



Titre: Traçage multi-niveaux d'orchestrateur d'application conteneurisée
Title:

Auteur: Benjamin Saint-Cyr
Author:

Date: 2023

Type: Mémoire ou thèse / Dissertation or Thesis

Référence: Saint-Cyr, B. (2023). Traçage multi-niveaux d'orchestrateur d'application conteneurisée [Mémoire de maîtrise, Polytechnique Montréal]. PolyPublie.
Citation: <https://publications.polymtl.ca/56735/>

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

URL de PolyPublie: <https://publications.polymtl.ca/56735/>
PolyPublie URL:

Directeurs de recherche: Michel Dagenais, & Heng Li
Advisors:

Programme: Génie informatique
Program:

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

Traçage multi-niveaux d'orchestrateur d'application conteneurisée

BENJAMIN SAINT-CYR

Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*

Génie informatique

Novembre 2023

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

Ce mémoire intitulé :

Traçage multi-niveaux d'orchestrateur d'application conteneurisée

présenté par **Benjamin SAINT-CYR**

en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*

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

Alejandro QUINTERO, président

Michel DAGENAIS, membre et directeur de recherche

Heng LI, membre et codirecteur de recherche

Foutse KHOMH, membre

DÉDICACE

À mes parents qui m'ont toujours supporté

REMERCIEMENTS

Merci à Mon directeur Michel Dagenais pour sa supervision.

RÉSUMÉ

Ce mémoire explore la complexité inhérente à la gestion des performances des applications dans un environnement Kubernetes, un orchestrateur de conteneurs largement adopté dans l'industrie. Dans ce contexte, l'unité la plus petite qui peut être orchestrée est désignée comme un Pod, qui regroupe un ensemble de conteneurs partageant la même adresse IP et système de fichiers. L'objectif est d'identifier et résoudre les problèmes de performance au démarrage et à l'exécution des Pods. Notre travail met en lumière les interactions entre la couche d'orchestration (Kubernetes) et la couche du système d'exploitation (noyau Linux).

Nous avons formulé l'hypothèse suivante : La visualisation des cycles de vie des déploiements et de leur consommation de ressources, provenant des traces du noyau et de l'orchestrateur, améliore considérablement l'observabilité du système. Cette stratégie intégrée vise à diagnostiquer précisément et résoudre efficacement les problèmes de performance rencontrés.

Les méthodes expérimentales employées comprennent l'instrumentation du noyau Linux et de Kubernetes, la collecte et l'analyse des traces, ainsi que la création de visualisations pour interpréter les données collectées. Ces méthodes ont permis de mettre en lumière les étapes critiques du cycle de vie des Pods qui contribuent aux retards dans le démarrage et l'exécution des Pods.

L'analyse des résultats a révélé l'efficacité de notre approche pour identifier les phases qui ralentissent le déploiement des Pods. Plusieurs visualisations ont été créées pour illustrer l'utilisation des ressources, l'impact de la limitation du CPU et la contention des ressources sur les performances des Pods.

Les principales conclusions de cette recherche suggèrent qu'une approche intégrée pour observer Kubernetes du point de vue du noyau offre une perspective plus holistique des performances du système. Elle permet non seulement d'identifier les goulots d'étranglement, mais aussi de fournir des directives pour optimiser les déploiements de manière plus efficace.

Ce mémoire se penche sur l'essence et la structure des conteneurs et des orchestrateurs, en explorant diverses méthodes pour extraire des métriques et des traces dans le contexte orchestral. Il met en avant les diverses visualisations et analyses qui peuvent être réalisées à partir de ces données extraites. Ensuite, nous présenterons notre article "Kube Compass : fine-grained observability for orchestrators and container runtime executions". Finalement, nous discuterons des résultats et de l'impact de notre approche.

ABSTRACT

The thesis delves into the inherent complexity of managing application performance within a Kubernetes environment, a widely adopted container orchestrator in the industry. In this context, the smallest unit that can be orchestrated is referred to as a Pod, which encompasses a set of containers sharing the same IP address and file system. The primary aim is to identify and address performance issues that may arise during Pod startup and during their runtime, shedding light on the interactions between the orchestration layer (Kubernetes) and the operating system layer (Linux kernel).

We have formulated the following hypothesis: Visualizing the life cycles of deployments and their resource consumption, derived from the traces of both the kernel and the orchestrator, significantly enhances system observability. This integrated strategy aims to diagnose accurately and resolve effectively the performance issues encountered.

The experimental methods employed include the instrumentation of the Linux kernel and Kubernetes, the collection and analysis of traces, as well as the creation of visualizations to interpret the collected data. These methods helped to highlight the critical stages of the Pod lifecycle that contribute to delays in Pod startup and execution.

The analysis of the results revealed that our approach can effectively identify the phases that slow down Pod deployment, and provides precious insights into the causes of such delays, be it CPU throttling, resource contention, or lock contention. Several visualizations were created to illustrate resource usage, the impact of CPU throttling, and resource contention on Pod performance.

The main conclusions of this research suggest that an integrated approach to observe Kubernetes from the kernel’s perspective offers a more holistic view of the system’s performance. It not only helps in identifying bottlenecks but also provides guidelines for optimizing deployments more effectively.

This thesis investigates the essence and structure of containers and orchestrators, exploring various methods to extract metrics and traces in the orchestral context. It highlights the diverse visualizations and analyses that can be carried out from these extracted data. Subsequently, we will introduce our article “Kube Compass: fine-grained observability for orchestrators and container runtime executions”. Finally, we will discuss the results and impact of our approach.

TABLE DES MATIÈRES

DÉDICACE	iii
REMERCIEMENTS	iv
RÉSUMÉ	v
ABSTRACT	vi
TABLE DES MATIÈRES	vii
LISTE DES TABLEAUX	x
LISTE DES FIGURES	xi
LISTE DES SIGLES ET ABRÉVIATIONS	xii
CHAPITRE 1 INTRODUCTION	1
1.1 Définitions et concepts de base	1
1.1.1 Conteneurs	1
1.1.2 Orchestrateurs	2
1.1.3 Traçage logiciel	2
1.2 Éléments de la problématique	4
1.3 Objectifs de recherche	4
1.4 Plan du mémoire	5
CHAPITRE 2 REVUE DE LITTÉRATURE	6
2.1 Conteneurs	6
2.1.1 Namespaces	7
2.1.2 Cgroups	8
2.1.3 LXC	9
2.1.4 Docker	10
2.2 Orchestrateurs	12
2.2.1 BORG	12
2.2.2 Kubernetes	14
2.2.3 Docker Swarm	17
2.2.4 Apache Mesos	17

2.3	Source d'information pour l'observabilité	19
2.3.1	Métriques	19
2.3.2	Traçage distribué	21
2.3.3	Traçage Noyau	24
2.4	Analyse d'information	30
2.4.1	Analyse des journaux	30
2.4.2	Analyse des métriques	31
2.4.3	Analyse des traces	33
2.5	Conclusion de la revue critique de la littérature	37
CHAPITRE 3 MÉTHODOLOGIE		39
3.1	Environnement utilisé	39
3.1.1	Environnement distribué	39
3.2	Outils développés	40
3.2.1	Bibliothèque d'Instrumentation pour Kubernetes	40
3.2.2	Configuration des Environnements de Test	40
3.2.3	Visualisations dans Trace Compass	40
3.3	Conclusion	40
CHAPITRE 4 ARTICLE 1 : KUBE COMPASS : FINE-GRAINED OBSERVABILITY FOR ORCHESTRATORS AND CONTAINER RUNTIME EXECUTIONS		42
4.1	Introduction	42
4.2	Related Work	44
4.2.1	Containers	44
4.2.2	Containerized Orchestrators	44
4.2.3	Google Cluster-Usage Traces	45
4.2.4	cAdvisor	45
4.2.5	eBPF	46
4.2.6	Open Telemetry	47
4.2.7	Tools Developed by SIG-Instrumentation	47
4.2.8	Other tools	48
4.2.9	Conclusion	48
4.3	Background	48
4.4	Methodology	49
4.4.1	Instrumenting the Orchestrator	50
4.4.2	Collecting Trace Data per Cgroup	53
4.4.3	Trace Compass Analysis	57

4.5	Implementation and Results	57
4.5.1	Environment and Data Collection	57
4.5.2	Instrumentation added	58
4.5.3	Lifecycle Analysis	58
4.5.4	Cgroup CPU Usage Analysis	59
4.5.5	Use Cases	61
4.6	Discussion and Future Work	65
4.7	Conclusion	65
CHAPITRE 5 DISCUSSION GÉNÉRALE		67
CHAPITRE 6 CONCLUSION		69
6.1	Synthèse des travaux	69
6.2	Limitations de la solution proposée	69
6.3	Améliorations futures	70
RÉFÉRENCES		72

LISTE DES TABLEAUX

Tableau 2.1	Liste des <i>namespaces</i> supportés par Linux	8
Tableau 2.2	Liste des <i>cgroups</i> v2	9
Table 4.1	Summary of tracepoints inserted for data collection.	52

LISTE DES FIGURES

Figure 2.1	L'Architecture de Docker	11
Figure 2.2	L'Architecture de Borg	13
Figure 2.3	L'Architecutre de Kubernetes	14
Figure 2.4	Architecture de Open Telemetry (OTel)	23
Figure 2.5	LTtNg Processus d'enregistrement	26
Figure 2.6	Architecture d'eBPF	27
Figure 2.7	Tableau de Bord Grafana. Source : [1] distribuée sous licence CC BY 2.0 Deed	32
Figure 2.8	Exemple d'une trace lue par babeltrace2	34
Figure 2.9	La vue <i>Control Flow</i> permet de visualiser la création des processus. Dans ce cas, nous pouvons voir la création d'un déploiement de <i>nginx</i> fait par <i>contaienrd</i> et <i>runc</i>	35
Figure 2.10	La vue <i>Critical Path Flow View</i> permet de visualiser en détail quels processus ont bloqué l'exécution d'un programme. <i>Aggregated Critical View</i> montre à un haut niveau qu'elles sont les processus ou ressources qui ont bloqué le processus. Dans ce que nous pouvons voir dans la figure, c'est le processus de <i>Containerd</i>	36
Figure 4.1	Tracing Architecture	50
Figure 4.2	The Pod Startup Process of Kubernetes	51
Figure 4.3	Example of Latency Introduced by CPU Limit	55
Figure 4.4	Example of Latency Due to Parallelization	56
Figure 4.5	CPU usage per Cgroup graph	59
Figure 4.6	Lifecycle Analysis	62
Figure 4.7	nginx server being throttled	63
Figure 4.8	Slow Pod Startup Caused by I/O Contention	64

LISTE DES SIGLES ET ABRÉVIATIONS

ABI	Application binary interface
API	Application Programming Interface
AWS	Amazon Web Services
BCC	BPF Compiler Collection
BPF	Berkeley Packet Filter
CFS	Completely Fair Scheduler
CNCF	Cloud Native Computing Foundation
CPU	Central processing unit
CRI	Container Runtime Interface
CTF	Common Trace Format
DNS	Domain Name System
DORSAL	Distributed Open Reliable Systems Analysis Lab
ELK	Elasticsearch, LogStash, Kibana
HTTP	Hypertext Transfer Protocol
IO	Input / Output
IP	Internet Protocol
IPC	Inter-process communication
NUMA	Non-uniform memory access
OCI	open container initiative
OS	operating system
RDMA	Remote Direct Memory Access
SDK	software development kit
SIG	special interest groups
SRE	Site reliability engineering
SSD	Solid State Drive
STDIN	Standard input
VM	Virtual Machine

CHAPITRE 1 INTRODUCTION

1.1 Définitions et concepts de base

L’infonuagique (ou “Cloud Computing”) offre la possibilité pour divers fournisseurs de mettre à disposition un parc informatique de grande envergure au grand public. Ceci offre aux entreprises, petites ou grandes, la chance de déployer leurs applications sur des centres de données hautement performants. Elle offre aux startups en croissance l’accès à des ressources informatiques d’une ampleur qui aurait été inimaginable auparavant, permettant une flexibilité et une adaptation dynamique aux besoins évolutifs. Par exemple, en cas d’augmentation de la demande, les entreprises peuvent louer des serveurs supplémentaires. Inversement, elles peuvent réduire l’échelle de leur architecture et restituer ces ressources au bassin commun, lorsque la demande diminue. Cette mise à l’échelle élastique permet aux entreprises de payer uniquement pour les ressources qu’elles utilisent, optimisant ainsi leur efficacité opérationnelle et financière.

1.1.1 Conteneurs

Au cœur de cette évolution vers une informatique plus flexible et plus efficiente se trouvent les conteneurs, qui ont radicalement transformé le déploiement et la gestion des applications dans les environnements infonuagiques.

Les conteneurs représentent une technique de virtualisation légère au niveau du système d’exploitation, qui repose sur la réutilisation des mécanismes du noyau Linux. Ils réutilisent les mêmes ordonnanceur et système de fichiers, tout en isolant l’utilisation des ressources grâce aux cgroups, et en limitant la visibilité du système grâce aux namespaces. Cette approche permet de réduire le surcoût d’exécution et le temps de démarrage, par rapport à la virtualisation traditionnelle basée sur des hyperviseurs.

Les conteneurs offrent également la possibilité d’encapsuler une application et ses dépendances dans une image, garantissant ainsi la reproductibilité de l’environnement, que ce soit en développement, en test ou en production.

En somme, grâce à leurs avantages en termes de performance, d’isolation, et de reproductibilité, les conteneurs sont rapidement devenus le choix privilégié des développeurs et des opérateurs, entraînant ainsi une adoption massive dans l’industrie.

1.1.2 Orchestrateurs

Les orchestrateurs sont des systèmes conçus pour gérer et orchestrer des conteneurs dans un environnement infonuagique. Ils permettent de gérer un bassin de ressources, comme des serveurs ou des nœuds, et de les allouer dynamiquement aux applications en fonction de leurs besoins.

L’une des principales caractéristiques des orchestrateurs est la possibilité pour les utilisateurs de décrire de manière déclarative l’état désiré de leur architecture, sans se préoccuper des détails de mise en œuvre. L’orchestrateur prend en charge la tâche de déployer, de redimensionner et de garantir la santé des applications, conformément à l’état déclaré par l’utilisateur. Cette abstraction permet aux utilisateurs de se concentrer sur la définition de l’état désiré de leur système, tandis que l’orchestrateur s’occupe de la réalisation pratique, offrant ainsi une manière plus robuste, fiable et efficace de gérer des applications à grande échelle dans un environnement infonuagique.

Les orchestrateurs peuvent être exploités dans divers environnements, que ce soit sur des infrastructures auto hébergées (“self-hosted”) ou sur des plateformes d’infonuagique publique, ce qui lui permet d’augmenter la taille du bassin de ressources dynamiquement.

1.1.3 Traçage logiciel

Une trace est l’ensemble des événements discrets produits par un processus. Ces données permettent une meilleure compréhension des incidents qui se sont produits au sein d’une application.

Le traçage logiciel consiste en une série de techniques afin de collecter cette trace. Il y a plusieurs techniques et stratégies sur quels types d’événements collecter, comment reconstruire ces événements, comment extraire le plus de contexte autour de ces événements, ou comment collecter un grand volume d’événements. Ce qui est difficile à atteindre est le traçage noyau. Énormément d’événements très rapides se produisent, ce qui génère une grande quantité d’information. Par contre, il permet une compréhension totale du système et permet de diagnostiquer des problèmes de performances très subtils.

Instrumentation

L’instrumentation est le processus d’insérer des instructions dans un programme, afin de collecter les événements.

Il y a deux principales approches afin de réaliser cette tâche. La première est l’instrumentation

statique. Les points de traces sont intégrés lors de la compilation, cette méthode est efficace en termes de performance, mais elle nécessite une planification préalable. Toute adaptation ultérieure exige une recompilation, ce qui peut être impraticable dans certains environnements. La deuxième approche est l'instrumentation dynamique. Elle permet d'ajuster les points de trace à la volée, pendant l'exécution du programme. Cependant, cette flexibilité s'accompagne d'un surcoût plus grand en ressources, pouvant impacter les performances du système.

Point de Trace

Le point de trace est une instruction insérée à un emplacement spécifique dans le programme, conçu pour envoyer des informations à un traceur, capturant ainsi l'état ou le comportement du programme à ce moment précis.

Événements

L'événement est un événement d'un certain type (nom) à un certain moment dans le temps. Il peut généralement contenir plus d'information et de contexte sur le moment et là où il a été exécuté. Il peut, par exemple, contenir dans quel processus, quel fil et quel cœur il a été exécuté, ou encore par quelle requête le processus est occupé.

Traceur

Le traceur est le logiciel qui s'occupe d'enregistrer les événements de trace.

1.2 Éléments de la problématique

Les orchestrateurs tels que Kubernetes offrent l'avantage significatif de permettre le déploiement déclaratif de programmes. Ceci facilite la gestion et la configuration des applications. Cependant, cette facilité d'utilisation peut également créer une couche d'abstraction qui masque des détails d'implémentation cruciaux. Ces détails, bien que souvent obscurs, peuvent avoir un impact significatif sur la performance des services déployés. Par exemple, la manière dont les ressources sont allouées et gérées, les stratégies de planification des tâches, ou encore la gestion de la mise en réseau sont autant d'éléments qui, bien que masqués, jouent un rôle déterminant dans la performance et la fiabilité des applications en production. Dans certains cas, cette opacité peut conduire à des inefficacités et à des difficultés lors du diagnostic et de la résolution des problèmes de performances [2] [3]. Des efforts continus sont donc nécessaires pour améliorer la transparence et la compréhension des mécanismes internes des orchestrateurs, permettant ainsi aux utilisateurs de mieux optimiser et ajuster leurs déploiements, en fonction des exigences spécifiques de performance et de fiabilité de leurs applications.

Dans le contexte de déploiements infonuagique, la performance est cruciale. Un démarrage rapide des services est essentiel pour répondre de manière élastique à des demandes fluctuantes et imprévisibles. De surcroît, une utilisation efficiente des technologies de conteneurisation et des stratégies de déploiement peut engendrer des économies significatives. Ceci est particulièrement pertinent à grande échelle où des inefficacités, même minimes, peuvent entraîner des coûts disproportionnés au fur et à mesure que les opérations se développent. Des démarrages de services rapides et une gestion optimisée des ressources sont donc des impératifs économiques et opérationnels dans les environnements infonuagiques contemporains.

Dans un environnement Kubernetes, il semble que les problèmes de performance puissent avoir plusieurs origines, notamment au niveau du noyau, sur lequel les conteneurs reposent, et au niveau global de la grappe de calcul que les orchestrateurs surveillent. Une stratégie pour aborder ces défis consiste à extraire des informations du noyau et à obtenir une vue détaillée de l'état interne des conteneurs et de l'orchestrateur. Une telle perspective offre des pistes pour naviguer à travers les défis de performance qui émergent à différents niveaux de l'architecture de Kubernetes.

