physically and temporally decoupled from the service providers [37]. The MOMs also typically offer synchronous request/reply messages (which is equivalent to RPCs) as one of several possible modes. Therefore, many MOMs are in fact supersets of RPCs.

Thus, there are two main models for communication in distributed systems: Remote Procedure Call (RPC), and Message-Oriented Middleware (MOM)[38]. These two methods will be briefly compared, and the Message-Oriented Middleware systems will then be examined in more detail.

Remote Procedure Calls (RPC) use a request-response communication protocol for process synchronization. RPC achieves the following tasks: the management of client and server stubs, the marshaling of data, and the asynchronous communication. RPC uses an interface description language (IDL) to provide an interface between servers and different clients.

Message-Oriented Middleware (MOM) allows application components to be distributed over different platforms and shield the application developer from concerns about operating systems and network interface differences. MOM allows a looser coupling between components as compared to RPC. They can be classified into two categories, *Queue-based middleware* where messages are stored and selected by queue, and *Publish/Subscribe middleware* where messages are managed by predicates [39].

The broker in a pub/sub system can be designed to deliver messages for a specified time slot but then stop attempting delivery, whether or not it has received confirmation of successful

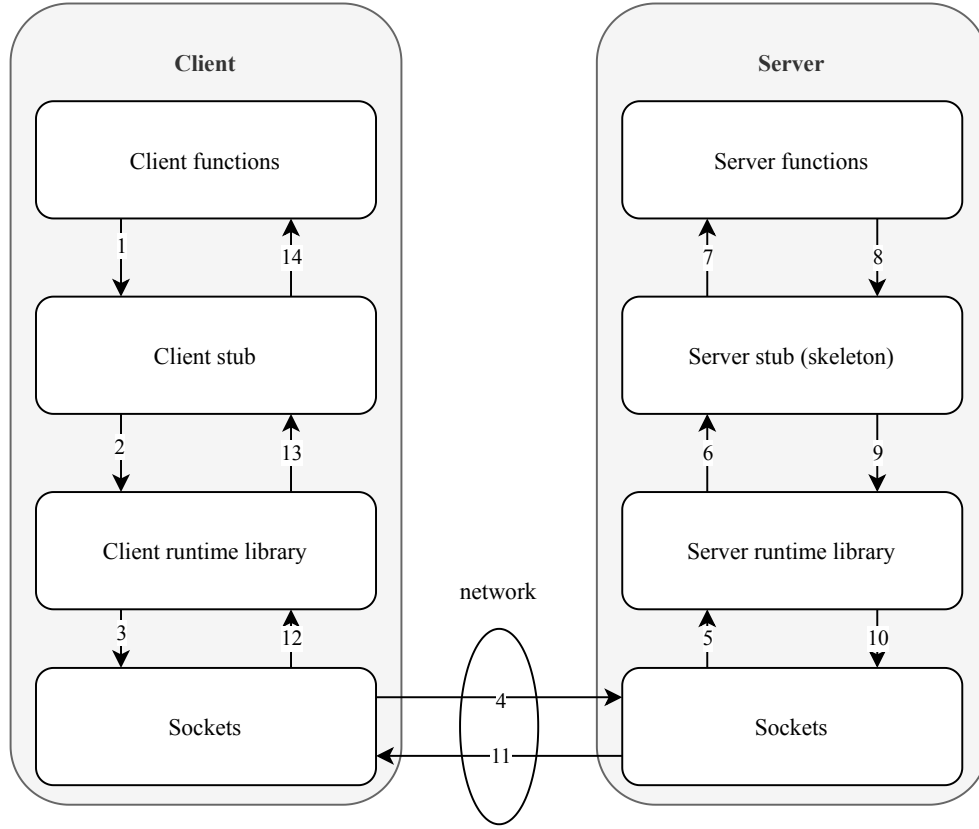


Figure 2.4 Steps of a RPC call

receipt of the message by all subscribers. The main advantage of this solution is the loose coupling between components and the potential for scalability. But message delivery issues are a side effect of decoupling publishers from subscribers.

According to [40] MOM is more appropriate for transactions that require long execution times, contrary to RPC that is more appropriate for service providers with small response times. This assessment is based on experiments on communications between an application program component and a service provider. This article provides a mathematical model to compare the execution times of RPC calls and MOM-based calls.

The default behavior of RPCs is to block the caller execution while the server process handles the request. By comparison, messaging systems are based on queuing: the producer service posts data in the queue and the consumer gets data from the queue [41].

2.3.1 Message-Oriented Middleware

Several MOMs exist and use different approaches for message transmission. These will be compared to identify the most effective and ideal use cases. A detailed qualitative and

quantitative feature comparison between four MOMs is available in this article [42]. Also, in [43], some MOMs are compared for parameters like message throughput, latency, messaging patterns, and security.

These comparative studies will be summarized in the following section, the advantages of each of the main MOMs will be stated, and a brokerless solution will be presented along with its main benefits.

AMQP

AMQP (*Advanced Message Queuing Protocol*) is an open-source standard protocol under an OASIS license. The **RabbitMQ** library is one of the most popular implementations of AMQP. The communication pattern is based on a broker. The functional and transport parts are separated. Indeed, the functional layer manages the queue and the exchanges, and the transport layer manages the channel between clients and brokers, and data encoding [44].

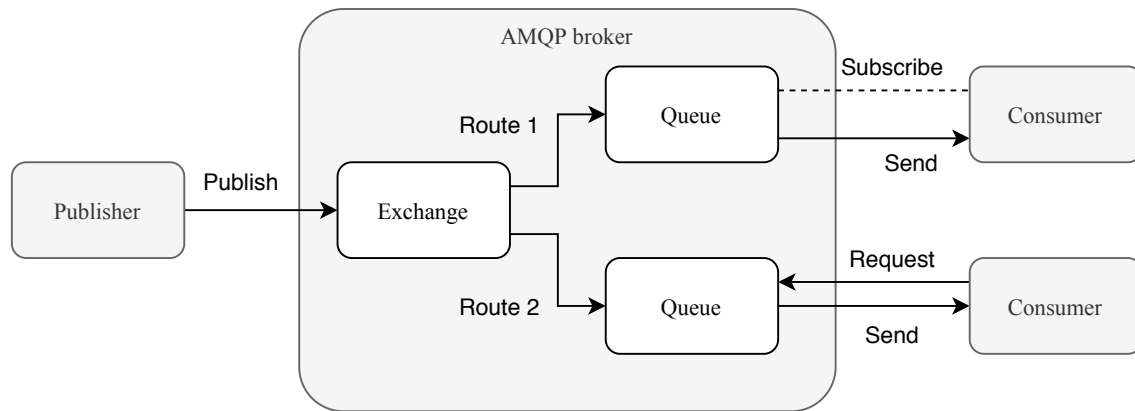


Figure 2.5 AMQP Model

AMQP supports peer-to-peer connections and multiplexing and is able to handle load balancing for the entire network. It enables different messaging patterns like round robin, message queue, combinations, and save and forward. [45] This messaging system is more performant with small messages, so it is suitable for transactional messaging [46].

MQTT

MQTT (*Message Queuing Telemetry Transport*) is an open-source messaging protocol under the OASIS license. Eclipse Mosquitto is one of the most popular implementations. MQTT is similar to AMQP in the protocol design but is suitable for small devices operating with

minimum bandwidth (like in IoT). MQTT uses a central broker and is based only on the pub/sub pattern. It does not support complex message routing, unlike AMQP.

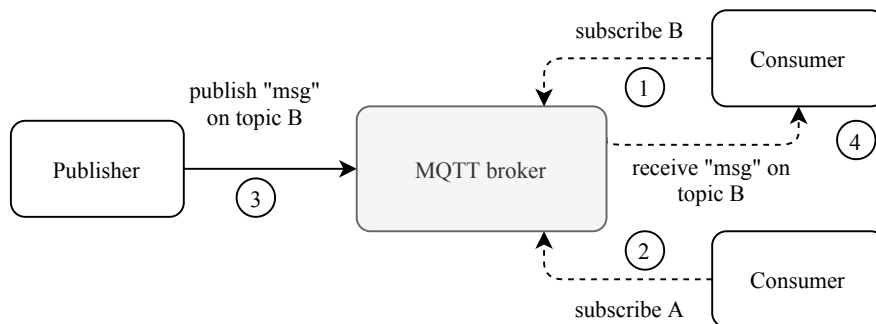


Figure 2.6 Example of MQTT publish/subscribe pattern

A system implementing **AMQP** can have multiple message namespaces, contrary to **MQTT** that groups all messages in a unique namespace, the topic.

Apache Kafka

Apache Kafka is an open-source distributed messaging library under the Apache License, developed in Java. Kafka supports only the Pub/Sub pattern. Unlike **MQTT**, **Apache Kafka** deals with high-velocity data ingestion and ensures batch processing, but not with M2M. **Kafka** is designed for massive datasets and heavy loads. It also has good scalability properties [47].

In opposition to the first two protocols described in this part, **Kafka** is a distributed system that can handle a massive amount of data. It offers high horizontal scalability and is suitable for metrics and log aggregation. The main drawback of **Kafka** is the lack of filters at the brokers. The filtering needs to be done by the consumer or the applications.

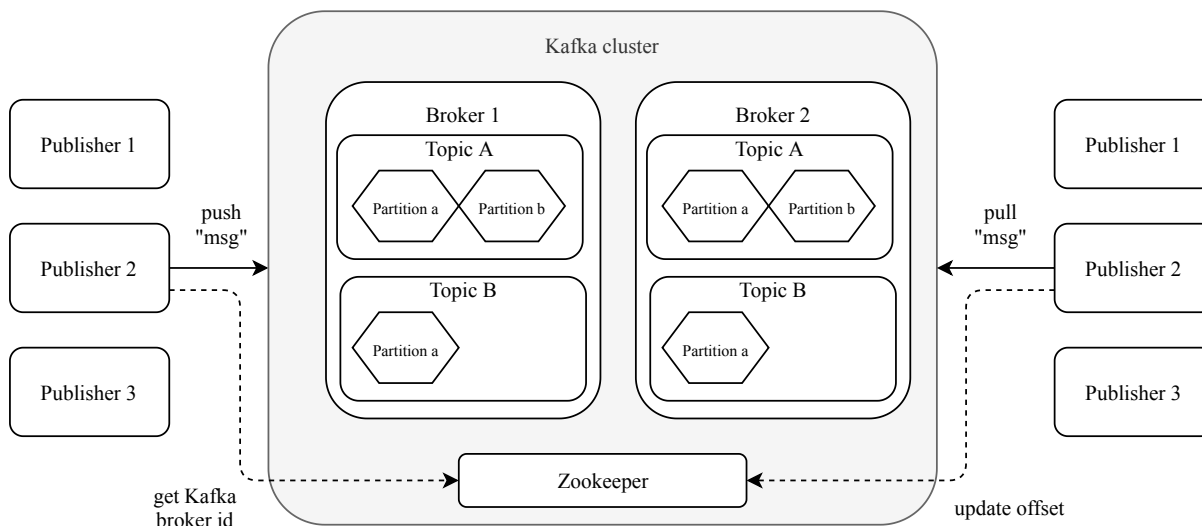


Figure 2.7 Example of Kafka cluster, ZooKeeper is used for managing and coordinating the Kafka broker

2.3.2 ZeroMQ

ZeroMQ (ZMQ) is an open source (under LGPLv3) messaging library, and use a message queue. ZMQ is based on a brokerless architecture, unlike the three other messaging protocols presented before. The operating mode is similar to Berkeley sockets [48]. On the one hand, there is no single point of failure. But on the other hand, there is no persistence support, so there is a greater chance of losing messages, as there is no quality of service implemented.

As said in this article [42], the brokerless architecture results in improved performance. As ZMQ is brokerless, messages are sent virtually immediately, and neither producer nor consumers reach their saturation point in the experiments. The only bottleneck observable is the physical network bandwidth. ZMQ also has the smallest overhead (it needs 52 bytes to frame a message, compared to Kafka with 192 bytes and AMPQ with 105 bytes).

The brokerless architecture gives ZMQ flexibility for different topologies and is the basis for other communication systems, like in this article [49].

To illustrate this statement, RabbitMQ (an implementation of AMQP) can handle 10 000 messages per second, and ZMQ can handle 66 000 messages per second on the same system. We can explain this difference through the architecture of the two systems. ZMQ has a concept of a subscription filter (messages are not stored on disk but in memory in small buffers), whereas RabbitMQ has a concept of queues that allow message persistence, so it is slower [50].

To sum up, **RabbitMQ** focuses on how messages are stored, filtered, and monitored, whereas **ZMQ** focuses on the transmission of messages over the network.

The user interacts with **ZMQ** using "sockets," similar to TCP sockets. The main difference is that each socket can handle communication with multiple peers. Here is a small view of the **ZMQ** architecture ⁴ :

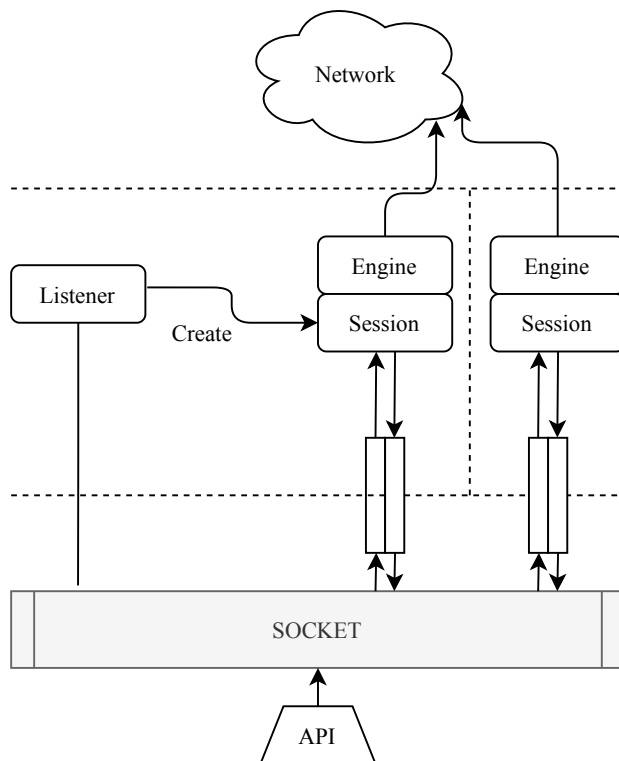


Figure 2.8 ZeroMQ architecture

The first frame of each message contains the logical name of the stream and the second frame contains the actual log record or message content.

ZMQ provides five basic patterns for message exchanges: synchronous Request/Response, Asynchronous Request/Response, Publish/Subscribe, Push/Pull, Exclusive Pair. Here is an example of two patterns commonly used. Other patterns may be found in the impressive guide of **ZMQ**⁵.

ZMQ allows using a higher-level API in several languages. For example, **CZMQ**, is a language binding for C that resembles an object model.

On the performance side, this article [51] analyzed the performance of **ZMQ**, for both delay

⁴<https://www.aosabook.org/en/zeromq.html>

⁵<http://zguide.zeromq.org/page:all>

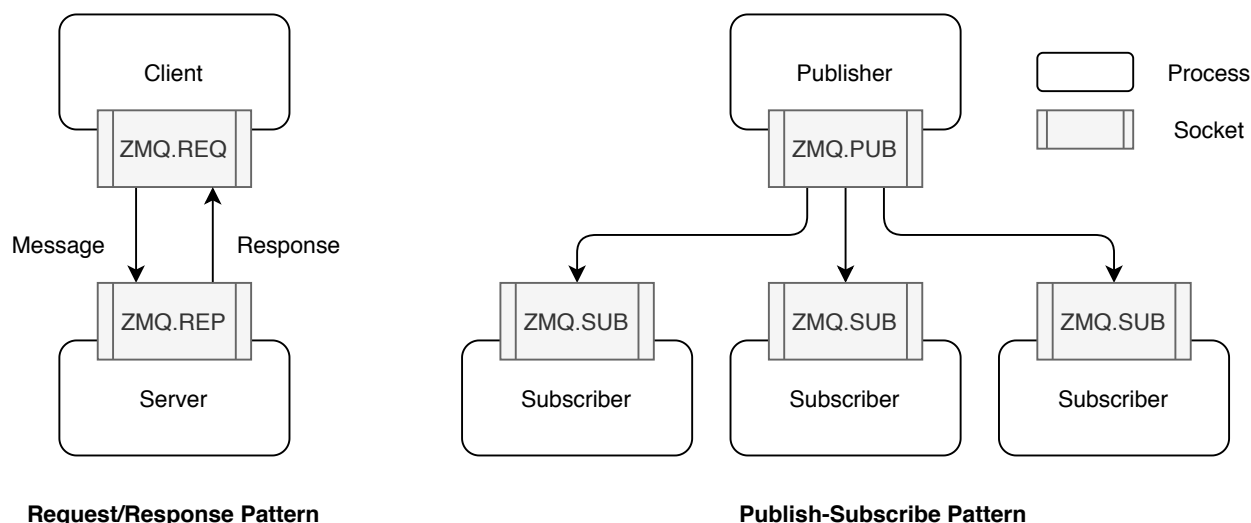


Figure 2.9 ZeroMQ patterns

and throughput. The tools used were Wireshark to capture packets, tools provided by ZMQ to measure the latency of message exchanges, and programmatic measurements. These experiments show a cruel lack of tools for messaging system performance. The tools used in this experiment were suitable for getting an overview of the performance, but they clearly cannot be used to find some issues in a system or to make a monitoring system. The important conclusion is that the publish/subscribe pattern performs better than the request/reply mode.

In conclusion, the choice of a messaging library is not straightforward, and quantitative benchmark results analysis is simple with two computers, like in this article, but is way more complicated in a production system with thousands of machines.

As seen previously, the brokerless design of ZMQ can lead to lost messages, and other messaging systems involve bottlenecks. The next part will analyze the methods available to measure the response time of communications between the components of distributed systems. Tracing tools are suitable for solving many of these issues.

2.4 Tracing Tools

Identifying bottlenecks is an important concern in container architectures. Performance problems may reside either in the host or container, and system metrics may be a useful tool for their detection. Tracing tools help to find bottlenecks in the applications or deeper in the kernel. A trace is a series of events over time. Events are collected at tracepoints during program execution. Each event has a type and payload. Tracing uses events as input for analysis and is used to profile applications, find abnormal states in applications, investigate real-time deadlines, find memory or load issues and investigate concurrency problems.

In order to analyze the Linux kernel or an application, there are two approaches. On one hand, the static analysis considers all possible executions and uses complex calculations on the source code, but without impact on execution, [52]. On the other hand, the dynamic analysis considers execution traces and has an impact proportional to the amount of information [53]. [54] Linux tracing systems are split into three parts [55], [56] :

- **Data source** : can be probes (kprobes for the kernel and uprobes for the userspace) or tracepoints (kernel tracepoints, **dtrace** probes or ltng-ust tracepoints for userspace programs)
- **Probe** : is when the kernel dynamically modifies the assembly of a program at runtime, while a tracepoint is inserted at compilation time.
- **Way to extract data** : use the virtual filesystem `/sys/kernel/debug/tracing`

To trace an application, the first way is to use built-in tools of Linux, like **perf_events** for CPU profiling, stack profiling, trace-dump, **ftrace** for live tracing and counting events, and finally **eBPF** for kernel summaries.

To collect performance data about containers, open source profilers like **mpstat**, **iostat** can be used from inside the container or from the host. But it can be challenging to find the origin of the data provided, especially when the number of containers grows[20]. Some tools are specifically designed to monitor containers like **docker stats** and **cAdvisor**. But there is no graphical user interface, and the information provided is more suited to monitoring than tracing [57].

Ftrace **ftrace** is a dynamic tracing system, so the kernel functions are changed when **ftrace** is activated. For every function, the kernel will add instructions to notify the tracer each time the function is called [58]. When **ftrace** is not activated, there is no overhead. The

overhead depends on the number of functions instrumented[59]. It can be used for analyzing latency and performance issues that take place in the kernel, outside of user space.

In conclusion, **ftrace** has very useful capabilities like tracing and counting, but the interpretation of results is really difficult on large architecture analysis [60].

Perf_events Perf is an event-oriented observability tool. It is part of the Linux kernel. This tool is suitable for event counting, profiling, and static and dynamic tracing. Static tracing is achieved with trace recording, whereas dynamic tracing is achieved with probes. It is a tool complementary to **ftrace**.

ktap **ktap** is a script-based dynamic tracing tool for Linux [61]. The main advantage of this tool is that it does not require compilation for every script, because it uses bytecode. So it does not require compiling a kernel module and has greater portability than others.

dtrace4linux **dtrace**⁶ is a performance analysis tool for operating system kernel and device drivers as well as user-level software. **dtrace** also provides a script language to filter data before being presented to the user. So it can reduce the overhead and is usable with sensitive data. This tool is a dynamic tracer, as it dynamically patches live running instructions. However, this tool comes from Solaris and implies license concern for Linux integration and GPL. **eBPF** and **systemTap** were responses to it.

eBPF Tracing scheduler events can be costly, but **eBPF** reduces this cost by using in-kernel maps to summarize data[62]. The main difference of this tool, compared to the two others presented earlier is the operating mode. **eBPF** is a virtual machine inside the kernel, which can run programs on events. It allows in-kernel trace aggregation and filtering. [63]

2.4.1 Lttng

LTTng is a high-efficiency full-system solution to trace bare metal, kernels, and user space. **LTTng** helps understand the issues in a system when some program fails and when the amount of data to record is huge. The community around the tool makes it one of the most popular kernel tracers. This article [64] explains the original motivation for the tool development, the different modules that implement its architecture, as well as the architectural choices made. It also explains the tool performance and its capabilities in userspace mode.

⁶<http://dtrace.org/blogs/about/>

This other article [65] explains the goals of the tracepoint hooking mechanisms. The advantage of tracepoints is the low overhead (no trap or system call), the dynamic activation, and the signal, thread, and NMI-safe execution. [66]

Here is an overview of the LTTng architecture.⁷

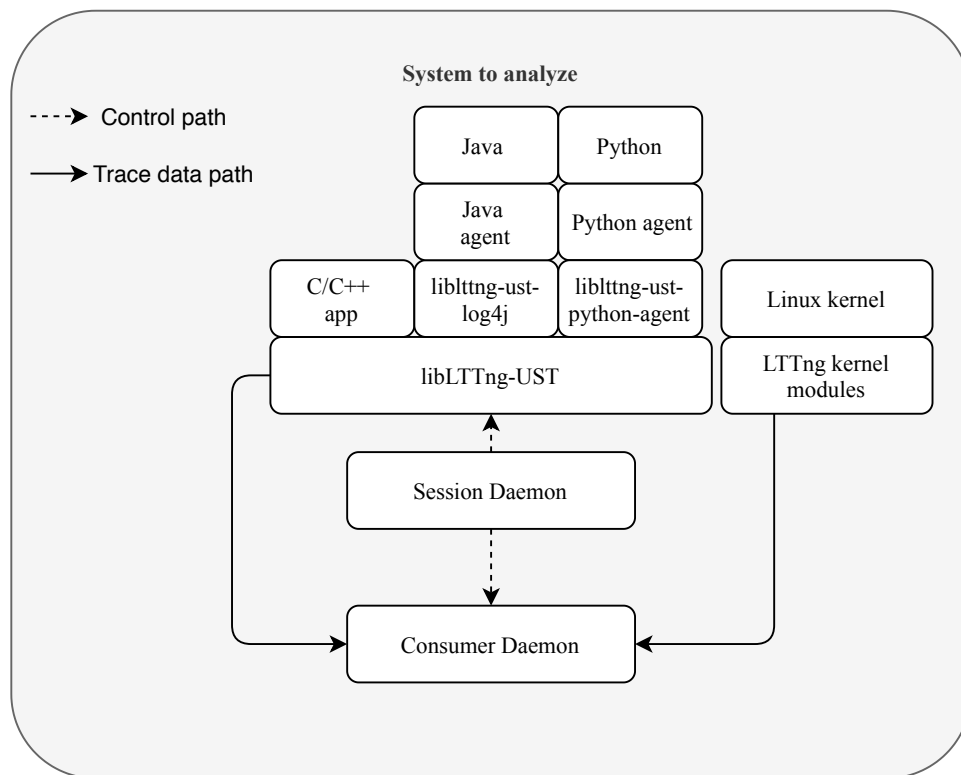


Figure 2.10 Overview of LTTng tracing tool components

2.4.2 Discussion

`ftrace`, `perf_events`, `eBPF`, `systemtap`, `ktap` and LTTng are some tools for collecting data with similar features. `Ftrace` was designed with the aim of grouping together all of the kernel tracing tools. The main advantage is that it is a standard tool, usable for the majority of kernel versions and architectures. It is, therefore, a tool more suited to the kernel and not to tracing applications.

However, `ftrace`, `perf_events`, and `eBPF` are actually part of the core Linux kernel; this avoids installing kernel modules. Linux `perf_events` is mainly used for CPU profiling.

LTTng is undeniably the most complete tool to date in terms of versatility. It efficiently collects information in user space and in kernel space.

A good practice for kernel tracing is to use built-in tools first and then `systemtap` or LTTng for more specific needs or for tracing applications in userspace mode.

⁷<https://lttng.org/docs/v2.12/#doc-tracing-session>

Some tools have also been specifically developed for tracing distributed systems, which is the subject of the following part.

2.5 Tracing methods for distributed systems

The tools described in the previous part are suitable for collecting traces on a userspace process, or the kernel, of a computing element. Some other tools have been developed specifically for distributed systems.

Some of them are now part of The Cloud Native Computing Foundation⁸ (CNCF) which is an initiative aimed at promoting the development and adoption of tools specifically developed for "cloud native"⁹ software.

Jaeger [67] is possibly the best example of tracer developed for distributed systems. It leverages the Open Tracing [68] instrumentation libraries and can be used for Root cause analysis, Distributed transaction monitoring, or Distributed context propagation.

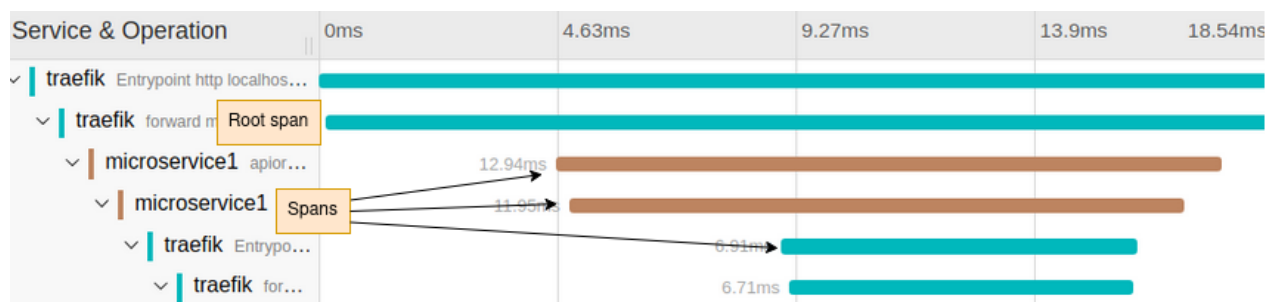


Figure 2.11 Example of trace and spans in Jaeger

Dapper a tool developed by Google[69] and able to handle large traces (generally of one terabyte per cluster per day). In the trace tree, nodes are basic units of work and are called *spans*, a key concept in distributed tracing. A trace represents the execution path (user request or transaction) of a distributed system, while a span refers to a single request within that execution path (function calls). And a trace is a series of tagged time intervals called spans. This concept has advantages like the ease of sampling and tracing nested distributed request, but also drawbacks like the lack of links with operating systems traces.

Zipkin [70] has its origins in **Dapper**, the distributed systems tracing infrastructure of Google[69].

These two tools share the same overall architecture and are comparable to the point of being API compatible. Indeed, it is possible to use the **Zipkin** instrumentation libraries with the **Jaeger** collector.

⁸<https://www.cncf.io/projects/>

⁹open source software stack to be containerized, where each part of the application is packaged in its own container, the global architecture is dynamically orchestrated

This article [71] summarizes the enterprise usage of tracing tools in 2017. According to a survey, **Zipkin** is the most represented tool in this paper, followed by some custom tools inspired by **Dapper**.

Thus, the choice of tools specific to distributed system monitoring revolves around those mentioned above with an active community.

The lack of standard in trace transmission is the subject of a working group, which is trying to develop a new tool called **OpenMetrics**¹⁰. The goal of this tool would be to standardize both the format and the traces transmission means. This is really relevant for decoupling the tracing tools from the trace visualization tools.

Although new tools and working groups are interested in this problem, it is not recent, as evidenced by these two articles [72] [73] dating from the early 2000s. They expose issues that are still topical in distributed systems. The number of components and the communication systems used inevitably lead to the need for efficient tracing. The three-part method of collecting traces, post-processing, and building visualizations exposed here is still used today. This shows the real need for appropriate trace collection and visualization solutions since the appearance of a large-scale distributed system. However, these methods applicable to RPC systems have not been adapted to current MOMs, creating a gap.

This article [74] presents the creation of a custom tracing facility system for a research laboratory. The authors testify to the absence of solutions meeting their needs at the time of writing the article. In these other articles [75] [76] [77] [78] [79] [80] [81] [82] [83], the authors presents platforms for collecting traces in distributed systems

Finally, the effectiveness of an instrumentation system also depends on the location and the care taken to optimize the instrumentation of the source code. This problem is illustrated by this article[84] in which the authors discuss the effective instrumentation techniques at the application code level.