1.3 Objectifs de recherche

L'objectif principal de cette recherche est de diagnostiquer et d'analyser les problèmes de performance, en particulier ceux liés à la latence, dans un système distribué de conteneurs orchestrés. L'approche adoptée consiste à collecter des traces d'exécution et à les analyser

pour identifier les causes des ralentissements et des inefficacités. Les objectifs spécifiques sont les suivants :

1. Instrumenter Kubernetes, un logiciel d'orchestration, avec LTTng, un logiciel de traçage.
2. Développer une visualisation dans Trace Compass pour diagnostiquer les problèmes de performance dans un système orchestré.

1.4 Plan du mémoire

La structure de ce mémoire est organisée de la manière suivante :

Le Chapitre 2 présente une revue de littérature, débutant avec un examen approfondi des conteneurs et des orchestrateurs de conteneurs, puis se poursuivant avec une exploration des données disponibles et des outils d'analyse correspondants.

Le Chapitre 3 décrit la méthodologie envisagée, détaillant les stratégies et les approches adoptées pour surmonter les défis mentionnés précédemment.

Le Chapitre 4 est consacré à l'article de journal intitulé “ Kube Compass : Fine-Grained Observability for Orchestrators and Container Runtime Executions ”, présentant un résumé et une analyse de cet article.

Le Chapitre 5 fournit une rétroaction critique sur l'article présenté dans le chapitre précédent, offrant une évaluation détaillée de ses mérites et de ses limitations.

Enfin, le Chapitre 6 conclut le mémoire, résumant les découvertes clés et esquissant les directions potentielles pour les recherches futures dans ce domaine.

CHAPITRE 2 REVUE DE LITTÉRATURE

Au cœur de la révolution de l’infonuagique, Kubernetes s’est solidement établi comme un élément central de la gestion des infrastructures distribuées. Son rôle prépondérant dans l’orchestration de conteneurs à grande échelle a été largement adopté par des entités allant des multinationales aux petites entreprises technologiques [4]. Bien que cette adoption massive témoigne de sa robustesse et de sa flexibilité, elle soulève également d’importants défis relatifs à l’efficacité et à la gestion des ressources.

En orchestrant d’immenses grappes de serveurs, et en gérant des milliers de conteneurs, Kubernetes peut engendrer des inefficacités coûteuses s’il est mal administré. Ces inefficacités se traduisent non seulement en matière de performance, mais aussi financièrement. Dans un contexte d’infonuagique où le paiement s’effectue à l’utilisation, de telles inefficacités peuvent avoir des conséquences financières substantielles.

Dans le paysage infonuagique actuel, la technologie des conteneurs a profondément transformé l’architecture des systèmes informatiques. Cette revue de littérature se propose de décomposer cette transformation en plusieurs axes d’analyse. Tout d’abord, nous explorerons les fondements même des conteneurs, de leurs mécanismes essentiels tels que les “namespaces” et “cgroups”, à des implémentations spécifiques comme Docker. Ensuite, l’accent sera mis sur les orchestrateurs, avec une attention particulière portée à Kubernetes, tout en offrant un regard sur ses prédécesseurs et concurrents. L’observabilité, essentielle pour garantir la performance, sera abordée par la suite, en détaillant les sources d’information et les méthodes d’analyse associées. Ces éléments, pris ensemble, offrent un éclairage sur l’importance et la complexité croissante de l’orchestration dans l’ère moderne.

2.1 Conteneurs

Les conteneurs partitionnent le système d’exploitation pour fournir aux applications un environnement d’exécution dédié, offrant une alternative plus légère et rapide par rapport aux machines virtuelles. Au lieu de recourir à un hyperviseur pour émuler le matériel, le système hôte partage son noyau avec le conteneur en utilisant des techniques de virtualisation légère, assurant ainsi une mise en marche accélérée [5]. Toutefois, il est crucial de reconnaître que la virtualisation légère, bien que bénéfique, comporte ses propres défis. Le niveau d’abstraction introduit par les conteneurs peut, dans certaines situations, masquer les détails de performance sous-jacents [6]. De plus, les mécanismes par lesquels les limites sont appliquées

peuvent ne pas toujours être parfaits, ce qui peut conduire à des incohérences ou des problèmes de performance imprévus [3]. La nécessité d'utiliser une version spécifique du noyau, ou la potentialité d'une contention accrue sur le noyau, sont également des considérations importantes à garder à l'esprit [7, Chapter 11].

L'idée des conteneurs a commencé en 1979 avec la commande *chroot* [7, Chapter 11]. Cette dernière modifie la racine du répertoire de fichiers d'un processus, restreignant ainsi sa vue du système et l'empêchant d'accéder aux fichiers des autres processus. Toutefois, bien que le système de fichiers soit isolé, le processus avait toujours accès à l'ensemble des autres ressources du système.

Face à cette limitation, FreeBSD a introduit en 1998 le concept de *Jails*. Ce mécanisme vise à offrir une isolation plus poussée des ressources du système d'exploitation pour les processus contenus à l'intérieur de la “prison” (ou “jail” en anglais). Ainsi, chaque “jail” peut se voir attribuer un hostname, une adresse IP et une vue distincte du système de fichiers, offrant une virtualisation légère des ressources [8].

De son côté, Linux a établi deux composants pour faire la gestion de ses conteneurs, soit les *namespaces* en 2002 et les *cgroups* en 2008 [7, Chapter 11]. Les *namespaces* limitent la vue des conteneurs et les *control groups* (*cgroups*) limitent l'utilisation des ressources. Il y a aussi *seccomp-bpf* pour restreindre les appels système des conteneurs [7, Chapter 11] [9].

Il est fondamental de préciser que, du point de vue du noyau, les conteneurs ne sont pas considérés comme des entités distinctes ou spéciales. En réalité, le noyau les traite comme tout autre processus. Cependant, ces processus “conteneurisés” exploitent des mécanismes spécifiques du noyau, tels que les *cgroups* et les *namespaces*, conçus précisément pour offrir les capacités d'isolation et de gestion des ressources nécessaires à la notion de conteneurisation. Par conséquent, pour bien comprendre les conteneurs, il est impératif d'approfondir notre connaissance de ces mécanismes sous-jacents.

2.1.1 Namespaces

L'idée des espaces de noms (*namespaces*) trouve ses racines dans le successeur d'Unix, Plan 9, qui visait à développer un système d'exploitation distribué. L'un des objectifs clés était d'utiliser des espaces de noms pour isoler les processus et fournir une interface uniforme pour l'interaction avec diverses ressources [10]. Inspiré par ce concept, Linux a introduit des *namespaces* initialement pour les systèmes de fichiers. En 2006, cette fonctionnalité a été étendue pour supporter différents types de ressources [11]. Le but était principalement de créer un mécanisme pour restaurer les programmes dans la zone utilisateur, lorsqu'ils étaient

migrés dans un autre nœud, dans une grappe de serveurs. Cependant, cette solution était assez générale pour être appliquée pour créer un mécanisme d’isolation des processus, comme les *Jails* de *FreeBSD*.

En 2023, il existe 8 types de *namespaces*, afin d’avoir plus de granularité dans l’isolation des ressources [12].

TABLEAU 2.1 Liste des *namespaces* supportés par Linux

Cgroup	Description
Cgroup	Isole le fichier racine des <i>cgroups</i>
IPC	Pour la visibilité des communications interprocessus
Network	Pour les réseaux
Mount	Pour les points d’attachement de fichiers
PID	Pour les identifiants de processus
Time	Pour séparer les horloges
User	Pour isoler les utilisateurs et les groupes
UTS	Pour le nom d’hôte (hostname) et le nom de domaine NIS

2.1.2 Cgroups

Les *cgroups* (ou “Control Groups”) sont un mécanisme essentiel du noyau Linux pour limiter, contrôler et isoler l’utilisation des ressources système par les processus. Deux éléments majeurs caractérisent les *cgroups* : d’une part, le noyau qui définit la hiérarchie des *cgroups* et, d’autre part, les contrôleurs qui se chargent de distribuer et de limiter l’utilisation des ressources par les processus [13].

La première version des *cgroups* a constitué une avancée notable dans la gestion des ressources au sein des systèmes Linux. Cependant, avec l’accroissement de sa popularité, certaines de ses lacunes, notamment la gestion complexe des hiérarchies multiples, sont devenues plus apparentes. C’est dans ce contexte que la version deux des *cgroups* a été développée, visant à remédier à ces insuffisances.

La seconde version des *cgroups*, souvent désignée sous *cgroups v2*, se distingue principalement de la première par sa hiérarchie unifiée. Au cœur de cette hiérarchie se trouve le “Root cgroup”, également connu sous le nom de *Cgroup* racine. Dans un système Linux, tous les autres *cgroups* dérivent soit directement de ce *Cgroup* Racine, soit d’un de ses descendants. Dans cette hiérarchie, les *cgroups* enfants héritent des restrictions imposées par leur *cgroup* parent, ce qui permet une gestion cohérente et structurée des ressources. Des outils tels que *systemd-cgls* permettent d’inspecter et de visualiser cette hiérarchie, facilitant ainsi la gestion

et la délégation des ressources.

Les *cgroups*, une fois établis, peuvent être associés à des contrôleurs spécifiques pour réguler l’usage des ressources. Ces contrôleurs peuvent :

- Attribuer des poids pour prioriser un cgroup par rapport à un autre, aidant ainsi à garantir que les ressources sont allouées de manière équitable ou selon les besoins spécifiques.
- Fixer des plafonds stricts pour l’utilisation des ressources, permettant de prévenir la surconsommation et d’assurer un partage équitable.
- Allouer une quantité précise de ressources à l’avance, garantissant ainsi une réservation de ressources pour des tâches critiques.
- Garantir un approvisionnement minimal pour prévenir les conflits d’accès et assurer une performance stable.

Voici les types de contrôleurs supportés dans le noyau Linux. Chacun s’occupe de leurs propres ressources, ce qui permet une granularité plus fine.

TABLEAU 2.2 Liste des *cgroups* v2

Cgroup	Description
cpu	Garantie un minimum de “CPU shares” et peut limiter, si c’est spécifié, la quantité de CPU même si le CPU est libre
cpuset	Associe un Cgroup à un groupe de CPUs ou un nœud NUMA
memory	Limite la mémoire du processus, noyau ou swap
pids	Limite le nombre de processus qui peuvent être créés
io	Limite l’accès à un périphérique d’entrée-sortie par blocs
hugetlb	Limite l’utilisation de grande page de mémoire
perf_event	Permet d’observer certains événements générés par <i>perf</i>
rdma	Permet une utilisation limitée de <i>RDMA</i>

2.1.3 LXC

Linux Containers (LXC) s’est positionné comme le précurseur dans l’exécution de conteneurs pour Linux. Il a judicieusement associé les concepts de *namespaces* et de *cgroups* pour offrir une isolation du contexte d’exécution, tout en contrôlant la consommation de ressources des machines virtuelles [14].

À l’opposé des machines virtuelles traditionnelles, LXC propose une solution plus légère et efficace. Étant donné que le noyau est partagé, les conteneurs n’ont pas besoin d’apporter leur

propre noyau. Par conséquent, démarrer un conteneur sous LXC ressemble davantage à lancer une série de processus qu'à initialiser une machine virtuelle complète. Cette légèreté s'avère particulièrement pertinente dans des scénarios où l'on souhaite provisionner rapidement de nombreux conteneurs pour répondre à une demande croissante [15].

Cependant, LXC présente certains inconvénients, malgré ses avantages notables. En raison de son architecture, LXC est limité à l'émulation de systèmes d'exploitation qui utilisent le noyau Linux. En outre, il offre un niveau de sécurité inférieur par rapport aux machines virtuelles traditionnelles, comme l'indique cet article [9]. LXC dépend de divers mécanismes du noyau Linux, tels que *AppArmor*, *seccomp-bpf* et les *capabilities*, pour fournir l'isolation des conteneurs. Cependant, un des défis de sécurité intrinsèque avec les conteneurs est que, si un processus malveillant parvient à s'échapper du conteneur, il peut compromettre l'ensemble du système, car il n'est pas isolé du noyau. De plus, bien que les conteneurs soient isolés de l'hôte, ce dernier peut toujours surveiller les activités se déroulant à l'intérieur des conteneurs, ce qui pose d'autres préoccupations en matière de sécurité et de confidentialité.

Il convient de noter que Docker, dans ses débuts, a adopté LXC comme moteur d'exécution par défaut. Cependant, avec le temps et l'évolution du paysage technologique, LXC a perdu de son attrait. Il voit aujourd'hui une diminution progressive de ses mises à jour et de son adoption dans l'industrie [14].

2.1.4 Docker

Docker est devenu le standard en moteur d'exécution de conteneur, et est principalement responsable de la popularité des conteneurs. Ce qui le distingue des machines virtuelles et des autres moteurs d'exécution est son habilité à emballer des programmes. Originellement, son moteur d'exécution était *LXC* avant de changer à *containerd* [14]. Docker se sert d'un *Dockerfile* pour définir et construire un environnement conteneurisé. Chaque commande dans ce fichier est traitée séquentiellement, générant une image intermédiaire qui capture l'état du système à ce moment précis, semblable à un "snapshot" dans les machines virtuelles. Une fois toutes les commandes exécutées, l'image finale est prête à être lancée. Ce processus garantit que chaque exécution du conteneur est basée sur un environnement exactement identique, assurant une reproductibilité parfaite entre les environnements de développement, de test et de production.

Le moteur d'exécution de Docker est composé de deux parties soit le *containerd* et *containerd-shim*, tel que présenté dans le diagramme 2.1 :

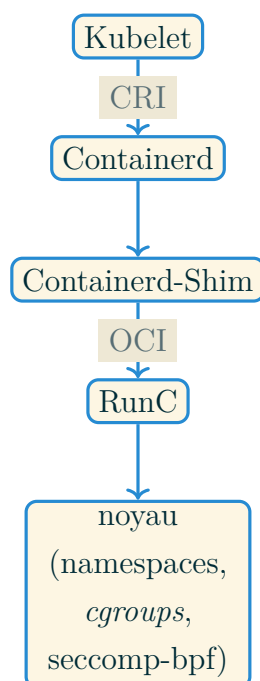


FIGURE 2.1 L'Architecture de Docker

La première est l'interface du conteneur appelé CRI (Container Runtime Interface). Dans le cas de Docker, cette interface est implémentée par *containerd* qui opère comme un démon Linux. Cette interface est ce que l'orchestrateur appelle quand il veut créer des conteneurs. Il va s'occuper de gérer les conteneurs sur la machine et leur cycle d'exécution. Il va télécharger les images, créer les interfaces réseau et il délègue la création des conteneurs au *Containerd-Shim*.

Le deuxième élément clé dans le fonctionnement des conteneurs est le moteur d'exécution du conteneur, dont les spécifications sont établies par l'Open Container Initiative (OCI). Bien qu'il existe plusieurs moteurs d'exécution, comme gVisor, axé sur la sécurité, et runhcs pour Windows, runC demeure le plus populaire. Comparé à ses homologues, runC privilégie avant tout l'efficacité, comme le démontre cet article [16]. Il maximise le partage de ressources avec le kernel et utilise principalement les *cgroups* et *namespaces* pour isoler les conteneurs. Cependant, bien que cela rende runC efficace, cela le rend également plus vulnérable, comparativement à d'autres plateformes, selon cet article [9].

En résumé

1. Les conteneurs sont une alternative plus légère que les machines virtuelles, car ils partagent le noyau de l'hôte.
2. Les conteneurs permettent de créer des images reproductibles de l'environnement de développement.
3. Du point de vue du noyau, les conteneurs sont perçus comme une collection de processus appartenant à un même *namespace* et *cgroup*.

2.2 Orchestrateurs

Dans le contexte actuel de l'infonuagique, les conteneurs sont devenus des outils incontournables pour encapsuler et exécuter de manière fiable des applications. Cependant, une fois ces conteneurs prêts, ils nécessitent une orchestration par une flotte de serveurs, tout comme les conteneurs maritimes doivent être transportés par une flotte de navires pour arriver à bon port. Les orchestrateurs entrent en scène pour remplir ce rôle de coordination, assurant une répartition efficace des tâches entre les conteneurs à travers la flotte de machines disponibles. Ils facilitent l'utilisation optimale des ressources, tout en veillant à la mise en échelle et à la résilience, éléments clés pour répondre aux demandes fluctuantes. Ceci garantit la continuité des services dans l'univers complexe et dynamique de l'infonuagique.

2.2.1 BORG

Borg est le premier orchestrateur développé par Google qui s'occupe d'orchestrer des conteneurs. Son but était de créer un orchestrateur qui serait capable de faire cohabiter des tâches gourmandes en CPU et des services sensibles à la latence. Ceci permet de fusionner deux bassins de ressources qui auraient normalement été séparés, offrant ainsi un bassin de ressources plus large et optimisant l'utilisation des ressources [17].

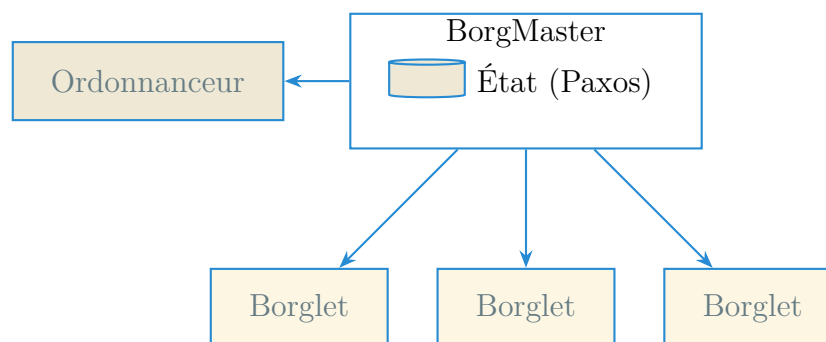


FIGURE 2.2 L'Architecture de Borg

Borg offre plusieurs bases que les orchestrateurs modernes utilisent encore. Tout d'abord, la grappe de serveurs est divisée en deux sections, soit le plan de contrôle et celui de travail, comme présenté dans la figure 2.2. Le plan de contrôle permet, selon une configuration déclarative de l'utilisateur, de déployer son service ou son calcul à travers un serveur dans le plan de travail de la grappe. Il permet aussi de s'assurer de le garder dans l'état désiré. De plus, afin d'assurer une résilience face aux pannes, l'état de l'orchestrateur est sauvegardé dans une base de données distribuée, dont la consistance est garantie par un algorithme d'élection tel que Paxos.