This patent [85] describes the challenges related to distributed tracing systems and presents an architecture based on RPC messages. This work raises the point of the transmission of messages related to tracing in a distributed system. Indeed, the instrumentation of messaging systems in distributed systems itself requires messages to transmit information. The overhead of this additional layer must be taken into account, and the desired level of reliability in the transmission of messages is also a significant issue.

In conclusion, the analysis of the literature underlines the interest and the need for tracing tools applied to distributed systems, but also the challenges that these tools must address.

¹⁰<https://openmetrics.io/>

Ongoing projects give rise to efficient tools specifically adapted to the tracing of distributed systems. However, none is targeted on the analysis of the messaging between elements of the distributed systems, and more precisely to the containers.

2.6 Trace viewers

After collecting information on a system using tracing software, it is necessary to interpret this data. However, these quantities of data in textual (or binary) data format are impossible to use without software for visualizing traces. The purpose of this type of software is to generate graphs and effectively format the data in tabular form, for example, to allow its exploitation.

Visualization tools can be classified into two categories. On the one hand, the tools made explicitly for trace visualization are usually suitable for a precise tracing tool, and on the other hand, the more general data visualization tools, can be used by converting the data format of the trace.

2.6.1 Tools for trace visualization

Trace Compass is an open-source trace viewer used to solve performance or reliability issues by reading and analyzing execution traces collected at the userspace or system level.

It provides views and graphs to extract helpful information from traces in a graphical way, with searching, filtering, and scripting.

Trace Compass handles traces larger than available memory, and can correlate traces of heterogeneous systems. Many trace types are supported, like **UFTrace**, **LTTng-UST**, **OpenTracing**, **CTF**, **Perf**, **PCAP** and many others, as well as custom XML. This tool can provide views like critical path, discs I/O, flame graphs and charts, scatter plots, and CPU usage.

Trace Compass is a plugin for the Eclipse IDE. Thus, the coupling with Eclipse was high and limited to this IDE. There was no possibility of scaling to multiple nodes within the Eclipse IDE for parallel trace analysis.

To bypass this limitation, the architecture has been reworked, with a smaller coupling between the UI and the data processing backend. **Trace Compass** is now using a standard HTTP REST API (TSP : trace server protocol) between the two components. This new implementation is developed in parallel with the regular **Trace Compass** version, and **Theia** is the new user interface ¹¹.

It is important to note that there are several approaches that were investigated to parallelize the analysis of traces in **Trace Compass** [86]. [87].

Theia is an open-source data viewer like **Trace Compass**, but based on a distributed architecture. It can run inside a docker container and provides a web frontend to the user through

¹¹<https://www.youtube.com/watch?v=Fysg1mOadik>

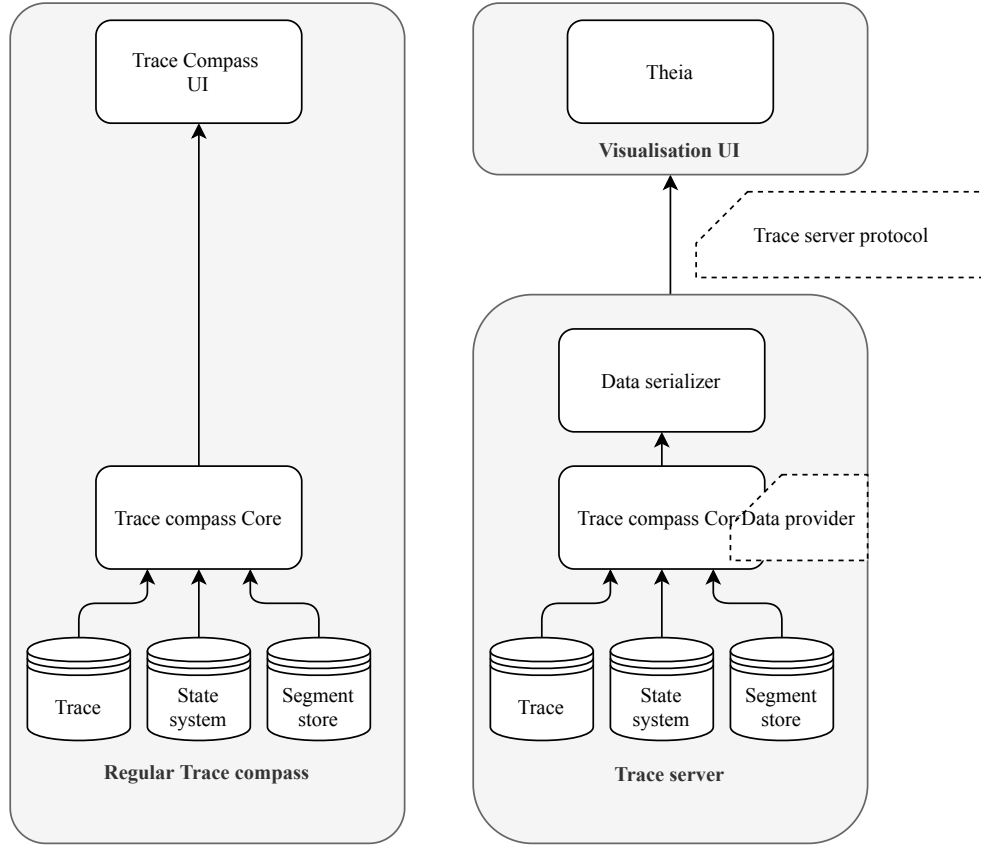


Figure 2.12 Trace Compass architecture

a browser. This tool is new and still under development.

As explained in the previous part, the (**trace server**) backend exposes an API (**Trace Server Protocol**) to provide JSON records for a trace composed as a time series (pair time-value). The API provided (Trace Server Protocol) by the Theia backend, Trace Server, can be diverted to be used by other analytics and interactive visualization Frameworks.

Theia is a browser extension developed in React with dynamic libraries like **Chart.js** and **agGrid**. Thanks to the loose coupling between the frontend and the reworked backend, the tool is now scalable and can handle traces larger than the local disk.

Kernelshark is a trace viewer for kernel trace, not as powerful as Trace Compass, but more focused on the analysis of CPU events [88]. It is a suitable frontend for **ftrace**. Again, the flexibility of the views is limited to the time graph.

Vampir [89] [90] supports the Open Trace Format and is designed for massively parallel programs. **Vampir** allows tracing parallel applications as well as accelerator-based paradigms

like CUDA or multi-processing like MPI. The interface is more suitable for event-based analysis, like CPU events. The views are more flexible than with the previous tool, and the performance radar view is really efficient for memory allocation analysis.

Paraver [91] is a tool to analyze distributed memory allocation and message passing in distributed systems. The visualization process is the same as previous tools but focuses on parallel code analysis.

HPCTOOLKIT [92] focuses on high-performance computing. It is a toolkit for both collecting performance measurements and presenting performance metrics (with the `hpcviewer` module). Three kinds of views are available, but with little flexibility for customizations.

To summarize, **Kernelshark** is a suitable tool to understand the CPU events, **Paraver** and **Vampir** are suitable tools for parallel applications analysis, and **HPCTOOLKIT** focuses on the HPC workflow analysis, with specific functions for that use case. For more general use, **Trace Compass** and **Theia** offer various functionalities for CPU events analysis and userspace applications analysis.

2.6.2 General data viewers

Other software tools are also available for analysis and the creation of graphs from data in text format. These tools are mainly intended for the analysis of logs coming from a system. However, by adapting the trace files, it is possible to use these tools for trace analysis.

Grafana is one of these tools and is used in many fields, with data visualizations from heatmaps to histograms. This tool provides a complete online solution, in the web browser, with authentication. It relies on Prometheus to store data as time series. The main advantage of an online solution is that a development team can share views and portions of graphs with time limits, and dashboards can be built to monitor a complete system.

Dashboards can be considered as a canvas for a painter. The user can put as many graphs as he wants on them, as well as tables. These dashboard setups can be saved and shared. The graphs are dynamic, and a lot of plugins can extend the variety of graphs. The database can be queried with the data source component that allows configuring one or several different data stores (Graphite, InfluxDB, SQL, JSON, log text...). It includes a query maker in order to get data from the database. The graphs include a lot of customization for data visualization and ranking.

Grafana is designed for analyzing and visualizing metrics [93] and does not allow full-text data querying. As a result, this tool is not usable out of the box to visualize the output of a tracing tool. However, with a small parser program that converts the CTF¹² format to JSON, it is possible to build powerful views with much customization.

Kibana provides a tool for exploring, visualizing, and building dashboards on top of the log data stored in **Elasticsearch**. This tool is used primarily for analyzing log messages. As it is tightly coupled with Elasticsearch, it does not support any other data source type. But this limitation can be overcome with programs that convert data with **Logstash**; before that, they can be used in **Kibana**.

Vizceral¹³ is an open-source data visualization tool to understand traffic patterns when a failover happens. This tool is really limited in functionalities and significantly less general than **Kibana** or **Grafana**. However, the view provided with a WebGL canvas of the traffic graph is interesting and can be used to illustrate the messaging between containers. It is a view suitable for the research work described in the next chapter.

Finally, there are several other open-source visualization tools, but they are less general, typically focused on system metrics visualization like CPU usage, disk activity, and web server load. This section aimed to explore the main tools used in industry and academia to visualize data.

2.7 Critical path computation and analysis

The critical path corresponds to the shortest path between the elements of a network and represents the dependencies between the tasks. The critical path has been applied early on to software analysis; notably, Yang and Miller [94] used an execution graph to determine the critical path of software execution in 1988. The critical path is a suitable method to solve performance problems and plan the realization of complex projects [95] in numerous domains such as activity networks with time constraints [96], networks with fuzzy activity [97], TCP transactions [98], and fuzzy numbers [99]. The reader is referred to the work of Montplaisir et al. [100] for a detailed introduction to the critical path.

Critical path allows tracking program execution and precisely profiling programs during execution. Indeed, a program execution speed is directly tied to that of the critical path [101].

¹²<https://diamon.org/ctf/>

¹³<https://github.com/Netflix/vizceral>

For instance, Tullsen and Calder [101] introduced a critical path profiler, and Ball and Larus [102] proposed a method for critical path profiling with minimal overhead.

Since the critical path computation is related to searching for the shortest path, one may use well-known algorithms such as Dijkstra [103]. However, due to the sequential nature of the Dijkstra shortest-path algorithm, parallelizing the computation yields only a marginal average speed-up of 10%. The reader is referred to the work of Giraldeau and Dagenais [104] for a thorough description of the main algorithm used to compute the critical path. This work has been used by [105] to process critical paths on web applications.

Critical path methods in distributed systems and virtualized environments require careful adaptations as described by Nemati et al. [106]. Notably, since the critical path comprises dependencies between multiple trace files collected on different nodes, ensuring a near-perfect time synchronization between traces is crucial. The time synchronization accuracy can be increased at collection time with an efficient clock synchronization of traced systems [107] such as with the Precision Time Protocol (PTP) or at analysis time with an offline trace synchronization algorithm such as the one presented by Poirier et al. [108]. The reader is referred to the work of Wu et al. [109] for a complete and detailed description of the analysis of the large distributed systems.

Some related work by [110] uses a clustering algorithm applied on critical path to detect anomalies in web applications.

Critical path computation requires sequentially processing every trace file on a single analysis node, which is problematic for numerous large traces. Schulz [111] showed that it is possible to extract the critical path for applications running on up to 128 processors. Nevertheless, there is a real and necessary need to improve the critical path computation.

2.8 General discussion

Container-based architectures are a suitable solution for distributed architectures that require isolation and high efficiency. This first part highlighted the lack of benchmarks in the literature on containers technologies targeting HPC systems. The experiments in the existing benchmarks were done on small architectures, and Singularity is rarely compared.

New distributed systems often involve a large number of computing elements located in different places and on heterogeneous hardware. So, messaging systems are fundamental. The efficiency and reliability of message-oriented middleware were discussed, and solutions with or without brokers were compared. This illustrates the challenges of inter-process communications and the need to analyze them in case of a slowdown or congestion.

As a response to this issue, numerous tracing tools are available. But many mostly focus on one usage, and the user interface is often far from being user-friendly. To counter this weakness, trace viewers convert data from tracing tools to graphs and metrics, for which there is a wider choice of visualization frameworks.

This literature review showed that most current trace analysis tools can not perform adequately on large traces, and do not support well container based systems communicating through Message Oriented Middlewares. A promising avenue to scale analysis tools is parallel distributed trace analysis. Nonetheless, one of the most difficult analysis to scale to large distributed systems is the critical path computation, since it may involve searching through the whole execution graph. The parallel distributed trace analysis, for critical path computation, in the context of large distributed container based systems, is therefore the focus of the work presented in this thesis.

CHAPTER 3 OVERVIEW

Three articles are presented, in submission chronological order, in chapters 4, 5, and 6. Each one addresses one of the research objectives defined in the first chapter and follows a common thread. The main objective of this thesis is to improve the analysis tools for large distributed systems analysis and more specifically container clusters using message-oriented middleware communication systems. The goal is to obtain the critical path of large distributed applications (cloud based or using distributed computation systems like MPI). A new optimized algorithm to compute the critical path is proposed.

A brief description of each article follows :

Chapter 4 presents the paper entitled "Advanced tracing methods for container messaging systems analysis" submitted to the Journal of Cloud Computing - Springer. This article provides an overview of the current architectures and methods used for inter-container messaging. Recurrent performance issues are characterized, as well as the ability of current tools to monitor this type of architecture and handle the massive amount of data produced during the tracing of such architectures. New methods, based on tracing, are proposed to understand better the origin of bottlenecks in large container cluster messaging systems. These two new methods provide a detailed picture of the analyzed systems, and take in account environmental constraints. This first contribution is the basis of subsequent work, as it allows trace collection on container clusters, for instance for whole system critical path computation.

Chapter 5 presents the paper entitled "Distributed Computation of the Critical Path from Execution Traces" submitted to Software: Practice and experience. Tracing methods allows for understanding performance issues, and it is now possible to collect traces on these due to the good scalability of tracing systems. However, it is often challenging to store the generated traces efficiently, and even more difficult to analyze them due to the poor scalability of current trace analysis software. This research aims to improve the scalability of tracing methods to provide the same granularity of analysis, even for very large distributed systems. The focus is on critical path analysis, a method used to identify the dependencies between process events on different nodes. The previous method did require to transfer and analyse all the traces on a single node, which was a strong limitation in terms of scalability. The new method introduces a distributed algorithm that allows critical path analysis of several traces located on remote nodes. File transfers are avoided, and the computation is now distributed on multiple nodes. Two architectures are also proposed for efficient trace collection and further analysis. These last introduce improvements to the data structures used to store traces, and

the communication patterns between computing nodes for trace file transfer. This second article highlights, as future work, some possible performance improvements linked to the indexation of trace files and usage of pre-computation.

Finally, Chapter 6 presents the paper entitled "Scalable Distributed Computation of the Critical Path from Execution Traces" submitted in "Concurrency and Computation: Practice and Experience", which improves on the performance of the distributed algorithm using pre-computation and indexing. Indeed the previous article highlighted the high cost of the critical path computation across multiple nodes, as well as scalability issues. This paper introduces a new algorithm to compute the critical path more efficiently, thanks to pre-computed indexes for the inter-node communications. It greatly reduces the computational cost of critical path processing on multiple remote nodes. A benchmark was conducted on representative usecases such as MPI scientific computations using a cluster of 20 nodes. This last article highlights the possibility to extend the pre-computation and usage of index files to the processing of other analysis and views.

CHAPTER 4 ARTICLE 1: ADVANCED TRACING METHODS FOR CONTAINER MESSAGING SYSTEMS ANALYSIS

Authors Pierre-Frédéric Denys, Michel R. Dagenais and Martin Pepin.

Submitted July 15th 2021 in Journal of Cloud Computing

Abstract Containers are increasingly used for software deployment, because of the modularity they offer for packaging and isolating applications. However, this implies a reliable communication system between computing elements in different containers. Thence, conventional messaging systems have evolved and adapted to increasing loads and Edge Computing. Inter-container communications, with Message Oriented Middleware, provide insight into the execution of distributed applications, as well as a deeper input for analysis. However detecting message losses and slowdowns on this type of infrastructure is a challenge. Actual tracing solutions for this task were compared to identify shortcomings and possible improvements. New tracing methods are proposed to address these shortcomings and open the door to more versatile tracing tools. This paper focuses on the approaches taken to extract information from messages, and achieve advanced analysis. Two new methods are presented, each providing a detailed picture of the distributed system, while being better suited for different use cases, depending on the environmental constraints.

4.1 Introduction

Over the last few years, container usage in software systems has steadily increased. Instead of deploying monolithic applications, microservices are now used to allow the different components of an application to be released and scaled separately.

For instance, in the IoT environment, connected cars or wearables often embed computing elements to pre-process some data. All these mobile devices need to exchange some messages with a remote service. To improve the response time in such environments, the application code is often located near to the final user. Moreover, it is relocated on another computing platform, when the mobile device moves. This paradigm is called *Edge Computing*, and the application code is typically embedded in Docker containers. Containers allow rapid launch, migration and scaling.

These usage scenarios rely on sending messages over the network, between computing nodes

inside a cluster, and between nodes distributed on public networks. This message transmission is a challenge from the performance, security and reliability point of view. To achieve an efficient data transfer, Message Oriented Middleware is most often used. These messaging systems are an evolution of the request-response communication protocols (like Remote Procedure Calls, RPC) previously used for inter-process communication in distributed systems. Messaging systems can suffer from data loss or bottlenecks, especially on public networks with the limited bandwidth available in IoT applications. To solve these issues, service providers and developers need to know where the bottlenecks are located, and be able to detect lost messages and latencies. Tracing tools are invaluable to understand many of these issues, by analysing the transmission time of messages in every layer of the architecture (library, kernel, network).

This article provides an overview of tracing solutions suitable for container-based environments like Docker. The actual solutions to collect the information will be compared in terms of capabilities, overhead and intrusiveness. The question is whether current tools can effectively monitor this type of architecture, including the ability to monitor several systems at once, and collect a huge amount of data efficiently.

The data collection can be performed from two points of view. On the system developer side, there is often no access to the container itself, but the system calls of the host can be analyzed. On the application developer side, there is only access to the containers and application code, and no access to the containers hosts. Two scenarios are therefore possible, depending on access constraints to the different components of the architecture.

The main contribution of this work is to provide efficient solutions for the instrumentation of inter-container messaging systems, taking into account the constraints in this type of environment. The first method focuses on the message transmission analysis, at the container host kernel level. The second method focuses on the messaging library instrumentation, inside the container. These two new methods share the advantage of being decoupled from the application source code. Another important contribution is the efficient analysis and presentation of the traces obtained.

4.2 Related Work

4.2.1 Container background

The invention of containers was motivated by the need for process isolation. Initially, **Chroot** (*for change root*) was developed to restrict the view of the file system for a process. Indeed, it is not possible for such a process to access files and binaries outside the restricted environment

[11]. Then, this was extended with the ability to isolate other resources, such as network interfaces. Two mechanisms have thus been added to the Linux kernel to allow for greater isolation granularity[12]:

cgroups (control groups) provide a mechanism to limit the amount of resources that a process group can use. They ensure that one or more containers do not drain all the system resources.

namespaces limit the scope of a process group, and what the processes in the group are able to see in the operating system.

The combination of **namespaces** and **cgroups** is the basis for a container. Using this more sophisticated isolation system, **LXC** was one of the first implementations of Linux container managers, before Docker. Finally, **Docker** offers an entire ecosystem for container management [112]. Figure 4.1 is an overview of the components of Docker, and the kernel support involved.

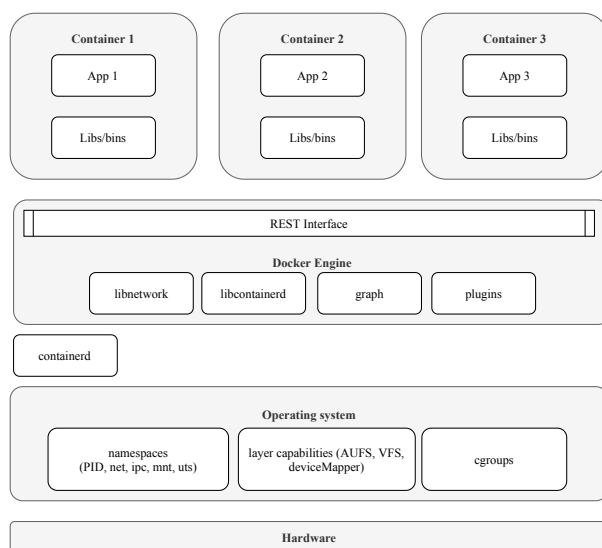


Figure 4.1 Docker architecture

This grouping into containers may cause problems when tracing the execution of processes. The container identifier is not available in the process metadata. Moreover, the (virtual) process ID seen from inside a container differs from the real host process ID. In addition, virtual networking is used for communications between containers on the same host or to connect to the external network. Therefore, monitoring a container from the host is interesting because it offers visibility on the processes, the isolating layer and the virtual networking.

4.2.2 Container communication patterns

Inter-process communication in distributed applications used to be based on RPC (remote Procedure calls). With time, a more advanced communication paradigm, MOM (message-oriented middleware) [40], became more prevalent. MOM supports additional communication patterns (request-reply, publish-subscribe)[38] as well as asynchronous communications [39]. Indeed, the default behaviour of RPCs is to block the caller execution, while the server process handles the request. By comparison, messaging systems, are based on queuing: the producer service posts data in the queue, and the consumer gets data from the queue [41].

Several MOM are available and use different approaches for message transmission. AMPQ, MQTT, Apache Kafka and ZeroMQ are some examples. These studies[42] [43] provide detailed comparisons on criteria like message throughput, latency, messaging patterns, and security.

ZeroMQ (ZMQ) is an open source messaging library, and uses a message queue. **ZMQ** is based on a brokerless architecture, unlike the three other messaging protocols cited before. The operating mode is similar to Berkeley sockets [48]. On the one hand, there is no single point of failure. But on the other hand, there is no persistence support, so there is a greater chance of losing messages, as there is no quality of service implemented.

As mentioned in [42], the brokerless architecture results in improved performance. As **ZeroMQ** is brokerless, messages are sent virtually immediately, neither producer nor consumers are reaching their saturation point in the experiments. The only bottleneck observable is the physical network bandwidth. The brokerless architecture gives **ZeroMQ** flexibility for different topologies, and is the basis for other communication systems like in [49].

ZeroMQ provides five basic patterns for message exchanges: synchronous Request/Response, Asynchronous Request/Response, Publish/Subscribe, Push/Pull, Exclusive Pair. Here is an example of two patterns commonly used. Other patterns may be found in the **ZMQ**¹ guide.

In conclusion, the brokerless design of **ZMQ** can lead to lost messages, while messaging systems with brokers are more likely to present bottlenecks. Therefore, a monitoring solution is needed for both communication patterns. On the one side, the time during which the application is blocked needs to be monitored and minimized. On the other side, the message queue needs to be supervised and lost messages need to be detected.

In the next section a set of available methods will be analyzed. These methods, mostly based on tracing, are used to measure the response time for communications between the components of a distributed system.

¹<http://zguide.zeromq.org/page:all>

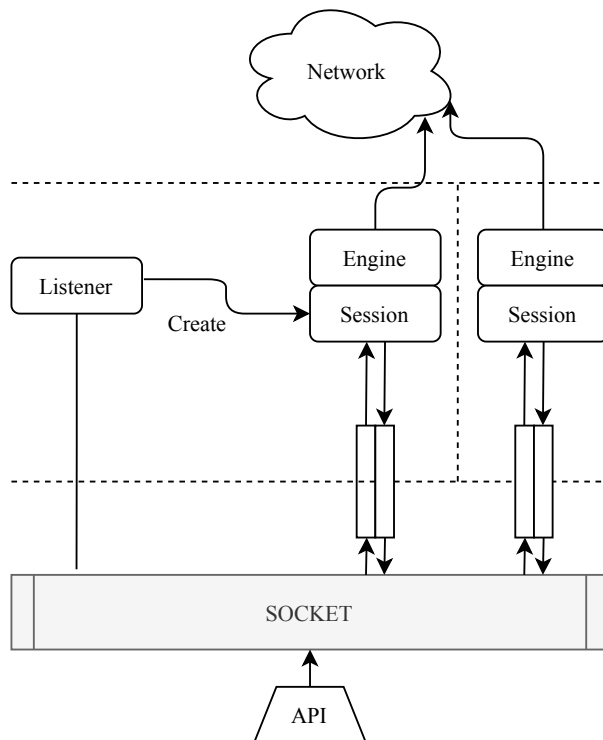


Figure 4.2 ZeroMQ architecture

4.2.3 Tracing tools

Identifying bottlenecks is an important concern in a container architecture. Performance problems may reside either in the host or in the container, and system metrics may be a useful tool for their detection. Tracing tools can extract all the raw data, from the applications or the kernel, to derive metrics and help to find bottlenecks.

A trace is a series of events over time. Events are collected at tracepoints during program execution. Each event has a type and payload. Tracing uses events as input for analysis and is used to profile applications, find abnormal states in applications, investigate real-time deadlines, find memory or load issues and investigate concurrency problems.

Tracing can be used for several aims:

Ensure correct system behaviour : Tracing can be used in quality assurance, to ensure that the system is responding as expected.

Monitoring real-time and high criticality applications : A tracing system can be implemented to monitor abnormal response time or behaviour.

In order to analyze the Linux kernel or an application, there are two approaches. On the one

hand, static analysis considers all possible executions and does not impact the execution, but uses complex calculations on the source code [52]. Because of that, it is often not applicable to large systems. On the other hand, dynamic analysis, or runtime verification, is easier to analyze but may impact the execution, and only covers the cases actually executed.

As the complexity of distributed systems has greatly increased with time, tracing has become indispensable in most performance monitoring solutions. The lack of standard in trace transmission is the subject of a working group, developing a new protocol named **OpenMetrics**². Although new tools and working groups are interested in this problem, it is not recent, as evidenced by these two articles [72] [73] dating from the early 2000s. They expose issues that are still topical in distributed systems. The number of components and the communication systems used inevitably lead to the need for efficient tracing. Some more targeted studies on containers [113] focus more specifically on the mode of instrumentation discussed in this article [114].

Several tracing tools are available on the kernel side:

ftrace - is part of linux kernel and is a versatile tool to trace the linux kernel. It offers function tracing and dynamic instrumentation.

LTTng - is set of kernel modules, to trace kernel and userspace applications. The architecture is designed to avoid synchronization between CPU cores, thanks to per-CPU variables and atomic operations.

perf - perf was initially designed for profiling and accessing performance counters. It also offers tracing functionality. It is suitable to characterize the behaviour of a program by profiling the time spent in each of its functions.

systemtap - was made in order to provide a scripting language to deploy custom probes (inserted in kernel with kprobes after compilation).

ebpf - newer and easier to deploy than systemtap, ebpf is more an aggregator than a tracing tool. It allows users to insert probes in the kernel using kprobes, and to associate code snippets to each probe.

And on the userspace side:

LTTng-ust - is the module of LTTng for userspace instrumentation. It features a low lockless design, and is usable even in signal handlers.

vampirtrace - focuses on parallel program instrumentation.

These tools are suitable to collect traces from userspace processes, or the operating system

²<https://openmetrics.io/>

kernel, of a computing element. Other tools were developed specifically for tracing distributed requests, like **Jaeger** [67], **Zipkin** [70] and other tools:

[75] [76] [77] [78] [79] [80] [81] [82] [83]

However, none of these methods, developed to trace nested requests in distributed systems, are suited to current MOMs and container architectures, and this creates a gap.

4.3 New instrumentation methods

A straightforward solution consists in instrumenting the source code of the application running inside the container. We saw in the previous section that existing tracing tools are suitable for this task. However, this instrumentation process is invasive and time consuming. It requires detailed intervention at the source code level, which must be updated for changes to the message sending functions, and repeated for new applications. Nevertheless, this method is actually widely used in the industry to debug and solve performance problems in monolithics applications.

When the instrumentation needs to be performed on hundreds of containers, a better solution would be to catch the messages at a lower level. It can be at the messaging library, at its language binding API, at the container API, at the host kernel, or at the network interface [115]. The interest of this approach is to avoid the labor intensive application specific instrumentation work. Since all the interactions between the different distributed components go through messages, this still provides a lot of insight into the application behaviour and performance.

This is the approach taken by the two new methods proposed in this paper. The first relies on instrumentation on the system kernel side, and the second on the messaging library side. Figure 4.3 shows the architecture used in both experiments, and the location of instrumentation.

4.3.1 Kernel Level Instrumentation

This new method allows to trace the messages exchanged between containers at the kernel level. The experiment has focused on the ZeroMQ library, but the instrumentation process would be the same for all Message-Oriented Middleware and other messaging systems. It does not require any modification of the messaging library or application source code.