2.2.2 Kubernetes

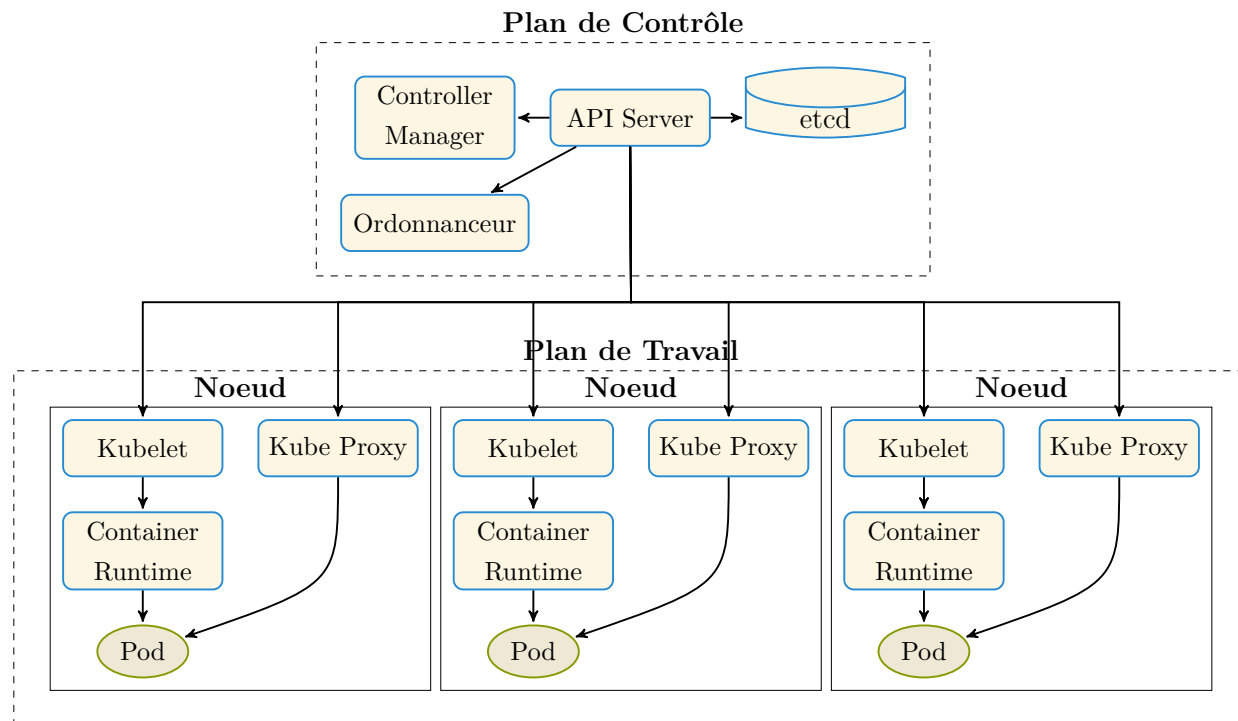


FIGURE 2.3 L'Architecture de Kubernetes

Kubernetes est le successeur de Borg. Il a été développé pour répondre à l'intérêt grandissant pour les conteneurs Linux et le besoin d'orchestrer leur déploiement [17]. Google a profité de l'expérience de Borg en développement d'orchestrateur de conteneurs, ainsi que de leur propre expérience à vendre des services avec Google Cloud, pour créer Kubernetes. L'idée était de créer une version plus distribuée de Borg, comme le démontre la figure 2.3. Il met aussi une emphase sur l'API et utilise une interface ainsi qu'une sémantique plus facile pour un utilisateur qui n'est pas familier avec les termes internes de Google. Il reprend l'idée d'avoir des configurations déclaratives de Borg et de partager les tâches ainsi que les services sous le même orchestrateur [17].

Dans Kubernetes, un *Pod* représente la plus petite unité de travail qui peut être déployée [18]. Un pod n'est pas simplement un conteneur, mais plutôt un groupe de conteneurs co-localisés qui fonctionnent en étroite synergie. Ces conteneurs partagent le même espace de noms pour le système de fichiers et de réseau. Ils disposent d'une configuration spécifiée sur la manière dont ils doivent être exécutés. Cette conception permet à plusieurs conteneurs au sein d'un même pod de communiquer facilement et de partager des ressources de manière transparente. En outre, pour assurer la fiabilité et la prévisibilité des applications, Kubernetes offre la

possibilité de réserver des ressources système pour ces Pods. Grâce aux *cgroups*, une fonctionnalité du noyau Linux, Kubernetes peut allouer des quantités spécifiques de ressources, comme le CPU ou la mémoire, à chaque pod. Cette allocation garantit une certaine qualité de service, veillant à ce que chaque application reçoive les ressources dont elle a besoin pour fonctionner efficacement.

Il y a trois types de qualité de services [19] :

1. **Best Effort** : C'est la classe la moins restrictive. Les pods "BestEffort" n'ont aucune limite de ressources spécifiées. Ils utilisent les ressources disponibles, mais sont les premiers à être évincés en cas de pression sur les ressources du nœud.
2. **Burstable** : Les pods "Burstable" ont des demandes de CPU définies, mais pas forcément de limites. Ils peuvent consommer davantage de CPU que ce qu'ils ont demandé (Burst), si le nœud a des ressources excédentaires, mais ils sont évincés après les pods "BestEffort" lors des contraintes.
3. **Guaranteed** : Pour être classé comme "Guaranteed", un pod doit spécifier des demandes de CPU identiques à ses limites pour chaque conteneur. Cela assure la disponibilité de ces ressources pour le pod, rendant ainsi ces pods les moins susceptibles d'être évincés. Cette classe de service est mise en œuvre de deux manières différentes, selon la configuration du Kubelet. Si le Kubelet applique les limites de ressources de manière statique, il utilisera le *cgroup* *cpuset* pour réserver un certain nombre de CPU pour le pod. Sinon, il emploiera le *cgroup* *cpu* pour attribuer un quota d'utilisation aux processus au sein du pod.

Pour déployer ces pods, Kubernetes se compose d'un ensemble de services répartis sur deux niveaux [18]. Le premier est le plan de contrôle. Il contient tous les services responsables de l'orchestration des déploiements.

Le API-Server est le serveur responsable de centraliser la communication entre tous les composants. Ceci permet de facilement cacher les détails d'implémentation, valider les objets, rajouter des valeurs par défaut et versionner les objets de Kubernetes. Dans ce cas, centraliser permet de facilement imposer certaines règles sémantiques et autres. Il est centralisé, mais puisque l'état est enregistré dans une base de données, la charge de requête à API-server peut être distribuée à travers plusieurs instances. Ceci permet d'avoir une grande disponibilité et une mise à l'échelle facile.

Ensuite dans le plan de contrôle il y a les contrôleurs qui sont similaires à des contrôleurs en robotique. Ils sont responsables d'observer l'état de leurs objets et faire les ajustements nécessaires pour atteindre l'état désiré du déploiement.

Pour maintenir un état consistant à travers la grappe, Kubernetes utilise etcd. etcd est une base de données de valeurs clés. Ils utilisent Raft comme algorithme de consensus. Raft est considéré comme une technique plus facile à comprendre que Paxos et tout aussi rapide [20].

Un autre service est l'ordonnanceur. L'ordonnanceur s'occupe d'aller voir quels pods sont censés être exécutés selon l'API-Serveur et s'occupe de choisir un nœud. L'ordonnanceur choisit en deux phases le nœud le plus apte à exécuter la tâche. Dans la première phase, il va filtrer ceux qui ne peuvent pas satisfaire les spécifications du déploiement. Dans la deuxième phase, il va noter quel nœud serait le meilleur pour le déploiement. Par défaut, l'ordonnanceur va prioriser les nœuds qui ont la plus grande capacité de CPU et de mémoire vive, mais la manière de filtrer et de noter les nœuds peut être personnalisée. Il peut prioriser la localité des différents nœuds, les SSD, si l'image est déjà installée sur le nœud, ou encore prioriser les nœuds déjà utilisés afin d'utiliser le moins de nœuds possible.

Le deuxième plan crucial dans l'architecture Kubernetes est le plan des travailleurs, aussi connu sous le nom de "Worker Plane".

Le Kubelet est le premier service essentiel dans l'architecture Kubernetes. Sa principale responsabilité est de s'assurer que les pods définis par l'API-Server soient correctement déployés et exécutés sur les nœuds. Il joue un rôle central en communiquant continuellement avec l'API-Server, en rapportant l'état actuel des ressources qu'il gère, et en vérifiant s'il y a de nouveaux pods à lancer. Pour mettre en place ces pods, le Kubelet s'interface avec le moteur d'exécution des conteneurs, qui doit être compatible avec le "Container Runtime Interface" (CRI). De plus, le Kubelet utilise cAdvisor, un outil intégré, pour recueillir et transmettre des statistiques détaillées sur les pods et les conteneurs qu'il gère.

Kube-proxy joue un rôle fondamental dans le modèle de réseau de Kubernetes, contribuant à simplifier le système et à le rendre plus rétrocompatible. Dans Kubernetes, chaque pod se voit attribuer sa propre adresse IP, unique au sein du réseau de la grappe de serveurs. Cette simplification du modèle réseau est particulièrement avantageuse, car elle élimine le besoin de créer explicitement des liens entre les pods ou de gérer la cartographie des ports de conteneur vers les ports hôte.

Le Kube-proxy agit comme une sorte de relais ou de répartiteur de trafic, facilitant la traduction des noms de services vers les adresses IP des pods. Il s'assure ainsi que les requêtes soient correctement acheminées vers les pods appropriés.

L'un des avantages majeurs de Kube-proxy est sa capacité à masquer l'ensemble des pods backend derrière un seul et unique "service". Cela offre une abstraction qui permet aux clients de communiquer avec un ensemble de pods via une seule adresse IP de service.

Kube-proxy intègre également des capacités de répartition de charge. En fonction de la demande et de la disponibilité, il répartit intelligemment le trafic entrant entre les pods, pour assurer une distribution équilibrée de la charge et optimiser les performances.

Au-delà de la simple répartition de charge, Kube-proxy prend également en charge les contrôles du trafic entrant. Cela permet de diriger le trafic entrant vers des services spécifiques en fonction de divers critères, comme l'URL ou le domaine. Cette fonction est particulièrement utile pour exposer des applications HTTP, des sites web ou des API, tout en conservant un contrôle granulaire sur la manière dont le trafic est dirigé et traité.

2.2.3 Docker Swarm

Docker Swarm est la solution proposée pour déployer les conteneurs et mettre à l'échelle un service créé avec le moteur Docker. Docker Swarm est intégré au moteur Docker et fournit le code pour les composants de contrôleur et de travailleur, facilitant ainsi la création d'une architecture plus distribuée [21]. Ceci le rend plus facile à déployer en production.

Docker swarm reprend une architecture similaire à Kubernetes, mais les composants sont tous intégrés dans le moteur Docker. Il divise le travail entre les managers et les travailleurs. Les managers s'occupent de maintenir l'état en utilisant Raft comme algorithme de consensus distribué, s'occupent de planifier le travail et sont l'interface HTTP pour la grappe de serveurs [21].

Une autre limitation de Docker swarm est qu'il n'est pas capable d'atteindre la même échelle que Kubernetes. Si Kubernetes peut déployer 300 000 conteneurs, selon leur test d'échelle de conteneurs, de son côté Docker swarm peut seulement déployer 30 000 conteneurs [22]. Il est aussi seulement conçu pour travailler avec Docker et ne supporte aucun autre moteur d'exécution, comme LXC ou rkt [22].

2.2.4 Apache Mesos

Mesos est un autre orchestrateur de conteneur populaire. Il est aussi très facile de le mettre à l'échelle. Il a été prouvé qu'il pouvait supporter jusqu'à 50 000 nœuds [23]. Mesos, comparé à Kubernetes et Docker Swarm, se veut plus comme un moyen de créer un bassin de ressources qui fait abstraction des détails spécifiques du matériel d'une grappe de serveurs. Ceci facilite l'exécution de divers logiciels comme Hadoop, Spark ou MPI. Il peut orchestrer directement des conteneurs via Mesos Containerizer, qui est son logiciel intégré pour les conteneurs.

Cet ordonnanceur communique avec le maître de Mesos pour demander des ressources. Une fois que le maître propose des ressources, c'est l'ordonnanceur qui décide lesquelles utiliser

en fonction des besoins. De plus, Mesos dispose d'un "exécuteur" dont le rôle principal est d'exécuter des tâches spécifiques sur les ressources qui ont été allouées.

Comparé à d'autres orchestrateurs, Mesos utilise Zookeeper et son algorithme Zab pour maintenir son état et élire son contrôleur [23]. L'état préservé dans Zookeeper permet à un nouveau contrôleur de pouvoir rapidement reprendre le travail. Les nouveaux contrôleurs sont gardés dans un arrêt chaud configuré par Zookeeper. Par contre, chaque cadriciel doit utiliser sa propre technique pour gérer la tolérance aux pannes. La majorité des autres orchestrateurs utilisent etcd et Raft, puisque les développeurs d'etcd ont fait des améliorations basées sur l'usage a posteriori de Zookeeper [24].

Mesos offre plusieurs cadriciels pour déployer des applications conteneurisées. Il y a l'option d'utiliser Docker pour l'exécution des conteneurs ou le Mesos Containerizer, un cadriciel natif à Mesos [25]. Ce dernier offre plusieurs isolateurs modulables, permettant une isolation flexible des ressources système entre les conteneurs.

En conclusion, Mesos se veut plus un moyen plus facile de déployer d'autres cadriciels d'orchestration comme MPI, Hadoop, Spark et Docker.

En résumé

1. Borg, en tant que prédécesseur de Kubernetes, a joué un rôle pionnier dans l'orchestration de conteneurs. Toutefois, sa conception monolithique et certaines complexités inhérentes ont limité son application au-delà des murs de Google. Ceci a influencé Kubernetes pour adopter une solution plus modulaire et ouverte.
2. Kubernetes s'est imposé comme l'orchestrateur de conteneurs de prédilection, grâce à sa flexibilité, sa communauté active et sa capacité à évoluer efficacement dans des environnements complexes.
3. Docker Swarm, bien qu'intuitif et facile à utiliser, ne parvient pas à rivaliser avec la capacité de Kubernetes ou de Mesos à gérer des infrastructures de grande échelle, ce qui le rend moins adapté pour des déploiements à grande échelle.
4. Mesos excelle dans la provision d'un bassin de ressources partagées pour divers cadriciels, mais il n'offre pas la simplicité et la focalisation sur l'orchestration de conteneurs que proposent Kubernetes ou Docker Swarm, le rendant moins idéal pour les déploiements de services à grande échelle.

2.3 Source d'information pour l'observabilité

L'observabilité, telle qu'elle est perçue dans les systèmes modernes, repose essentiellement sur trois piliers fondamentaux [26] : la journalisation (ou logs), les métriques et le traçage. Ces trois facettes offrent différentes perspectives sur le comportement et la performance des systèmes. Chacune d'entre elles fournit des renseignements uniques qui, lorsqu'ils sont combinés, donnent une vue d'ensemble complète de l'état et des opérations d'un système. Il est généralement considéré que les trois piliers sont essentiels à l'atteinte d'une observabilité robuste. Dans cette section, nous explorerons comment Kubernetes et ses orchestrateurs offrent ces sources d'information, permettant ainsi d'examiner la richesse des données disponibles et d'évaluer comment elles peuvent être utilisées pour déchiffrer les intrications du système.

2.3.1 Métriques

Les métriques sont une valeur dans un système de mesure sur une période. Ces métriques peuvent être l'utilisation des ressources et l'état des contrôleurs. Ces métriques peuvent ensuite être agrégées et analysées afin de mieux comprendre le système.

Google Cluster-Usage Trace

Alors que les nouveaux projets utilisent Kubernetes, tous les projets préexistants chez Google utilisent Borg. Borg est toujours maintenu et, en 2011 et 2019, Google a publié un mois de données de trace sur leur utilisation de Borg [27]. Ces données, collectées à travers des événements et l'utilisation des ressources, puis envoyées via gRPC à un collecteur [28], offrent un aperçu unique des dynamiques internes d'une infrastructure d'envergure mondiale.

L'objectif principal de la publication de ces traces est de fournir une meilleure compréhension des charges de travail que les orchestrateurs, tels que Borg, doivent gérer au quotidien. Ces traces, en dévoilant la complexité et la diversité des tâches orchestrées au sein d'un géant comme Google, ont également servi de fondement à l'élaboration d'outils de mise à l'échelle automatique tels qu'Autopilot [29].

Cette étude se distingue non seulement par sa précision, mais aussi par le fait qu'elle représente l'une des premières instrumentations d'orchestrateur à une échelle aussi large. Ces informations permettent d'appréhender les enjeux et les défis rencontrés par une grande organisation comme Google et offrent la possibilité de comparer ces données avec d'autres études concernant des grappes de serveurs à grande échelle.

cAdvisor

Développé par Google, cAdvisor est un outil conçu pour collecter des métriques liées à l'utilisation des conteneurs. Depuis son introduction en 2014, il est devenu le collecteur de métriques standard pour les nœuds de Kubernetes [23].

cAdvisor s'adresse principalement à ceux qui cherchent à obtenir un aperçu rapide et global du comportement des conteneurs. Il est doté d'une interface web, permettant aux utilisateurs d'avoir un aperçu en continu des ressources utilisées. Grâce à une détection automatique des *cgroups* existants et nouvellement créés, cAdvisor nécessite peu de configuration, simplifiant ainsi sa mise en œuvre.

L'outil offre une gamme variée de métriques. Elle inclut la consommation de CPU, mémoire, disque et réseau, ainsi que des métriques plus spécifiques. Ces dernières englobent les compteurs de performances et des détails matériels comme le nombre de cœurs, la capacité mémoire et cache. Ces informations sont inestimables pour les administrateurs système et les développeurs, leur permettant d'analyser et d'optimiser les performances de leurs applications conteneurisées [30].

Bien que cAdvisor collecte de précieuses métriques, il n'a pas la capacité intrinsèque de les stocker sur le long terme. Ainsi, il se repose sur des bases de données tierces pour cette finalité. Des solutions renommées comme Prometheus, Elasticsearch et InfluxDB ont intégré le support pour cAdvisor, assurant ainsi une collecte et un stockage efficaces des métriques.

Toutefois, il est crucial de noter que, malgré ses atouts, cAdvisor présente certaines limitations en matière de granularité. Bien qu'il offre une vue globale des métriques, il pourrait ne pas suffire aux ingénieurs système ou aux SRE recherchant une analyse détaillée des performances. Sa précision par défaut est de 10 secondes, ce qui, dans certaines situations, pourrait ne pas être assez détaillé pour diagnostiquer des anomalies ou optimiser des performances au niveau du noyau [31] [32] [33].

Kube State Metric Server

Le *Kube-State-Metrics Server* est un outil développé par SIG-Instrumentation de Kubernetes. Des outils comme cAdvisor se focalisent sur les métriques relatives à l'utilisation des ressources par les conteneurs. En revanche, Kube-State-Metrics se distingue en exposant des métriques concernant l'état des objets au sein de Kubernetes [34]. Il n'est pas intégré à Kubernetes, il doit être rajouté comme serveur.

L'objectif principal du *Kube-State-Metrics Server* est d'observer l'état de grappe de serveurs, sans être directement couplé à l'API-Server. En le faisant, il fournit une source de métriques

pouvant être utilisée pour configurer des alertes en cas de changements d'état indésirables ou inattendus.