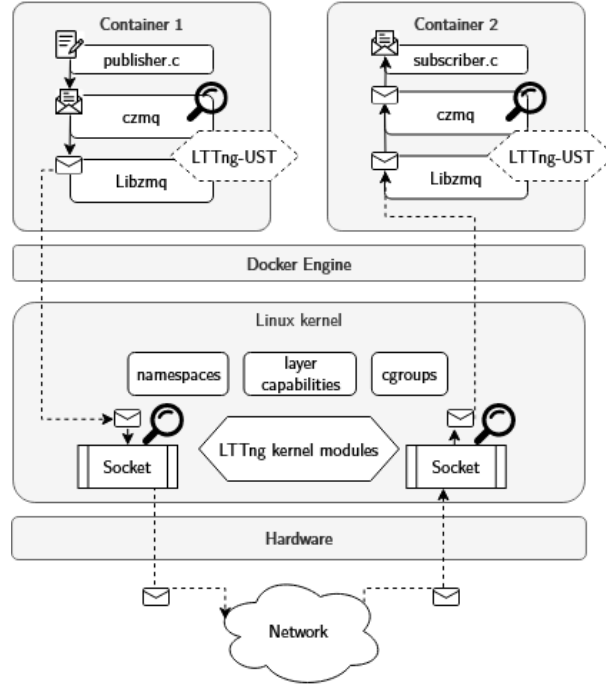


Figure 4.3 Architecture used for the two methods and instrumentation location

System Architecture

The typical architecture consists of applications, running within docker containers, that exchange messages through their network interface. It can be several applications, or a single one, split into microservices. Hundreds of containers may run on several hosts. The messages are exchanged through the internal virtual network between containers within a node, or on the external network. In this case, the amount of data collected is significant and a filtering system is needed.

In contrast, on Edge Computing networks, a small number of containers are running on each host. As a result, the number of Edge Computing hosts is higher than in the first case. Thus, there is less data to collect on each host, but a larger number of trace files are produced. An efficient correlation system is needed to bring this data together into a global analysis node or cluster. Even if this is not the core of this paper, some guidelines will be given here on that topic.

The goal of this first method is to be able to trace the messages exchanged between containers from their host. This method implies a full access to the host operating system kernel.

Instrumentation Algorithm

In order to collect information about messages exchanged between containers, one possibility is to trace the kernel of the container host. The challenge is to get the process number of the program running inside the container, from the host point of view. As Docker uses namespace isolation between containers, getting this information is not straight forward. The instrumentation method needs to watch the system calls linked to network sockets on the host kernel, to trace the messages sent by the applications inside the containers.

The second step is to identify formally the process running inside containers. The `lsns` tool³ used to list all namespaces on a linux system can provide the namespace linked to a particular task with its `pid`. This method allows the classification of processes by containers id.

At this point, it is possible to filter by containers the processes running on the host machine. The final step is to add a context field to the trace when a `sendto()` system call happens. This field contains the information about the message exchanged: the message payload and the recipient. This step needs to be carried out during the trace collection. In this experiment, LTTng was used as tracing tool. A context field `fdpf` is added for tracing. When it is activated during tracing, it will run a function to get the payload and recipient of the message.

This function uses the kernel structures in order to get this information and is triggered for `sendto` syscalls only.

- For each `sendto` syscalls, get the file descriptor table linked to the process with the kernel data structure.
- For each file descriptor, get the inode number
- If the filedescriptor is a socket
 - Get the kernel structure linked to the socket
 - Then, get the `inet` structure linked to the socket
 - Collect all information about the message like the recipient IP

The same process is performed for the `recvfrom` syscall to get the information about received messages. After the compilation of the LTTng tracing software, with the new context and associated handler, the kernel tracing becomes available.

The recipient IP address is visible in the traces, as well as the message payload. Each trace file contains the data about one or more containers, and the tracepoints are identified

³<https://man7.org/linux/man-pages/man8/lsns.8.html>

with the container ID. Then, these files are analysed together, with a trace viewer like Trace Compass, to provide a global view of the system. This shows the messages exchanged between containers running on different hosts. As a result, messages can be followed from the sender to the receiver container.

With this approach, it is possible to compute the transmission time, and detect if a message never arrives at destination. The interest of this analysis is the added value brought by the kernel activity recorded. It is possible to make a link, between an abnormal situation in the message transmission, and what is happening on the host at this moment.

The originality of this method lies in the use of namespaces at the host level, in order to follow the routing of messages, between containers located on the same host and on other hosts. To understand the usefulness of this method, a representative use case will be presented in the next section, and the overhead of system tracing on the host will be evaluated.

4.3.2 Messaging library level instrumentation

The need for kernel access (and as a result, high permission level) on the host kernel is the main drawback of the first method. In some production environments, there is no or limited access to the kernel host, for security reasons. Therefore, it was important to find a way to instrument the container communications from inside the container.

System Architecture

Due to environment constraints, host kernel access is not always possible. Moreover, in some cases, the containers are running on a third party cloud provider, where it would be unthinkable to instrument the host kernel. In such circumstances, the instrumentation mechanism must be enclosed inside the container.

Applications running inside containers often use a messaging library to communicate over the network. As highlighted in the literature review, Message Oriented Middleware systems are typically used. The application then uses a library to format, encapsulate and send messages on the network sockets. A distinct process, running inside the container, is in charge of the message transmission and queuing, and a higher level library is available for the application. As a result, the instrumentation daemon must run inside the container, and collect the data at the messaging library process level or higher.

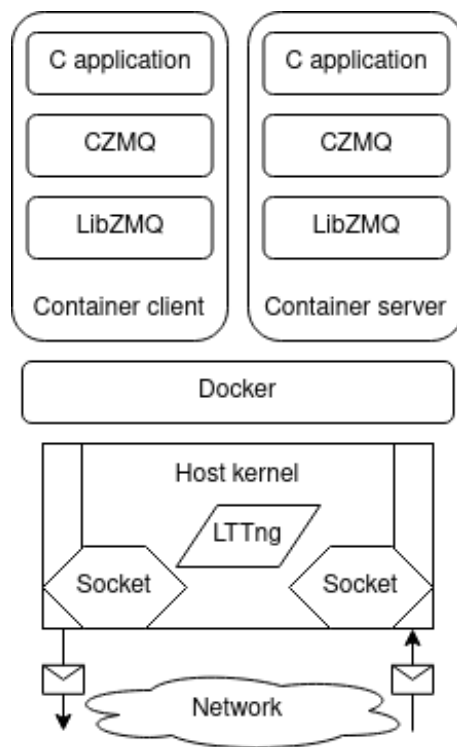


Figure 4.4 Architecture

Instrumentation Algorithm

Instrumentation of the application source code itself would not be a challenge, but would not offer a generic solution, being tightly coupled to the source code of a specific application. Moreover, the information would be limited, it is only possible to detect that a message is sent by the application. It is not possible to see what is happening during the message queuing and transmission, and there is no way to ensure that the message has been sent to the socket. Finally, it is not always possible to modify the application source code, to insert the instrumentation.

The messaging library **ZeroMQ** was chosen here to illustrate the method, but the process is identical with other messaging libraries. To avoid the drawbacks of application instrumentation, the instrumentation was inserted in the **zmq** library, as it is the **ZeroMQ** core. Tracepoints were added to all the functions involved in the message transmission on the sockets, in order to catch the information sent over the network at every step of the transmission.

This solution is suitable but is not ideal, since the binding API **CZMQ**, used between the application code and the library, needs to be recompiled to work with the **zeromq** instrumented library. Also some informations could not be extracted about the messages, as their body

was already formatted for transport by the intermediate library `czmq`. As a result, a relevant solution is to add some tracepoints inside the CZMQ API functions. The API allows to get the information on the message transmission until the message is sent to the socket.

To sum up, various points have to be considered for the userspace instrumentation, especially in the case of an existing application. The level of details available for the analysis depends on the location of the instrumentation. We will now focus on the last solution and the instrumentation process.

In order to work on a real use case, on a production system, the library used by an industrial partner (**ZeroMQ**) was chosen for the experiments. However, the instrumentation method and concerns would be similar for other message-oriented middleware.

On the instrumentation side, an `LTtng` trace point provider was added to the `czmq` source code. Thanks to a compilation flag, it can be activated or not at compilation time. Environment variables could also be used to activate or deactivate tracepoints, which is convenient for applications deployed with a `Dockerfile`, for example. The instrumentation is triggered during the deployment.

In order to obtain detailed information about the messages, the following classes were instrumented: `zstr`, `zmsg`, `zframe` and `zsock`. Depending on the communication pattern (publish-subscribe, router-dealer...), several methods are available to send messages. Also, depending of the type of data to send, the sending method is different. The tracepoints are able to collect the sender parameters (port, address), message parameters (content, destination, `routing_id`, type, UUID), and receiver parameters (port, address).

An issue arised for multipart messages, sliced in frames. It was a challenge to identify those messages. Indeed, in some communication patterns, especially in some message-oriented middleware, it is not always possible to clearly identify the sender of a message received, and if the message is only a frame, part of another message. For example, with an architecture involving several publishers, how to reconstitute the message sequence on the subscriber?

The preferred alternative solution is to add the publisher identifier in the message header. However, since it is not always possible to do so in the source code, an universal and standardized solution was needed. The solution adopted was to set a routing id, in the socket sending function. The id is composed of a date, a flag to tell if the message is a frame of multipart message, and the sender identification (if not already available in the header). As a result, each message comes with an identifier, and a message on the receiver side contains all the needed information for source identification.

Another suitable solution is to override the message transmission functions, and add the pos-

sibility to send a routing id into the body or header of the message. In this case, the message identification is easier, as it is possible to add very specific information, to be collected in trace files.

4.4 Results

This section presents the experiments carried out to validate the effectiveness of the two methods. The detailed specifications of the environment used and the full results are available in this git repository ⁴.

The experiments conducted with industrial partners, involved in this project, enabled us to list the main challenges and common problems, related to the communication between containers, in representative industrial use cases. The most common problems are in the queues, and the storage of messages. Depending on the level of quality of service, when a customer is unreachable, messages can be dropped, or stored. These issues are common on edge networks with unstable wireless networks, for example. It is complicated to monitor the number of lost messages, or to monitor the number of messages waiting in queues, without a custom monitoring system or instrumentation. Nonetheless, this is an important concern for resilient communications.

This paper proposes efficient generic methods in order to solve effectively a large part of the problems encountered. The use cases proposed represent only a small part of the possible problems, but are representative of the most frequent problem classes encountered in industrial use cases.

4.4.1 Kernel level instrumentation

Use case

In the publish-subscribe pattern, one of the most common issues is the slow subscriber. It can lead to overloaded message queues, the container running out of memory, and a crash. A solution to prevent this issue is to set a *high water mark* value, in order to stop to queuing messages when this threshold is reached. When the number of messages queued reaches the high water mark value, the additional messages are dropped silently.

This section presents experiments involving these configuration problems, and how the kernel level instrumentation of the container host allows to detect them. Also, a solution to set correctly the amount of memory, and high water mark value, is proposed to prevent message

⁴<https://gitlab.com/pierrefrederick.denys/tracing-methods-for-containers-messaging>

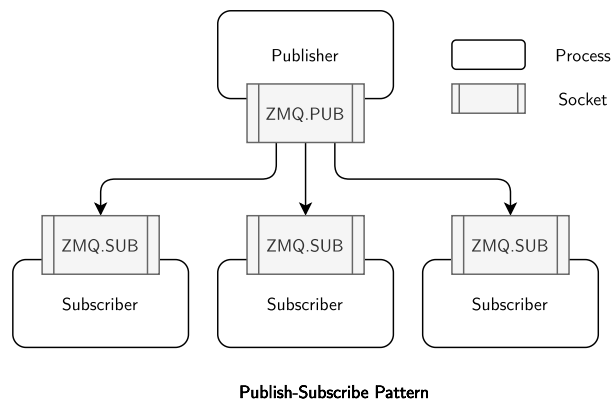


Figure 4.5 Publish subscribe pattern in zeromq

loss. Finally, it will be shown that kernel tracing is a suitable solution to identify the cause of subscriber slowness.

The architecture consists of two applications, running in Docker containers, that exchange messages through the Zeromq library. This simple architecture reproduces the behaviour of an application, deployed as microservices running in Docker containers. A first container is launched with a server implementation, and sends a message with the zeromq library to a second container, launched with a client implementation that listens for messages with the zeromq library.

A server container, configured as described in the previous paragraph, runs a *publisher* python program, binding a `PUB` type socket on port 5601. This program sends messages to the socket in an infinite loop, with the current timestamp as unique content. The timestamp allows the unambiguous identification of messages. It simulates a fast publisher.

A client container, runs a *client* script binding a `SUB` socket and listening on the same port as configured for the server. A pause of a variable random duration, of several ms on average, is added to the receiving loop, to simulate a busy machine with a slow response time. This results in a slow subscriber.

The two containers are running on the same virtual machine, and the kernel of the machine is traced. In order to produce an out of memory situation, the memory of the server container, running the publisher program, is strictly limited, and no swap is provisioned. It is achieved by using the `memory="512m"` and `memory-swap="0g"` options of the Docker API. Then, the tracing is launched with two filters, on the PID namespaces of the two containers.

The `docker stats` allow to display the increasing memory usage in the server container, until it reaches the limit. When the limit is reached, the publishing process is killed. Thereafter,

the tracing session is stopped. When the trace is open with the Tracecompass⁵ trace viewer, it is possible to count the number of messages sent on the socket for a time interval. If the traces of the two containers are opened with the same time axis, it is possible to see the increasing delay between when a message is sent from the publisher container, and when it is processed in the subscriber.

The number of messages queued comes from the difference between the number of messages sent by the publisher and the number of messages processed on the subscriber in the time interval considered. The optimal high water mark of the publisher, and amount of memory needed for the queue, is determined by this analysis.

In this case, the messaging analysis allowed to set up an optimal queue on the publisher, to counter the slowness of the subscriber. Moreover, it also allowed to detect a problem on the subscriber. The queuing of messages on the publisher is a solution for short periods of higher latencies in a subscriber. However, it can lead to dropped messages, when the subscriber remains too slow for a longer period of time, and the high water mark is reached. In that case, a solution should be investigated to increase the subscriber message processing capacity.

While the traces contain the information to compute the average message processing capacity of the subscriber, it also contains information about the whole kernel, like the thread scheduling events and the system calls, if the corresponding tracepoints are activated. The analysis of these kernel events brings a deep insight into the execution of the processes involved, and provides detailed information, useful to identify the root cause of the subscriber slowness.

The defaults in this configuration were the incorrect *high water mark*, the low amount of memory allocated to the container, and the big difference between the number of message published and capacity of subscriber. The inverse situation is when the *high water mark* is set low. The default value for the zeromq library is 1000. In that case, only 1000 messages can be queued (a total of 24*1000 bytes in the current experiment). The behaviour of zeromq in this case is to drop messages when the queue is full. As a result, it leads to an important loss of data.

In the case of a predictable system, for example a sensor that sends data every second, the loss is easily detected on the subscriber. Some of the expected messages in the serie will be missing. The time at which the message was lost can also be easily determined. In the case of an unpredictable publishing system, like a logging system, the detection is more difficult, since there is no expected time, or number of messages, for reception on the subscriber.

For example, in a system where publishers are servers, on which sensors are connected, and

⁵<https://www.eclipse.org/tracecompass/>

the subscriber is a log server, the messaging pattern is used to provide an asynchronous system for log collection. During an interval in which the sensors state changes quickly, the rate of messages can increase for a short time. It will result in a message queue overflow, and the additional messages will be dropped. This situation is undetectable on the subscriber side, and most of the time undetectable on the publisher side, depending on the networking protocol used.

This data loss is not detected by kernel message tracing nor by conventional tracing tools, because the messages are never sent to the socket, and are rather stored on publisher side queue. The tracing system needs to collect information about messages upstream of sockets. The only solution is then instrumentation at the messaging library level .

Performance and Overhead

In order to characterize the overhead of tracing, the performance impact was measured on kernel operations. The experiment was run under different load configurations. The impact is the ratio between the CPU cycles with and without tracing. In order to record the system activity and measure the tracing overhead, the **sysstat** set of tools is used. The **sadf** tool provides information about system resource usage. The method used is the same as in [116].

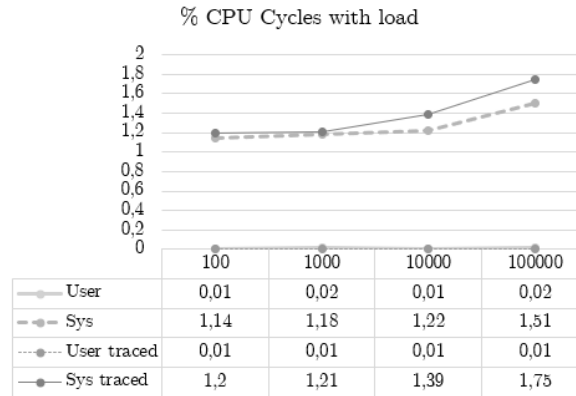


Figure 4.6 Impact of kernel tracing on CPU cycles depending of the messaging load (in messages per seconds)

Fig. 4.6 presents the impact of kernel tracing on the percentage of CPU cycles. The CPU usage is measured with the **sar** tool, (**sar -u -o sysdata 2 50**), to get 50 data points, one every 2 seconds. The average is computed separately for user space and system space.

In order to measure accurately the messaging rate, a **time.sleep()** is set in the message sending loop, and the value is adjusted to obtain the desired number of messages sent in one

second. The real messaging rate is checked with the analysis of the subscriber trace (count of messages received in one second). The impact of LTTng tracing on the CPU cycles is around 3% for a medium load, and around 15% for a very high load.

Another suitable method to estimate the overhead of the system is to monitor the maximum number of messages that the application is able to handle, with and without tracing. In order to obtain these measurements, the sleep was removed from the program loop, and the program was thus run at full rate. The average maximum was 2,544,000,000 messages per second without tracing, and 1,212,000,000 with tracing. The experiments were conducted without a memory limit, and with an infinite water-mark value, to avoid the interference of message queues overflow.

The overhead, induced by tracing the system at maximum messaging load, can be explained by the need to write the trace to disk. Indeed, for every message, a trace event is produced, and thus a huge amount of data is collected and stored. As a result, the data collected in the trace needs to be kept to a strict minimum. One byte saved on every traced message can lead to gigabytes saved on the final trace file.

Discussion

The results obtained during the various experiments confirm the efficiency and low overhead of the method. In addition to the zeromq experiments, the method has been also tested with a RPC library and the Mosquitto message broker. This illustrates that the method is easily adapted to all libraries using networks sockets for message transmission. The context and associated handler, added to LTTng for `sendto` and `recvfrom` events, will become available by default in future LTTng releases. Thanks to this integration, the instrumentation will be available directly in the packaged version of LTTng, without the need for a special enhanced version. The context handler is where the message payload is processed. As a result, it is easy to add a custom encryption or anonymising function in the handler, to remove sensitive data from the trace content. Similarly, if a specific library is used to encode the data, e.g. protobuf, a deserialization function can be provided.

The main disadvantages, of kernel level instrumentation, are the need for privileges to access the host kernel, and the lack of observability on MOM internal queues. The alternative method studied and presented in the next section does not suffer from these disadvantages.

4.4.2 Messaging library level instrumentation

Many messaging systems offer some resilience, to avoid the loss of messages in case of system failure. However, while this fault masking is interesting from a functional point of view, the storage or retransmission of messages consumes resources and degrades performance. Therefore, for performance optimization purposes, it is important to have some observability into these mechanisms such as internal queues.

Several different types of problems may be encountered. The message queues may be kept in memory but also written to disk. Are the messages kept long enough to allow retransmission in case of errors, and purged soon enough to minimize memory consumption? Routing policy errors may happen and can be detected if all the possible recipients are traced. The message delivery policy can also impact the quality of service and the loss of messages. As a final example, a failure may be caused by a malformed message content.

It is difficult to find the cause of such failures only with disconnected software logs. Information about the global system state is often required. Message tracing provides an overview of the whole communication system (global number of messages exchanged, ratio of errors) as well as a finer granularity analysis at the container socket level. This finer analysis can efficiently identify several communication problems and misconfigurations. This method allows identifying low level problems, and determine the most suitable communication parameters.

These experiments were conducted in collaboration with an industrial partner, to get the opportunity to test the system in a production environment. It allows the overhead characterization of the tracing method. The source code of the instrumented library can be found on the repository cited earlier in this section. For the current version of the instrumentation, the tracepoints are only in the `czmq` library. The modifications were made on version `czmq 4.2.0`. Everything is in a `.patch` file that can be applied directly to the source repository.

Use case

A major advantage of this solution is the ability to identify the messages. When neither the host system nor the application code is accessible, this solution allows adding some metadata to messages to identify them. The sender and destination can be recorded during the trace collection, as well as other information about the message transmission. The proposed instrumentation enabled to track messages from the publishers to the subscribers, including the acknowledgment.

If the messages do not contain information about the sender, because of concerns about the increased size, and depending on the communication pattern, it is not always possible to

identify the source of the message on the recipient. Also, while messages usually contain a timestamp in their header, this is not always true, in which case it is difficult to determine precisely the send time.

As the instrumentation daemon is located inside the container, the trace collection stops at the same time as the application process itself. As a consequence, it is more difficult, for example, to find what happened in the case of an out of memory situation that crashes the container. Also, during the experiments, the out of memory situation caused an abnormal end of tracing session, resulting in some issues to read the trace data afterwards.

Performance and overhead

The execution time of a program is the major criteria for developers in most case. The overhead refers to the difference of execution time between the application with and without tracing. In order to ensure the reliability of the results, each test was run several times. Long program executions were targeted, because it reduces the influence of unlinked system events. 4.7 presents the impact of tracing on the CPU cycles of the host.

The results show a slightly higher overhead when compared to kernel instrumentation. It should be noted however that tracing occurs twice in this case. Indeed, a tracing daemon runs inside each of the two containers, rather than a single one on the kernel host.

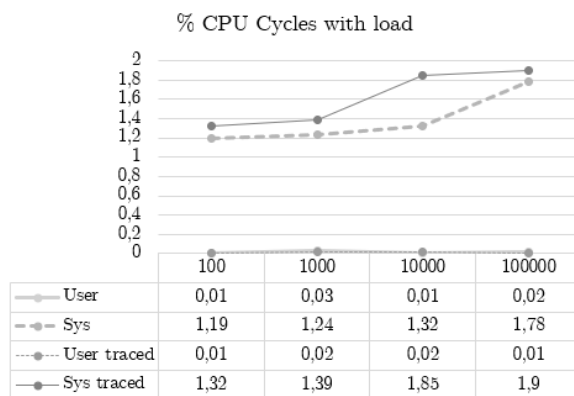


Figure 4.7 Impact of kernel tracing on CPU cycles depending of the messaging load (in messages per seconds)

Another experiment performed on a production cluster of an industrial partner, shows that the overhead was around 4 % in terms of CPU cycles of the container hosts. The cluster contains a hundred containers, and the czmq library was patched and integrated during the build of the container image. In this experiment, analyzing the traces from hundreds of

containers proved challenging. Trace viewers like Trace Compass handle without problems many trace files, but some parts of the data analysis currently do not scale too well to hundreds of trace files.

Discussion

This method is an alternative to host kernel instrumentation, when a privileged access to the host kernel is not available. This solution is not coupled to the application source code, and is thus much less invasive than the direct application instrumentation. This instrumentation is the only way to detect overflows on the internal queues. Indeed since the messages are not yet sent outside of the container, they are not detectable from the containers kernel host.

4.5 Conclusion

The different challenges and advantages of the two proposed solutions are reviewed.

Intrusiveness The advantage of the kernel level instrumentation solution is the low intrusiveness in the source code for the tracing procedure. None of the two solutions require to add tracepoints inside the application source code. This is important in projects where the development is separated from integration. Tracing is often conducted during the integration part of a project. There may be little interaction with development.

Environment constraints The main drawback of this solution is the need for administrator (root) level privileges to trace the host kernel of the container. In some environments this access is not possible for security reasons. This is especially the case if the containers are deployed in a shared context or public cloud. A relevant alternative is then to use the second proposed solution, which moves the instrumentation, from the host kernel, to the messaging library inside the container.

Source code dependency This tracing method is also weakly coupled of the source code, as there is no tracepoint added inside the application or container. The tracing system only needs to be updated when the messaging library changes. The evolution of application versions remains independent of the tracing system.

Scalability The proposed tracing architecture is highly scalable. Indeed, the number of containers can widely vary without impacting the tracing part. For the first solution, the new

instances are added on the fly in the tracing system. For the instrumented library solution, the pre-compiled version with tracing code can be integrated inside the container Docker image.

Overhead Analysis Experiments The overhead comparison showed that the library source code instrumentation was more costly and the kernel instrumentation was least costly, in terms of resources. The impact was only measured in a small scale cluster and is not necessarily representative of a bigger architecture. This would be particularly interesting to study in the context of very large High Performance Computing clusters, where container-based architectures are increasingly used.

Security concern In a multi-user environment, data security is an important concern. Kernel level tracing data may contain sensitive information. On the same cluster, the containers can host different client applications, and tracing the whole system host may expose data from these different applications in the same tracing files.

Moreover, messages between containers are exchanged on an isolated network, so they are not always encrypted. The sensitivity of the data that can be collected by the instrumentation should be examined. Since the content, source and destination of the messages are available in the trace files, the storage and sharing of these files may require a special attention. The content of the messages can be encrypted on the fly, at the trace collection, if needed.

Another concern is that the trace files contain information about the application behaviour: messages exchanged, number and size of messages, and possibly the name of functions if Remote Procedure Calls are used. In some case, these informations need to be protected, or obfuscated, before storage, trace transmission or trace analysis.

The two new methods proposed in this article improve the efficiency and bring new possibilities for Message Oriented Middleware analysis. Both methods are complementary and take into account environmental constraints. The flexibility of these methods allows to adjust the granularity of the analysis, and therefore the ideal amount of data collected for analysis.

CHAPTER 5 ARTICLE 2: DISTRIBUTED COMPUTATION OF THE CRITICAL PATH FROM EXECUTION TRACES

Authors Pierre-Frédéric Denys, Michel R. Dagenais and Quentin Fournier.

Submitted July 27th 2022 in Software: Practice and Experience

Abstract Due to the ever-increasing number of computer nodes in distributed systems, efficient and effective tools have become crucial for their analysis. Although several efficient methods have been proposed to monitor and profile distributed systems, tracing remains the most effective solution for in-depth system analysis. Tracing is the act of collecting a trace, which is a sequence of low-level events generated by the kernel or the userspace. After data collection, the most important part is the event analysis. The paradigm and choice of graphs determine the ability of the user to detect abnormal behaviors and identify their root cause. Although tracing is a highly effective approach to analyzing complex systems, the scalability of the current analysis tools is limited. As a consequence, tracing is often impractical for large distributed systems. This paper identifies the shortcomings of the current approaches, most notably the critical path computation and the trace file transfer between nodes. Then, this paper proposes new solutions to these drawbacks, most notably a distributed algorithm to compute the critical path, that does not aggregate all traces in a single node, and an efficient architecture to perform tracing on distributed systems. These new solutions are made publically available.

Keywords Distributed Systems, Critical Path, Active Path, Trace Compass, Container Clusters, Docker, Tracing, MPI, Parallel Computing, Message-Oriented Middleware.

5.1 Introduction

Monitoring is widely applied to distributed systems to efficiently collect metrics on each node. This approach produces discrete data that provides an overview of the system load and behavior over time. However, individual performance metrics often cannot reveal the root cause of performance problems in distributed systems. This limitation of monitoring is especially true in remote systems and large clusters, such as production servers, high-performance computing centers, and distributed systems.

Instead, *tracing* provides a continuous view of the behavior of the system and allows a deeper analysis of the system. In particular, tracing allows developers to find performance problems originating from the kernel and the userspace. Furthermore, tracing allows kernel and userspace events from multiple systems to be observed and analyzed on a single graph, thanks to an accurate time synchronization. Several analysis methods have been proposed to efficiently exploit these large amounts of data, including finding the critical path across an application execution graph.

Critical path analysis highlights the events linked to a process execution, across a full execution graph with all system events. The result of the processing is presented to the user as a time graph, where each involved process is represented on a separate line and where kernel events are represented across the timeline. Arrows between events represent signals between events, which allow the user to understand the application execution and to discover the cause of blocking events and errors.

This article surveys previous work on critical path analysis, including methods that could be improved to achieve large-scale distributed system analysis. In particular, this paper puts under the microscope the poor scalability of the current algorithms used to compute the critical path of a process. The limitations of current analysis tools, data collection, storage, and distribution during the analysis are characterized by benchmarks. Then, this paper introduces solutions and mitigations to the aforementioned limitations. Although this paper focuses on the critical path computation, the proposed methods are suitable to scale other analyses, such as time and flame graphs. This article aims to answer the following research question: *How to improve the efficiency of the current methods to analyze traces collected on large distributed systems?*

The main contributions of this work are (1) the identification of limitations of the current tools, (2) the introduction of a distributed critical path algorithm, (3) efficient communication patterns to exchange information between nodes during analysis, and (4) the architecture to compute the critical path in parallel.

The remainder of this paper is organized as follows. Section 6.2 surveys the related work. Section 6.3 introduces the methodology to parallelize the critical path computation efficiently. Section 6.4 presents and discusses the results of the proposed approach on three use-cases. Finally, Section 6.5 proposes interesting future work and concludes this paper.

5.2 Related Work

This section surveys state-of-the-art methods related to every aspect of the critical path analysis: data collection, data storage, data transfer, critical path computation, and visualization. In particular, this section investigates the ability of the methods to perform well on distributed systems.