Parmi les utilisations notables du Kube-State-Metrics, on trouve avant tout sa capacité à une surveillance proactive. En effet, cet outil peut être mobilisé pour observer attentivement des éléments essentiels, tels que le nombre de nœuds ou encore le statut des pods, pour n'en citer que quelques-uns. De plus, la clarté de la vision qu'il offre est un atout précieux pour l'optimisation des ressources. Cela permet aux administrateurs de moduler les ressources allouées en se basant sur l'état actuel et les besoins manifestes de grappes de serveurs. En outre, le Kube-State-Metrics s'avère être un excellent moyen d'avoir une vue d'ensemble sur l'historique des tendances, ainsi que sur l'évolution générale de l'état de la grappe de serveurs au fil du temps. La compagnie Datadog, qui développe une plateforme d'observabilité, l'utilise notamment pour surveiller la santé des déploiements, des nœuds et des pods, ce qui sous-tend plusieurs de ses produits [35].

Cependant, tout outil présente ses limites, et le Kube-State-Metrics ne fait pas exception à la règle. Même s'il propose une vue d'ensemble complète de l'état de la grappe de serveurs, il ne vise pas à détailler en profondeur l'utilisation des ressources au niveau des conteneurs. Pour une telle granularité, d'autres outils, à l'instar de cAdvisor, seraient plus appropriés.

2.3.2 Traçage distribué

Les infrastructures désormais sont de plus en plus distribuées. Une requête va généralement devoir passer par plusieurs autres services et bases de données afin de répondre à la demande d'un client. Ceci rend le système plus complexe à comprendre et analyser. Pour remédier à ce problème, le concept de traçage distribué a vu le jour.

Travail de pionniers du traçage distribué

Le travail de *magpie* [36] peut être considéré comme pionnier dans le traçage distribué. Bien qu'il ne soit pas aussi riche en fonctionnalités que les outils modernes, il jetait les bases de la surveillance des requêtes par une instrumentation ciblée de certaines parties du noyau Windows NT. Trois défis clés y étaient déjà abordés : la synchronisation des traces, l'identification des événements causalement liés, et la limitation de l'impact sur la performance.

Dapper

Dapper se distingue des premiers travaux sur les traceurs distribués puisqu'il est le premier à avoir été déployé en production à grande échelle. L'article sur Dapper a été publié après

2 ans d'expérimentation avec les traceurs distribués et offre une perspective pratique sur le sujet [37].

L'article de Dapper [37] dénote deux principales approches : l'approche annotée ou l'approche black-box. L'approche annotée consiste à annoter chaque requête afin de déterminer la cause et son effet, ce qui, bien que précis, ajoute un surcoût d'exécution et s'avère intrusif. L'approche black-box enregistre les messages par les différents services et utilise des statistiques pour déterminer le chemin critique d'une requête. Cette technique a l'avantage de réduire le surcoût d'exécution en évitant la propagation du contexte de requête, mais elle souffre d'un caractère non déterministe, ce qui complique la tâche pour un déverminage efficace. Une autre méthode pour éviter la propagation du contexte est de spécifier les causes et effets des divers événements. Toutefois, cette méthode manque de précision et de transparence, et requiert un effort supplémentaire de la part du développeur.

Pour ces raisons, Dapper décide d'annoter l'identifiant d'une requête afin de facilement pouvoir tracer une requête et utilise quelques techniques pour pallier différentes lacunes de cette approche.

Tout d'abord, Dapper effectue de l'échantillonnage de requêtes, afin de réduire le surcoût d'exécution du traçage. C'est un compromis acceptable, car il suffit d'une requête bien choisie pour déverminer un problème. Il est important de noter que, bien que seulement un sous-ensemble des requêtes soit tracé, chaque requête choisie pour le traçage est tracée au complet. Ceci permet d'obtenir une vision détaillée des interactions au sein du système pour ces requêtes spécifiques. Un faible surcoût d'exécution rend l'adoption du traçage distribué plus acceptable pour les administrateurs de système.

Pour éviter que les développeurs aient à instrumenter leurs applications, les développeurs de Google ont instrumenté les bibliothèques communes de réseautage pour annoter les requêtes. Ainsi, avoir du traçage distribué ne demande aucun effort de la part des développeurs.

Dapper se base sur les intervalles de temps (span) comme unités fondamentales de trace. Chaque intervalle de temps possède son propre identifiant ainsi qu'un identifiant parent, afin d'illustrer les relations causales entre les unités de travail. Ces relations parent-enfant, combinées aux intervalles de temps, forment un arbre de trace qui permet de visualiser le chemin d'exécution d'une requête.

OpenTelemetry

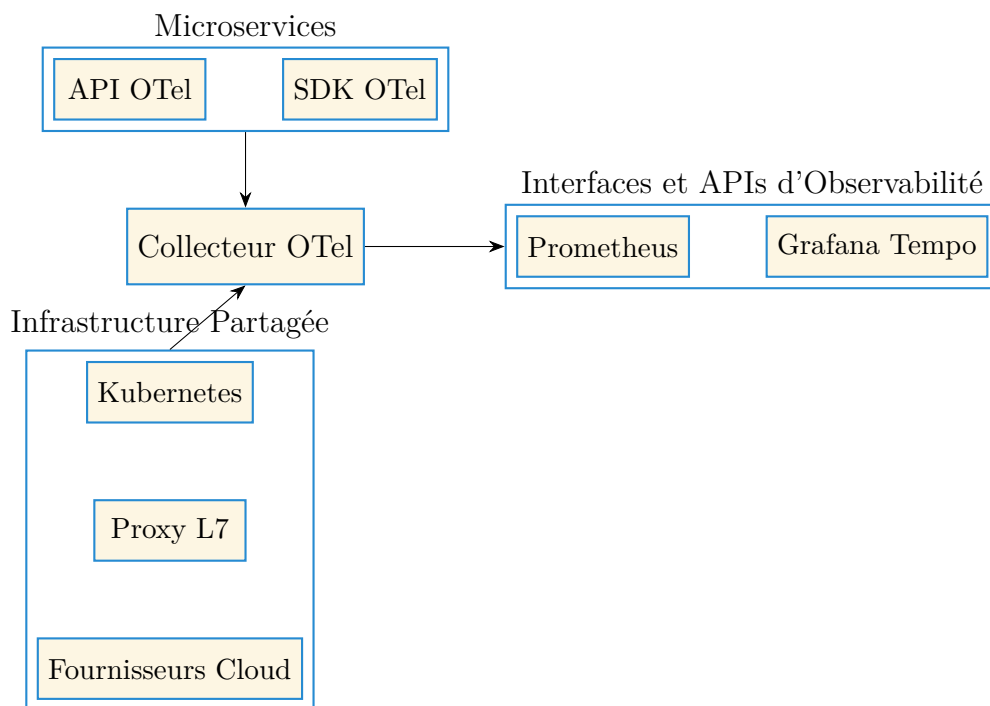


FIGURE 2.4 Architecture de Open Telemetry (OTel)

OpenTelemetry est né de la fusion des projets OpenTracing et OpenCensus dans le but d'établir une norme pour la collecte de traces et de métriques dans le monde des applications distribuées [38]. L'initiative vise à fournir un ensemble d'APIs et d'instrumentation pour simplifier l'intégration des outils de surveillance et de traçage.

OpenTelemetry prend en charge à la fois les métriques et les traces, ce qui le rend unique par rapport à d'autres outils de traçage. Cette capacité à capturer et à visualiser les données, à la fois du côté des métriques et du traçage, donne aux développeurs et aux opérateurs une vision plus complète de la performance et de la santé de leurs systèmes.

Parmi les implémentations les plus populaires de *OpenTelemetry*, il y a tout d'abord Zipkin (2012) [39] qui fut le premier traceur distribué en source libre créé par Twitter. Jaeger, initialement développé par Uber en 2017 [40], est un outil de traçage distribué en source libre qui est utilisé pour surveiller et résoudre les problèmes de performance dans les microservices. S'inspirant des concepts de Dapper et de Zipkin (2012), Jaeger offre une solution pour tracer les demandes à travers plusieurs services distribués. Finalement, il y a l'implémentation faite par Open Telemetry (OTel) qui est illustrée dans la figure 2.4.

Il y a un effort entrepris par David Ashpole du SIG-Instrumentation pour instrumenter certains des composants de Kubernetes avec *OpenTelemetry* [41] [42]. La possibilité d'instrumenter des éléments clés de Kubernetes, tel que l'Api-Server, etcd et Kubelet, permet de mieux comprendre les interactions entre les différents composants de Kubernetes et améliore l'observabilité. Cette amélioration de l'observabilité est cruciale, car elle permet aux administrateurs et aux développeurs de diagnostiquer et résoudre les problèmes plus efficacement. De plus, elle fournit des aperçus précieux sur les performances et l'utilisation des ressources dans les déploiements de Kubernetes.

Comme l'article sur Dapper le mentionne, une des manières de rendre l'instrumentation plus transparente fut d'utiliser des bibliothèques communes dans l'infrastructure de Google [37]. Plusieurs des microservices utilisent Istio pour faire le maillage de services. Il y a donc quelques études sur comment instrumenter de manière très transparente les microservices déployés sur Kubernetes en utilisant Istio [43]. Quelques études, telles que celle de Wang et al. [44], ont exploré et documenté des cas d'utilisation spécifiques de l'instrumentation des microservices dans des environnements Kubernetes en utilisant Istio [44].

En résumé

1. La synchronisation des traces, l'identification des événements causalement liés et la limitation de l'impact sur la performance sont des facteurs importants pour l'implémentation d'un traceur distribué efficace.
2. L'architecture de Dapper a inspiré les traceurs distribués modernes. Les principes d'échantillonnage, d'intervalle et d'annotation demeurent très populaires.
3. *OpenTelemetry* se positionne comme le standard moderne pour le traçage distribué, offrant un cadre robuste pour la collecte de métriques et de traces dans les applications distribuées.

2.3.3 Traçage Noyau

Il y a deux catégories de traceur noyau. Premièrement, il y a les traceurs qui ne font qu'enregistrer l'événement. Deuxièmement, il y a les agrégateurs. Ceux-ci vont faire un calcul lorsque l'événement est rapporté. Les traceurs, utilisés pour réduire au minimum les surcoûts d'exécution, permettent une analyse efficace des systèmes fonctionnant en temps réel dur, ainsi que des analyses post-mortem. Les agrégateurs, quant à eux, sont utilisés pour la surveillance spécifique de certaines mesures.

Perf

Perf, souvent référencé comme le profileur officiel de Linux, est préinstallé sur la plupart des distributions Linux [7, Chapter 13]. C’est un outil robuste conçu pour profiler et tracer le noyau Linux et les applications en espace utilisateur.

L’une des forces majeures de Perf réside dans sa capacité à tracer différents types d’événements, tels que les points de trace et les points de trace statiques. Son principal atout est cependant sa capacité à collecter des événements matériels, offrant ainsi une compréhension détaillée sur les performances du système.

Perf recueille des informations soit par des compteurs d’événements, soit en stockant des événements dans un tampon de mémoire circulaire. Sa particularité réside dans l’échantillonnage des compteurs de performance matériels pour un processus spécifique, offrant ainsi une vue précise de son comportement.

Après la collecte, les événements peuvent être lus à l’aide de *Perf script*. Cette sortie peut ensuite être analysée avec des outils tels que *AWK*, *Perl*, ou *Python* pour générer des visualisations détaillées, facilitant ainsi l’analyse des performances comme le “flame graph” créé par Brendan Gregg [45] [46].

Dans le contexte de Kubernetes, l’importance de Perf est mise en évidence par des outils comme *ddprof*, le profileur de Datadog est une plateforme de surveillance et d’analyse pour les infrastructures infonuagiques. *ddprof* utilise Perf pour échantillonner et déterminer le temps CPU consommé par chaque processus ainsi que leurs allocations, offrant une vue détaillée des performances au sein d’un environnement Kubernetes [47].

Ftrace

Ftrace, intégré au noyau Linux, sert de traceur et de profileur pour offrir une introspection profonde des opérations du noyau.

Son fonctionnement repose principalement sur l’exploitation de différents types d’événements. Les points de trace statique sont définis par `TRACE_EVENT`. Les points de trace dynamiques du noyau sont représentés par les kprobes, tandis que les points de trace dynamiques utilisateurs sont représentés par les uprobes. Enfin, les hist sont des événements uniques à Ftrace qui permettent de générer des histogrammes uniques.

L’interaction avec Ftrace est principalement réalisée à travers Tracefs, un système de fichiers virtuel dédié. Cela le rend particulièrement utile pour les systèmes embarqués avec des contraintes d’espace [7]. Toutefois, dans un contexte serveur, l’édition directe des fichiers peut

s'avérer ardue. C'est là, que Trace-cmd entre en jeu, fournissant une interface de commande de haut niveau pour interagir avec Ftrace.

Cependant, Ftrace n'est pas sans ses défis, en particulier dans des environnements orchestrés. Il est important de noter qu'il ne traite pas nativement l'espace utilisateur, ce qui peut limiter son utilité pour des analyses complètes. De plus, selon l'article [48], Ftrace peut rencontrer des difficultés de mise à l'échelle, en particulier sur des systèmes multicœurs.

LTTng

LTTng est un traceur originalement créé au Laboratoire DORSAL de Polytechnique et est désormais maintenu par Efficios. Ce traceur se concentre principalement sur la collecte d'événements de traces avec un faible surcoût d'exécution [49].

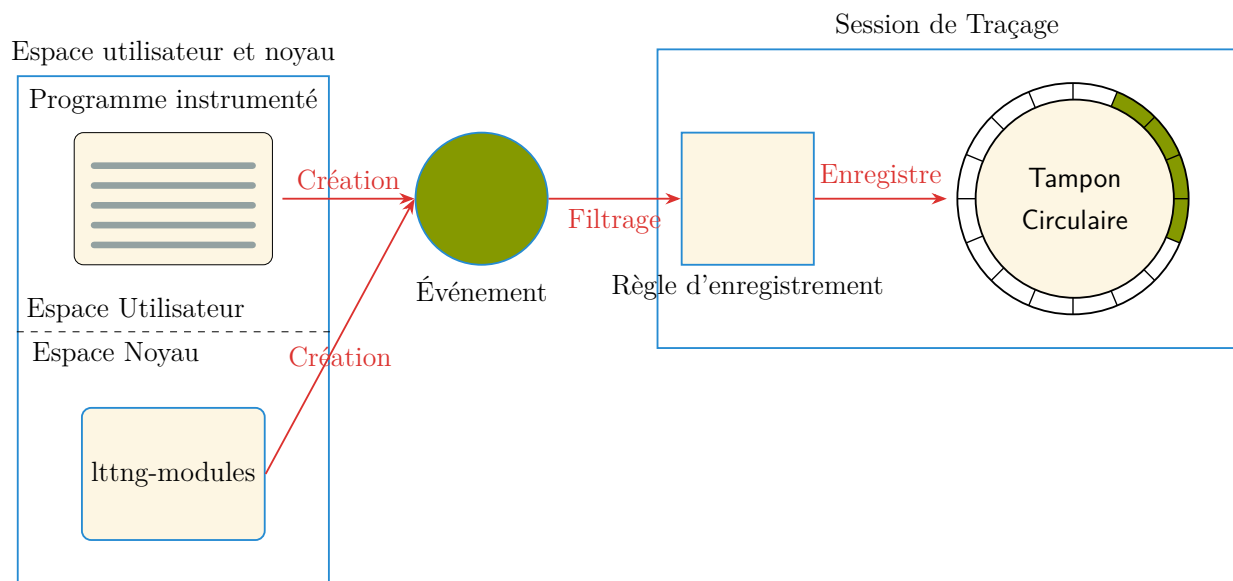


FIGURE 2.5 LTTng Processus d'enregistrement

Ce traceur se concentre principalement sur la collecte de points de trace statiques produits par la Macro `TRACE_EVENT`. L'avantage des points de trace statiques générés par `TRACE_EVENT` est leur rapidité et leur simplicité, mais ils doivent être insérés avant la compilation. Rajouter un point de trace demande de recompiler le noyau et éventuellement soumettre ce correctif au noyau. Il supporte aussi les points de trace dynamiques générés par kprobes. Les kprobes peuvent être appliqués au noyau à l'exécution, mais avec surcoût d'exécution plus élevé comparé aux `TRACE_EVENT`.

Ce traceur noyau a deux qualités qui le distinguent.

Premièrement, il est un traceur noyau très performant. Il collecte chaque événement dans un tampon de mémoire circulaire, tel que montré à la figure 2.5. Pour rendre la mise en échelle plus rapide, chaque CPU a son propre tampon, ce qui retire le besoin pour des verrous. Il est alors l'un des traceurs noyau les plus rapides avec un grand nombre de cœurs [48]. Ils utilisent aussi le format CTF qui a été conçu pour réduire le surcoût d'écriture.

Deuxièmement, il a un traceur en espace utilisateur afin d'offrir de l'observabilité à chaque couche du noyau [50]. Il est l'un des rares traceurs à offrir ce genre de facilité. On peut utiliser des points de trace USDT ou uprobes afin de tracer l'entrée d'une fonction, et il peut lire les arguments en entrée ou en sortie. Par les points de trace, LTTng permet d'aller chercher de l'information à l'intérieur de la fonction.

Un autre avantage particulièrement utile dans le contexte des conteneurs est l'ajout de contexte sur les événements, comme les points de trace noyau et utilisateur. LTTng peut rajouter de l'information comme le pid ou le tid. Il est aussi désormais particulièrement utile pour les conteneurs [51]. Il peut rajouter le pid virtuel et est capable de rajouter l'identifiant des sept sur huit *namespaces* possibles (pid_ns, cgroup_ns, etc.).

eBPF

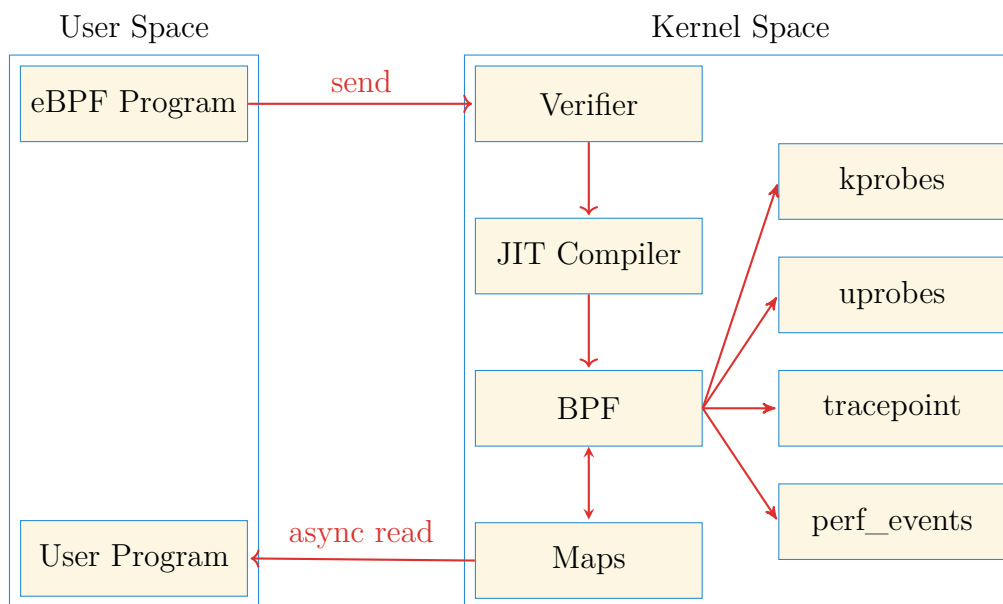


FIGURE 2.6 Architecture d'eBPF

eBPF est une version étendue de BPF (Berkeley Packet Filter) [52]. Il est un programme exécuté dans un environnement isolé par une machine virtuelle, comme montré dans la figure

2.6. La version étendue de BPF rajoute certaines fonctionnalités pour permettre d'étendre le noyau Linux sans utiliser des modules noyau. Comparé à des modules noyau, eBPF utilise un sous-ensemble de C, afin de garantir que l'exécution du programme ne peut pas entraîner un dysfonctionnement de l'exécution du noyau. Il s'exécute suite à un événement comme l'invocation d'un appel de système, un kprobe, un uprobe, un intervalle de temps, etc. Ces événements et l'information agrégés sont ensuite rapportés par les BPF-map pour communiquer entre un programme eBPF et le plan utilisateur.