5.2.1 Trace Data Collection

Tracing is a non-intrusive and lightweight method to record system activity by collecting low-level events from the kernel space, including system calls, interruptions, and network events, and from the userspace (program execution). Events are stored in a trace and comprise a name, a timestamp, and possibly a payload. Most notably, tracing helps detect abnormal behavior such as real-time deadlines, unexpected response time, and memory, load, or concurrency issues. As stated by Poirier et al. [108], tracing may be seen as a high-performance `printf`.

Tracers apply, and have been applied, to a wide range of systems [117] such as application servers, clients, real-time systems, embedded systems, and heterogeneous distributed systems. The low overhead of tracers makes them suitable for heavily loaded systems and low-resource devices such as IoT systems and containers.

Several tracers have been proposed throughout the years as listed by Toupin [118]. The Linux Trace Toolkit: next generation (LTTng) [119] is the preferred choice for this research, as it is lightweight and suitable for kernel and userspace tracing. Note that the proposed methods apply to the other tracing tools.

Tracers typically produce a large amount of data, and since tracing is used to analyze loaded systems with complex architectures [120], the traces must be stored efficiently.

5.2.2 Trace Data Storage

Due to the high throughput of tracers, data structures optimized for time series, such as described by Prieur-Drevon [121], are mandatory to store traces and critical paths.

While B-trees [122] are a widely popular data structure for ordered collections stored in files and databases, they are not suitable for interval storage. As a solution, State History Trees [121] were developed, a data structure specifically designed for large time interval data that stores the data on disk rather than in memory. The State History Tree is organized as a tree where multiple intervals are stored on-the-fly in each node and committed to the

disk when full without rebalancing thanks to the chronological ordering. Additionally, this method is suitable for short time intervals to the main structure (trace file) and can be used in a distributed paradigm [123]. As of the writing of this article, State History Trees are the preferred data structure to store traces.

Alternatively, Chan et al. [124] introduced another hierarchical trace file format called SLOG2, a scalable logfile format. However, this format is slower and less versatile than State History Trees and is therefore not as popular.

The manipulation of large trace files is challenging because of their size. As a solution, Noeth et al. [125] proposed to add compression at the trace file creation on creation nodes. This method is part of the ScalaTrace project and has been tested on an MPI cluster, a representative example of a massive trace data generator.

Although efficient data structures exist to store traces, they are often analyzed on a different node after collection, and communication between nodes remains challenging.

5.2.3 Communication Between Nodes

The trace data collected on each node by a daemon is then transferred to a storage node or directly to the analysis node. In both cases, trace data must be sent over the network.

The communication between distributed nodes is generally achieved with Remote Procedure Calls (RPC) or, more recently, with Message Oriented Middleware (MOM) [126]. While RPC standard behavior is blocking, MOM benefits from a straightforward implementation of queues for asynchronous communication and quality of service on unreliable networks [127, 128]. Consequently, Message Oriented Middleware tends to be preferred.

The communication between tracing nodes and the analysis node is better achieved with a specific protocol such as the Trace Server Analysis Protocol (TASP, also called TSP) [129]. This protocol is similar to the Langage Server Protocol (LSP) [130], except that instead of being based on JSON-RPC, TSP is based on HTTP Representational State Transfer (REST). The main advantage of this protocol is the possibility to encode data using a binary format.

The communication capacity of distributed systems is variable. Indeed, networks are often unreliable and slow, and the bandwidth is prioritized for services rather than monitoring, tracing, or backup in production environments. As a result, the amount of data and the communication frequency must be reduced. In particular, the transmission of complete trace files must be avoided as much as possible. One solution to solve the communication challenge is for tracing daemons to periodically send the trace files to storage nodes using asynchronous communication (e.g., MOM) and to perform as much pre-computation as possible on the

storage nodes. Then, only the necessary data for the analysis is sent to the analysis node using a specifically-designed protocol (e.g., TSP).

5.2.4 Critical Path Computation and Analysis

The critical path, also called the active path, corresponds to the shortest path between the elements of a network and represents the dependencies between the tasks. The critical path has been applied early on to software analysis; notably, Yang and Miller [94] used an execution graph to determine the critical path of software execution in 1988. The critical path is a suitable method to solve performance problems and plan the realization of complex projects [95] in numerous domains such as activity networks with time constraints [96], networks with fuzzy activity [97], TCP transactions [98], and fuzzy numbers [99]. The reader is referred to the work of Montplaisir et al. [100] for a detailed introduction to the critical path.

Interestingly, the critical path allows tracking program execution and precisely profiling programs during execution. Indeed, a program execution speed is directly tied to that of the critical path [101]. For instance, Tullsen and Calder [101] introduced a critical path profiler, and Ball and Larus [102] proposed a method for critical path profiling with minimal overhead.

Since the critical path computation is related to searching for the shortest path, one may use well-known algorithms such as Dijkstra [103]. However, due to the sequential nature of the Dijkstra shortest-path algorithm, parallelizing the computation yields only a marginal average speed-up of 10%. The reader is referred to the work of Giraldeau and Dagenais [104] for a thorough description of the main algorithm used to compute the critical path.

Critical path methods in distributed systems and virtualized environments require careful adaptations as described by Nemati et al. [106]. Notably, since the critical path comprises dependencies between multiple trace files collected on different nodes, ensuring a near-perfect time synchronization between traces is crucial. The time synchronization accuracy can be increased at collection time with an efficient clock synchronization of traced systems [107] such as with the Precision Time Protocol (PTP) or at analysis time with an offline trace synchronization algorithm such as the one presented by Poirier et al. [108]. The reader is referred to the work of Wu et al. [109] for a complete and detailed description of the analysis of the large distributed systems. Critical path computation requires sequentially processing every trace file on a single analysis node, which is problematic for numerous large traces. Schulz [111] showed that it is possible to extract the critical path for applications running on up to 128 processors. Nevertheless, there is a real and necessary need to improve the critical path computation.

5.2.5 Visualization of Large Traces

Time graphs are a popular visualization for trace analysis. As illustrated in Figure 6.1, time is represented along the x-axis, and each vertical line represents a processor tree and its cores, or a process tree and its children. On each line, events are represented by a rectangle corresponding to the run time state of this core or process. Notably, time graphs allow a fine-grained representation of kernel or network-related events.

One way to mitigate the aforementioned communication and computation challenges is to limit the amount of data fetched, and computations, to the period selected by the user in the visualization software. For instance, the graph displayed to the user can be partitioned into frames (areas of pixels), each of which can be processed independently. Note that this segmentation is possible across time but also across resources. Although such an approach is straightforward when the content of the frames is independent of each other, it becomes much more challenging when data from other frames are required for the computation of the selected frame. Chan et al. [124] pointed out this challenge and proposed a solution: the bounded-box method based on a specific trace format. Note that their solution is suitable for other trace file formats with annotations.

The end-user analysis environments play an essential role and should be carefully considered. In our opinion, there are two typical users: software developers or architects, and system integrators or administrators. Let us first consider the analysis environment of software developers.

Developers often have to observe the performance of their application and solve issues such as race conditions. They generally work on test clusters with a limited number of nodes, and they can control the trace collection and thus the amount of data. As a result, the analysis tool is ideally integrated into their IDE, and a fine-grained integration can be configured between the trace data and the source code. Notably, function calls found in traces can be

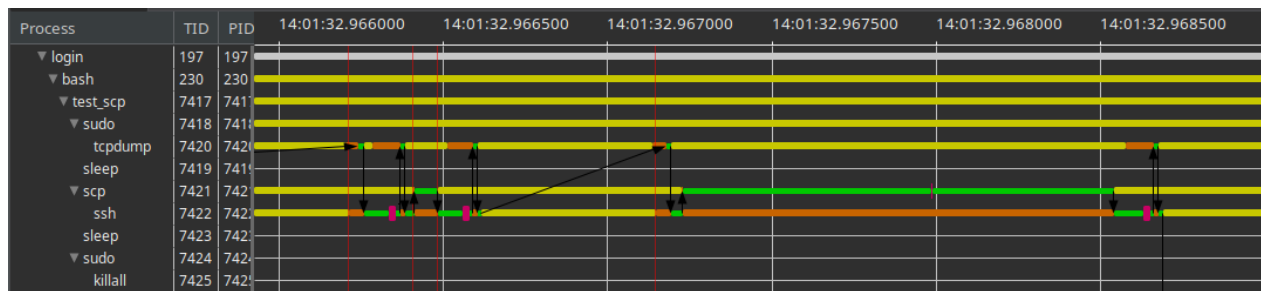


Figure 5.1 A time graph in Trace Compass.

linked to code files, and the developer can easily find the code related to a system event or unexpected behavior. Trace compass is one of these tools and is integrated in the Eclipse IDE [131] as illustrated in Figure 6.2.

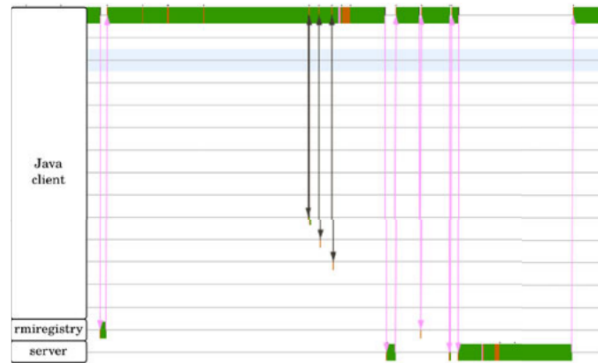


Figure 5.2 The critical path view of a process in Trace Compass.

Unlike developers, system integrators do not control the data since they work on already generated trace files. The data collection policy is configured at the system set up and is typically exhaustive, as they need a high level of details to understand the issues. Consequently, integrators and administrators typically handle larger traces than developers. Additionally, the same time-expensive processing is usually performed for each of the multiple users involved in the system monitoring and analysis. In this case, a better approach is to rely on a shared visualization service, and preprocess the data on tracing nodes, such that fewer resources are needed to perform final computations and views. Several effective methods are available for monitoring systems, but there is a gap on the trace analysis side. Indeed, the Theia Trace Viewer, released in 2017, is one of the first. Datadog can also be part of the solution. None of the current solutions offers a complete method for the efficient trace analysis on large clusters. This article will propose solutions to solve these gaps.

An interesting example of a distributed analysis architecture is the use of Prometheus nodes to collect metrics in a web application, and a Grafana client to display graphs. Indeed, a Prometheus node, linked to a storage service, runs the application periodically to retrieve the metrics and store them in the Prometheus database. These Prometheus nodes are generally replicated and work in clusters. When a user asks for a graph of the metrics on the Grafana client, the client sends a request to one of the Prometheus nodes. The Prometheus node processes the raw data and sends the resulting graph data to the client. As a result, the Prometheus nodes, similarly to computing nodes, perform the processing and transfer only the necessary information to display the graphs to the Grafana client, similarly to viewer nodes. This pattern could be applied to trace files analysis, to avoid file transfers and

perform a parallel pre-computation.

For a review of tracing tools for large distributed environments, the reader is referred to the survey of Las-Casas et al. [132]. Numerous projects are available to achieve this goal, most notably Zipkin [133] and Opentelemetry [134].

5.3 Methodology to Parallelize the Critical Path Computation

This article presents two improvements to the critical path computation that increase the scalability in terms of system size and load: computation of the critical path and a proposal of an efficient architecture for the transmission of trace files between analysis nodes.

5.3.1 Requirements and Challenges

The goal of the critical path is to identify the dependencies between events collected simultaneously on different nodes. The traces recorded on the different nodes are transferred to the analysis node, where they must be synchronized due to the time difference between the system clocks. After synchronization, the analysis starts by extracting the execution graph for each trace. Then, the critical path of a process is computed at the user request.

First, let us focus on the system calls and signals required to compute the critical path. Contrary to the thread states `running`, `preempted` and `interrupted`, the `blocking` thread state can modify a program control flow. The event that unblocks the thread is either from the kernel mode or an interrupt. Contrary to the former, interruptions allow knowing the device that caused the waiting. Additionally, kernel events such as `sched_switch`, `sched_ttwu`, `interrupt_entry`, `interrupt_exit`, `inet_sock_local_in`, and `inet_sock_local_out` delimit events and interruptions. As a result, thread states and interruptions are the minimal information required to determine the unblocking relations between processes, and thus compute the critical path. The execution graph is processed with task states represented by horizontal edges and signals represented by vertical edges.

The two main steps of the critical path processing are the execution graph processing and the critical path computation. The first step can be easily parallelized as there is no dependency between traces. Each execution graph can be processed independently and simply stacked in the viewer with the same time scale. Parallelizing the second step is more challenging. Indeed, the computation of the critical path requires finding the links between multiple trace events, and the parallelization of the algorithm requires modifications. Since no pre-computation is possible, due to dependencies along the computing process, the critical path cannot be computed upstream on remote traces.

5.3.2 Methodology

Let us briefly introduce the current algorithm used to compute the critical path. The reader is referred to Giraldeau and Dagenais [104] for a detailed description of the algorithm.

Let us assume that the entire traces recorded on the nodes have been sent to the analysis node and synchronized. First, the execution graph must be processed for all the collected traces. Then, the critical path is computed at the demand of the final user by selecting a process.

After that, the system states of the selected process on the execution graph are iterated until a blocking state is encountered, and a procedure is called to retrieve the origin of this blocking state in other threads or host processes. Finally, this procedure performs a backward iteration until the interval start. In the process, if an incoming blocking state is detected, a recursion is made.

The primary drawback of the algorithm described above is the amount of memory usage due to (1) all trace files must be opened in the same environment to identify remote dependencies, and (2) the graph elements are typically stored in memory. Additionally, the algorithm as presented is not easily parallelizable due to the dependencies between the traces. The improved methodology described hereinafter addresses these two critical limitations of the current approach.

5.3.3 Improved Methodology

This section introduces improvements to (1) the scalability of the algorithm and (2) the data structure and communication between computing nodes.

Algorithm Parallelisation

Let us assume the simple architecture depicted in Figure 5.3 for the sake of simplifying the algorithm optimizations, and eliminating environmental conditions such as network delays and speed. Let us also assume that the trace files are stored and opened locally in the viewer node.

A distinct data provider module is associated with each trace in the proposed methodology. As a result, traces are seen by the computing module as if they were stored on different hosts. Separate computing operations can be performed on each trace, and there is no coupling with trace storage. Additionally, a computational sub-module is associated with each data provider, to perform pre-computing operations.

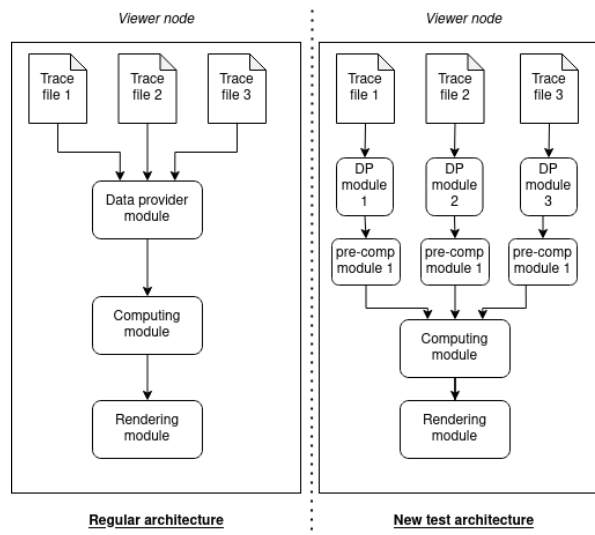


Figure 5.3 Architecture to retrieve, compute, and display the critical path.

The first straightforward approach investigated involves resolving outbound edges: when a blocking state on the thread execution trace is encountered, a message is sent to all tracing nodes to find the cause of the interruption, based on the event time and attributes. This method was quickly discarded because of the communication burden between nodes, as illustrated in Figure 5.4.

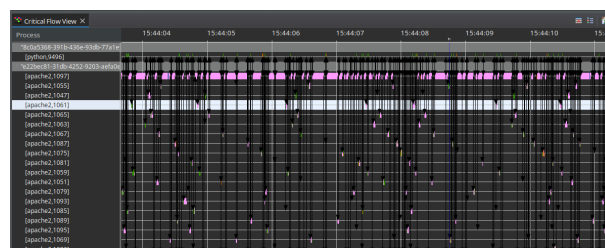


Figure 5.4 Loaded critical path view.

The second approach considered relied on the pre-computation of the graph and critical path. Instead of drawing the complete execution graph, the primary computation node sends a call to all preprocessing nodes in order to retrieve the graph elements linked to the selected time window. The computation node then performs the time synchronization, and the execution graph is displayed to the user thanks to the viewer node. Secondly, the user selects a process on which computing the critical path. At this point, the critical path is calculated on the current trace, and remote dependencies (also named blocking edges) are identified in the lower part of Figure 5.5. At this point, the viewer node displays the first graph for the user without the remote dependencies solved, and a data structure containing the remote dependencies is

sent to other computing nodes to solve them. The user can then see right away the different states of the process of interest, for instance executing in user or system mode, preempted, blocked waiting for the disk or another local process, or waiting for a remote dependency. If the process is blocked for a significant proportion of the time on remote dependencies, the user will want to know what other processes and events the process is waiting for. This is the asynchronous parallel computation that was sent to the other computing nodes for solving.

Remote dependencies are resolved using relevant identifiers; for example, an acknowledgment number is present for network events on the sender and receiver packet. This approach is time- and cost-efficient, since this computation is only performed when the user modifies the time window or process, and since the pre-computed intervals can be stored in a cache on the viewer node and reused. Lastly, the graph is updated step by step during the remote dependencies resolution. As a result, the user gets a first display way faster than with previous methods. The result is an uninterrupted link between events represented by arrows between system events across processes and traces. It is the critical path related to the selected process.

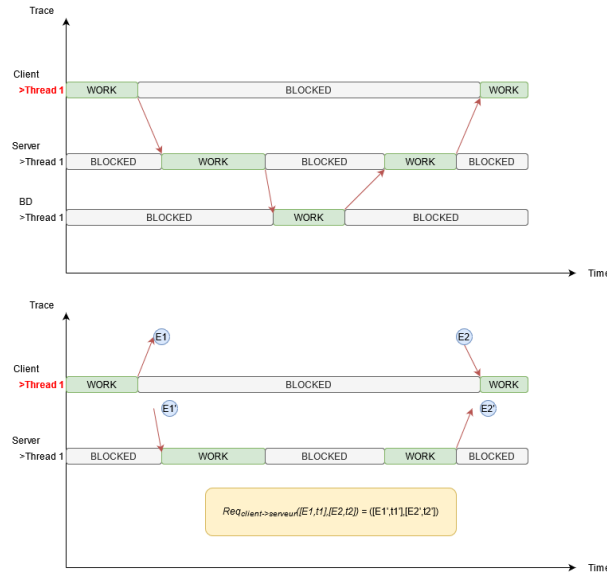


Figure 5.5 Methodology to identify edges.

Several challenges have been encountered: (1) the origin of network interruptions is typically in remote traces, (2) the number of processes and nodes in the critical path is not known in advance, (3) the identification and retrieval of the interruption source depend on the nature of the event, and (4) the trace of the source event is not always available, which can lead to unidentified origin events that need to be identified on the graph.

The simplified pseudo-code of the method to resolve recursive remote dependencies is given in Algorithm 2. Contrary to previous methods, users do not have to wait for the entire graph processing and dependency resolving, since the function $\text{DisplayCP}(CP)$ updates the graph during processing (Line 14). The user gets a first display very quickly, and more details appear as the remote computations proceed.

Although the proposed algorithm is not fully parallelizable, the new algorithm allows users to almost instantly see the critical path during processing.

Architecture for Remote Trace Analysis

Now that the critical path computation is possible on a distributed architecture, the traces and critical path pieces must be transferred from the computation nodes to client nodes with an efficient communication pattern.

Before introducing the proposed approach, the size, number, and frequency of the packets sent must be characterized. Previous experiments have concluded that the costlier transmissions are the trace files. Indeed, typically one or a few large trace files per node in the cluster must be transferred from the computation nodes to the analysis node.

In practice, the viewer node is often a laptop. The limited amount of resources available on the viewer node restricts the number and size of traces that can be processed. Such a limitation often results in long computation times, or inability to complete the computation. As a result, crucial improvements must be implemented to avoid transmitting complete trace files. For instance, the data transmission can be asynchronous, and the processing can begin as data comes in.

Thanks to the new algorithm presented in the previous section, there is no need to transfer all trace files to a single analysis node. As a result, two architectures are possible and answer different needs.

The first architecture, Figure 5.6 is based on several computing nodes that collect traces from traced systems. These nodes collect traces and perform the pre-computation of the traces (execution graph processing). The user connects with his browser on one of these nodes, and it becomes a viewer node. As a result, this node is in charge of collecting data from others and computing final graphs for the user.

The main advantage of this architecture is scalability, as no trace files are on the traced systems. Also, it adds less overhead to the traced systems because traces are not analyzed on them. In this case, the parallelization depends on the number of computing nodes. This architecture is preferred when the elasticity of the system is high (the number of traced

systems is variable), and the storage is not persistent, such as in container environments.

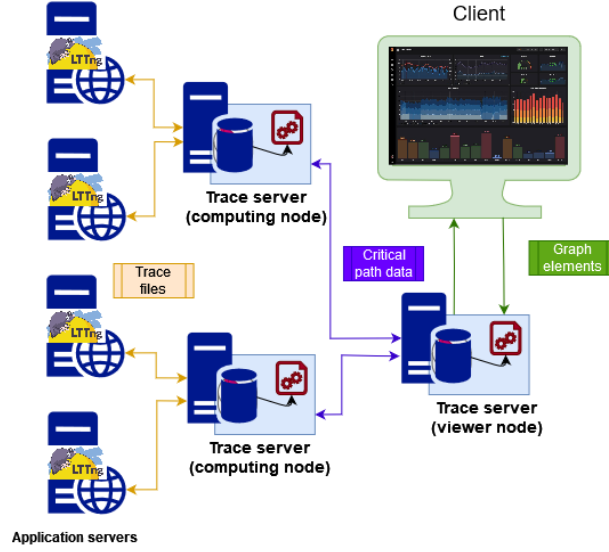


Figure 5.6 Edges Identification

The second architecture is more conventional: traced nodes store their own traces and play the role of computing nodes. Another dedicated trace server serves the requests of the clients and distributes the analysis between computing nodes.

This architecture is more efficient when the number of nodes does not vary and when traced nodes are able to handle both the application load and the tracing computation load. There is also less network load because trace files stay on trace nodes. However, the overhead is higher on the traced nodes. This disadvantage is counterbalanced by the greater parallelization of the critical path computation, due to the larger number of computing nodes.

These two architectures did not require any file transfer between the client and traced nodes, and allowed the distribution of trace analysis computation. The next section will focus on the parallelization speedup of the algorithm.

5.4 Results

The detailed results obtained in this paper and the source code of the experiments are available in this git repository ¹.

Let us define the following concepts to understand this section better:

- The *viewer* is the software that transforms the processed data into a visual graph.

¹<https://gitlab.com/pierrefrederick.denys/trace-compass-critical-path-distributed>

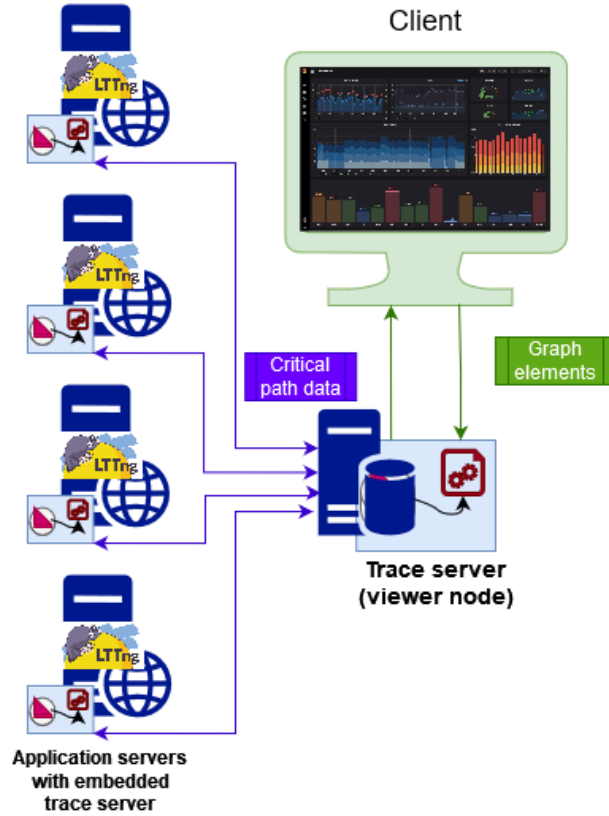


Figure 5.7 Edges Identification

- The *viewer node* is the middleware node that runs a trace server to process the data sent by computing nodes and to answer requests from the viewer.
- *Computing nodes* are trace servers running on the application server or the trace collection nodes. The computing nodes store traces, retrieve and precompute graph elements from the traces, and send the results to viewer nodes.
- *Graph elements* are preprocessed data (events, vertices, and edges) computed from traces and transferred from computing to viewer nodes.

5.4.1 Representative Use-Case

In order to illustrate the contribution of the new method, a representative and demanding critical path computation, over several large traces, is performed with both the existing method and the proposed new algorithm. One key parameter to compare the two methods is the time required to display the first graph to the user. This first example traces a 10Gb file copy between four nodes, resulting in four traces of around 3.2Gb each. With the current

algorithm, 63 minutes are required to perform the traces indexation, analysis, and display of the graph to the user, when performed on a single node. The proposed new algorithm completes the full analysis in only 7 minutes and displays the first graph in only 45 seconds. As a result, the global analysis is 9 times faster with the new method, and the first graph display happens in less than one minute, rather than in an hour with the current method. Note that the computing time has been evaluated with several parameters, such as the trace size and the number of traced nodes, and that each experiment has been repeated 10 times.

5.4.2 Performance and Overhead

Processing Time

The first benchmark compares the processing time of the current and improved algorithms. In this experiment, the *total processing time* refers to the elapsed time between the selection of a process by the user and the display of the complete critical path. The *first display* is only defined for the new algorithm and refers to the elapsed time between the selection of a process by the user and the display of the critical path with unresolved remote dependencies. In other words, only dependencies involving other processes running on the same node are displayed.

For this benchmark, SCP transfers have been recorded between four nodes. In order to remove network fluctuations, the four machines are on the same isolated network. The node activity and transfer time are proportional to the transferred file size, and so are the trace size and number of events recorded. Therefore, four different sizes have been considered: 100 Mb, 1Gb, 10Gb, and 100Gb. The analysis has been performed 10 times in order to alleviate the fluctuations.

Figure 5.8 reveals that, for smaller traces, the total processing time of the new algorithm is slightly higher. This slowdown is due to the transmission time of packets on the network for resolving remote dependencies. However, for larger traces, the total processing time of the new algorithm is lower, as the distribution of the processing between computing nodes more than compensates for the transmission time. Additionally, the “first display” time indicates that the user virtually instantly obtains a first graphical display. As a result, users can explore several critical paths of processes sooner, without having to wait for the entire remote dependencies to resolve.

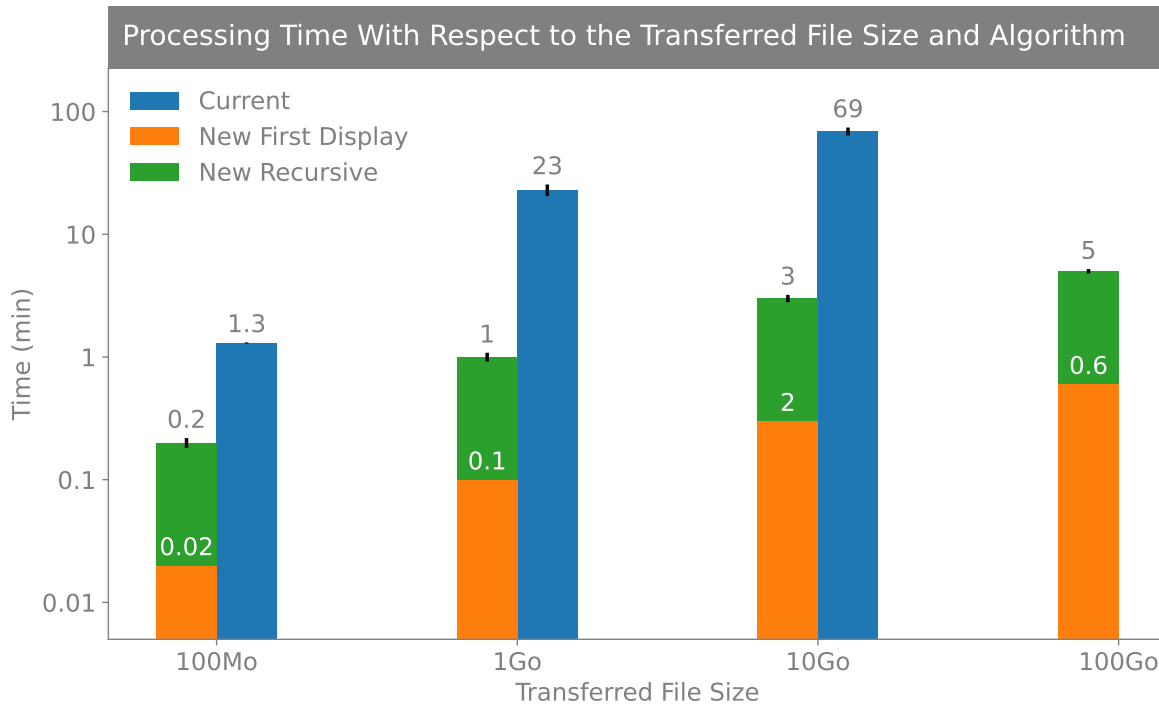


Figure 5.8 Processing time of both algorithm versions.