Il existe plusieurs utilitaires afin de faciliter le développement de programmes eBPF. Bpfttrace est le plus facile à exploiter. Il utilise un langage simple, similaire à *AWK*, et rapporte l'information au STDIN. Ceci permet de rapidement calculer des statistiques sur le système et réaliser un prototype. Ensuite, pour développer des outils d'analyse de performance, il y a BCC qui est une collection d'outils pour développer des programmes eBPF.

Il est populaire dans le contexte de Kubernetes, puisqu'il y a une extension à *kubectrl*, appelée *kubectrl trace*, qui permet de réutiliser l'infrastructure de Kubernetes pour déployer ses programmes eBPF.

Il y a aussi Pixie, développé par New Relics, qui permet de surveiller Kubernetes en utilisant l'information collectée par eBPF. Utiliser eBPF au lieu d'un autre traceur permet de créer une intégration transparente avec l'environnement de Kubernetes.

eBPF inclut XDP qui permet d'exécuter les programmes eBPF directement sur les cartes réseau. Ceci permet de générer la distribution de charge et la mitigation de DDoS très rapidement. Cette fonctionnalité a permis de développer plusieurs programmes comme Cilium [53], Hubble et Katran [54]. eBPF permet aussi d'avoir plus de contexte sur les appels réseau et de savoir quel processus appelle qui et pourquoi [55].

Bien que eBPF puisse être très pratique pour résoudre des erreurs et développer des prototypes de moniteurs, ceci implique un certain surcoût d'exécution. Aussi la mise en échelle d'eBPF pour les systèmes à plusieurs cœurs est difficile selon cet article [48].

SystemTap

SystemTap est une plateforme robuste de traçage qui s'inscrit dans la même veine que eBPF. Il offre la possibilité d'exécuter du code suite à un événement de trace dans le noyau Linux, permettant une introspection détaillée du système. Sa syntaxe, inspirée de C et AWK, le rend accessible pour ceux familiarisés avec ces langages [56].

Là où SystemTap se distingue réellement de eBPF, c'est dans son approche d'exécution du code. Au lieu d'utiliser une machine virtuelle dans le noyau, comme le fait eBPF, SystemTap

traduit les scripts en modules noyau. Cela peut donner à SystemTap une plus grande flexibilité dans ses opérations, mais cela implique un coût. La compilation en modules noyau peut rendre le démarrage de SystemTap plus lent comparativement à eBPF. Il est également plus difficile de garantir la sécurité d'un programme avec un module noyau, comme le note Brendan Gregg [57].

Cependant tout comme eBPF, la performance de SystemTap est grandement impactée par un système qui a un grand nombre de cœurs [48].

En résumé

1. Perf, Ftrace et eBPF sont très pratiques puisqu'ils sont inclus dans le noyau de Linux. Par contre, Ftrace et eBPF souffrent de problèmes de performance quand il y a plusieurs cœurs.
2. eBPF et SystemTap sont très flexibles puisqu'ils permettent d'exécuter du code lorsqu'un événement survient. Ceci permet de calculer progressivement certaines métriques ou statistiques. C'est pratique, mais cela vient au coût d'utiliser plus de ressources lors de l'exécution du programme, et souffre aussi de problèmes de performance quand il y a plusieurs cœurs.
3. LTTng fait une chose et le fait bien. Il enregistre les événements de trace très rapidement et impacte peu l'exécution des programmes.

2.4 Analyse d'information

La collecte de données n'est que la première étape d'un processus d'analyse en profondeur. Une fois ces données réunies, la tâche essentielle est de les analyser et de les visualiser de manière à obtenir des informations pertinentes et exploitables. Dans le cadre spécifique de Kubernetes, une multitude d'outils se présentent pour assurer cette fonction, comme le démontre une référence en la matière présentée sur ce site [58]. Dans cette section, nous étudierons les outils en code source libre les plus populaires destinés à l'analyse de logs, de métriques, et de traces, tant pour les orchestrateurs que pour le noyau Linux.

2.4.1 Analyse des journaux

Elasticsearch (ELK)

Elasticsearch est l'une des plateformes les plus populaires pour l'indexation, la recherche et l'analyse des journaux et des métriques. Basé sur Apache Lucene [59], un moteur de recherche en code source libre hautement performant, Elasticsearch est conçu pour traiter de grands volumes de données avec rapidité et efficacité. Elasticsearch enrichit les capacités de Lucene en offrant une couche de distribution. Cette distribution permet à Elasticsearch de partitionner les données sur plusieurs nœuds, optimisant ainsi les performances et la résilience lors de l'analyse de vastes quantités de données. Grâce à son architecture distribuée, Elasticsearch peut indexer de grands volumes de journaux rapidement [60]. Sa capacité de recherche est presque instantanée et permet aux utilisateurs d'extraire des informations pertinentes des journaux en quelques millisecondes.

LogStash est un composant essentiel de la pile Elastic, souvent appelée la pile ELK (Elasticsearch, LogStash, Kibana) [61]. Il s'agit d'un serveur de traitement des événements qui prend en entrée les journaux, les transforme, et les achemine vers diverses destinations, notamment Elasticsearch.

Kibana est l'interface graphique dédiée à la visualisation des données indexées par Elasticsearch. Il permet aux utilisateurs de créer des tableaux de bord interactifs pour visualiser leurs journaux et métriques. Kibana facilite la compréhension des tendances, des modèles et des anomalies dans les données.

Bien que la combinaison initiale d'Elasticsearch, LogStash et Kibana soit connue sous le nom de pile ELK, l'écosystème Elastic a évolué pour inclure d'autres outils comme Beats pour la collecte de données. Cette pile complète offre une solution de bout en bout pour la collecte, le stockage, l'analyse et la visualisation des données.

2.4.2 Analyse des métriques

Prometheus

Prometheus est une base de données de séries chronologiques offerte en source libre, développée originellement par SoundCloud et maintenant prise en charge par le CNCF [62]. Elle s'occupe d'aller chercher l'information offerte par des points appelés *Prometheus Exporter* spécifiés par l'utilisateur. Prometheus est flexible et permet de collecter toutes sortes de métriques. Il supporte des métriques personnalisées à l'application. Il existe aussi un support pour des métriques sur Kubernetes. Il existe des exporteurs pour les sources de bas niveau comme le *node_exporter* qui est capable d'aller chercher des métriques sur l'utilisation des ressources dans Linux et autres systèmes UNIX. Pour les métriques spécifiques aux *cgroups*, il supporte aussi *cAdvisor*. Il y a le *ebpf_exporter* développé par CloudFlare qui est capable d'exporter des données de traçage noyau [63]. Ces données sont ensuite enregistrées à long terme par Prometheus dans une base de données de séries chronologiques. Prometheus offre un langage d'interrogation afin de chercher facilement à travers les métriques collectées. Cette information peut ensuite être collectée par une interface web pour le monitoring. Le fait qu'ils utilisent un langage flexible permet de facilement personnaliser les vues utilisateur et facilement créer des requêtes pour ses propres métriques définies dans l'exporteur. Il s'intègre très bien alors avec des visualisations génériques offertes par *Grafana*.

Le fait de collecter des métriques en continu lui permet aussi d'alerter les administrateurs réseau si une métrique dépasse un certain niveau, ou si un événement spécifique se produit. Il existe un agent Prometheus qui peut alerter par toutes sortes de messageries les administrateurs réseau. Ces administrateurs peuvent aussi créer leurs propres règles personnalisées, afin d'avoir des alertes plus précises en cas de certains incidents.

L'association d'un langage d'interrogation intuitif, de métriques personnalisables et d'un système d'alerte efficace confère à Prometheus une grande extensibilité, comme en témoigne une étude récente. Dans cette publication, Prometheus est employé efficacement pour anticiper les anomalies, automatisant ainsi le processus de surveillance et renforçant la proactivité dans la gestion des incidents [64].

Grafana



FIGURE 2.7 Tableau de Bord Grafana. Source : [1] distribuée sous licence CC BY 2.0 Deed

Développée par Grafana Labs, Grafana est une plateforme d'analyse en source libre qui offre une interface web pour visualiser, interroger et surveiller les journaux, métriques et traces générés par des grappes de serveurs ou des déploiements divers [55].

L'une des principales forces de Grafana réside dans sa capacité à intégrer une multitude de sources de données. Bien que Prometheus soit l'une des bases de données de séries temporelles les plus couramment utilisées avec Grafana, la plateforme prend en charge de nombreuses autres sources. Parmi ces sources, on compte trois serveurs spécialisés de Grafana Labs : Loki pour la gestion des journaux, Mimir pour les métriques et Tempo pour les traces Open Telemetry. Cette modularité permet à Grafana de s'adapter à divers écosystèmes et besoins.

En termes de visualisation, Grafana est dotée d'une riche bibliothèque de tableaux de bord et de widgets, comme le montre la figure 2.7. La visualisation des graphiques de nœuds, par exemple, est essentielle pour avoir un aperçu instantané de l'état et de la performance d'une grappe de serveurs. De plus, Grafana excelle dans la représentation des traces distribuées, offrant ainsi une vue d'ensemble de la performance des applications dans des architectures

microservices. Toutefois, il est à noter que Grafana n'a pas été initialement conçu pour visualiser les traces au niveau du noyau. Cependant, avec l'évolution des technologies, des initiatives ont vu le jour pour utiliser eBPF (Extended Berkeley Packet Filter) afin de générer des logs, des métriques et des traces directement exploitables par Grafana.

2.4.3 Analyse des traces

SysDig

SysDig se distingue comme un outil de surveillance puissant, axé sur la profondeur et la flexibilité. Plutôt que de se limiter à une seule source de métriques, SysDig utilise son propre module noyau et supporte également eBPF pour collecter des informations. L'utilisation d'eBPF permet à SysDig d'être moins intrusif, tout en offrant une granularité élevée des données collectées, ce qui est essentiel pour des analyses de performance approfondies.

L'un des avantages de SysDig réside dans sa capacité à fusionner et à analyser des métriques provenant de différentes sources. Par exemple, il peut intégrer des données de Prometheus, un système de monitoring et d'alerte en source libre, pour enrichir ses analyses et fournir une perspective plus complète de l'état du système.

Un autre atout majeur de SysDig est sa compatibilité avec différentes plateformes de conteneurs. Alors que certains outils, comme cAdvisor, sont principalement axés sur les environnements Docker, SysDig élargit son champ d'application en supportant également d'autres plateformes telles que LXC et rkt [22]. Cette polyvalence en fait un choix idéal pour les environnements hétérogènes.

De plus, SysDig offre *CSysDig*, une interface utilisateur en ligne de commande pour la visualisation. Cette interface permet aux administrateurs système d'avoir un aperçu rapide et interactif de l'état de leur système.

Pixie

Pixie est un logiciel libre développé par New Relics [65]. Cet outil est particulier puisqu'il combine les journaux, les métriques de Kubernetes et les traces collectées par ses programmes eBPF pour créer ses visualisations.

Pixie s'appuie sur son propre ensemble de points de trace pour produire des visualisations. Il possède ses propres programmes eBPF qui collectent des informations de trace sur la grappe de serveurs. Il peut indiquer le flux réseau, le flux des requêtes DNS, les pertes TCP et les retransmissions, à la fois globalement et pour les pods. Cela permet de surveiller le réseau et

anomalies.

Par contre, ce n'est pas une solution qui peut être mise à l'échelle puisqu'il est relativement lent pour les traces de grand format.

Trace Compass

Trace Compass est un projet développé par le laboratoire DORSAL de Polytechnique Montréal, en partenariat avec Ericsson et Ciena. C'est un outil d'analyse post-mortem qui peut traiter une variété de traces, qu'il s'agisse de traces noyau, utilisateur, distribuées ou encore d'autres formats tels que JSON ou XML. Plusieurs particularités distinguent Trace Compass des autres outils d'analyse de traces.

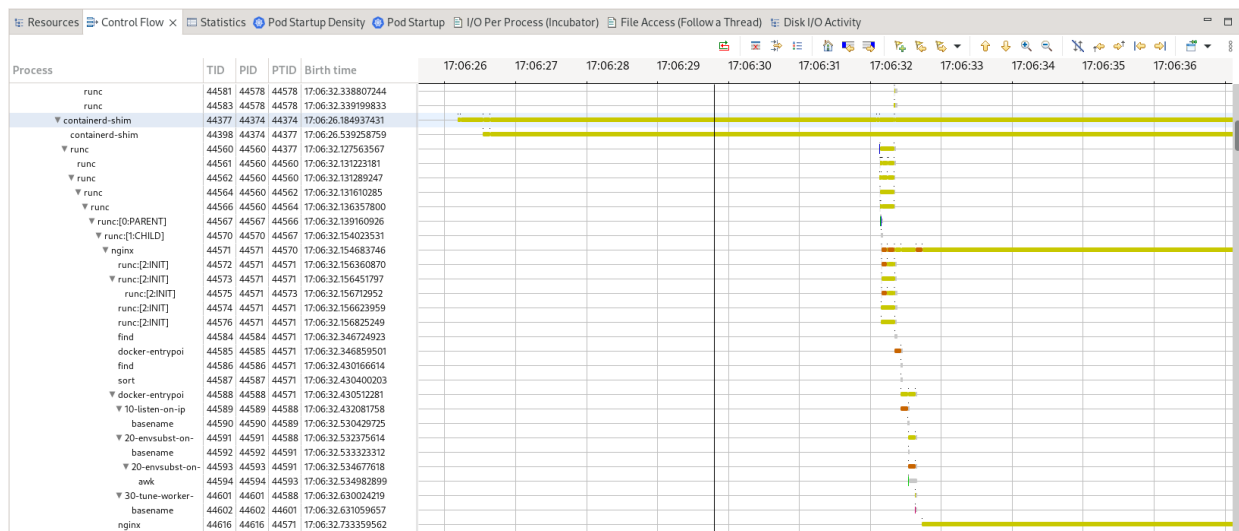


FIGURE 2.9 La vue *Control Flow* permet de visualiser la création des processus. Dans ce cas, nous pouvons voir la création d'un déploiement de *nginx* fait par *containerd* et *runc*

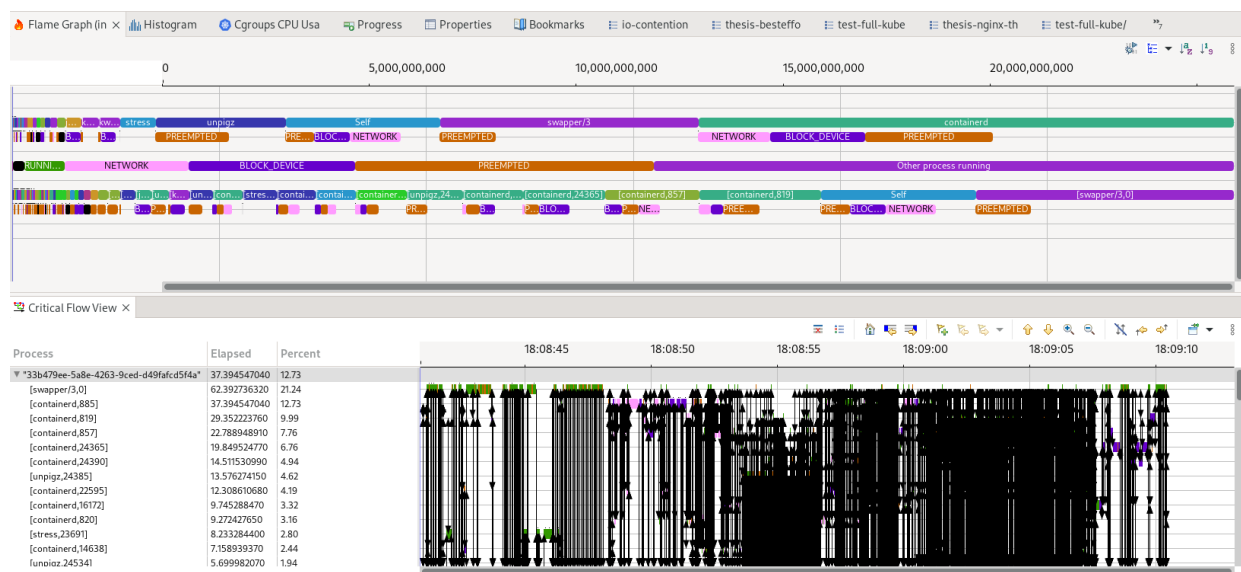


FIGURE 2.10 La vue *Critical Path Flow View* permet de visualiser en détail quels processus ont bloqué l'exécution d'un programme. *Aggregated Critical View* montre à un haut niveau qu'elles sont les processus ou ressources qui ont bloqué le processus. Dans ce que nous pouvons voir dans la figure, c'est le processus de *Containerd*.

Sa plus grande particularité, comparée aux autres outils, est son habilité à faire le calcul du chemin critique. Avec une trace de noyau, il peut montrer tous les processus qui l'ont interrompu et les ressources pour lesquelles il a attendu [67].

La deuxième particularité importante pour analyser les systèmes distribués est la synchronisation des traces. Les horloges des ordinateurs ne sont pas parfaitement synchronisées et vont généralement avoir des temps légèrement différents. Ce défaut est vu lors du traçage noyau, puisque les événements peuvent arriver à une granularité dans les nanosecondes. Pour remédier à ce problème, Trace Compass va synchroniser les traces en utilisant des événements causals comme les événements du réseau [68]. Cela permet de savoir si une horloge n'est pas synchronisée, puisque recevoir un paquet avant qu'il soit envoyé est absurde.

Troisièmement, Trace Compass a été conçu pour analyser des traces de grands volumes. Les traces noyau génèrent rapidement plusieurs Gigabites d'information. Il est alors important de pouvoir analyser cette quantité d'information rapidement. Il est capable de gérer des traces qui font des téraoctets [69]

Finalement, Trace Compass est très extensible. Il a été conçu pour générer des analyses personnalisées.