Number of Traced Nodes

A second benchmark was performed with the same setup as the first one, except that the number of traced nodes is variable. For this experiment, an SCP process has been created in parallel between one node and a variable number of other nodes. Experiments have been repeated for several trace sizes to evaluate the processing time with respect to the number of nodes and trace size.

On Figure 5.9, each curve is linked to a defined trace size, and the variable is the number of nodes involved in the computation. The graph shows that the best results are obtained when each computing node processes only one trace file. The processing time increases with the trace file size. As a result, the tracing architecture should use a regular trace file rotation to get multiple small trace files rather than a bigger one.

Performance

Significant performance gains are observed in the viewer: the critical path computation is faster and requires almost no resources on the client side. Since the client resources are no longer a bottleneck, the computing capacity has greatly increased. Additionally, since the

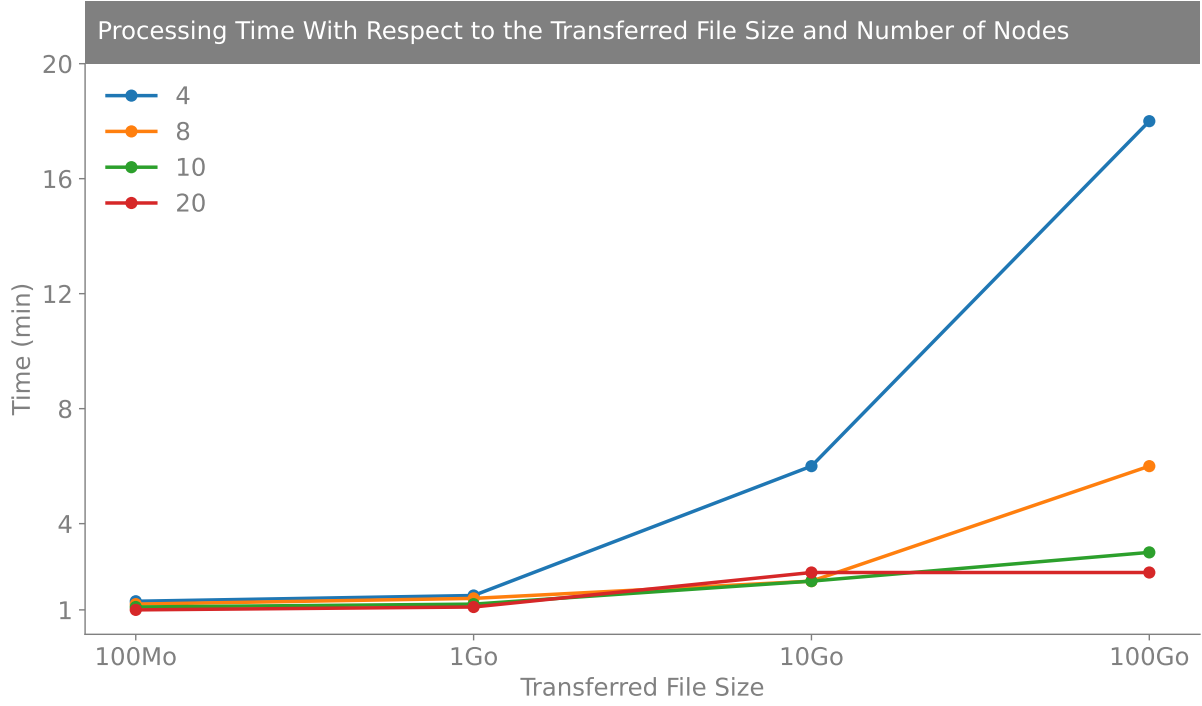


Figure 5.9 Processing time of the new algorithm in terms of the number of nodes and trace size.

critical path processing is performed on a time interval, the overhead is independent of the trace size and duration; only the trace data from the interval $[t, T]$ are transmitted from computing nodes to the viewer node.

5.4.3 Other Use-Cases

The expected use-case for the new method is to analyze distributed systems. The advantages of the new method will be highlighted below in two typical cases: MPI clusters and inter-container communication with message-oriented middleware.

MPI Clusters

MPI clusters are designed to perform massively parallel computations. Nonetheless, the synchronization between threads and barriers that causes busy-wait are the source of performance problems that are difficult to detect. Tracing helps follow communications between threads, and thus detect abnormal delays or patterns.

The main characteristic of these computing clusters is the number of nodes and messages

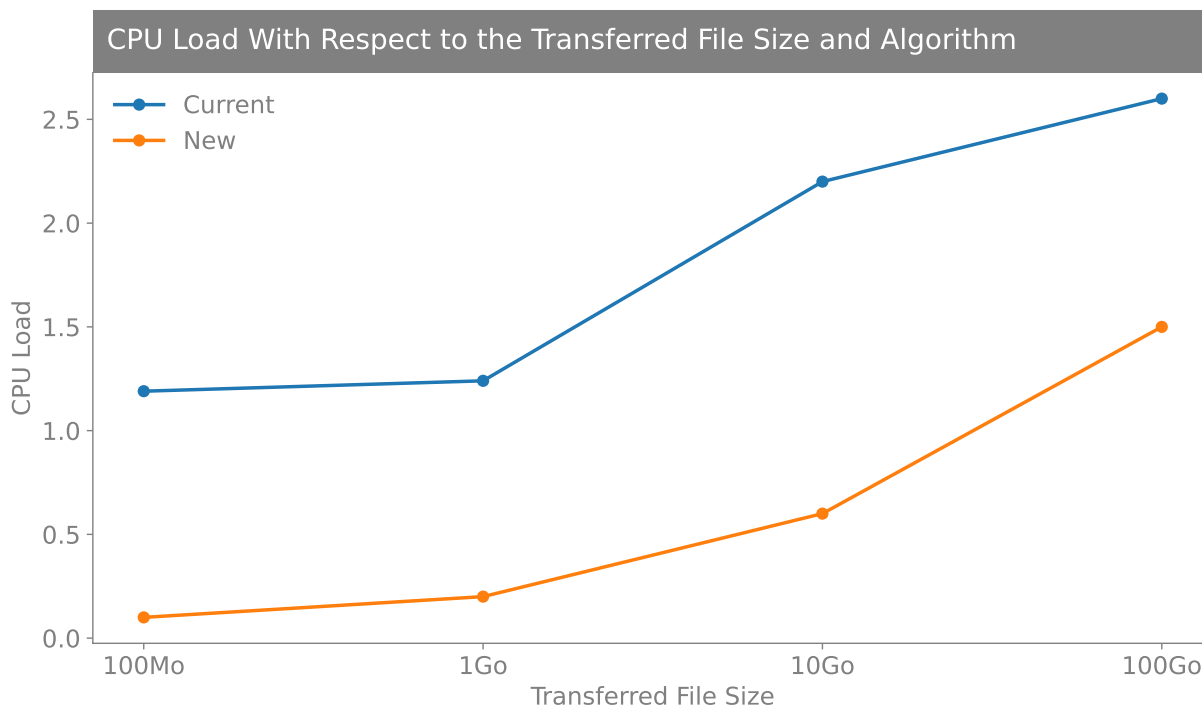


Figure 5.10 Comparison of performance of both algorithms on the viewer node.

exchanged. As a result, traces collected on each node are often enormous, and the execution graph will present many events due to the massive usage of messages between nodes. The critical path computation on such execution graphs requires substantial resources. With the conventional method, trace files should be transferred from each node to a central trace server in order to compute the critical path on this group of files.

The new method avoids transferring complete trace files. Instead, only the required graph elements are transferred from computing to viewer nodes. Generally, a good option is to avoid transferring trace files to avoid a network overload in the cluster. Furthermore, as resources are usually dedicated, the overhead of trace server execution on the calculation node is negligible.

Inter-container Communication with Message-Oriented Middleware

In clusters of containers, the communication between nodes is mandatory and efficiently achieved with Message-Oriented Middleware such as ZeroMQ. These communications are often asynchronous and, depending on the quality-of-service configured, some lost messages can be sent multiple times. Tracing in this environment is especially helpful in identifying

slow communications or lost messages. However, the result data is typically challenging to exploit. The critical path is helpful and allows following messages between containers. If the message tag has been recorded in the trace, the critical path also allows the detection of duplicate messages. The second method explained in Denys et al. [135] allows collecting comprehensive information on messages. Then, it is possible to follow a message even if it passes through several containers. Thus, the critical path is useful to determine the full path of messages.

5.4.4 Discussion

The tracing cost has been compared between the current algorithm and the new one proposed in this article. Even though a full parallelization of the algorithm is impossible, a reasonable acceleration has been observed with the distributed algorithm.

The processing time is comparable, or even slightly longer, for small traces and many nodes, as the transmission time on the network is larger than the processing time on the local node. The best improvement is observed with a larger number of analysis nodes and smaller traces. As a result, a good tracing practice is to use file rotation and a larger number of tracing nodes to get a good distribution of file traces and, consequently, a better performance.

Several improvements could be explored in subsequent work and will be detailed in the next section.

5.5 Future work

The improvements made to the algorithm and the experiments assumed that trace files were already time-synchronized. If the traced nodes are out of sync, a synchronization process must be done on raw trace files. One interesting extension of this work would be implementing a distributed synchronization method into the tracing tool.

A valuable extension would be standardising the communications between the viewer and computing nodes, that is to propose a single API for several viewer tools.

Another research direction would be to focus on the viewer side to further improve the efficiency and extend the analysis. In our opinion, one of the most compelling features would be to filter trace files stored on remote computing nodes.

A noticeable improvement, especially in environments with a large number of nodes, consists in a method for optimised trace placement. Is it more efficient to use 20 analysis nodes for 20 traces, or make smarter placement of similar traces in groups? This question could be

explored in subsequent work.

5.6 Conclusion

Critical path analysis is a valuable tool for identifying bottlenecks in a distributed environment. However, the current algorithm to compute the critical path lacks the efficiency required for large distributed systems. As a solution, this paper introduces a method for parallelizing and scaling the current algorithm. The proposed architecture improves the storage and processing efficiency of traces, thereby reducing the limitation on the number of trace-points and increasing the analysis capacity. Furthermore, the time required to display the graph is greatly reduced, allowing the final user to get the results faster. We believe this work to be the starting point for adapting current trace analysis tools to very large distributed systems.

Algorithm 1: Pseudo-code of the simplified algorithm to resolve the recursive remote dependencies in the viewer node.

Input: execution graph G , process P
Output: critical path CP

```

1 function resolveBlockedStates( $G$ , optional  $P$ )
2    $BlockingStates[] \leftarrow$  new JSON data structure
3    $ReturnedEdges[] \leftarrow$  new JSON data structure
4   while  $event \neq eventInit$  do           // Backward exploration of the process
      states
5     if  $event$  is blocking then           //  $P$  is blocked by another local process
6        $state \leftarrow Resolve()$ 
7     else if  $event$  is blocked by network then   //  $P$  is blocked by another
      remote process
8        $BlockingStates[] \leftarrow event$ 
9     else
10       $state \leftarrow Unresolved$            // Blocked state is unresolved
11  forall  $ComputingNodes$  do
12     $ReturnedEdges[] \leftarrow SolveBlockingStatesRemote(BlockingStates[])$ 
13     $CP \leftarrow ReturnedEdges[]$ 
14     $DisplayCP(CP)$                        // Update CP view during processing
15     $resolveBlockedStates(CP)$            // Recursively solve blocking states in
      returned paths
16  return  $CP$ 

17 function SolveBlockingStatesRemote( $BlockingStates[]$ )
18   for  $BlockingStates[]$  as  $BlockingState$  do
19      $states = FindCorrespondingProcessState(BlockingState.parameters)$ 
20     if  $states$  is not null then
21        $ReturnedEdges[] \leftarrow states$ 
22     else
23        $state \leftarrow UnresolvedNetwork$ 
24        $ReturnedEdges[] \leftarrow state$ 
25   return  $ReturnedEdges[]$ 

26 function FindCorrespondingProcessState( $BlockingState.parameters$ )
27   if  $BlockingState.parameters.seq == currentState.parameters.ack\_seq$  then
28     while  $currentState.Start\_time \geq BlockingState.Start\_time$  do
29        $states \leftarrow currentState$ 
30   return  $states$ 

```

CHAPTER 6 ARTICLE 3: SCALABLE DISTRIBUTED COMPUTATION OF THE CRITICAL PATH FROM EXECUTION TRACES

Authors Pierre-Frédéric Denys, Michel R. Dagenais.

Submitted January 13th 2023 in Concurrency and Computation: Practice and Experience

Abstract Large distributed systems are challenging to analyze because latencies can quickly accumulate. This typically requires efficient tools such as tracing, to collect with low overhead a sequence of low-level events generated at runtime by the operating system (kernel space) and applications (user space). Most notably, tracing allows computing the longest path of dependencies between threads and resources (critical path), a powerful approach to investigate and understand performance issues in large clusters. However, the computation of the critical path across multiple nodes is costly and more difficult to scale. Several solutions have recently been proposed, such as a distributed algorithm for computing the critical path that bypasses the transfer of large trace files between nodes, thus speeding up the rendering of visualizations. Nevertheless, the current algorithms for the parallel computation of the critical path require much more computational resources for a moderate acceleration. This paper introduces a new algorithm to compute critical paths based on pre-computed indexes of the inter-node communications. It greatly reduces the computational cost of later queries for the critical path of different threads and requests. The algorithm has been implemented in Trace Compass and is publicly available.

6.1 Introduction

Nowadays, a wide range of applications, such as scientific simulations, depend on large distributed systems. Such environments often comprise tens of thousands of nodes, each comprising multiple CPU cores and possibly GPUs. Due to their scale and complexity, analyzing their performance poses several new challenges: most prominently, the algorithms and data structures must be efficient and scalable. However, analysis tools for large distributed systems are challenging to develop and optimize, as they are large parallel applications themselves.

One suitable approach for analyzing the performance of large distributed systems is *tracing*. Tracing collects a sequence of low-level events, generated by the operating system (kernel space) and applications (user space) whenever a specific instruction, called *tracepoint*, is encountered at runtime. Notably, when the thread scheduling related events are traced, it

is possible to compute the critical path corresponding to the longest path of dependencies between threads and resources. The critical path, for a thread or request, is an efficient and effective approach to identify and explain the root cause of bottlenecks in large-scale environments, such as network latency, storage contention, and cache issues [104].

The critical path computation can ultimately depend on all the threads in a system, and thus has been the main bottleneck for analyzing large traces lately. The initial algorithm introduced by Giraldeau and Dagenais [104] computes the critical path from the complete execution graph stored in memory. The available memory thus became the limiting factor to analyze very large traces. A new optimised disk based structure was developed to store the execution graph on the disk, while allowing quick access for traversing the graph to compute the critical path¹, thereby mitigating memory limitations.

More recently, a parallel algorithm to incrementally compute the critical path from distributed traces was proposed [136]. It brought interesting improvements, as larger traces could be handled by a cluster of parallel trace analysis nodes. Furthermore, the incremental computation improved the user experience by quickly showing the status of the thread for which the critical path computation was requested, including the periods where it was idle, waiting for some external events. Then, incrementally, were shown the details of the critical paths leading to those external events, recursively. However, the computation of the critical path remains a global computation, on the combined execution graph, and involves work from all the parallel analysis nodes.

This paper presents a new algorithm that can quickly prune the critical path computation, to only involve a small subset of the parallel analysis nodes, by indexing trace files using an *external communication index* (ECI). The three contributions of this research are (1) the identification of the limitations of the current tools, (2) common patterns of distributed trace analysis problems, and (3) a new algorithm that indexes the external communications, in order to reduce the computational cost of the distributed critical path computation.

The remainder of this paper is structured as follows. Section 6.2 surveys the related work and discusses the limitations of the existing tools. Section 6.3 introduces a method that remotely indexes the external communications in trace files to compute the critical path more efficiently. Section 6.4 evaluates the new proposed method with real use cases, and discusses its benefits as well as limitations. Finally, Section 6.5 concludes this paper by discussing relevant future research directions.

¹Unpublished work by Genevieve Bastien <https://git.eclipse.org/r/c/tracecompass/org.eclipse.tracecompass/+183418>

6.2 Related Work

The following literature review briefly surveys the related work on (1) the trace collection and storage, (2) the communication between nodes, (3) the computation of the critical path, (4) the visualization of distributed traces, and (5) the use of indexes in distributed systems. Since this paper focuses on large distributed systems, the efficiency of the methods is the main focus.

6.2.1 Trace Collection and Storage

Tracing is often used as a non-intrusive and lightweight method to record the system activity by collecting low-level events generated at runtime by the operating system (kernel space) and by programs (user space). Kernel space events include system calls, interruptions, and network events, which are necessary to build the execution graph and, subsequently, the critical path. Tracing is particularly helpful in detecting abnormal behaviors such as unmet real-time deadlines, unexpected response times, and memory, load, or concurrency issues. As observed by Poirier et al. [108], tracing is similar to a high-performance `printf`.

Tracing has been applied to a wide range of systems [117], including real-time, embedded, heterogeneous, and distributed systems. Due to their low overhead, tracers are particularly suitable for investigating heavily loaded systems, such as HPC clusters, and low-resource devices, such as IoT systems and containers. Several tracers have been proposed throughout the years as listed by Toupin [118]. The preferred choice for this research is the Linux Trace Toolkit: next generation (LTTng) [119], as it is lightweight and suitable for both kernel and userspace tracing. Nonetheless, the proposed methods apply to other tracing tools as well.

As tracers collect low-level events, they typically produce a large amount of data. Consequently, traces must be stored efficiently, especially on heavily loaded systems with complex architectures [120], and manipulating large trace files is challenging. As a solution, a compression of trace files has been used on the traced nodes during the collection. These methods have been compared by Noeth et al. [125] and are used to trace MPI programs on a large number of nodes, one of the use cases targeted by this project and a representative example of a source of massive traces. Nonetheless, scaling remains challenging, even with compression and efficient data structures, as the volume of traces generated grows with the number of nodes. After the traces are collected, further communication is necessary to merge the information or metadata from all the nodes, to compute global properties like the critical path. For this reason, traces are often centralized in order to be analyzed, which further poses a communication challenge.

6.2.2 Communication Between Nodes

Traces are either analyzed on the node where they were collected, or sent over to analysis nodes. Regardless, the result of the analysis must be transferred to the viewer nodes, and thus some data must be sent over the network [131]. However, the communication capacity of distributed systems is variable. Indeed, some networks are unreliable and slow or already saturated. Moreover, the bandwidth is prioritized for services rather than monitoring, tracing, or backup, in production environments. As a result, the amount of data and the communication frequency must be reduced. In particular, the transmission of complete trace files should be avoided as much as possible. We discussed earlier the importance of having an efficient trace format. One solution to this communication challenge is for tracing daemons to periodically send the trace files to storage nodes using asynchronous communication (e.g., MOM) and to perform as much precomputation as possible on the nodes on which the traces are stored, such as filtering, compression, and snipping. Then, only the necessary data for the analysis is sent to the analysis node.

Different approaches are available to transfer the results from analysis requests. The communication between distributed nodes is generally achieved with Remote Procedure Calls (RPC) or, more recently, with Message Oriented Middleware (MOM) [126]. While RPC standard behavior is blocking, MOM benefits from a straightforward implementation of queues for asynchronous communication, and quality of service on unreliable networks [127, 128]. Consequently, Message Oriented Middlewares tend to be preferred. The communication between analysis nodes and the user is better achieved with a specific protocol such as the Trace Server Protocol (TSP) [129]. This protocol is similar to the Langage Server Protocol (LSP) [130], except that instead of being based on JSON-RPC, TSP is based on HTTP Representational State Transfer (REST). The main advantage of this protocol is the possibility of encoding data using a binary format.

6.2.3 Critical Path Computation and Analysis

The critical path, also called the active path, corresponds to the longest non idle path between the elements of a network and represents the wait dependencies between the tasks. A thorough description of the main algorithm used to compute the critical path on trace files is provided by Giraldeau and Dagenais [104] and Montplaisir et al. [137].

The critical path algorithm has been executed on a wide range of applications in order to detect and explain performance issues or logic faults. Rezazadeh et al. [138] presents a use case of the critical path for multi-threaded applications. It has also been applied to Chromium

by Ezzati-Jivan et al. [139] to detect and analyze the root causes of performance issues. The critical path is also used by Nemati et al. [106] in clusters of virtual machines to diagnose performance issues, without interacting directly with the kernel of the guest machines. The execution states extracted from the results of the active path have also been used by Fournier et al. [110] to detect latencies automatically.

Critical path methods in distributed systems require careful adaptations, as described by Nemati et al. [106]. Notably, since the critical path comprises dependencies between multiple trace files collected on different nodes, ensuring a near-perfect time synchronization between traces is crucial. Critical path computation also requires sequentially processing every trace file on a single analysis node, which is problematic for numerous large traces. Schulz [111] showed that it is possible to extract the critical path for applications running on up to 128 processors. Nevertheless, there is a real and necessary need to improve the critical path computation. The algorithm from Giraldeau and Dagenais [104] has been tested [136], and several performance issues have been identified. As a result, this article proposes an improved distributed version of the algorithm.

6.2.4 Visualization Tools for Large Traces

A layout view can give a good overview of the system on a small scale, with one or two nodes. However, as the number of nodes increases, the analysis needs to scale in order to provide a readable picture to the user.

Moreover, even if tracing tools are able to collect information about the performance of the system and communications, the main issue is the ability to analyze such an amount of data. More often than not, visualization tools do not run on multiple nodes and, therefore, only scale vertically. As traces become larger, the user interface suffers from a lack of performance. In that case, the only solution with these tools is to reduce the amount of tracing data recorded, which impacts the ability to find the root causes of performance issues.

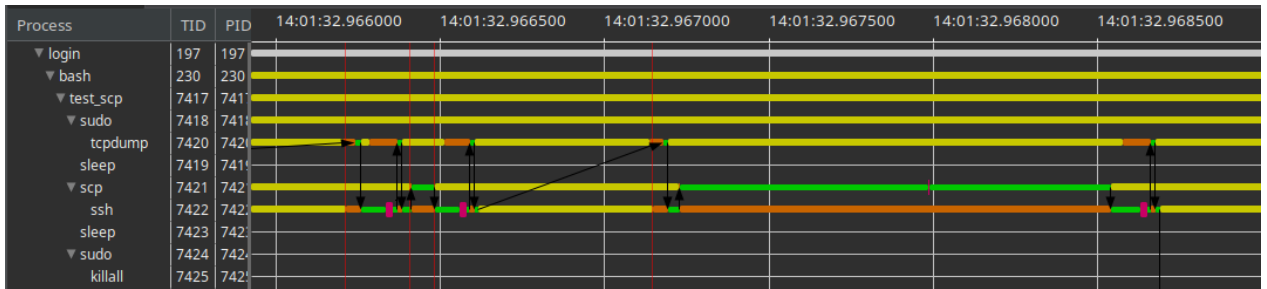


Figure 6.1 An example of a time graph displayed in Trace Compass.

Time graphs are a popular visualization for trace analysis. As illustrated in Figure 6.1, time is represented along the x-axis, and each horizontal line represents a resource that changes state over time (e.g., a thread, a core, a program). Usually, an event signals this state change. The period of time during which the state of a resource stays the same is called a state interval. On each row, rectangles represent state intervals, and a specific color represents each different state. Time graphs help keep track of the state of a system.

Trace Compass is integrated into the Eclipse IDE [131] and provides a critical path analysis module, as illustrated in Figure 6.2.

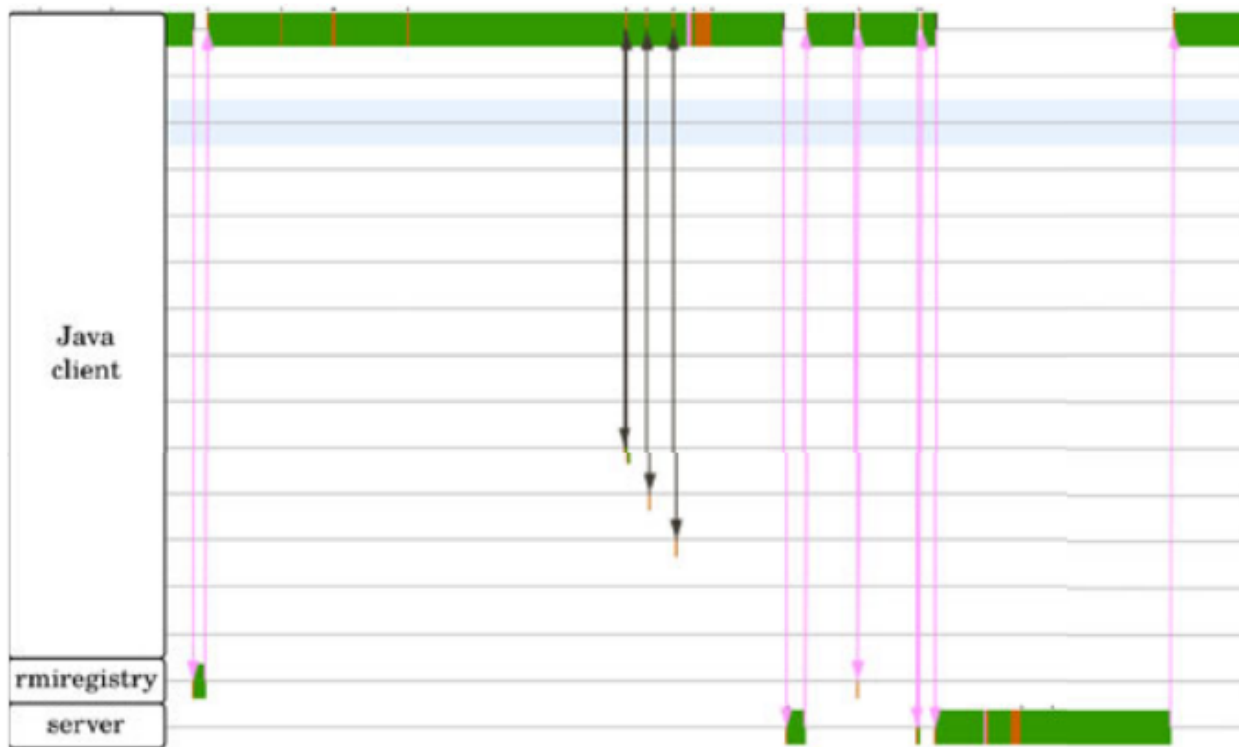


Figure 6.2 The critical path view of a process as displayed in Trace Compass.

When performing trace analysis, which is time-expensive, most tools require the users to process the trace on their own machine. This is usually performed multiple times, as each user involved in the system monitoring and development is trying to find performance defects or faults. In that case, a better approach would be to rely on a shared visualization service and use preprocessing of the data on tracing nodes. This way, fewer resources are needed to perform final computations and views.

None of the current solutions offer a complete method for efficient trace analysis on large clusters. This article will propose solutions to solve these gaps. It is important to note that

there are several approaches to parallelize the analysis of traces in Trace Compass [86]. The work of Martin [87] introduces a multi-threading algorithm for analysis. For a review of tracing tools for large distributed environments, the reader is referred to the survey of Las-Casas et al. [132]. Numerous projects are available to achieve this goal [140], most notably Zipkin [133] and Jaeger.

6.2.5 Index usage in Distributed Systems

Data locality is often the bottleneck in distributed and multi-threaded applications [141]. In our case, the traces are distributed among multiple nodes, making critical path computation more difficult. In order to balance the load on the different analysis nodes, an efficient method has to be applied to find file locations across different servers. In particular, we will focus on the use of an external communication index.

Data is often organized using a tree hierarchy. Moreover, striping data across many disks or data servers is common for data replication or system performance. Indexation, using meta-data associated with files or system elements, helps the organization and search of files [142, 143].

On the distributed file systems side, Vesta [144] and Inter-Mezzo use pathname hashing for data allocation and location. [145] evaluates a metadata management system for in-memory space efficiency. Lazy hybrid (LH) is a method presented in [146] that uses hashing for metadata distribution across the metadata servers. It uses a Metadata Look-Up Table, a file replicated between the clients and metadata servers.

In many current parallel file system implementations, clusters are organized with several file servers on which the files are effectively stored, and one or more servers that store information about the files. This information is then used to build indexes and provide an efficient way to know the location of a file.

In distributed systems, trace files are generated locally and often stored on the same machine, or moved to several nodes dedicated to trace storage and analysis. Trace file storage is a challenge, and most options that are available are used for micro services architectures [140].

As a result, a new method is needed in order to improve the indexation of trace files and improve the performance of distributed trace computation algorithms. Ultimately, a global overview of the tracing data available for the whole cluster should be available. Indexing is a good option to achieve that, as this method is widely used in distributed file systems.

6.3 Methodology to Optimise Critical Path Computation Using Index

This article presents a new distributed algorithm for the critical path computation. The focus is not on the critical path computation algorithm itself, but on the efficient communications between the distributed nodes, through the indexation of external communication events in trace files. In this section, we identify the challenges with the current methodology, and propose an improved method.

6.3.1 Requirements and Challenges

The goal of the critical path is to identify and link the dependencies between events collected simultaneously on different nodes. The first step to compute the critical path is to transfer the traces recorded on the different nodes to the analysis node, if necessary, where they must be synchronized due to the time difference between the system clocks. Then, the analysis begins by extracting the execution graph for each trace. At this point, the user needs to select a thread, or a section thereof that represents a query, on which to compute the critical path.

The execution graph of a trace is processed, with task states represented by horizontal edges, and signals represented by vertical edges, as illustrated in Figure 6.3. It is currently implemented by looking at kernel signals, scheduling events, and network calls of running processes. In the case of the Linux kernel, the `blocking` thread state can indicate a wait dependency on other processes, unlike the `running`, `preempted`, and `interrupted` thread states. Kernel events such as `sched_switch`, `sched_ttwu`, `interrupt_entry`, `interrupt_exit`, `inet_sock_local_in`, and `inet_sock_local_out` delimit events and interruptions. Interrupt requests allow determining the device causing the wait. As a result, thread states and interrupt requests are the minimal information required to determine the dependencies between processes and thus compute the critical path. Note that the critical path analysis may be applied to other operating systems, network interactions between virtual machines, as well as distributed software executions such as with the Message Passing Interface (MPI).

During the computation of the critical path of a thread, the local execution graph is scanned backward to replace all idle states, initiated by blocking events and ended by unblocking events, with the execution segments in other threads that created the unblocking events. For example, when a thread is waiting on a socket and is unblocked by receiving the expected data, the network events are responsible for the unblocking. The unblocking event origin, and the associated thread segment, should be identified in the trace of the remote node sending the data. An example of this would be a client, requesting a Web page. The client initiates the

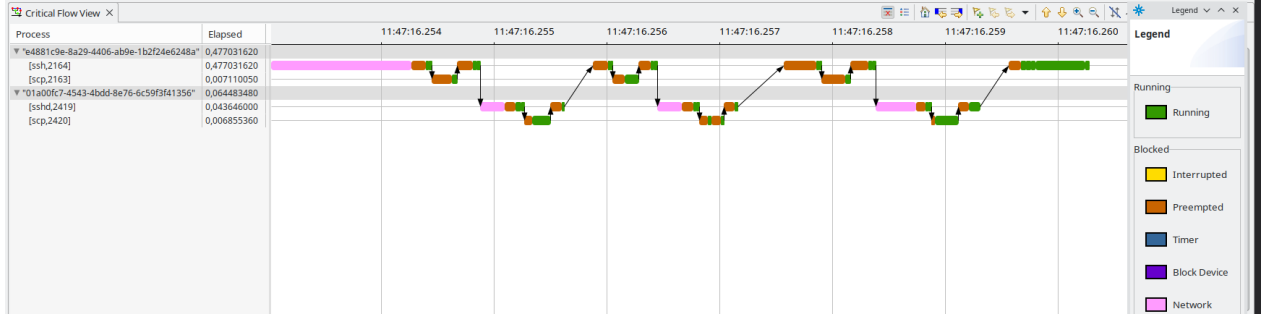


Figure 6.3 The critical path as displayed in Trace Compass

request and waits for the answer. The critical path computation is thus initiated for the client thread. The client thread is unblocked when receiving the answer from the remote Apache Web server. The Web server itself may be waiting for a while for information requested from a remote MySQL database. The critical path is composed of the client preparing the request, the Web server serving the request (while the client is blocked), the database replying to the request from the Web server (while the client and Web server are blocked), and finally the Web server sending the reply to the client and the client displaying the result.

In summary, the first step for computing the critical path is getting the execution graph from each trace, which can be easily parallelized if we initially do not follow the dependencies between traces. Then, the execution graph for the distributed system comprising multiple nodes can be obtained by stacking and connecting the graph from each node, after synchronizing the different traces to the same time reference Poirier et al. [108]. However, distributing the computation of the critical path is more challenging. Indeed, it requires finding the links between multiple trace events, and the parallelization of the longest path algorithm. Since no precomputation is possible, due to the trace data being distributed, the critical path cannot be computed without communication.

When analyzing a distributed system trace, the user will interactively navigate through the different threads and select a thread and time interval of interest (e.g. a request exhibiting an undue latency) for which computing the critical path. Precomputing all critical paths, for the whole duration of all threads, is therefore a waste of time. However, when preprocessing the traces, identifying all the wait dependencies and associated edges of the execution graph is worthwhile; all that remains is scanning the execution graph to identify the critical path for the thread and time interval specified by the user.

In our opinion, the three main contributions necessary to significantly improve the efficiency of the critical path computation, and enable analyzing large trace files, are (1) the efficient distribution of the critical path computation, (2) the distribution of traces, and (3) an efficient

communication protocol to link the trace analysis nodes. The following section describes the work introduced to complete these milestones.

6.3.2 Current Methodology

Instead of performing the critical path computation on a single machine, Denys and Dagenais [136] introduced a method to distribute the processing across multiple nodes. This section provides a brief overview of their algorithm and highlights the remaining limitations.

The original algorithm for computing the critical path synchronizes the time, computes the execution graph, and performs a backward iteration over every blocking state, to look for their origin in other threads or host processes, as detailed by Giraldeau and Dagenais [104]. However, the original algorithm presents two notable drawbacks: the large memory footprint due to the in-memory computation of the complete execution graph, and the poor scalability due to the centralized nature of the computation. First, the large memory footprint was solved previously by storing the graph on disk. Then, Denys and Dagenais [136] introduced an improved methodology based on a distributed computation of the critical path, and a new data structure for the communication between nodes.

The distributed algorithm precomputes the execution graph on each node, and synchronizes the time. Then, when the user selects a thread interval on which to compute the critical path, the execution graph is parsed, and remote dependencies are stored in a list. After completion, this list is broadcasted to all analysis nodes. Next, every node will try to resolve the remote dependencies, with their own precomputed execution graph, by matching events using relevant identifiers, such as TCP port and sequence numbers. Finally, a list of graph elements containing states corresponding to remote dependencies is sent back to the main node. The original execution graph and the remote events are then merged together. The result displayed to the user is an uninterrupted event sequence which is the critical path related to the initially selected thread interval.

The distributed algorithm enables the precomputation of trace files, thus avoiding to transfer complete trace files between nodes. However, several challenges have been encountered while using this method on large distributed traces, as identified by Denys and Dagenais [136]. Firstly, it is currently impossible to know whether a trace located on a remote host contains some events related to the currently analyzed thread. As a result, the execution graph must be processed for each remote trace. Furthermore, this method requires broadcasting the remote dependencies list to each node. As a consequence, the whole network is loaded, and all analysis nodes are busy during those computations. Ideally, only a small number of analysis nodes would be involved in a critical path computation, leaving the other nodes available to

perform other tasks, like answer other trace analysis requests from other developers.

6.3.3 Improved Methodology

This section introduces new algorithms for (1) the indexation of trace files using an *external communication index* (ECI) and (2) the use of preprocessing for the critical path computation optimization.

External Communication Index Computation

This first step aims to create an ECI for each trace file, located on analysis nodes, based on metrics about the traces, such as the number of tracepoints.

Efficient methods such as PTP (*Precision Time Protocol*) are available to synchronize traces located on different nodes while tracing and Poirier et al. [108] devised an algorithm to synchronize traces offline. Moreover, time synchronization is not necessary on systems based on the same clock, such as containers, as well as systems accurately synchronized, depending on the granularity level of events. We will not dwell on this aspect, as it is not a limitation, and traces stored on every node are considered time synchronized in all experiments in this paper.

When the preprocessing is triggered by a user request or a script, the system calls and kernel signals are parsed, in each trace file located on analysis nodes. Note that the traces have either been generated in place, or previously transferred to analysis nodes from other traced hosts. Specifically, during the trace preprocessing, system calls and signals such as `irq_handler_entry`, `irq_handler_exit`, `irq_softirq_entry`, `irq_softirq_exit` and `irq_softirq_raise` are parsed, and blocking thread states are targeted. For events related to network communications, the algorithm targets events such as `inet_socket_local_in` and `inet_socket_local_out`, or `net_dev_queue` and `net_if_receive_skb`.

These signals allow identifying the remote nodes involved in the network exchanges. A loop iterates over every blocking state and collects information about the remote node involved. After reduction, an ECI is produced, containing metrics about the trace in the header, and a list of node identifiers, involved in network exchanges, found in the trace file.

```

1 {
2   ...
3   {nodeId : "0xc0a81e02", traceFile : "8949844vdfd98", nodeLocation : "0
    xc0a81e19"},
4   {nodeId : "0xc0a81e04", traceFile : "8949844vdfd98", nodeLocation : "0
    xc0a81e19"}
5   ...
6 }

```

Listing 6.1 Example of external communication index entries

This new method is inspired by the lazy hybrid method and its Metadata Look-Up Table [146]. Thanks to this index, it is then possible to know in advance if a trace contains network exchanges with a particular remote node. Other information is also usable for a better file distribution on nodes, which may be the subject of future work.

External Communication Index Usage for an Optimised Critical Path Computation

In a cluster of multiple analysis nodes, on which multiple trace files have already been pre-processed, and the ECI files have already been generated, the first challenge is related to the communication index propagation between the analysis nodes.

Every time a trace file is open or closed on an analysis node, a data reduction process in the algorithm merges all the local ECIs into one. The result is a matrix on which each line is a combination of traced node identifier, trace file name and analysis node (where the file is stored) identifier.

Then, the propagation of data between the analysis nodes is needed. A file exchange is then triggered in order to share or update the ECI between each analysis node. A first method consists in sending a first request to the nodes to get the serial number of the ECI. The serial number is a combination of the current date of update and a unique identifier placed at the beginning of the ECI. If the serial number is the same as the copy stored locally, no further action is required. If the serial number is more recent, a request to get a new copy of the ECI is sent to the node. The second method consists in sending only one request to get the latest version of the ECI each time.

As the file size of the ECI is typically negligible, it adds more overhead to send two requests, one for the update verification and another for the file transfer. As a result, the second option is preferred, and the algorithm on the viewer node merges all the received files together. At this step, the viewer node has obtained the up to date information about the traced node

identifiers for each trace, and their location on analysis nodes.

When the new proposed critical path computation method is triggered by a user, the backward iteration on the execution graph related to the current analysed trace begins. If a blocking event is encountered and cannot be solved locally, and is network related, the packet origin is searched in the local copy of the ECI. The identifier of the analysis node on which the related traces are stored is determined, using the ECI, as illustrated in Figure 6.5.

Algorithm 2: The pseudo-code of the simplified algorithm to resolve the recursive remote dependencies in the viewer node using the ECI file.

Input: execution graph G , process P
Output: critical path CP

```

1 ...
2 function resolveBlockedStates( $G$ , optional  $P$ )
3    $BlockingStates[] \leftarrow$  new JSON data structure
4    $ReturnedEdges[] \leftarrow$  new JSON data structure
5   ...
6   forall  $BlockingStates[]$  as  $BlockingState$            // Identify the node which
   causes blocking state
7   do
8      $currentNodeId =$  getNodeIdFromSignals( $BlockingState$ )
9     if  $currentNodeId$  is in  $IndexCorrespondingNodes[]$  then
10       $IndexFileCorrespondingNodes[] \leftarrow currentNodeId$ 
11    else
12       $state \leftarrow UnresolvedNetwork$ 
13  forall  $IndexFileCorrespondingNodes[]$            // Send blocking edges to each
   respective identified nodes
14  do
15     $ReturnedEdges[] \leftarrow$  SolveBlockingStatesRemote( $BlockingStates[], nodeId$ )
16     $CP \leftarrow ReturnedEdges[]$ 
17    DisplayCP( $CP$ )           // Update CP view during processing
18    resolveBlockedStates( $CP$ ) // Recursively solve blocking states in
   returned paths
19  return  $CP$ 
20 ...

```

As a result, the viewer node is able to know which trace file on which analysis node contains the resolution of a blocking state located in the currently analyzed trace. Thus, the main node does not interrogate other analysis nodes with trace files without events related to the unblocking.

The previous distributed critical path computation algorithm was finding the origin of an

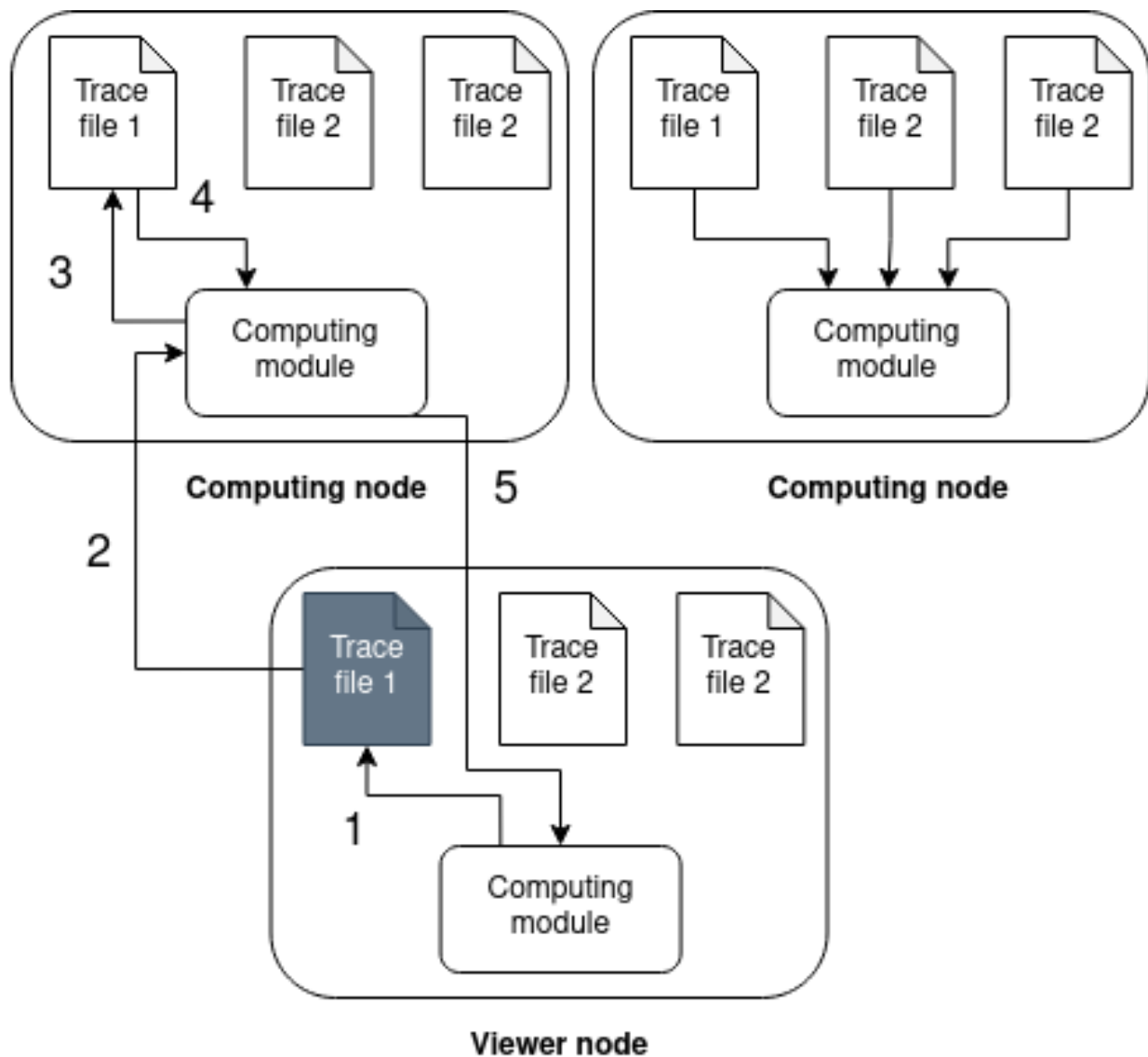


Figure 6.4 The algorithm steps illustrated on the architecture elements.

unblocking (network event) in remote traces hosted on different nodes by sending messages to all nodes.

Rather than using a broadcast paradigm, we want to use a unicast or at least a multicast messaging system. Instead of sending a message to each remote host, we can send a message only to nodes with traces in which outgoing packets correspond to the identifier of the analysis node. It leads to faster critical path processing, and less load on the cluster of analysis nodes, as only nodes involved in the computation are contacted.

Furthermore, thanks to the ECI information, instead of displaying to the user that the

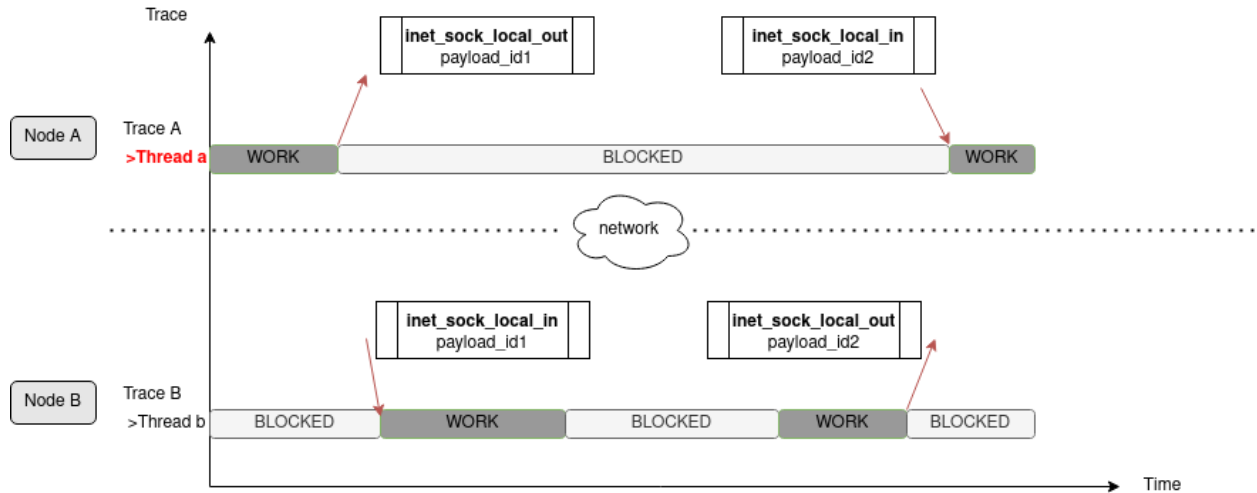


Figure 6.5 The remote node matching with common ID in the payload of events.

blocking origin has not been found, it is now possible to notify the user that the origin may be on a server that is not online at the time of analysis. This allows a better indication of missing information.

6.4 Results

This section demonstrates the effectiveness of the proposed method, on representative use cases, and shows a comparison with two earlier methods. The results of the two benchmarks detailed hereafter, and the source code of the experiments, are available in a GitHub repository².

Let us introduce this part with a simplified theoretical use-case, based on a large cluster with 20 nodes communicating together. Suppose that a thread *a* in node A sent a message to two other threads *b* and *c* in nodes B and C, respectively. Furthermore, let us consider that traces are recorded and stored locally, a trace server is running on each node, and the execution graph is preprocessed on each host. The user wants to obtain the critical path of thread *a*. This triggers the execution graph Analysis. Locally sourced blocking events are replaced by their cause found in other threads, and network blocking events are stored in a list. This list is sent to the 19 other nodes to find the origin. Each of the 19 other nodes triggers their execution graph Analysis, to find the origin of these blocked states. If the origin is found, the node replies with the sequence of events causing the blocking state. Therefore, apart from the 3 relevant nodes, A, B and C, 17 other nodes trigger their execution graph

²<https://gitlab.com/pierrefrederick.denys/critical-path-distributed-with-preprocessing>

Analysis for nothing. This costly operation could be avoided if the nodes involved in each blocking were indexed during the trace analysis.

With the new proposed approach, only two requests are sent to nodes B and C, because A was aware that B and C had events related to blockings in trace A. As a result, only the minimum number of requests are sent, each time a user triggers a critical path analysis on a thread:

$$T(j) = \max_j \mathcal{F}(i, j) + \alpha \mathcal{C}(n), \quad (6.1)$$

where j is the number of nodes, α is the number of search iterations, $\mathcal{F}$ is the processing time, and $\mathcal{C}$ is the transfer time between nodes. We will now confirm this assertion on real usecases.

6.4.1 Test Architecture

For the two benchmarks presented hereafter, and in order to have a reasonable basis for comparison, the experiments are executed on the three algorithms for the critical path analysis: the conventional algorithm currently used in Trace Compass [104] (noted **AL1** in the results), the more recent distributed computation algorithm [136] (noted **AL2**), and finally, the new distributed computation algorithm proposed here, using ECI files, (noted **AL3**). Each experiment was repeated ten times for good accuracy, and the average is displayed along with the standard deviation. The cluster comprises 21 physical computers, 20 configured as computing nodes and one more as a viewer. Each computer contains an i5-9400 at 2.9Ghz with six cores and 16GB of memory. More details on the experimental setup are available in the repository of the project.

A first small and simple use case has been used in order to characterize the overhead and gain of the new algorithm. It can be viewed as good case for the new algorithm. Indeed, 20 remote file copies are issued in parallel, and the main script then waits for all the copies to complete. Therefore, there is a single *rendez-vous* and the critical path computation will only involve the node that finished last its copy. A more representative use case, based on an industry standard benchmark that executes real parallel scientific applications, will be presented after.

6.4.2 SCP usecase : Performance and Overhead

The algorithms were tested on traces generated during a parallel SCP file copy. The copy operations were run in parallel, from the main node to a variable number, from 1 to 20, of computing nodes. As a result, a maximum total number of 21 traces are generated. Even

if the machines, on which the file is copied, are identical in terms of software and hardware, there is some variability on the time needed to complete the copy operation, because of network delays, background system tasks, and the variability of processes scheduling. The parallel copy script on the main node is waiting until all copies are completed, and thus eventually waits after the slower node on which the file is copied. In addition to the number of nodes, the other variable tested with this use case is the size of the file copied and, thus, the execution duration and the size of the trace.

Trace Files Characteristics

This part presents a characterization of the traces generated with the benchmark, and the processing time observed during the analysis with the different algorithms.

Table 6.1 details the trace files used for the experiments. Each line gives the size of the copied file and the number of nodes on which it is copied in parallel. The size of the trace file generated on the main node, and the average size of the trace file generated on each of the computing nodes, are given in columns 2 and 3. The standard deviation (noted SD) is based on the 10 executions for the main node, and on the average trace file size across the 10 executions and all the computing nodes for the computing node entry.

We can observe that the number of events increases, in the trace generated from the main node, when the number of computing nodes increases. The trace file size given in the second column is for each computation node, as their size is almost equal between nodes. For example, the last row of Table 6.1 relates to the case where a 10GB file is copied in parallel to 20 nodes, generating a total of 20 trace files of about 19.5GB, and one trace file of 5.2GB on the main node, for a total of more than 390GB of trace files to analyse. This was impossible to process with the conventional version of Trace Compass, due to memory limitations when the execution graph was stored in main memory, in the approach from Giraldeau and Dagenais [104]. Finally, the last columns contain the percentage of events, within traces, used to identify blocked states, caused by interrupt request events (`irq`) and remote dependencies (`net`), as explained in the methodology. As the size of the file transferred is the same for each repetition of the tests, the standard deviation for the proportion of network events is close to zero.

According to Table 6.1, kernel events related to interrupt requests (`irq`) represents less than 10%, and network events less than 2%. As a result, only 2% of the events need to be transferred for remote dependencies computing.

Table 6.1 The trace file used for the benchmarks : average trace sizes and events number

	File size	Trace file on each Computing node	(SD)	Trace file on main node	(SD)	Events Main	Events Dest	Events IRQ	Events Network
1	100 MB	28 MB	(1.86)	61 MB	(1.68)	1,026,089	2,205,019	9.1 %	1.0 %
	1 GB	181 MB	(1.92)	277 MB	(1.83)	6,757,000	9,981,329	8.9 %	1.6 %
	10 GB	1.8 GB	(0.33)	3.8 GB	(0.07)	69,414,305	124,240,618	9.5 %	1.5 %
2	100 MB	68 MB	(1.93)	105 MB	(2.12)	261,926	388,210	7.5 %	1.0 %
	1 GB	346 MB	(1.18)	252 MB	(1.56)	12,494,815	9,113,923	8.2 %	1.2 %
	10 GB	3.2 GB	(0.12)	2.4 GB	(0.56)	118,495,717	86,933,604	9.2 %	1.4 %
3	100 MB	71 MB	(1.58)	75 MB	(1.96)	2,644,924	2,793,396	8.4 %	1.1 %
	1 GB	443 MB	(1.12)	266 MB	(1.36)	15,984,498	9,604,204	9.2 %	1.3 %
	10 GB	4.4 GB	(0.18)	2.4 GB	(0.45)	158,781,607	88,181,239	8.6 %	1.7 %
10	100 MB	83 MB	(1.69)	78 MB	(1.85)	2,993,919	285,664	8.8 %	1.8 %
	1 GB	4.1 GB	(0.09)	262 MB	(0.66)	43,285,500	9,565,111	7.2 %	1.9 %
	10 GB	8.8 GB	(0.06)	6.8 GB	(0.05)	317,427,586	248,481,236	9.1 %	1.1 %
20	100 MB	120 MB	(1.20)	71 MB	(1.81)	3,205,774	269,877	7.8 %	1.4 %
	1 GB	9.1 GB	(0.08)	271 MB	(1.76)	334,953,487	9,854,156	9.4 %	1.2 %
	10 GB	19.5 GB	(0.05)	5.2 GB	(0.06)	698,340,689	222,199,310.2	8.7 %	1.9 %
Average								8.6 %	1.4 %

External Communication Index Computation Overhead

In this first benchmark, the computing time to display the full critical path is compared between each algorithm. The number of nodes on which the file is transmitted with scp process varies from 1 to 20. Table 6.2 gives the average total time needed to display the entire critical path of a thread of interest. The columns *Mean Processing Time* gives the time needed to fully display the critical path for each algorithm. The processing time for AL2 and AL3 is given for the first execution (column "First") and subsequent executions (column "Sub"). Indeed, the ECI processing and execution graph generation is only done once, at the first critical path request, and remains available on disk thereafter. The subsequent requests are faster (provided that the same trace set is selected) as preprocessing is already done. The last column gives the overhead induced by the ECI file pre-computing in AL3, compared to AL2, for the first critical path request. It is computed as the time difference in the trace initial processing. The average overhead is around 2% of the trace indexation and synchronization time. It is important to note that this overhead is only applied at the first opening of the trace. Indeed, once the ECI file is computed, it is reused for further critical path analysis, as long as new trace files are not added or deleted. It is important to note that AL2 and AL3 are incremental algorithms; a first display is available to the user after a few seconds. The display is then updated incrementally, as the blocked states are replaced by the remote threads and their state, while remote dependencies are resolved.

Algorithm (AL1) refers to the conventional sequential algorithm, for which the full processing of the execution graph is needed, on a single analysis node, for displaying the critical path.

Table 6.2 The external communication index computation overhead : Mean total processing time of each algorithm.

Nodes	File size	Mean Processing Time (s)					Time overhead (SD)
		AL1	AL2		AL3		
		First	First	Sub	First	Sub	
1	100MB	7.35	3.01	1.26	3.07	1.25	2.11% (0.10)
	1GB	43.6	15.7	1.96	16.0	2.01	2.02% (0.07)
	10GB	570	210	3.23	214	3.42	2.11% (0.05)
2	100MB	23.2	4.98	1.34	5.11	1.38	2.25% (0.23)
	1GB	69.2	19.4	2.01	19.3	2.08	2.03% (0.12)
	10GB	632	235	3.28	240	3.34	2.41% (0.06)
3	100MB	212	67.9	1.36	69.3	1.39	2.01% (0.18)
	1GB	687	210	2.59	214	2.21	2.17% (0.09)
	10GB	958	297	3.32	303	3.39	2.12% (0.06)
10	100MB	685	80.6	1.58	79.0	1.68	2.03% (0.19)
	1GB	9890	356	2.63	364	2.64	2.41% (0.12)
	10GB	NP	1,916	4.01	1,957	4.18	2.15% (0.08)
20	100MB	1360	75.2	1.62	76.6	1.65	2.14% (0.21)
	1GB	NP	38.3	2.67	38.8	2.69	2.06% (0.14)
	10GB	NP	3,780	3.89	3,867	3.81	2.32% (0.07)

That explains the impossibility to handle the larger traces and the "NP" mention in the table for *not possible*.

The new proposed algorithm AL3 does not really improve the elapsed time for computing the critical path, when the computing nodes are not doing anything else, as compared to AL2. However, by only sending remote requests to the remote nodes containing trace files affecting the critical path, a lot of computing resources are saved. This is measured both in terms of network bandwidth and CPU time usage, across the cluster of analysis nodes. The results are presented in the next two subsections.

Transferred Data Amount Improvement

The total amount of data exchanged over the network is compared for algorithms AL2 and AL3, in order to characterize the gain of the new algorithm. These results have been obtained with the network traffic capture capabilities of the Wireshark tool, and its "conversation" filter, to only measure the amount of data related to the critical path computing, and exclude unrelated packets.