En résumé

1. Elasticsearch et la suite ELK offrent un moyen efficace de fouiller et retrouver des informations spécifiques au sein de vastes quantités de journaux.
2. SysDig est une plateforme de surveillance qui utilise sa propre instrumentation du noyau pour analyser les conteneurs.
3. Prometheus est une base de données robuste pour stocker et interroger des métriques. Il est particulièrement adapté pour s'intégrer à des systèmes d'alerte ou à des interfaces graphiques.
4. Grafana est une bonne plateforme pour visualiser des métriques et des logs avec des tableaux, des graphiques et des alertes, mais elle supporte seulement le format Open Telemetry pour les traces et n'est pas conçue pour les Traces Noyau.
5. Pixie permet d'analyser une grappe de serveurs Kubernetes avec une instrumentation transparente aux applications et permet de mieux comprendre comment ils fonctionnent.
6. Trace Compass permet d'analyser les traces noyau de grand volume efficacement. Il permet d'analyser le chemin critique des processus et synchronise les traces même si les horloges sont légèrement désynchronisées.

2.5 Conclusion de la revue critique de la littérature

La montée en puissance des conteneurs et des orchestrateurs a radicalement transformé la manière dont sont conçues, déployées et gérées les applications. Toutefois, cette transformation n'est pas exempte de défis, en particulier en ce qui concerne l'observabilité. Bien qu'il existe une myriade d'outils qui promettent une vision claire de nos systèmes, la plupart se concentrent sur des aperçus de haut niveau, sacrifiant souvent la granularité pour la simplicité et l'efficacité.

Cette approche simplifiée peut être suffisante pour une surveillance opérationnelle courante. Cependant, lorsqu'il s'agit de diagnostiquer des problèmes complexes qui traversent plusieurs couches du système, de l'orchestrateur au noyau du système d'exploitation, une vue d'ensemble de haut niveau ne suffit plus. C'est là que des outils comme LTTng et Trace Compass se montrent très utiles. Leur capacité à fournir une analyse granulaire post-mortem peut révéler des nuances et des détails qui seraient autrement manqués.

Dans le cadre de cette recherche, nous explorerons comment le traçage logiciel peut être utilisé pour diagnostiquer des problèmes de performance dans des environnements de conteneurs orchestrés. Nous utiliserons LTTng et LTTng-UST pour le traçage, dû à son faible surcoût,

et Trace Compass dû à son habileté à synchroniser les horloges. Nous utiliserons Kubernetes et containerd en raison de leur popularité.

L'objectif est de combler le fossé entre l'observabilité de haut niveau et la nécessité d'une analyse détaillée, offrant ainsi une solution complète pour la surveillance et le débogage dans des environnements d'orchestration de conteneurs.

CHAPITRE 3 MÉTHODOLOGIE

3.1 Environnement utilisé

Dans le cadre de cette recherche, toutes les simulations numériques et analyses de données ont été effectuées sur une plateforme informatique spécifique. Les spécifications détaillées de l’ordinateur utilisé sont présentées ci-dessous :

OS	Debian 12 bookworm
Kernel	x86_64 Linux 6.1.0-10-amd64
CPU	Intel Core i7-8550U @ 8x 4GHz
RAM	15738MiB

3.1.1 Environnement distribué

Pour nos expérimentations avec Kubernetes, nous avons établi et utilisé deux environnements de test via Vagrant.

Le premier est un environnement mononœud, mis en place grâce au script de développement local fourni par Kubernetes. Cette configuration permet une initiation rapide de Kubernetes sur une machine virtuelle, facilitant ainsi le traçage et le débogage. Cependant, bien que pratique pour les tests initiaux, cette version ne reflète pas l’objectif principal de Kubernetes, qui est de fonctionner dans un environnement distribué.

À cet égard, la création d’un environnement distribué a représenté un défi puisque nous devons instrumenter le noyau de l’hôte et l’orchestrateur. Des outils comme *Kind* ou *minikube* ne peuvent donc pas être utilisés, puisqu’ils n’offrent pas un hôte facilement instrumentable avec LTTng. Des outils comme *kube-init* et *kubeadm* téléchargent et exécutent le plan de contrôle dans des conteneurs, donc nous ne pouvons pas utiliser notre propre version instrumentée de Kubernetes avec ces outils. Alors, la solution a été de réutiliser une machine Vagrant basée sur le tutoriel “Kubernetes the hard way” [70] [71] afin de déployer notre machine de manière distribuée. Nous avons configuré un environnement Vagrant qui comprend trois nœuds de contrôle et trois nœuds de travail. Cette configuration reproduit un déploiement standard de Kubernetes, et permet des expérimentations plus proches des conditions de production réelles, favorisant ainsi une compréhension approfondie des interactions et des performances au sein d’un cluster Kubernetes.

3.2 Outils développés

Cette section détaille les ressources et outils développés spécifiquement pour notre recherche, qui ont facilité l'instrumentation approfondie de Kubernetes et l'analyse subséquente des données de trace.

3.2.1 Bibliothèque d'Instrumentation pour Kubernetes

Nous avons élaboré une bibliothèque dédiée, optimisée pour le traçage des activités internes de Kubernetes. Cette bibliothèque s'intègre de manière transparente avec LTTng, permettant une collecte de traces cohérente et complète pendant l'exécution de Kubernetes.

Le code source et la documentation pour la bibliothèque peuvent être trouvés ici :

<https://github.com/BenjaminSaintCyr/k8s-lttng-tpp.git>

3.2.2 Configuration des Environnements de Test

Avec un accent sur la flexibilité et la précision des tests, nous avons configuré un projet qui automatise la construction de Kubernetes instrumenté. Ce projet intègre également deux variantes d'environnements de test : un mononœud et un multinœud.

Les détails de construction et les scripts d'initialisation sont accessibles ici :

<https://github.com/BenjaminSaintCyr/thesis-project.git>

3.2.3 Visualisations dans Trace Compass

Pour compléter notre approche analytique, nous avons créé trois visualisations distinctes dans Trace Compass.

Les extensions de visualisation, ainsi que les guides pour leur utilisation, sont disponibles à cette adresse :

<https://github.com/BenjaminSaintCyr/tracecompass-incubator>

3.3 Conclusion

Ce chapitre a établi la méthodologie, l'environnement et les outils développés essentiels pour notre étude. Dans le chapitre suivant, nous appliquerons ces éléments méthodologiques à travers un article qui explore les défis de l'observabilité au sein de l'orchestration des conteneurs, et propose une approche innovante pour diagnostiquer les goulets d'étranglement de

performance.

CHAPITRE 4 ARTICLE 1 : KUBE COMPASS : FINE-GRAINED OBSERVABILITY FOR ORCHESTRATORS AND CONTAINER RUNTIME EXECUTIONS

benjamin.saint-cyr@polymtl.ca michel.dagenais@polymtl.ca Department of Computer and Software Engineering Polytechnique Montréal Montréal, QC Canada

Submitted to: Transactions on Parallel and Distributed Systems, October 25, 2023

This paper discusses the challenges faced in obtaining in-depth observability within container orchestration, and the inherent limitations in diagnosing performance bottlenecks. Using software tracing, we have developed an approach that combines the information from the scheduler of the orchestrator, and the Linux scheduler, to provide a comprehensive view of the system, essential for resolving performance bottlenecks. The methodology involves instrumenting both control and worker planes of the orchestrator. Additionally, a technique has been developed to associate trace events with Linux Cgroup namespaces, which facilitates the attribution of resource usage. This is complemented by a visualization for the lifecycle of objects in the cluster, such as Pods, deployments and services. The results are two-fold: Firstly, they present an insightful visualization that clearly depicts the status or condition of every individual object within a given cluster. Secondly, they provide a thorough examination and breakdown of how the CPU is utilized, specifically focusing on the consumption of each Cgroup. These advances offer a more effective approach to diagnosing performance issues, and pave the way for future improvements in orchestrator observability.

Orchestrator, Kubernetes, Docker, Go, Tracing, Performance Analysis

4.1 Introduction

The rapid evolution of cloud computing over the last decade has given rise to multiple technologies to manage and deploy services. At the forefront, orchestrating containerized applications has become the standard way to deploy services in the cloud. Like shipping containers, platforms such as Docker offer a seamless way to package and move a service from development to production. Containers accomplish lightweight virtualization by sharing the host kernel with the container. Using existing kernel mechanisms such as Cgroups and namespaces, it ensures containers are isolated, safeguarding them from intrusive and disruptive neighbouring processes. This technique is more lightweight than traditional virtual machines because containers do not require their own kernel; instead, they utilize the kernel of the host

system. This makes containers faster to download and execute than virtual machines. All these reasons make it a good choice for efficient and reliable deployments.

These containers are then scheduled over a cluster by an orchestrator. It is in charge of managing resources, scheduling, load balancing, scaling, and handling config maps and secrets. Several container orchestrators exist, such as Docker Swarms, Kubernetes (k8s) and K3s. However, Kubernetes has emerged as the dominant platform. Its ascendance can be attributed to three primary factors: extensive community support, superior scalability, and a robust declarative configuration system. It was created by Google as the successor of Borg [17]. It features a distributed control plane for enhanced scalability.

As the scale of a cluster grows and multiple services are deployed, the complexity of managing and understanding the system escalates dramatically. While existing tools can provide a high-level view of the cluster, they often lack the depth required to understand performance intricacies that directly impact both the quality of service and optimal resource utilization. Therefore, the need for tools that bridge the gap between the orchestration layer and the underlying kernel is clear. Such tools should facilitate in-depth diagnosis and resolution of complex issues inherent in these large, distributed systems.

LTTng enables software tracing at both kernel and user space levels [72]. This work leverages LTTng to integrate insights from the Kubernetes scheduler, with those of the Linux scheduler. This provides a comprehensive system view, essential for addressing performance issues. Our contributions are as follows:

1. **Instrumentation Enhancement of Kubernetes:** The aim is to enrich event data collection of Kubernetes, specifically aligning it with kernel events, by doing it in a way that can be correlated with kernel events. This will allow to bridge the gap between kernel and orchestrator observability. This data then can be used to troubleshoot performance issues, optimize resource utilization, and better comprehend system behaviour.
2. **Lifecycle Visualization:** Another key aim is to visually represent the lifecycle of Kubernetes objects. The visualization allows developers to easily track the progress of each object, identify bottlenecks or errors in the process, and understand the dependencies and relationships between different objects. This comprehensive view of the object lifecycle can help developers plan their deployments and make more informed decisions about resource allocation and error handling.

This dual focus aims to augment both the comprehensibility and the manageability of Kubernetes deployments, contributing to more efficient and effective system operations.

4.2 Related Work

Previous studies have looked at many levels of performance, in order to increase the efficiency of the orchestrator. From distributed tracing, to workload and low-level metrics, these studies provide critical insights into the behaviours of applications, and the allocation of resources within distributed environments. In the following sections, we delve into some of the key findings from these studies and discuss the technologies that have been instrumental in this domain.

4.2.1 Containers

The primary objective of containerization is to allow multiple tenants to share the same operating system, and isolate them from each other, to ensure that intrusive and disruptive processes cannot affect other tenants. The first effort was in 1979 with `chroot(8)` (for change root) [7, Chapter 11]. This command would change the top-level directory, as seen by the process being isolated, to prevent it from snooping through the whole file system. This approach works well for a simple task, but the rest of the system was still completely shared. In 1998, FreeBSD improved on that concept with Jails, by virtualizing the access to the file system, the set of users, and the networking subsystem [8]. In 2002, Linux implemented a similar feature called namespaces [7, Chapter 11]. This would allow a set of processes to see each other, but not any other process. There is a type of namespace for each resource, in order to give a more granular level of isolation. To provide resource isolation, control groups were introduced to Linux in 2008. In 2008 LXC Combined the two in order to create a container runtime [7, Chapter 11]. Finally, in 2013, docker arrived [7, Chapter 11]. It provided a way to compile a program and its dependencies into image, and run it on top of its runtime (originally LXC). Docker has emerged as the most popular container runtime, mostly because it provided an easy and convenient interface.

4.2.2 Containerized Orchestrators

Borg

Borg is the first container orchestrator developed by Google. It was designed to manage both long-lasting jobs and batch jobs on a single cluster [17].

Kubernetes

Kubernetes was created as an open source successor to Borg, aimed to serve as a platform for cloud-based containerized services. It improves on the Borg monolithic design by distributing the scheduling, control and storage to other systems. This allows for better scalability and resiliency. It still keeps one centralized API-Server, in order to make abstraction of the storage, and enforce object validation, defaulting and versioning [17].

4.2.3 Google Cluster-Usage Traces

While new projects use Kubernetes, all pre-existing projects at Google use Borg. Borg is still being maintained, and in 2019 Google released one month of trace data about their usage of Borg [27]. The instrumentation within the infrastructure of Google captures both cluster activities and resource usage metrics. This data is then transmitted to a collector through the gRPC protocol, as detailed in [28]. This trace data shows the real world usage of a large-scale orchestrator, using a real workload. This data set is helpful to develop autoscaling tools, like Autopilot from Google [29].

The goal of this instrumentation is mainly to observe workload patterns and trends, rather than diagnose specific performance issues.

4.2.4 cAdvisor

cAdvisor is the tool developed by Google to collect resource usage metrics for each container. It uses a library, like libcontainer and libpfm, to collect container-level resource usage metrics and performance events. These metrics can then be recorded to databases, like Prometheus or Elastic search, for monitoring and analysis. It is the standard way to collect metrics from containers, since it is shipped with every Kubelet [73].

One limitation of cAdvisor is that it collects relative metrics from /cgroup virtual files. These metrics are not very precise, and this mechanism is not the most efficient way to access this information [31] [32].

The primary emphasis of cAdvisor is on harvesting metrics directly from cgroups, rather than offering insights into the workload or events of the orchestrator. While it is integrated with Kubernetes by default, its design allows compatibility with other orchestrators as well.

4.2.5 eBPF

To collect lower-level information, a tool that has gained a lot of popularity is eBPF. eBPF is a specialized virtual machine within the Linux kernel that provides programmable hooks, allowing users to write and execute custom programs in response to various kernel events. These programs, once loaded, can be executed in response to events such as system calls, tracepoints, and more. This mechanism offers a way to extend kernel functionalities without traditional kernel modules, while ensuring the safety and efficiency of code execution through a rigorous verification processes [52, Chapter 1].

eBPF has also been used to build networking tools like load balancers and containers network interface. Cilium is an example that is meant to replace components such as iptables and Kube-proxy using eBPF [53]. There is also Katran, a load balancer developed by Meta, meant to replace a hardware load balancer [54]. They say a key advantage of eBPF is that it allows to have built-in observability.

eBPF programs can execute around system calls and can act like a tracer. Since it is programmable, it can easily be used to aggregate and record certain metrics. Compared to a tracer like LTTng, eBPF offers greater convenience and flexibility. It allows what is called transparent tracing, due to the lack of static instrumentation, and provides seamless integration. This is the main reason for the popularity of eBPF [74].

eBPF is already well supported by the Kubernetes community. There is a plugin to kubectl called kubectl trace that allows scheduling eBPF programs around the cluster [52, Chapter 15] [52, Chapter 17].

One interesting open source tool is Pixie, developed by New Relic [65]. This tool combines a Daemonset, eBPF and its own custom instrumented libraries, to give both a global and detailed view of what is happening in a Kubernetes Cluster. This approach has several advantages. The first is that the instrumentation is non-invasive. It combines information at all levels, and reuses mechanisms inside of Kubernetes, to deploy it across clusters and collect data.

eBPF, with its ability to embed programmable hooks in the kernel, has enhanced observability in the Kubernetes ecosystem. While tools like Cilium and Pixie leverage eBPF for networking and observability, there remains an unexplored potential: merging kernel-level events with user-defined Kubernetes tracepoints. Our work seeks to fill this gap, correlating kernel and user-space data for a richer understanding of system behaviours and performance bottlenecks.

4.2.6 Open Telemetry

Open Telemetry is standard for generating logs, metrics and trace. Distributed tracing does record the spans of each request, including their sub-requests, to display the execution graph of a request. This then allows to pinpoint where the latency occurs in a distributed system. This approach mainly focuses on propagating a request context, in order to trace where the request went and its duration. This information is then aggregated and analyzed in an execution graph. It is a way to avoid searching through logs on multiple machines for a specific Pod UID. Since it is a known standard, it allows various backends to collect these tracepoints.

A lot of progress has been made on adding distributed tracing to Kubernetes. Most of the work was done with Open Telemetry, but it is still under development [41] [42].

The limitation of this approach is the lack of granularity. Open Telemetry shows where requests go, and which system takes the most time to process. It does not produce information at the runtime level, or the kernel level. This can be an issue if the performance issues is due to an error at that level. Locking, CPU, disk or even network contention can be an issue for such problems and Open Telemetry does not show it.

4.2.7 Tools Developed by SIG-Instrumentation

SIG-Instrumentation is a dedicated team within the Kubernetes community that focuses on integrating observability tools into Kubernetes. Their primary aim is to enhance the monitoring and debugging capabilities of Kubernetes, enabling administrators and developers to obtain deeper insights into the state and performance of their workloads.

One tool is Kube-State-Metrics that generates metrics based on the state of each object in a Kubernetes cluster [34]. It accomplishes this task by listening to the API-Server of the cluster. These metrics range from Pods, nodes, deployments and StatefulSets. This can be integrated into the monitoring infrastructure, like Prometheus and Grafana, to get alerts.

Finally, there is Metric Server [75]. It collects resource utilization of the CPU and memory, quickly and efficiently. This data is then used in auto scaling pipelines. It is not meant to be an accurate metric.

These tools, while unintrusive, due to their reliance on the API-Server, come with inherent limitations and costs. Continuously querying vast amounts of plaintext data can escalate cloud server expenses, as highlighted in the documentation [34]. Furthermore, being confined to the metrics exposed by the API-server, along with its low clock resolution, can pose challenges when attempting to correlate these with granular low-level metrics.

4.2.8 Other tools

Additional tools commonly employed within the context of Kubernetes include Datadog, Grafana, Elasticsearch and Fluentd, among others [58]. These tools each provide unique perspectives on observability and data interpretation, yet they fundamentally depend on the same instrumentation techniques already discussed in the literature review.

4.2.9 Conclusion

The literature review indicates that most tools offer only a partial view of ongoing operations, in both the operating system and the orchestrator. This limited view complicates the debugging process when performance issues surface across the orchestration layer, container runtime, and the operating system. While containers allow for lighter virtualization by sharing the kernel, it does add more pressure to it. Factors like CPU, IO, network, and lock contention can slow down Pod startups and affect the quality of service. These types of issues occur both at the orchestrator and kernel level. A Pod being throttled or bursting are other issues that will happen at the kernel and orchestrator level.

4.3 Background

For a comprehensive understanding of the work presented, it is essential to review some foundational concepts of Kubernetes, as detailed in the official documentation [18].

The heart of Kubernetes is a cluster. This is a set of machines, termed ‘nodes’, which run containerized applications. Nodes in the cluster are categorized as either ‘worker nodes’ or ‘control nodes’. Worker nodes handle the execution of containerized workloads. Each worker node runs pods, which are the smallest deployable units in Kubernetes that can be created and managed. The control nodes oversee the worker nodes, ensuring that the cluster maintains the desired state, as defined by the user. They host the components of the control plane, orchestrating the operations of the cluster.