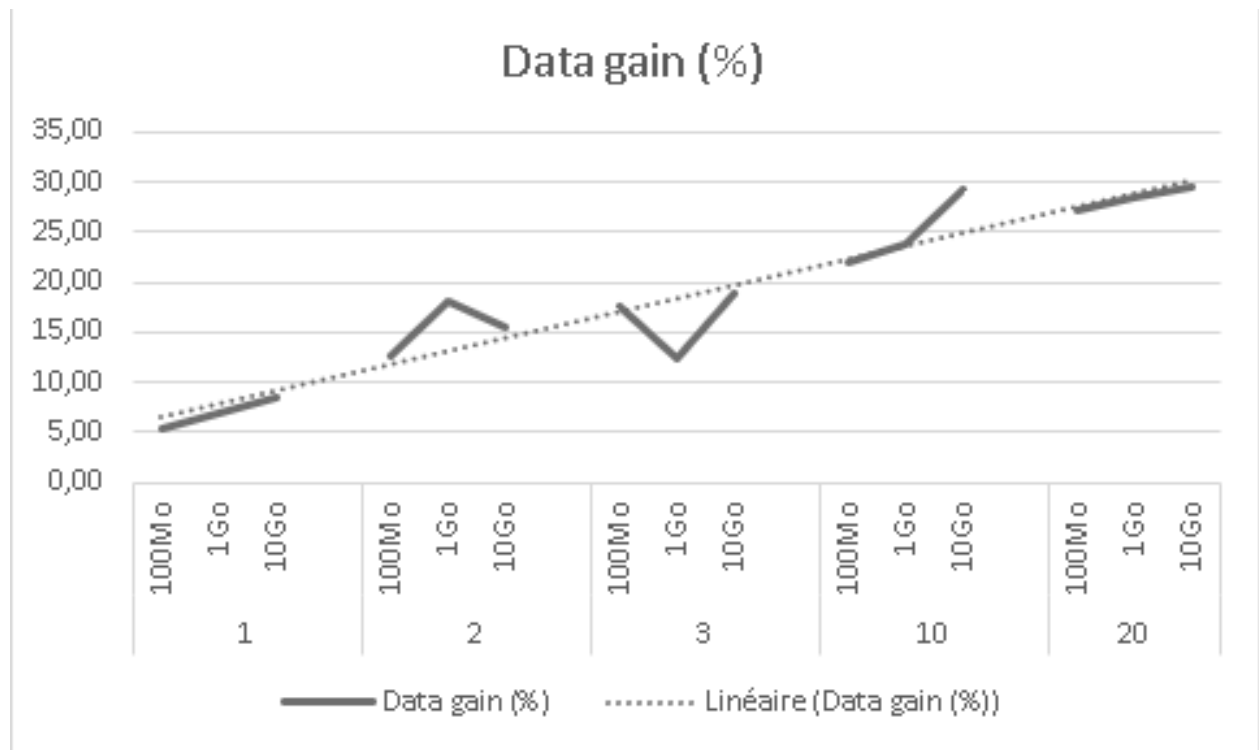


Figure 6.6 The data gain on transferred data processing

We observed a gain in the amount of data transferred between nodes across the network when a critical path is computed. This is an average gain of 18% of the total transferred data, and the maximum gain observed for this benchmark is near 30%. We can see that the improvement increases with the number of traced nodes. This improvement seems limited because it takes into account the additional data exchanged between nodes on the first iteration. When a user needs to process several critical path analyses, the pre-computing is done only once, and this gain becomes even larger.

Computation CPU Time Gain

For this experiment, the CPU usage of the Trace Compass process has been measured on each computing node, during the critical path processing, in the previous experiment. Then, these results are added for both algorithms AL2 and AL3, with and without the ECI. Table 6.7 gives the percentage of improvement with the usage of the ECI, depending on the number of nodes in the traced cluster. The CPU load has been recorded with the `mpstat` tool, to give the average CPU load on each node.

$$\frac{\sum_{i=0}^{20} CPU_{load}}{20}$$

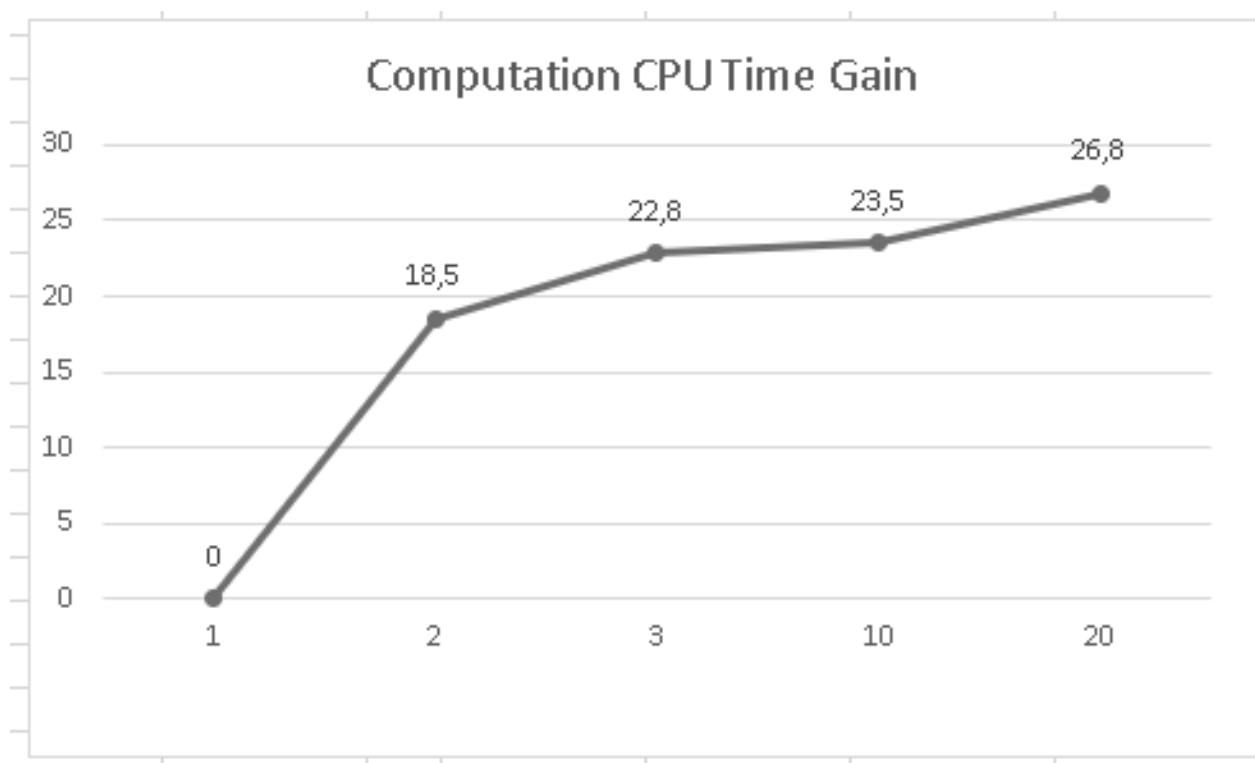


Figure 6.7 The gain in terms of global CPU usage of the analysis nodes.

A significant gain in terms of processing time has been made with the distributed algorithm AL3 compared to AL2, even if the gain in terms of processing time is variable with the ECI. If many nodes are involved in network communications, the computation node must send multiple requests, and the performance improvement is limited. If only two or three nodes are required to be queried for remote dependencies resolution, the gain in performance is substantial. As a result, the new method is particularly efficient for large traces and many nodes.

6.4.3 MPI usecase : Performance and Overhead

For this usecase, the industry standard SPEChpcTM 2021 benchmark [147] of the "Standard Performance Evaluation Corporation" (SPEC[®]) has been used. It comprises nine production HPC applications from various scientific domains and several programming languages. This test suite comes with useful benchmarking tools and is suitable for an extensive cluster size range from one to a few hundred nodes. This test suite is used to reproduce a near-to-real computation cluster running HPC workloads and using the Message Passing Interface (MPI) library. Each of the 20 computing nodes has a 6-core CPU. The experiments are run with 120 MPI ranks, one for each core of each node. More details about the environment are available in the public repository.

Trace Files Characteristics

For this use case, four benchmark applications are used and run with different input sizes to generate different trace file sizes.

- *SOMA Offers Monte-Carlo Acceleration*: is a High-Performance Computing Monte-Carlo simulation for soft coarse-grained polymers. The variable load is the number of simulated polymers.
- *Tealeaf*: solves the linear heat conduction equation on a spatially decomposed regular grid. The variable load is the number of cells.
- *Minisweep*: is a Nuclear Engineering and radiation transport simulation. The variable load is the grid cell size.
- *SPH-EXA*: performs hydrodynamical and computational fluid dynamics simulations. The variable load is the power of 3 of the number of particles.

Table 6.3 presents the traces obtained during application executions. The parameters refers to the variable used to obtain a variation of runtime. More details about their signification are available in the repository. The most relevant informations are the size of the generated trace files. The largest use case is Tealeaf with parameter 32768 which generated 20 times 3.94GB for the remote nodes and 83.2GB for the main node, for a total of 162GB of trace files to analyse to compute the critical path.

Table 6.3 The trace file used for the benchmarks

Benchmark	Parameter	Exec.	Trace size		Event		
		Time (s)	avg.	per node	Total	Irq (%)	Network (%)
Soma	1400	6	7.5 MB	(3.52)	158 MB (1.21)	12.3	1.08
	14000	21	29 MB	(1.82)	613 MB (1.38)	12.9	1.14
	140000	261	256 MB	(1.42)	5.4 GB (1.24)	13.1	1.85
	1400000	2355	2.83 GB	(1.02)	63.8 GB (0.88)	13.6	1.90
Tealeaf	512	61	73MB	(2.12)	1.5 GB (1.28)	13.6	1.91
	1024	78	93 MB	(1.21)	1.9 GB (1.02)	12.6	1.92
	4096	411	494 MB	(1.52)	10.45 GB (0.56)	13.2	1.34
	32768	3120	3.94 GB	(0.83)	83.2 GB (0.08)	12.9	1.41
Minisweep	32	75	90 MB	(2.04)	1.9 GB (1.21)	13.6	1.75
	64	150	182 MB	(1.84)	3.85 GB (0.81)	13.6	1.28
	128	315	379 MB	(1.62)	20.3 GB (0.93)	12.9	1.71
	768	1890	2.27 GB	(1.13)	48.2 GB (0.38)	12.3	1.05
SPH-EXA	20	246	296 MB	(1.57)	6.26 GB (1.09)	13.6	1.00
	40	470	565 MB	(1.84)	11.9 GB (1.28)	12.8	1.02
	60	705	846 MB	(1.92)	17.9 GB (0.52)	13.7	1.13
	120	1410	1.69 GB	(1.07)	36 GB (0.31)	13.0	1.54

Table 6.4 The overhead of External Communication Index Computation usage

Benchmark	Parameter	Mean Processing Time (s)					Time overhead (SD)
		AL1	AL2		AL3		
		First	First	Sub	First	Sub	
Soma	1400	143	68.3	1.08	69.5	1.16	1.8 (0.12)
	14000	501	203	1.59	206	1.68	1.6 (0.15)
	140000	6223	1244	2.58	1267	2.79	1.9 (0.11)
	1400000	NA	2807	3.11	2857	3.30	1.8 (0.09)
Tealeaf	512	1455	715	1.63	729	1.69	1.9 (0.16)
	1024	1860	695	1.68	711	1.73	2.3 (0.17)
	4096	9800	1960	2.54	1999	2.68	2.0 (0.14)
	32768	NA	3719	3.68	3787	3.74	1.8 (0.08)
Minisweep	32	1788	894	1.72	912	1.75	2.0 (0.15)
	64	3577	1192	1.88	1213	1.98	1.7 (0.16)
	128	7511	2504	3.12	2555	3.21	2.1 (0.19)
	768	NA	2253	3.19	2295	3.28	1.9 (0.12)
SPH-EXA	20	5866	2704	2.06	2757	2.16	1.9 (0.06)
	40	11207	3735	3.39	3825	3.48	2.4 (0.10)
	60	16810	3362	3.59	3443	3.68	2.4 (0.14)
	120	NA	4689	3.17	4779	3.29	1.9 (0.13)

External Communication Index Computation Overhead

The table 6.4 gives the global average overhead on computation time observed for several trace files sizes and different traced programs using MPI.

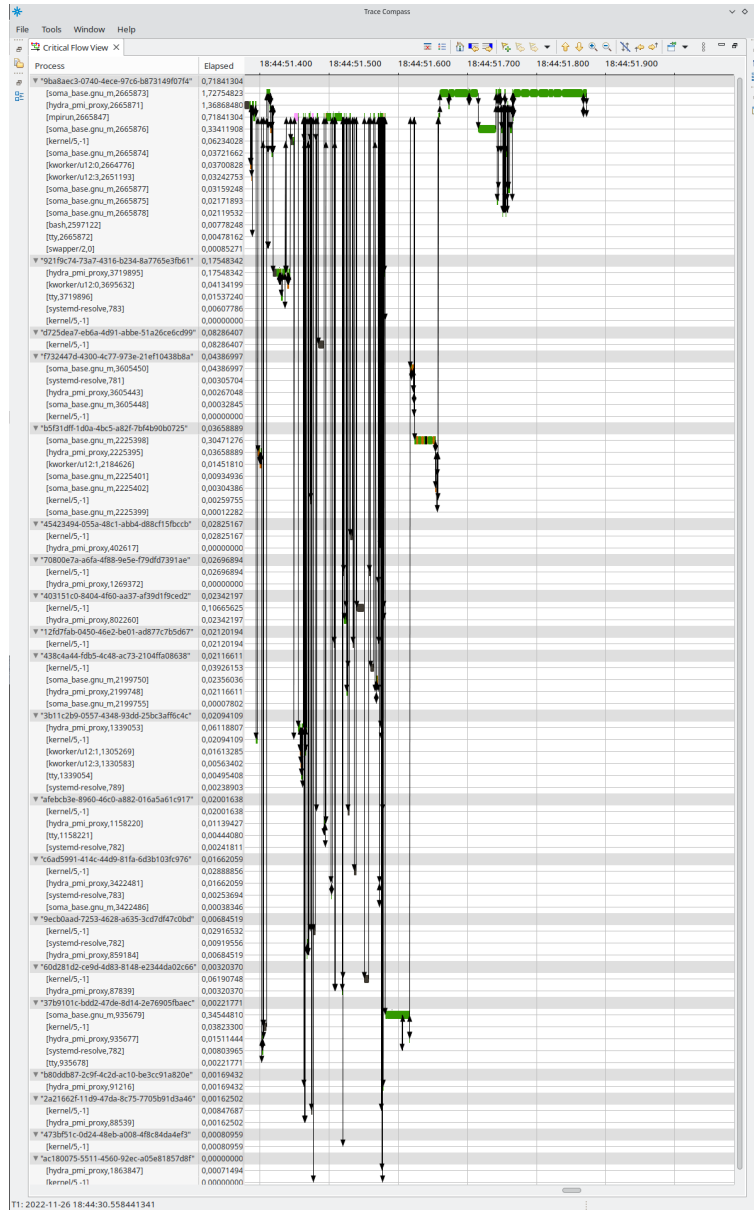


Figure 6.8 Example of global critical path view for Soma benchmark

Although the elapsed processing time is slightly higher with AL3 as compared to AL2, there is no significant impact on the user experience, as the display of the critical path is incremental, being updated during processing. The first display of the critical path begins a few seconds after being requested. The times in the table are the total elapsed time to obtain the full critical path graph.

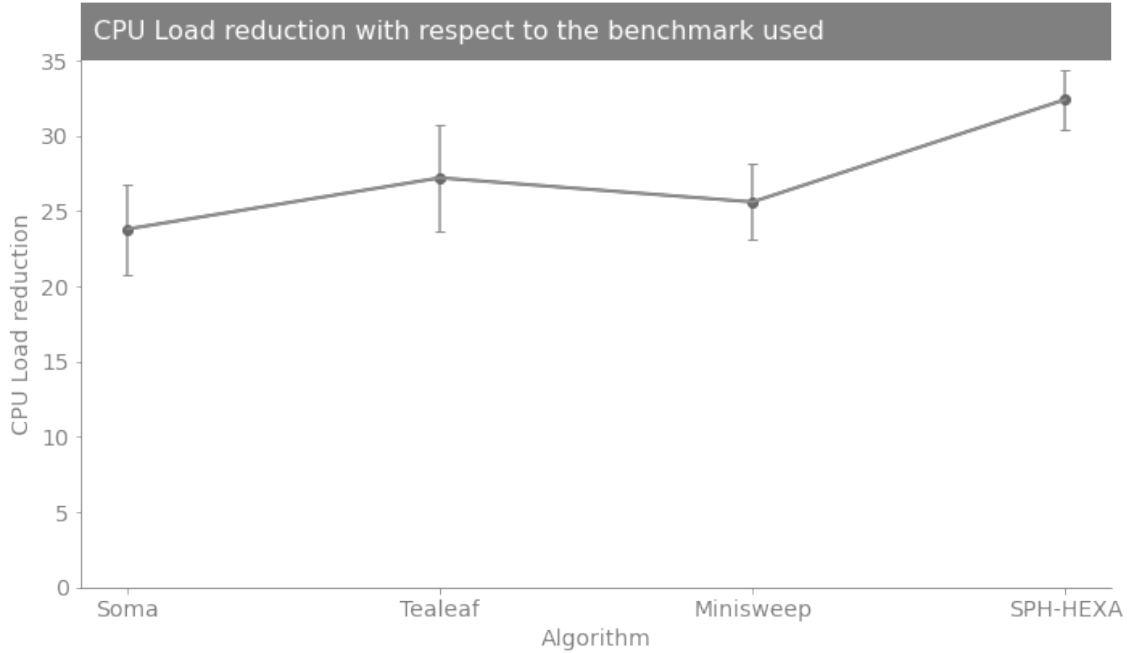


Figure 6.9 CPU load reduction with respect to the benchmark used

Computation CPU Time Gain

The graph 6.9 shows the global average CPU load reduction, with AL3 as compared to AL2, observed across the computing nodes. We can see that the CPU load reduction is around 25% on average with AL3. The gain in terms of CPU load is appreciable and limits the load on the computing nodes.

6.4.4 Discussion

In real-world use cases, such as MPI programs on HPC clusters, calculations are performed by a large number of nodes with the same architecture and resources. The challenge is to properly distribute the computations over the massively parallel cluster. Bottlenecks are created when the load is not perfectly balanced among all the nodes. This appears as nodes being blocked by synchronization barriers, where the computation cannot proceed and some nodes are blocked until all the nodes have finished. In such cases, the slowest node is the bottleneck in the system, and a blocking states will be found in the other nodes, especially in the coordinator node in charge of distributing the computation. Tracing helps to identify the slowest node and, ultimately, the cause of this slowdown. The MPI benchmarks presented in this article are representative of real usecases and give an overview of the ability of the new critical path distributed method to handle large traces, while computing the critical path in

a reasonable amount of time.

The results show a 2% overhead on the computation time for executing the first request for the critical path. This time includes the preprocessing of traces, the ECI file computation, and building the execution graph. However, there is no significant overhead on subsequent requests. This is advantageous in the case where multiple users work on the same set of traces. The traces preprocessing is only needed once and subsequent requests by the same user or other users can proceed very quickly. For this usecase as well, the new proposed algorithm AL3 brings a 25% reduction in data transmissions and a 27% CPU time gain observed for a cluster of 20 nodes.

6.5 Conclusion

In conclusion, the new algorithm presented in this article allows a more targeted and efficient distribution of the critical path computation, especially for large distributed systems with many traced nodes and analysis nodes. This method based on indexing the remote dependencies for the blockings identified in each trace file, to avoid useless computations and communications between nodes. It leads to a better response time when the critical path analysis is triggered. It is another step for adapting the current trace analysis tools to very large distributed systems analysis.

In future work, three aspects will be examined for possible optimisations. First, it would be interesting to study the trace files placement between nodes, and propose a technique to efficiently and automatically migrate trace files to balance the load and minimize the communications. Notably, such an improvement would add elasticity to the system and improve the resources usage. Second, the performance of a distributed analysis system depends on a reliable communication pattern. Therefore, a more efficient communication protocol and messaging system between nodes would further optimize the proposed method. It would be relevant to consider tests over larger clusters. Finally, the proposed technique can be extended to other distributed analyses, to improve their efficiency by speeding up information retrieval.

6.6 Acknowledgements

We would like to gratefully acknowledge the Natural Sciences and Engineering Research Council of Canada (NSERC), Prompt, Ericsson, Ciena, AMD, and EfficiOS for funding this project.

CHAPTER 7 GENERAL DISCUSSION

We discuss in this part the limitations of the solution presented, before concluding the thesis with recommendations and future work.

A first limitation which appeared to us during this research project is the difficulty to find distributed applications ready to be instrumented. It is difficult to reproduce a cluster of hundreds of nodes. The volume of kernel or userspace events traced has a major impact on the size of the generated traces. To counter that, we attempted to reproduce a representative architecture, as used in high performance computing systems. We used an industry standard benchmark relying on widely used applications like MPI. This allowed to get representative results, and observe tendencies related to larger clusters.

For the message oriented middleware analysis, the tests have been performed on one of the most popular messaging libraries. This demonstrates the approach and similar results should be obtained with other messaging libraries. Nonetheless, while it is clearly possible to instrument other libraries, the exact overhead may vary and has not been measured. However, no concerns were identified that would lead us to believe that the results would significantly differ with other messaging libraries. Such instrumentation raises security issues and traces need to be suitably protected, or their content obfuscated. On the kernel instrumentation side, it requires elevated access to the host machine and, in some case, the access can be restricted or difficult to obtain. For an easier analysis of traces generated through these methods, specific views could be developed as future work. The number of nodes involved, and accordingly of trace files to manage, implies efficient filtering capabilities, as well as the ability to provide a global overview of the traced system.

On the distributed analysis method side, a major gain in terms of total processing time is only observed on systems with many small traces and analysis nodes. The gain on systems with large traces, but a limited number of nodes, is rather small, considering the additional overhead of the analysis. However, in high performance computing clusters, due to the number of computing nodes, the distribution of trace files is often easily achieved.

On the last contribution, in the third paper, we observe a small overhead on the computation time due to the ECI file computation. Nonetheless, being as low as 2%, it presents a negligible impact on the user experience, since it only affects the pre-computation part. This pre-computation is performed in background, before the user browses the trace interactively and issues requests to compute the critical path for a specific thread or part thereof. Furthermore, it should be noted that subsequent requests for critical path computations are almost

instantaneous, due to this pre-computation step.

Finally, some operations such as traces time synchronisation, and grouping as an experiment all the relevant trace files comprising an MPI (or other) parallel application spanning several computing nodes, were achieved with external scripts for the purpose of prototyping and validating the new proposed algorithms. These operations will easily be integrated in the trace analysis software workflow in the future, for an even better user experience.

In addition to performance improvements provided by this work, some related future work has been identified on the trace selection side. An easier method could be implemented to specify nodes and trace files that need to be grouped, since they are related to a same application or execution. For example, this could be used to find all the trace files related to an MPI application execution. For now, this action is performed manually or scripted, but could be integrated in trace analysis software. A file indexation could be performed periodically, or at the end of a tracing session, and ECI files could be created at this time automatically. The proposed distributed method, as well as the external communication index, can be extended to other analyses. Pre-computing could be used to compute metrics on each remote node, and reduce the computations and amount of data transferred to trace viewing client node. The goal of this research project was indeed to allow faster and easier large distributed traces analysis, for developers and system administrators of high performance computing clusters.

CHAPTER 8 CONCLUSION

A summary of the work carried out within the scope of this research project is provided hereafter. We conclude this thesis by suggesting potential research avenues for future investigations.

8.1 Summary of Works

The goal of this research work was to solve the scalability issue with current tracing and visualization tools, observed for large cluster analysis. The new algorithms proposed bring solutions to distribute as much as possible the computing work of trace analysis and optimize it as well.

The two new methods proposed in Chapter 4 improve the efficiency of Message Oriented Middleware analysis. It provides analysis for packets exchanged with message-oriented middleware libraries, queue size, queue filling, processing time, or lost packets. All that provides a better understanding of bottlenecks and performance issues. The first method based on kernel instrumentation allows the collection of related traces to exchanged messages between containers, from the container host level. An alternate method, based on messaging library-level instrumentation, has a lower intrusiveness but requires elevated access to the host machine. Both methods are complementary, and their selection should depend on environmental constraints that can be observed in production environments. The flexibility of these methods allows for the adjustment of the granularity of the analysis and, therefore, the ideal amount of data collected for analysis. The overhead of the library instrumentation on the CPU cycles consumed is around 3%, for a medium load, and around 15%, for a higher load. This impact is not negligible and depends on the granularity of traced events selected by the user. The kernel tracing performed on a production cluster shows an overhead of 4% on average. These methods present low intrusiveness and avoid source code instrumentation of the application. It is, as a result, totally decoupled from the application source code.

The research work presented in Chapter 5 proposes a method that offers reusable patterns for the distribution of the analysis algorithm. It focuses on the critical path analysis computation. The view allows the user to quickly identify bottlenecks and explain the latency areas present in the time interval covered by the trace. It allows intrinsically to understand the actual system state during the blocked state intervals. The incremental method allows the user to get the first display almost instantly, and an updated view as the analysis progresses. Very large

systems analysis was not possible before, because the transfer of all traces to the analysis node was needed. It led to memory overflow due to the memory storage requirements of system states, and very long analysis times due to the sequential paradigm. This new distributed algorithm can be extended to several other analyses for parallelizing the computation. This solution brings an improvement on the security side, as trace file transmission between nodes is not required. Sensitive data is less exposed. The graph elements, like events for the execution graph, and system states for the critical path, can be filtered on the computing node.

Finally, in Chapter 6, we investigated how to improve the efficiency of the distributed algorithm proposed in 5. This new method, based on using an External Communication Index, optimizes the critical path computation by avoiding useless communications between computing nodes. The pre-computation of the traces, on the tracing nodes, improves the performance of the distributed analysis and the overall user experience. Indeed the combination of improvements brought by 5 and 6 allow to interactively query and display the critical path for a selected thread of a portion thereof. As a result, a near-instantaneous first display of the critical path is presented, and is updated while the nested remote blockings are resolved as the computation proceeds. This method can also be expanded for other analyses, to know in advance the content and localization of traces, over the parallel analysis nodes. For example, when an application of interest runs on a portion of nodes of a cluster, it is possible with the index to quickly identify and locate the nodes and associated traces that contain trace data relevant to the analysis requested. This new method brings a 25 % reduction in data transmission between computing nodes, and a 27% CPU time gain, for an MPI application running on a 20 nodes cluster.

8.2 Future work

For the first research axis, the new method allows the analysis of packets exchanged with message-oriented middlewares in container architectures. The library instrumented was ZeroMQ, but the instrumentation can be extended to other popular messaging libraries. The integration of the instrumentation to a newer version of ZeroMQ is planned. The collected data in traces allows for learning about queue size, processing time, and lost packets. Nevertheless, it would be interesting to develop more graphical views to display these metrics in trace viewers, in order to analyze the generated traces effectively. Similar work has been done with DPDK, where several new graphical views were developed. This work can be reused to achieve this goal. On the kernel tracing side, the integration to future versions of LTTng needs to be achieved, and more specific views could be developed in order to follow messages

between nodes, and identify clearly abnormal behaviours like blocked and lost messages. This is especially relevant for systems with a very large number of nodes, and on which manual analysis is not conceivable.

Concerning the second research axis, the proposed method is ready to use and integrate into the current version of Trace Compass. It can also be implemented in other trace analysis software tools, to allow distributed trace analysis. We identified several improvements for future work. A distributed synchronization method needs to be implemented to time synchronize traces before analysis. Some suitable algorithms are available. A module in charge of the communication between analysis and traced nodes, with a standardized API, is currently under development in our research group. This module, called Trace compass coordinator, achieves the communication between computing nodes and viewer nodes. Node management can be achieved with it, as well as the External Communication Index repartition. Using a standard API also allows distributing the computation of other analyses. Last but not least, it allows other trace viewers to use the trace analysis system as a backend.

This research has proposed new methods for the distributed performance analysis of large-scale systems. It allows faster rendering for the analysis of large traces over multiple nodes. The fields of applications are many, such as High-Performance Computing, Cloud Computing, or IoT systems. Finally, we would like to highlight that the code of the algorithms developed, as well as the data used for the benchmarks, are all available in public repositories.

REFERENCES

- [1] A. S. Tanenbaum and M. Van Steen, *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.
- [2] I. Mavridis and H. Karatza, “Combining containers and virtual machines to enhance isolation and extend functionality on cloud computing,” *Future Generation Computer Systems*, vol. 94, pp. 674–696, May 2019. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0167739X18305764>
- [3] A. M. Joy, “Performance comparison between Linux containers and virtual machines,” in *2015 International Conference on Advances in Computer Engineering and Applications*, Mar. 2015, pp. 342–346, iSSN: null.
- [4] R. K. Barik *et al.*, “Performance analysis of virtual machines and containers in cloud computing,” in *2016 International Conference on Computing, Communication and Automation (ICCCA)*, Apr. 2016, pp. 1204–1210, iSSN: null.
- [5] —, “Performance analysis of virtual machines and containers in cloud computing,” in *2016 International Conference on Computing, Communication and Automation (ICCCA)*. IEEE, 2016, pp. 1204–1210.
- [6] S. H. Davis *et al.*, “Using virtual machine containers in a virtualized computing platform,” Patent, Apr., 2016, library Catalog: Google Scholar Publisher: Google Patents.
- [7] F. Manco *et al.*, “My VM is Lighter (and Safer) than your Container,” in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 218–233.
- [8] Z. Li *et al.*, “Performance overhead comparison between hypervisor and container based virtualization,” in *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*. IEEE, 2017, pp. 955–962.
- [9] T. Salah *et al.*, “Performance comparison between container-based and VM-based services,” in *2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)*. IEEE, 2017, pp. 185–190.
- [10] W. Felter *et al.*, “An updated performance comparison of virtual machines and Linux containers,” in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2015, pp. 171–172.

- [11] A. Haque, A. Ayyar, and S. Singh, “Chroot Hardening Through System Calls Modification,” in *2018 8th International Conference on Cloud Computing, Data Science Engineering (Confluence)*, Jan. 2018, pp. 491–496, iSSN: null.
- [12] E. W. Biederman and L. Networkx, “Multiple instances of the global linux namespaces,” in *Proceedings of the Linux Symposium*, vol. 1. Citeseer, 2006, pp. 101–112.
- [13] D. Bernstein, “Containers and Cloud: From LXC to Docker to Kubernetes,” *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, Sep. 2014, conference Name: IEEE Cloud Computing.
- [14] Kovács, “Comparison of different Linux containers,” in *2017 40th International Conference on Telecommunications and Signal Processing (TSP)*, Jul. 2017, pp. 47–51, iSSN: null.
- [15] T. Combe, A. Martin, and R. Di Pietro, “To Docker or Not to Docker: A Security Perspective,” *IEEE Cloud Computing*, vol. 3, no. 5, pp. 54–62, Sep. 2016, conference Name: IEEE Cloud Computing.
- [16] D. Beserra *et al.*, “Performance Analysis of LXC for HPC Environments,” in *2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems*, Jul. 2015, pp. 358–363, iSSN: null.
- [17] C. Anderson, “Docker [Software engineering],” *IEEE Software*, vol. 32, no. 3, pp. 102–c3, May 2015, conference Name: IEEE Software.
- [18] N. Naik, “Building a virtual system of systems using docker swarm in multiple clouds,” in *2016 IEEE International Symposium on Systems Engineering (ISSE)*, Oct. 2016, pp. 1–3.
- [19] L. Ma, S. Yi, and Q. Li, “Efficient service handoff across edge servers via docker container migration,” in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, ser. SEC ’17. San Jose, California: Association for Computing Machinery, Oct. 2017, pp. 1–13. [Online]. Available: <https://doi.org/10.1145/3132211.3134460>
- [20] E. Casalicchio and V. Perciballi, “Measuring Docker Performance: What a Mess!!!” in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, ser. ICPE ’17 Companion. L'Aquila, Italy: Association for Computing Machinery, Apr. 2017, pp. 11–16. [Online]. Available: <https://doi.org/10.1145/3053600.3053605>

- [21] G. M. Kurtzer, V. Sochat, and M. W. Bauer, “Singularity: Scientific containers for mobility of compute,” *PLoS ONE*, vol. 12, no. 5, May 2017. [Online]. Available: <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5426675/>
- [22] G. Hu, Y. Zhang, and W. Chen, “Exploring the Performance of Singularity for High Performance Computing Scenarios,” in *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, Aug. 2019, pp. 2587–2593, iSSN: null.
- [23] L. Gerber, *Containerization for HPC in the Cloud: Docker vs Singularity-A Comparative Performance Benchmark*, 2018.
- [24] A. Khan, “Key Characteristics of a Container Orchestration Platform to Enable a Modern Application,” *IEEE Cloud Computing*, vol. 4, no. 5, pp. 42–48, Sep. 2017, conference Name: IEEE Cloud Computing.
- [25] L. Mercl and J. Pavlik, “The comparison of container orchestrators,” in *Third International Congress on Information and Communication Technology*. Springer, 2019, pp. 677–685.
- [26] A. Tosatto, P. Ruiiu, and A. Attanasio, “Container-based orchestration in cloud: state of the art and challenges,” in *2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems*. IEEE, 2015, pp. 70–75.
- [27] A. Leung, A. Spyker, and T. Bozarth, “Titus: introducing containers to the Netflix cloud,” *Communications of the ACM*, vol. 61, no. 2, pp. 38–45, Jan. 2018. [Online]. Available: <https://doi.org/10.1145/3152529>
- [28] E. Casalicchio and S. Iannucci, “The state-of-the-art in container technologies: Application, orchestration and security,” *Concurrency and Computation: Practice and Experience*, vol. n/a, no. n/a, p. e5668, _eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.5668>. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5668>
- [29] W. Gerlach *et al.*, “Container Orchestration for Scientific Workflows,” in *2015 IEEE International Conference on Cloud Engineering*, Mar. 2015, pp. 377–378, iSSN: null.
- [30] N. Dragoni *et al.*, “Microservices: How To Make Your Application Scale,” in *Perspectives of System Informatics*, ser. Lecture Notes in Computer Science, A. K. Petrenko and A. Voronkov, Eds. Springer International Publishing, 2018, pp. 95–104.

- [31] M. G. Xavier *et al.*, “Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments,” in *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, Feb. 2013, pp. 233–240, iSSN: 2377-5750.
- [32] W. Li, A. Kanso, and A. Gherbi, “Leveraging Linux Containers to Achieve High Availability for Cloud Services,” in *2015 IEEE International Conference on Cloud Engineering*, Mar. 2015, pp. 76–83, iSSN: null.
- [33] G. L. Lee *et al.*, “Lessons learned at 208k: towards debugging millions of cores,” in *SC’08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. IEEE, 2008, pp. 1–9.
- [34] I. Laguna *et al.*, “Debugging high-performance computing applications at massive scales,” *Communications of the ACM*, vol. 58, no. 9, pp. 72–81, 2015, publisher: ACM New York, NY, USA.
- [35] J. Higgins, V. Holmes, and C. Venters, “Securing user defined containers for scientific computing,” in *2016 International Conference on High Performance Computing Simulation (HPCS)*, Jul. 2016, pp. 449–453.
- [36] —, “Orchestrating Docker Containers in the HPC Environment,” in *High Performance Computing*, ser. Lecture Notes in Computer Science, J. M. Kunkel and T. Ludwig, Eds. Cham: Springer International Publishing, 2015, pp. 506–513.
- [37] “remote procedure call - an overview | ScienceDirect Topics.” [Online]. Available: <https://www.sciencedirect.com/topics/computer-science/remote-procedure-call>
- [38] L. Qilin and Z. Mintian, “The State of the Art in Middleware,” in *2010 International Forum on Information Technology and Applications*, vol. 1, Jul. 2010, pp. 83–85, iSSN: null.
- [39] V. Issarny, M. Caporuscio, and N. Georgantas, “A Perspective on the Future of Middleware-based Software Engineering,” in *Future of Software Engineering (FOSE ’07)*, May 2007, pp. 244–258, iSSN: null.
- [40] D. Menasce, “MOM vs. RPC: communication models for distributed applications,” *IEEE Internet Computing*, vol. 9, no. 2, pp. 90–93, Mar. 2005, conference Name: IEEE Internet Computing.

- [41] S. Vinoski, "Where is is middleware," *IEEE Internet Computing*, vol. 6, no. 2, pp. 83–85, Mar. 2002, conference Name: IEEE Internet Computing.
- [42] P. Sommer *et al.*, "Message-oriented Middleware for Industrial Production Systems," in *2018 IEEE 14th International Conference on Automation Science and Engineering (CASE)*, Aug. 2018, pp. 1217–1223, iSSN: 2161-8070.
- [43] S. Patro, M. Potey, and A. Golhani, "Comparative study of middleware solutions for control and monitoring systems," in *2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT)*, Feb. 2017, pp. 1–10, iSSN: null.
- [44] H. Subramoni *et al.*, "Design and evaluation of benchmarks for financial applications using Advanced Message Queuing Protocol (AMQP) over InfiniBand," in *2008 Workshop on High Performance Computational Finance*, Nov. 2008, pp. 1–8.
- [45] S. Vinoski, "Advanced Message Queuing Protocol," *IEEE Internet Computing*, vol. 10, no. 6, pp. 87–89, Nov. 2006, conference Name: IEEE Internet Computing.
- [46] G. Marsh *et al.*, "Scaling advanced message queuing protocol (amqp) architecture with broker federation and infiniband," *Ohio State University, Tech. Rep. OSU-CISRC-5/09-TR17*, p. 38, 2008.
- [47] Z. Wang *et al.*, "Kafka and Its Using in High-throughput and Reliable Message Distribution," in *2015 8th International Conference on Intelligent Networks and Intelligent Systems (ICINIS)*, Nov. 2015, pp. 117–120, iSSN: null.
- [48] P. Hintjens, *ZeroMQ: Messaging for Many Applications*. "O'Reilly Media, Inc.", Mar. 2013, google-Books-ID: KWtT5CJc6FYC.
- [49] J. Moore *et al.*, "Zest: REST over ZeroMQ," in *2019 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, Mar. 2019, pp. 1015–1019, iSSN: null.
- [50] N. Estrada and H. Astudillo, "Comparing scalability of message queue system: ZeroMQ vs RabbitMQ," in *2015 Latin American Computing Conference (CLEI)*, Oct. 2015, pp. 1–6, iSSN: null.
- [51] Y. Yang, L. Zhang, and X. Wang, "Time Delay Performance Analysis of Distributed Communication Platform Based on ZeroMQ," in *2019 International Conference on*

- Communications, Information System and Computer Engineering (CISCE)*, Jul. 2019, pp. 319–323, iSSN: null.
- [52] R. T. Evans, J. C. Browne, and W. L. Barth, “Understanding application and system performance through system-wide monitoring,” in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2016, pp. 1702–1710.
 - [53] “IFT6315 - Analyse et compréhension de programmes.” [Online]. Available: <http://www-labs.iro.umontreal.ca/~dufour/cours/ift6315/docs/2-dynamique.pdf>
 - [54] B. Gregg, “Velocity 2015 linux perf tools.” [Online]. Available: <https://www.slideshare.net/brendangregg/velocity-2015-linux-perf-tools>
 - [55] “Choosing a Linux Tracer (2015).” [Online]. Available: <https://www.brendangregg.com/blog/2015-07-08/choosing-a-linux-tracer.html>
 - [56] “Container Performance Analysis at DockerCon 2017.” [Online]. Available: <https://www.brendangregg.com/blog/2017-05-15/container-performance-analysis-dockercon-2017.html>
 - [57] J. Evans, “Linux tracing systems & how they fit together.” [Online]. Available: <https://jvns.ca/blog/2017/07/05/linux-tracing-systems/>
 - [58] Steven Rostedt, “ftrace - Function Tracer.” [Online]. Available: <https://www.kernel.org/doc/Documentation/trace/ftrace.txt>
 - [59] —, “Secrets of the Ftrace function tracer [LWN.net],” Jan. 2010. [Online]. Available: <https://lwn.net/Articles/370423/>
 - [60] T. Bird, “Measuring function duration with ftrace,” in *Proceedings of the Linux Symposium*. Citeseer, 2009, pp. 47–54.
 - [61] M. Gebai and M. R. Dagenais, “Survey and Analysis of Kernel and Userspace Tracers on Linux: Design, Implementation, and Overhead,” *ACM Computing Surveys*, vol. 51, no. 2, pp. 26:1–26:33, Mar. 2018. [Online]. Available: <https://doi.org/10.1145/3158644>
 - [62] S. McCanne and V. Jacobson, “The BSD Packet Filter: A New Architecture for User-level Packet Capture.” in *USENIX winter*, vol. 46, 1993.
 - [63] J. Evans, “Notes on BPF & eBPF.” [Online]. Available: <https://jvns.ca/blog/2017/06/28/notes-on-bpf---ebpf/>

- [64] M. Desnoyers and M. R. Dagenais, “The ltng tracer: A low impact performance and behavior monitor for gnu/linux,” in *OLS (Ottawa Linux Symposium)*, vol. 2006. Citeseer, 2006, pp. 209–224.
- [65] D. Toupin, “Using Tracing to Diagnose or Monitor Systems,” *IEEE Software; Los Alamitos*, vol. 28, no. 1, pp. 87–91, Feb. 2011, num Pages: 87-91 Place: Los Alamitos, United States, Los Alamitos Publisher: IEEE Computer Society. [Online]. Available: <https://search.proquest.com/docview/818397878/abstract/8F403357CB184AF1PQ/1>
- [66] T. Bertauld and M. R. Dagenais, “Low-level trace correlation on heterogeneous embedded systems,” *EURASIP Journal on Embedded Systems*, vol. 2017, no. 1, p. 18, Jan. 2017. [Online]. Available: <https://doi.org/10.1186/s13639-016-0067-1>
- [67] “Jaeger: open source, end-to-end distributed tracing,” library Catalog: www.jaegertracing.io. [Online]. Available: <https://www.jaegertracing.io/>
- [68] “The OpenTracing project,” library Catalog: opentracing.io. [Online]. Available: <https://opentracing.io/>
- [69] B. H. Sigelman *et al.*, “Dapper, a Large-Scale Distributed Systems Tracing Infrastructure,” Google, Inc., Tech. Rep., 2010. [Online]. Available: <https://research.google.com/archive/papers/dapper-2010-1.pdf>
- [70] “OpenZipkin · A distributed tracing system.” [Online]. Available: <https://zipkin.io/>
- [71] J. Mace, “End-to-End Tracing: Adoption and Use Cases,” *Survey, Brown University*, 2017.
- [72] J. Moc and D. A. Carr, “Understanding distributed systems via execution trace data,” in *Proceedings 9th International Workshop on Program Comprehension. IWPC 2001*. IEEE, 2001, pp. 60–67.
- [73] J. Moe and D. A. Carr, “Using execution trace data to improve distributed systems,” *Software: Practice and Experience*, vol. 32, no. 9, pp. 889–906, 2002, publisher: Wiley Online Library.
- [74] F. Ehm and A. Dworak, “A remote tracing facility for distributed systems,” in *Conf. Proc.*, vol. 111010, 2011, p. WEMAU001, issue: CERN-ATS-2011-200.
- [75] Erlingsson *et al.*, “Fay: extensible distributed tracing from kernels to clusters,” *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 4, pp. 1–35, 2012, publisher: ACM New York, NY, USA.

- [76] R. Fonseca *et al.*, “X-Trace: A Pervasive Network Tracing Framework,” 2007. [Online]. Available: <https://www.usenix.org/conference/nsdi-07/x-trace-pervasive-network-tracing-framework>
- [77] A. Chanda, A. L. Cox, and W. Zwaenepoel, “Whodunit: transactional profiling for multi-tier applications,” in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys ’07. Lisbon, Portugal: Association for Computing Machinery, Mar. 2007, pp. 17–30. [Online]. Available: <https://doi.org/10.1145/1272996.1273001>
- [78] R. R. Sambasivan *et al.*, “Diagnosing Performance Changes by Comparing Request Flows.” in *NSDI*, vol. 5, 2011, pp. 1–1.
- [79] R. Fonseca, M. J. Freedman, and G. Porter, “Experiences with Tracing Causality in Networked Services.” *INM/WREN*, vol. 10, no. 10, 2010.
- [80] P. Barham *et al.*, “Using Magpie for request extraction and workload modelling.” in *OSDI*, vol. 4, 2004, pp. 18–18.
- [81] J. Zhou *et al.*, “MTracer: A trace-oriented monitoring framework for medium-scale distributed systems,” in *2014 IEEE 8th International Symposium on Service Oriented System Engineering*. IEEE, 2014, pp. 266–271.
- [82] J. Mace, R. Roelke, and R. Fonseca, “Pivot tracing: Dynamic causal monitoring for distributed systems,” *ACM Transactions on Computer Systems (TOCS)*, vol. 35, no. 4, pp. 1–28, 2018, publisher: ACM New York, NY, USA.
- [83] J. Kaldor *et al.*, “Canopy: An end-to-end performance tracing and analysis system,” in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 34–50.
- [84] P. Avgustinov *et al.*, “Efficient trace monitoring,” in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, ser. OOPSLA ’06. Portland, Oregon, USA: Association for Computing Machinery, Oct. 2006, pp. 685–686. [Online]. Available: <https://doi.org/10.1145/1176617.1176673>
- [85] P. Voccio and M. C. Dietz, “Methods and systems of distributed tracing,” Patent, Mar., 2018, library Catalog: Google Scholar Publisher: Google Patents.
- [86] F. Reumont-Locke, “Méthodes efficaces de parallélisation de l’analyse de traces noyau,” Master’s thesis, École Polytechnique de Montréal, Aug. 2015. [Online]. Available: <https://publications.polymtl.ca/1899/>

- [87] M. Martin, “Analyse détaillée de trace en dépit d’événements manquants,” Master’s thesis, École Polytechnique de Montréal, Aug. 2018. [Online]. Available: <https://publications.polymtl.ca/3248/>
- [88] “Using KernelShark to analyze the real-time scheduler [LWN.net].” [Online]. Available: <https://lwn.net/Articles/425583/>
- [89] A. Knüpfer *et al.*, “The vampir performance analysis tool-set,” in *Tools for high performance computing*. Springer, 2008, pp. 139–155.
- [90] W. E. Nagel *et al.*, “VAMPIR: Visualization and analysis of MPI resources,” 1996, publisher: Citeseer.
- [91] V. Pillet *et al.*, “Paraver: A tool to visualize and analyze parallel code,” in *Proceedings of WoTUG-18: transputer and occam developments*, vol. 44. Citeseer, 1995, pp. 17–31, issue: 1.
- [92] L. Adhianto *et al.*, “HPCTOOLKIT: tools for performance analysis of optimized parallel programs,” *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010, _eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.1553>. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1553>
- [93] E. Betke and J. Kunkel, “Real-Time I/O-Monitoring of HPC Applications with SIOX, Elasticsearch, Grafana and FUSE,” in *High Performance Computing*, ser. Lecture Notes in Computer Science, J. M. Kunkel *et al.*, Eds. Cham: Springer International Publishing, 2017, pp. 174–186.
- [94] C.-Q. Yang and B. Miller, “Critical path analysis for the execution of parallel and distributed programs,” in *[1988] Proceedings. The 8th International Conference on Distributed*, Jun. 1988, pp. 366–373, aRT2.
- [95] R. J. Willis, “Critical path analysis and resource constrained project scheduling — Theory and practice,” *European Journal of Operational Research*, vol. 21, no. 2, pp. 149–155, Aug. 1985, aRT2.
- [96] Y.-L. Chen, D. Rinks, and K. Tang, “Critical path in an activity network with time constraints,” *European Journal of Operational Research*, vol. 100, no. 1, pp. 122–133, Jul. 1997, aRT2.
- [97] S. Chanas and P. Zieliński, “Critical path analysis in the network with fuzzy activity times,” *Fuzzy Sets and Systems*, vol. 122, no. 2, pp. 195–204, Sep. 2001, aRT2.

- [98] P. Barford and M. Crovella, “Critical path analysis of TCP transactions,” *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 2 supplement, pp. 80–102, Apr. 2001, aRT2.
- [99] S. Nasution, “Fuzzy critical path method,” *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 24, no. 1, pp. 48–57, Jan. 1994, aRT2.
- [100] A. Montplaisir *et al.*, “Efficient Model to Query and Visualize the System States Extracted from Trace Data,” in *Runtime Verification*, ser. Lecture Notes in Computer Science, A. Legay and S. Bensalem, Eds. Berlin, Heidelberg: Springer, 2013, pp. 219–234, aRT2.
- [101] D. M. Tullsen and B. Calder, “Computing along the critical path,” University of California, Tech. Rep., 1998.
- [102] T. Ball and J. R. Larus, “Efficient path profiling,” in *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, ser. MICRO 29. USA: IEEE Computer Society, Dec. 1996, pp. 46–57, aRT2.
- [103] N. Jasika *et al.*, “Dijkstra’s shortest path algorithm serial and parallel execution performance analysis,” in *2012 Proceedings of the 35th International Convention MIPRO*, May 2012, pp. 1811–1815, aRT2.
- [104] F. Giraldeau and M. Dagenais, “Wait Analysis of Distributed Systems Using Kernel Tracing,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 8, pp. 2450–2461, Aug. 2016, conference Name: IEEE Transactions on Parallel and Distributed Systems.
- [105] L. Gelle, N. Ezzati-Jivan, and M. R. Dagenais, “Combining Distributed and Kernel Tracing for Performance Analysis of Cloud Applications,” *Electronics*, vol. 10, no. 21, p. 2610, Jan. 2021, number: 21 Publisher: Multidisciplinary Digital Publishing Institute. [Online]. Available: <https://www.mdpi.com/2079-9292/10/21/2610>
- [106] H. Nemati *et al.*, “Critical Path Analysis through Hierarchical Distributed Virtualized Environments using Host Kernel Tracing,” *IEEE Transactions on Cloud Computing*, pp. 1–1, 2019, aRT2.
- [107] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” in *Concurrency: the Works of Leslie Lamport*. New York, NY, USA: Association for Computing Machinery, Oct. 2019, pp. 179–196, aRT2.

- [108] B. Poirier, R. Roy, and M. Dagenais, “Accurate offline synchronization of distributed traces using kernel-level events,” *ACM SIGOPS Operating Systems Review*, vol. 44, no. 3, pp. 75–87, Aug. 2010, aRT2.
- [109] C. Wu *et al.*, “From Trace Generation to Visualization: A Performance Framework for Distributed Parallel Systems,” in *SC '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, Nov. 2000, pp. 50–50, aRT2.
- [110] Q. Fournier *et al.*, “Automatic Cause Detection of Performance Problems in Web Applications,” in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Oct. 2019, pp. 398–405.
- [111] M. Schulz, “Extracting Critical Path Graphs from MPI Applications,” in *2005 IEEE International Conference on Cluster Computing*, Sep. 2005, pp. 1–10, aRT2.
- [112] “aquasec-history-docker.” [Online]. Available: <https://blog.aquasec.com/a-brief-history-of-containers-from-1970s-chroot-to-docker-2016>
- [113] “LISA2017.” [Online]. Available: http://www.brendangregg.com/Slides/LISA2017_Container_Performance_Analysis.pdf
- [114] M. Großmann and C. Klug, “Monitoring container services at the network edge,” in *2017 29th International Teletraffic Congress (ITC 29)*, vol. 1, 2017, pp. 130–133.
- [115] S. Zeng and Q. Hao, “Network i/o path analysis in the kernel-based virtual machine environment through tracing,” in *2009 First International Conference on Information Science and Engineering*, 2009, pp. 2658–2661.
- [116] R. Guha Anjoy and S. K. Chakraborty, “Efficiency of lttng as a kernel and userspace tracer on multicore environment,” 2010.
- [117] T. Bertauld and M. R. Dagenais, “Low-level trace correlation on heterogeneous embedded systems,” *EURASIP Journal on Embedded Systems*, vol. 2017, no. 1, p. 18, Jan. 2017.
- [118] D. Toupin, “Using Tracing to Diagnose or Monitor Systems,” *IEEE Software; Los Alamitos*, vol. 28, no. 1, pp. 87–91, Feb. 2011, num Pages: 87-91 Place: Los Alamitos, United States, Los Alamitos Publisher: IEEE Computer Society.
- [119] M. Desnoyers and M. R. Dagenais, “The lttng tracer: A low impact performance and behavior monitor for gnu/linux,” in *OLS (Ottawa Linux Symposium)*, vol. 2006. Citeseer, 2006, pp. 209–224.

- [120] M. Desnoyers, “Low-Impact Operating System Tracing,” phd, École Polytechnique de Montréal, Dec. 2009, aRT2.
- [121] L. Prieur-Drevon, “Structures de données hautement extensibles pour le stockage sur disque de séries temporelles hétérogènes,” M.A.Sc., Ecole Polytechnique, Montreal (Canada), Canada, 2021, aRT2.
- [122] G. Graefe and H. Kuno, “Modern B-tree techniques,” in *2011 IEEE 27th International Conference on Data Engineering*, Apr. 2011, pp. 1370–1373, iSSN: 2375-026X.
- [123] M. K. Aguilera, W. Golab, and M. A. Shah, “A practical scalable distributed B-tree,” *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 598–609, Aug. 2008.
- [124] A. Chan, W. Gropp, and E. Lusk, “An efficient format for nearly constant-time access to arbitrary time intervals in large trace files,” *Scientific Programming*, vol. 16, no. 2-3, pp. 155–165, Jan. 2008, aRT2.
- [125] M. Noeth *et al.*, “Scalable Compression and Replay of Communication Traces in Massively Parallel Environments,” in *2007 IEEE International Parallel and Distributed Processing Symposium*, Mar. 2007, pp. 1–11, iSSN: 1530-2075.
- [126] L. Qilin and Z. Mintian, “The State of the Art in Middleware,” in *2010 International Forum on Information Technology and Applications*, vol. 1, Jul. 2010, pp. 83–85, iSSN: null.
- [127] D. Menasce, “MOM vs. RPC: communication models for distributed applications,” *IEEE Internet Computing*, vol. 9, no. 2, pp. 90–93, Mar. 2005, conference Name: IEEE Internet Computing.
- [128] S. Vinoski, “Where is is middleware,” *IEEE Internet Computing*, vol. 6, no. 2, pp. 83–85, Mar. 2002, conference Name: IEEE Internet Computing.
- [129] Y. Chen Kuang Piao, “Nouvelle Architecture pour les Environnements de Développement Intégré et Traçage de Logiciel,” M.A.Sc., Ecole Polytechnique, Montreal (Canada), Canada, 2022.
- [130] R. Rodriguez-Echeverria *et al.*, “Towards a Language Server Protocol Infrastructure for Graphical Modeling,” in *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS ’18. New York, NY, USA: Association for Computing Machinery, Oct. 2018, pp. 370–380.

- [131] Y. Chen Kuang Piao, N. Ezzati-jivan, and M. R. Dagenais, “Distributed Architecture for an Integrated Development Environment, Large Trace Analysis, and Visualization,” *Sensors*, vol. 21, no. 16, p. 5560, Jan. 2021, aRT2.
- [132] P. Las-Casas *et al.*, “Sifter: Scalable sampling for distributed traces, without feature engineering,” in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 312–324. [Online]. Available: <https://doi.org/10.1145/3357223.3362736>
- [133] S. Mallanna and M. Devika, “Distributed request tracing using zipkin and spring boot sleuth,” *Int. J. Comput. Appl.*, vol. 975, p. 8887, 2020.
- [134] A. Tobey and S. Spees, “Tracing bare metal with OpenTelemetry.” San Francisco, CA: USENIX Association, Mar. 2022.
- [135] P.-F. Denys, M. R. Dagenais, and M. Pepin, “Advanced tracing methods for container messaging systems analysis,” 2021.
- [136] F. Q. Denys, Pierre-Frédéric and M. R. Dagenais, “Distributed computation of the critical path from execution traces,” 2022.
- [137] A. Montplaisir *et al.*, “Efficient Model to Query and Visualize the System States Extracted from Trace Data,” in *Runtime Verification*, ser. Lecture Notes in Computer Science, A. Legay and S. Bensalem, Eds. Berlin, Heidelberg: Springer, 2013, pp. 219–234, aRT2.
- [138] M. Rezazadeh *et al.*, “Performance evaluation of complex multi-thread applications through execution path analysis,” *Performance Evaluation*, vol. 155-156, p. 102289, Jun. 2022. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0166531622000049>
- [139] N. Ezzati-Jivan, H. Daoud, and M. R. Dagenais, “Debugging of Performance Degradation in Distributed Requests Handling Using Multilevel Trace Analysis,” *Wireless Communications and Mobile Computing*, vol. 2021, p. e8478076, Nov. 2021, publisher: Hindawi. [Online]. Available: <https://www.hindawi.com/journals/wcmc/2021/8478076/>
- [140] L. Cheng *et al.*, “Network-aware locality scheduling for distributed data operators in data centers,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 6, pp. 1494–1510, 2021.

- [141] Y. Chen Kuang Piao, N. Ezzati-jivan, and M. R. Dagenais, “Distributed architecture for an integrated development environment, large trace analysis, and visualization,” *Sensors*, vol. 21, no. 16, 2021. [Online]. Available: <https://www.mdpi.com/1424-8220/21/16/5560>
- [142] J. Smith and P. Schirling, “Metadata standards roundup,” *IEEE MultiMedia*, vol. 13, no. 2, pp. 84–88, Apr. 2006, conference Name: IEEE MultiMedia.
- [143] A. Devulapalli and P. Wyckoff, “File Creation Strategies in a Distributed Metadata File System,” in *2007 IEEE International Parallel and Distributed Processing Symposium*, Mar. 2007, pp. 1–10, iSSN: 1530-2075.
- [144] P. F. Corbett and D. G. Feitelson, “The Vesta parallel file system,” *ACM Transactions on Computer Systems*, vol. 14, no. 3, pp. 225–264, Aug. 1996. [Online]. Available: <https://doi.org/10.1145/233557.233558>
- [145] N. Y. Song *et al.*, “Optimizing of metadata management in large-scale file systems,” *Cluster Computing*, vol. 21, no. 4, pp. 1865–1879, Dec. 2018. [Online]. Available: <https://doi.org/10.1007/s10586-018-2814-7>
- [146] S. Brandt *et al.*, “Efficient metadata management in large distributed storage systems,” in *20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies, 2003. (MSST 2003). Proceedings.*, Apr. 2003, pp. 290–298.
- [147] J. Li *et al.*, “SPEC_{hpc} 2021 Benchmark Suites for Modern HPC Systems,” in *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering*. Beijing China: ACM, Jul. 2022, pp. 15–16. [Online]. Available: <https://dl.acm.org/doi/10.1145/3491204.3527498>