Not to be confused with containers, a pod can encapsulate one or more containers. Containers within a pod share the same network IP, port space, and storage, allowing them to communicate effortlessly. Importantly, pods also manage resource allocation, ensuring that each container adheres to specified CPU and memory limits. Pods are described using a declarative configuration known as a ‘manifest’.

Within the control plane, the API-server is Acting as the gatekeeper and the central communication hub for Kubernetes. It processes and validates REST requests, updating the state

of the cluster accordingly. The API-server is responsible for interpreting the manifest and instantiating the appropriate controller. This controller, a fundamental component in Kubernetes, ensures that the pod remains in its desired state, overseeing its lifecycle and managing any discrepancies between the current and desired states. The API-server will store the state of the cluster in ETCD, a highly available key-value store used by Kubernetes to maintain all cluster data, ensuring fault tolerance. When a pod is created and remains unscheduled, the responsibility of the scheduler is to assign it to a node. It determines the most appropriate node by considering factors such as resource availability, manifest constraints, scheduling policies, and strategies like bin packing, to optimize resource utilization.

The principal component in the worker plane is the kubelet. The Kubelet will constantly poll the API-Server, in order to confirm it is alive, and see if any new pods are assigned to it. If there is one, the Kubelet will create a pod with the container runtime installed on the system. Any runtime that satisfies the standard Container Runtime Interface (CRI) can be used, but the most popular is containerd.

4.4 Methodology

Against this backdrop, a more detailed understanding of Pod behaviour is crucial. A slow Pod startup not only impedes resource utilization efficiency but also constrains service scalability in response to demand. Slow Pod deletion also constitutes a significant resource drain. Identifying errors in the cluster induced by the operating system kernel is challenging, due to the fragmented observability landscape.

Containers heavily rely on kernel mechanisms, like Cgroups and namespaces, to provide resource control and isolation. Understanding which container made which system call gives a deeper insight into their performance characteristics. It will show what file they access, what CPU is used, and many more related issues.

In this article, a new solution is proposed to enhance the observability of Kubernetes, using Kernel Tracing and a bespoke instrumentation of the control plane and worker plane. This approach can collect precise events, from the kernel to the orchestrator, providing precise insight into the behaviour of Kubernetes, thereby enabling accurate identification of performance bottlenecks. An overview of our architecture is presented in Figure 4.1. All of the tracing data will be reported to a lttnng-relayd node.

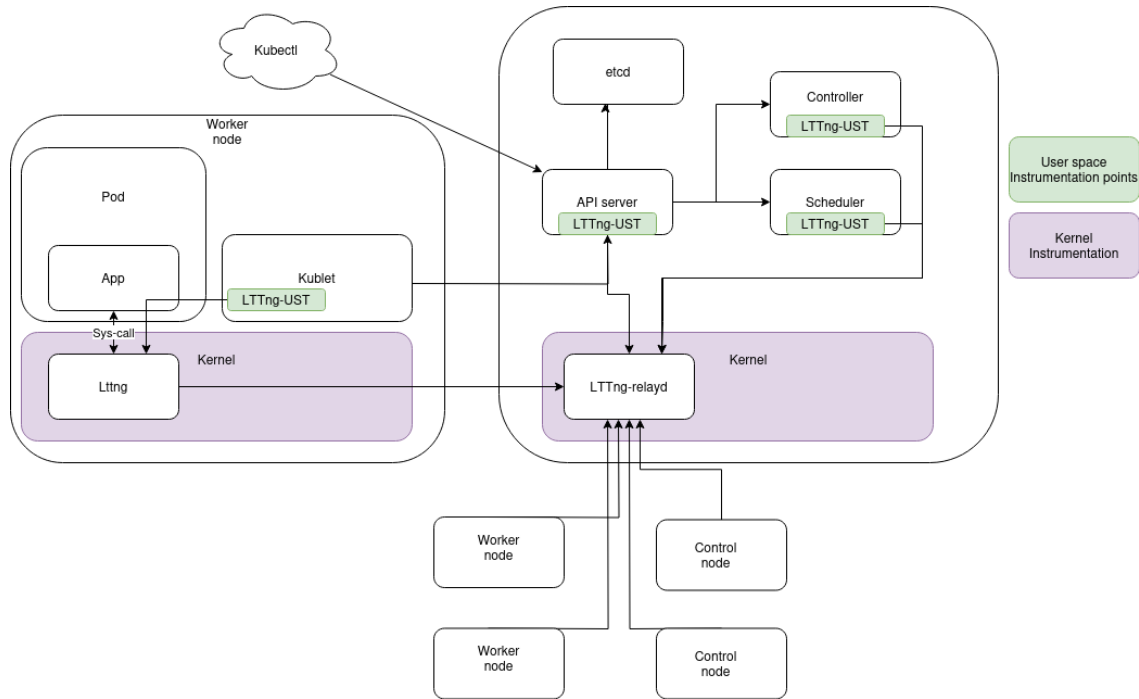


Figure 4.1 Tracing Architecture

4.4.1 Instrumenting the Orchestrator

Trace Collector

To bridge the gap between the orchestration layer and the operating system, we needed a method to correlate orchestrator events with kernel events.

Using the current Event logging framework from Kubernetes, you can expect to get events with a timestamp at the microseconds granularity [76]. This is sufficient to get a global overview of what is happening, but not to correlate with what is happening in the kernel.

Distributed tracers could be interesting to collect information. They provide a convenient way to propagate the trace context and collect the trace data. However, distributed tracers do not collect the execution context from the kernel, like the Cgroup, PID and TID. This makes it hard to correlate trace information generated by the tracer with kernel events.

Another approach would be to make fake system calls to the kernel, near user-level events [77], in order to increase the accuracy and correlate this information with other system calls. This is less intrusive way to associate kernel and user space data, but it is not very efficient and thus scalable.

LTTng-UST is a good approach to collect user-space events. It is efficient [48] and allows

correlating user-space events with kernel-space events. It integrates well with kernel traces collected with the LTTng kernel module and analysis tools like Trace Compass. LTTng is written in C and offers agents for C/C++, Java and Python. Kubernetes is written in Go, and unfortunately LTTng-UST does not directly support Go. There are several ways to use LTTng-UST in Go. The simplest is to use C bindings through CGO. In the development of our prototype, we made a deliberate configuration choice to enable CGO during the compilation of Kubernetes. While standard practice dictates that Kubernetes is compiled with CGO disabled to ensure memory safety, our customized version has it activated. Another approach would be to write events to the LTTng logger. This would enable to trace Kubernetes without using CGO. The issue is that the logger is meant for shell scripts and is not as efficient as regular tracepoints.

Ultimately CGO was used due to its ease of use and acceptable performance.

Instrumenting the Lifecycle

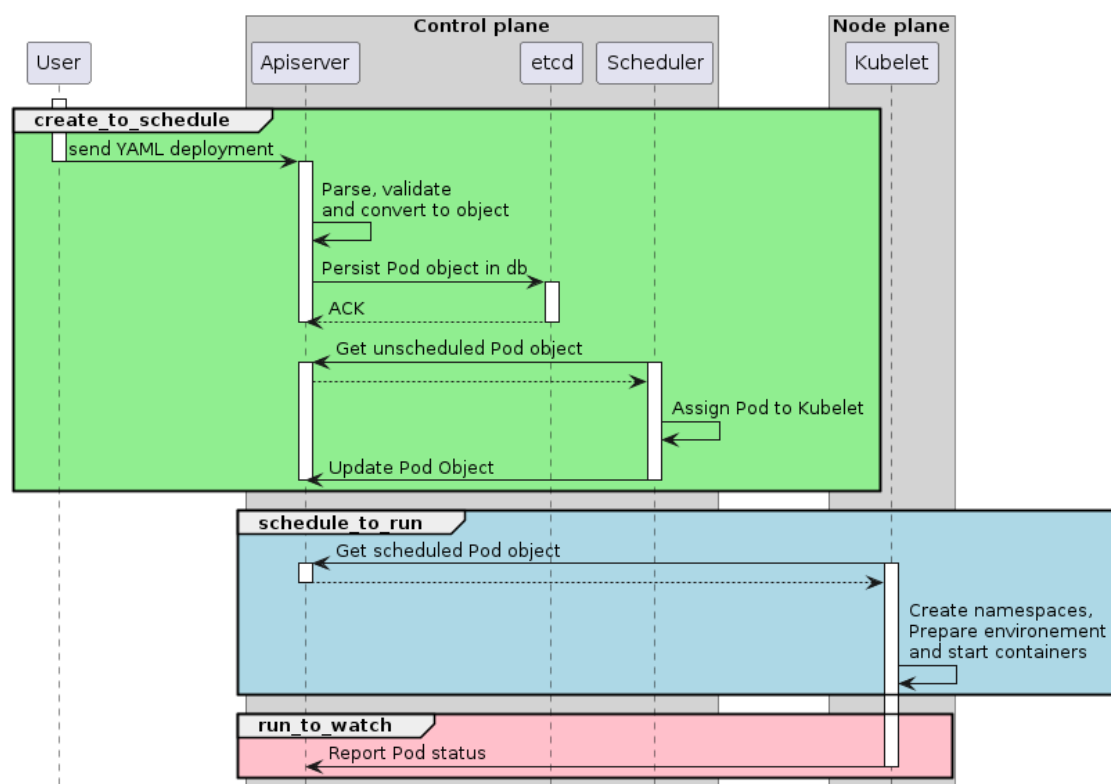


Figure 4.2 The Pod Startup Process of Kubernetes

Pods are the central object in Kubernetes. They are critical to the deployment of an application. Understanding the reasons behind a longer Pod startup time is vital for achieving

dynamic scalability, a concept often referred to as elastic scaling. Additionally, it facilitates a quicker recovery in the event of pod failures, thereby fostering system resilience and self-healing capabilities.

The Pod startup process can be divided into three primary stages, each marked by four significant events:

1. **Create Event:** Triggered when the deployment description is first received. The apiserver will parse and record this deployment in etcd until a controller reads it and asks the scheduler to create the Pod.
2. **Schedule Event:** Occurs when the Pod is assigned to a node.
3. **Run Event:** Marks when the Pod is operational
4. **Watch Event:** Represents when the Kubelet notified the apiserver that the Pod is up and running.

These events create three phases: the `create_to_schedule`, `schedule_to_run` and `run_to_watch` phase shown in Figure 4.2. These phases are used in monitoring tools such as Perf Dash by SIG-Scalability.

A lot of efforts were spent to improve introspection in Kubernetes [73]. One of these efforts is to have a standard way to record events. Each component uses the same library to report events, such as starting a pod, evicting a resource, or indicating an error. It gives a high-level overview of the startup process and general lifecycle updates. This is what tools like clusterloader2 uses to record each main event of a Pod startup. There is also standard way to record the condition of a Pod for more details on the state of the Pod.

Table 4.1 Summary of tracepoints inserted for data collection.

Tracepoint Names	Location (Module)	Purpose	Key Data Collected
Tracepoint 1	Event Logger	This is where state changes are logged	Pod Creation, Errors, Host event
Tracepoint 2	Condition Logger	This is where the pod lifecycle is logged	Pod Scheduling
Tracepoint 3	Pod Worker Loop	This is the pod controller loop	Pod termination

In the course of this research, the existing event logging framework was adapted to accommodate the `ltng-ust` tracepoint providers. The first logging framework is the event logger. It logs every event that can represent a state change [76]. The second logging framework is

the pod condition that tells if it is managed by the Kubelet, to know if the pod has or not met certain criteria [78]. These two loggers in Kubernetes successfully capture a considerable portion of the internal state of each Kubernetes object; however, there are still areas for improvement. One of these challenges pertains to context propagation. Detailed data about the objects being logged, such as their ownership or finalization, is not readily available to these loggers.

Moreover, certain events, like the termination of a Pod, are not currently included. To address this, such an event had to be manually integrated into the custom instrumentation. Therefore, while the existing logging infrastructure provides a foundational understanding, additional customization and enhancements were necessary, to provide a more comprehensive coverage of Kubernetes operations and events. A summary of the tracepoints is shown in Table 4.1.

4.4.2 Collecting Trace Data per Cgroup

Another frequent issue with Pods is settings up correct resources usages. Without a specified limit, each pod competes for resources, leading to potential contention between latency-sensitive pods and CPU-intensive ones. If the limit is too low, this will introduce too much latency, since the Pod will be throttled during the critical execution.

A common misconception arises from how limits are specified in Kubernetes, versus their enforcement by the Linux CFS scheduler. It is a common issue discussed in this talk with AWS [2] and this internal document at Twitter [3].

Kubernetes offers three classes of Quality of Service (QoS), based on the CPU request and limits. The request value indicates the desired amount of CPU resources for the pod, and the pod can utilize more CPU resources if available on the system. In contrast, the limit value delineates the maximum amount of CPU resources that the pod is allowed to use. The three classes are: BestEffort, Burstable and Guaranteed [19].

BestEffort is for when no resource requests or limits are specified.

Burstable is when the request is lower than the limit or the limit is a fraction. It enforces its CPU request and limits with CPU cgroup that uses the cfs bandwidth control as the underlying throttling mechanism. Since the request is a soft limit, it can be exceeded if the host CPU is idle, but it will not go over the limit.

Guaranteed is when the request and the limit are the same as the request, which is the case by default if the limit is not specified. It can use the Cpusets cgroup that will enforce the limit by allocating a whole CPU to the Pod, but only if the CPU-manager-policy of the

Kubelet is configured to static at the Kubelet startup. Otherwise it will also use the CPU cgroup.

The configuration issue comes with the CPU cgroup. Since the limit specified in Kubernetes is in *cores*, and the limit enforced in the kernel is in *time*, this can lead to some misconceptions.

There are 4 files in the cgroupfs that control and report statistics about the CPU controller [79]:

1. **cpu.cfs__period__us**: Scheduling period. After this period of time elapses, the CPU quotas will reset
2. **cpu.cfs__quota__us**: Runtime allocated within a period. A negative value means there is no bandwidth restriction.
3. **cpu.cfs__burst__us**: How much unused bandwidth the cgroup can accumulate to use in another period. A value of 0 indicates no accumulation is allowed.
4. **cpu.stat**: Periods, Burst and Throttling statistics

Given these definitions, we can derive the following:

1. **Quota Calculation**: The quota determined by the CPU limits set in the configuration can be calculated as:

$$Quota = \text{Cores Limit} \times \text{Period}$$

This equation represents how the quota scales linearly with the number of specified cores limit for a given period.

2. **Throttling Determination**: If the number of cores used exceeds the limit, throttling occurs. The time for which throttling will happen is given by:

$$Throttled\ Time = \text{Period} \times (\text{Cores Used} - \text{Cores Limit})$$

This equation highlights the proportional relationship between exceeded core usage and the time the CPU is throttled.



The app needed 100ms
The app was throttled 120ms

Figure 4.3 Example of Latency Introduced by CPU Limit

This leads to possible misunderstandings when setting limits. Consider a scenario where a service has been allocated 400 millicores, with the default period of 100 ms.

Firstly, we determine the effective quota:

$$Quota(ms) = Cores \times Period(ms)$$

$$Quota = 0.4 \times 100 = 40 \text{ ms}$$

Thus, with a 40 ms quota, if a request takes 100 ms to process, it will utilize its entire quota in the first 40 ms. After this, the service will experience throttling for the remaining duration of the period (i.e., 60 ms) until the quota is reset.

This can be visualized in terms of latency:

$$Latency \text{ Due To Throttling} = Request \text{ Time} \times (Cores \text{ Used} - Cores \text{ Limit})$$

$$Latency = 100 \text{ ms} \times (1 - 0.4) = 60 \text{ ms}$$

The total time taken, including the throttling, becomes:

$$Total \text{ Latency} = Request \text{ Time} + Latency \text{ Due To Throttling}$$

$$Total \text{ Latency} = 100 \text{ ms} + 60 \text{ ms} = 160 \text{ ms}$$

This additional latency, as demonstrated in Figure 4.3, underscores the importance of correctly configuring core limits. If, for example, the quota was set at 1000 millicores, the request could be serviced without any throttling-induced latency. This issue could also have been solved by increasing the period to 250 ms, but you can not configure it from Kubernetes.

Expectation		Reality			
Running	Running	Running	PREEMPTED	Running	PREEMPTED
Running	Running	Running	PREEMPTED	Running	PREEMPTED
Idle	Idle	Running	PREEMPTED	Running	PREEMPTED
Idle	Idle	Running	PREEMPTED	Running	PREEMPTED
Idle	Idle	Running	PREEMPTED	Running	PREEMPTED
Idle	Idle	Running	PREEMPTED	Running	PREEMPTED
Idle	Idle	Running	PREEMPTED	Running	PREEMPTED
Idle	Idle	Running	PREEMPTED	Running	PREEMPTED
Idle	Idle	Running	PREEMPTED	Running	PREEMPTED

Figure 4.4 Example of Latency Due to Parallelization

The second misconception can arise when an application uses its own thread pool, and selects the number of threads based on the number of cores, like Apache httpd does. The issue is that if one allocates 2000 millicores, this will be translated into a 200 ms quota within a 100 ms period. If there are 8 cores, then this will be translated into 25 ms per CPU, as shown in Figure 4.4. The CPU bandwidth used by each process in the Cgroup will be accounted every 5 ms. The interval can be adjusted in the `procs` [79]. If the request takes more than 25 ms to be processed, it will have to wait 75 ms before it is able to continue.

This becomes an issue when you introduce a horizontal autoscaler based on the latency of the request. The latency added by throttling can trigger scaling needlessly to more instances. A notable real example of this involved an AWS client who went from 73,000 Pods to 4 after fixing this issue [2].

Each container in a Pod shares the same Cgroup. Therefore, grouping scheduling events by Cgroup is interesting, to better understand when a Pod was throttled due to the limit. Most approaches to collect the Cgroup data do collect relative information reported by virtual files in the `/cgroup` directory. This approach is simple and relies on mechanisms native to most languages. The issue is that it is an inefficient approach. It is also less precise, due to the update delay. In response to these limitations, we propose an alternative approach. We augment the context of each system call trace event with the Cgroup namespace. This method offers detailed and up-to-date insights into CPU usage for individual Cgroups, providing a more granular perspective on resource utilization. However, there are a few inherent limitations with this approach. A prominent one is the inability to collect throttling information solely from system calls. Throttling data is accounted for in the `cpu.stat` virtual file. While additional events could potentially capture this information, it would necessitate kernel instrumentation, which is a more complex and invasive solution. That being said, it is easy to deduce throttling, since the process is periodically waiting for CPU, despite the CPU being idle. Moreover, this approach can also be used to associate Cgroup namespace ids with Pods UIDs. Docker relies on `bind` mount, `volumes` and `tmpfs` mount [80] which

all rely on the mount system call. All these directories reuse the Pod uid and container id as a naming convention. Therefore, by searching for an uid, we can associate it with the Cgroup namespace that called it. The advantage of this approach is that one does not need to instrument the container runtime.

This enhances the versatility and practicality of our tool, providing a richer and more informative view of resource usage, within Kubernetes environments.

4.4.3 Trace Compass Analysis

Given the importance of understanding thread interactions and synchronizing data from multiple sources, our methodology leans heavily on Trace Compass. Trace Compass provides two useful features in our case: The Critical Path Analysis and The Trace Synchronization.

Critical Path Analysis in Trace Compass [67] involves tracking the sequence of steps that a process takes to completion. This feature not only presents the states the process goes through, but also highlights the interruptions by other processes and further details about those interrupting processes. By using the aggregated view of this information, it provides a comprehensive view of the resources a process awaits, aiding in identifying potential bottlenecks or slowdowns.

Trace Synchronization addresses the challenge of aligning traces from different machines, which might have unsynchronized clocks [68]. In a distributed system context, this is indispensable. Without such synchronization, analyzing performance across a distributed environment would become considerably more complex.

4.5 Implementation and Results

4.5.1 Environment and Data Collection

Our Kubernetes environment was established on multiple virtual machines, using VirtualBox as the VM provider, Ubuntu Jammy as the operating system, and Vagrant for provisioning. This setup afforded us considerable flexibility with the operating system, enabling the deployment of our custom Kubernetes version v1.27.3 and easily installing version 2.13.1-1 of lttng-modules-dkms. We used a single node to analyse the trace data, and 3 controller nodes and 3 worker nodes to measure the overhead.

To collect the trace data, we used LTTng-relayd on the first controller node, as illustrated in Figure 4.1.

4.5.2 Instrumentation added

As outlined in Table 4.1, 3 tracepoints were added using our Go LTTng-UST library. Reusing the event logging, the pod condition and extra tracepoints in the event loop of the pod worker, we are able to get a detailed insight into the state of Kubernetes.

What sets these custom tracepoints apart is their precision. They are backed by a high-resolution clock that is synchronized with kernel events, allowing for precise timing analysis. Moreover, they offer an enriched context, capturing invaluable details, such as the process ID and the name of the originating process.

Additionally, kernel-space instrumentation was carried out using LTTng-modules. The inclusion of the `cgroup_ns` context to these kernel tracepoints ensured a better understanding of cgroup behaviour.

4.5.3 Lifecycle Analysis

Understanding the different lifecycle stages of a Kubernetes Pod can be challenging. When the cluster is under load, the extended Pod startup time, possibly due to resource contention and other underpinning issues, is a recognized challenge in the Kubernetes ecosystem [81]. While kernel tracing has been proposed as a potential remedy, the gap between the practical operations of Kubernetes and the kernel is far from trivial [82].

The introduction of a lifecycle analysis offers a structured framework to demystify this. As depicted in Figure 4.6, the essence of this analysis is to provide a granular understanding of the Pod startup process. Equipped with this refined insight, developers can discern which Pods are in the latency tail, identify the phases consuming the most time, and investigate kernel activities accordingly. The Lifecycle analysis not only unveils the ‘what’ of Pod operations but also delves deeply into the ‘why’, addressing concerns such as extended startup times, repeated failures, or evictions.

There are two approaches to visualize the Pod startup. One could see the perspective of each component of Kubernetes (Api-Server, Scheduler, Kubelet, etc.), but this approach is not abstract enough. Most developers are only aware of the objects generated by their deployments. It becomes increasingly complex to analyze a large deployment, since everything is done asynchronously. The state of each component constantly changes and deployments would have to be tracked through dependency arrows, which only add to the complexity.

Our approach then is to show the state of each object in spans, and position them in the hierarchy based on their owner. The lifecycle analysis shows in which state the object was, and is colour coded based on the phase it is in. The phases are: Scheduling, Creating,

Running, Failed and Unknown. It enables to figure out which process is responsible for each phase, and analyze their critical path in Trace Compass. In the visualization, the hierarchy of objects needs to be represented, to better understand the relationships with objects. In Kubernetes, objects can have owners in their metadata to help differentiate parts, for example the Pods created by the deployment. This information is collected by the instrumentation, and will be represented in the visualization. A deployment can have thousands of objects. It is hard to manually find the outliers in such a large number of spans. A complementary density chart was created to help find the outliers. This way, developers can pinpoint the pod that took the most time, and then see in the lifecycle analysis which phase took the most time and when. A developer can then see which stateful, controller and deployment set it belongs to, and what processes were involved.

Logs can record these state changes, but they do not offer the same level of clock precision as tracing. They also are not synchronized with the kernel and do not collect kernel information, like PIDs and TIDs. This makes tracing more precise and easier to associate with kernel traces, for deeper analysis.

4.5.4 Cgroup CPU Usage Analysis

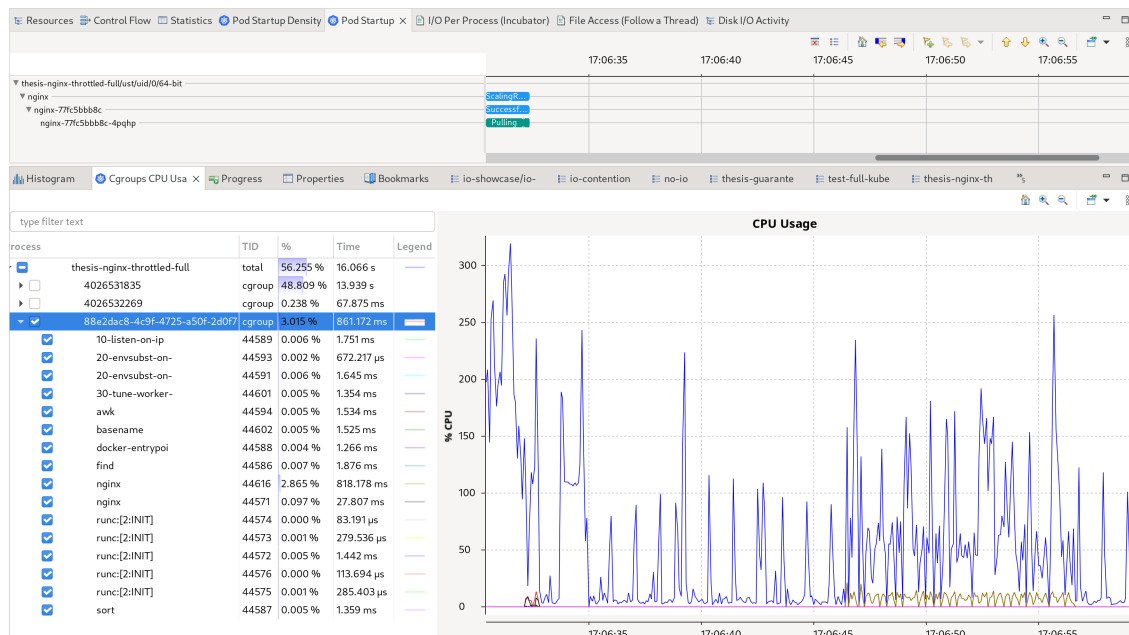


Figure 4.5 CPU usage per Cgroup graph

Managing resources efficiently in a Kubernetes environment is crucial to ensure consistent application performance. The Kubernetes Pod, which bundles one or more containers, serves

as the unit for enforcing resource allocation policies. Limits set too restrictively will cause CPU throttling, affecting application performance. Conversely, excessively liberal limits can result in resource contention between Pods, a phenomenon known as bursting.

To shed light on these issues, we propose a dedicated CPU usage analysis, visualized as a XY chart like the one shown in Figure 4.5. This graphical representation allows monitoring the CPU utilization at a global level as well as for individual Cgroups (control groups). By indicating the percentage of CPU time consumed by each Cgroup, relative to the total CPU time, this visualization provides a comprehensive overview of resource usage patterns. The state of the CPU is either on or off, depending on the point in time. The percentage is calculated by doing the sum when the CPU was on or off. Each process is then grouped by its Cgroups namespace.

Algorithm 1 CPU usage calculation per Cgroup

```

1: Define the function normalize(prevTime, time, cpuTime) as  $\frac{\text{cpuTime}}{\text{time} - \text{prevTime}} \times 100$ 
2: procedure ANALYZECPUUSAGE(fetchParameters, stateSystem)
3:   Initialize totalCpu  $\leftarrow 0$ 
4:   Initialize prevTime  $\leftarrow$  Get the start time from fetchParameters
5:   Initialize xAxis  $\leftarrow$  Initialize xAxis based on the time range and resolution
6:   Initialize map threadUsage
    $\leftarrow$  Initialize the cpu usage of each thread based on stateSystem
7:   Initialize map CgroupUsage
    $\leftarrow$  Initialize the cpu usage of each cgroup based on stateSystem
8:   for i  $\leftarrow 0$  to xAxis.length - 1 do
9:     time  $\leftarrow$  xAxis[i]
10:    for all (threadName, cpuTime)  $\in$  fetchThreadsCpuUsage(time) do
11:      totalCpu  $\leftarrow$  totalCpu + cpuTime
12:      normalizedThreadCpuUsage  $\leftarrow$  normalize(prevTime, time, cpuTime)
13:      threadUsage[threadName][i]  $\leftarrow$  normalizedThreadCpuUsage
14:      if TidToCgroup.contains(threadName) then
15:        key  $\leftarrow$  TidToCgroup.get(threadName)
16:        CgroupUsage[key][i]  $\leftarrow$  CgroupUsage[key][i] +
          normalizedThreadCpuUsage
17:      end if
18:    end for
19:    prevTime  $\leftarrow$  time
20:  end for
21:  return CgroupUsage, threadUsage
22: end procedure

```

This tool can be instrumental in identifying CPU throttling scenarios. When the CPU usage for a particular Cgroup is observed to be unexpectedly low, despite demanding workloads,

it might be indicative of CPU throttling, due to overly restrictive resource limits. Similarly, instances where a single Pod dominates the CPU utilization can be easily spotted, allowing developers to diagnose and address resource hogging issues.

By juxtaposing this view with the data on Pod lifecycles and systemwide CPU utilization, one can correlate high CPU usage or CPU throttling instances with specific Pods, and the workloads they are running. This could reveal whether certain tasks are especially CPU-intensive, whether specific Pods are inefficiently using CPU resources, or whether resource limits need to be adjusted to better reflect the demands of the workloads.

Comparatively, while other tools such as cAdvisor provide a measure of CPU usage, they may not offer the granular, per-Cgroup visibility provided by this approach. This level of details, coupled with the ability to visualize data over time, can be valuable in diagnosing performance issues and optimizing resource allocation strategies.

Instrumentation Overhead

In the instrumentation, there are two primary facets: the cost associated with recording events, and the latency induced by all the logged events. Using the built-in benchmarking framework in Go [83], the data suggests that the time required to record a single event falls between 600 ns and 900 ns. Furthermore, when using clusterloader2, to gauge the latency during pod startups, the overhead observed is minor, approximately 3.1%.

4.5.5 Use Cases

Diagnosing Slow Pod Startup

During load testing at scale, some Pod startup can be affected by resource contention and various other issues. There is currently an issue raised by the developers of Kubernetes about this [81]. Kernel Tracing was a proposed solution, but the gap between Kubernetes and the kernel is wide [82]. The tool proposed in this paper aims to respond to such issues.

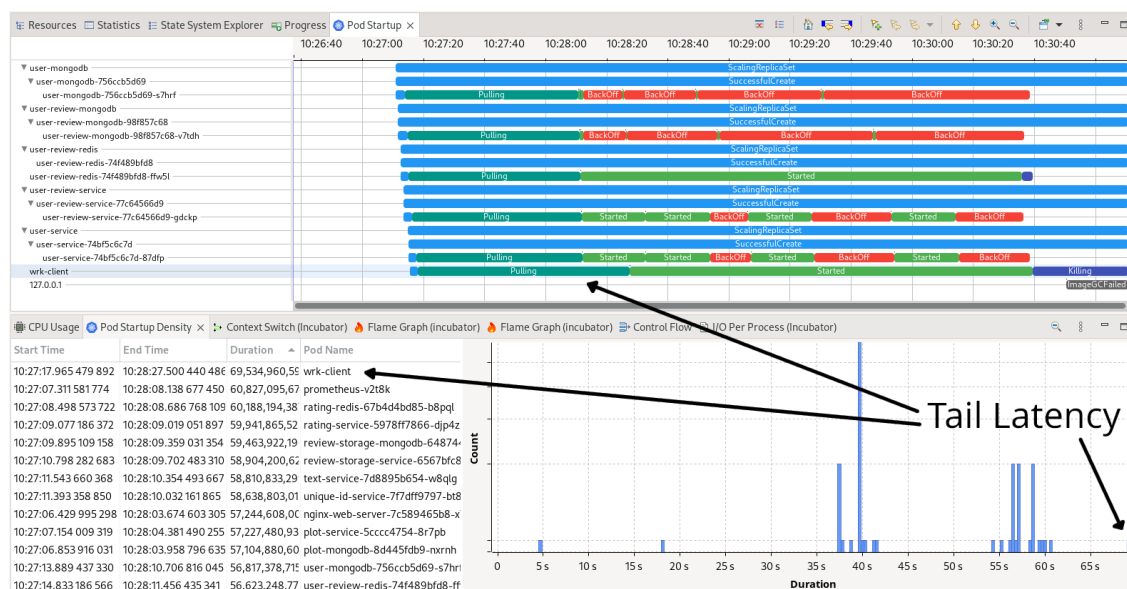


Figure 4.6 Lifecycle Analysis

The goal of the lifecycle analysis is to explain the Pod startup process with more granularity. Using the Pod Startup Density view, as shown in Figure 4.6, a developer can start identifying which Pods belong to the tail latency. Once a Pod is selected, the developer can investigate it in the lifecycle Analysis above in Figure 4.6. It can first find which phase took the most time. Once the phase is identified, then the developer can further investigate the cause by inspecting what was happening in the kernel. It could also use this lifecycle view to analyze why a Pod keeps failing, or why it was evicted.

Common factors that can impact slow Pod startups in Kubernetes is multitenancy and pulling images. Having to pull an image will increase the time needed for deployment. Multitenancy can introduce unpredictable workloads, leading to Pods waiting for resources in shared cluster environments. Once these issues are identified, then they can be addressed by the scheduler of Kubernetes. The insight collected by the analyses can then be used to tune the scheduler, by prioritizing image locality or other metrics.

Latency Issues Due to the CPU Limits

A common issue in Kubernetes is optimizing CPU limits. Setting a limit that is too low will result into throttling, and this will cause latency issues. Setting the limit to high may result in CPU contingency between Pods, also called bursting.

CPU limits can easily be improperly configured due to a misconception about limits. Limits are specified as cores in Kubernetes, but are enforced by the CFS bandwidth control as

quotas. Therefore, if one core is specified, but the application is multithreaded, then all the threads will use up the quota, and the Pod will be throttled. This particularly becomes an issue when your request takes more time than what is allocated to it. If the hard limit is too low, the Pod will wait for CPU, even if the CPU is available. This artificially limits the Pods. This issue will snowball if the horizontal autoscaling is based on the requests per seconds. The Horizontal autoscaler will then create more Pods to respond to this artificial limitation. Gaining deeper insight into the scheduler of Linux allows the developer to better understand the resource usage and how it affects a deployment.

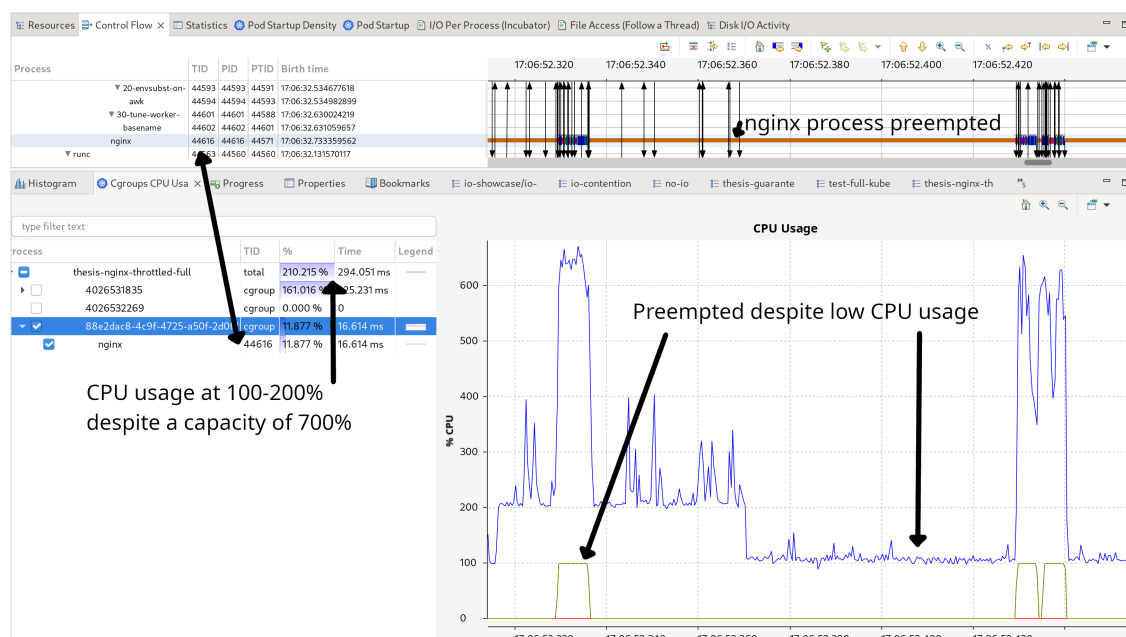


Figure 4.7 nginx server being throttled

Having a visualization that shows the resource utilization for each Cgroup is interesting to diagnose latency issues caused by the CPU. Looking at Figure 4.7, you get a global and local understanding of the system. If the total CPU time is 100%, then it is clear that some bursting is happening. When a Cgroup is periodically shutdown, even if the CPU is not at full capacity, it hints that the CPU is being throttled. To better help the developer diagnose throttling, developers can look at the control flow view, to see the time *waiting for CPU*. It then becomes obvious that the execution was limited by the CPU usage. When we can see that, on top of this, the CPU is available, then it becomes clear that the Pod was being throttled, and the limits should be rearranged. Figure 4.7 illustrates the impact of throttling on latency. Using a simple nginx deployment, and using *hey* to generate load, we recorded the difference between throttling and no throttling. Unthrottled, this deployment was able

to reply within 1.7 ms. With a 80 millicores limit, and 40 millicores request, the response time increased to 37 ms. Kernel tracing shows this issue. Despite the CPU being mostly idle, the process was waiting for the CPU. This view also shows the time and percentage used by each Cgroup, enabling the developer to quickly identify which Cgroup hogs the resources.

Latency Issues Due to Resource Contention

Other views of Trace Compass can be reused in the context of Kubernetes. One of them is the Aggregated Critical Path view. Once a slow request or a slow Pod startup is identified, a developer can use the critical path view to dig deeper into the issue. Looking at the critical path of a process, a developer can identify what took the most time. It will show if it was waiting for the CPU, I/O, Networking, Locking, or if it was just slow to execute. Then, there is the critical path flow of the process to diagnose the slow CPU. There is also a view that shows the I/O for each process.

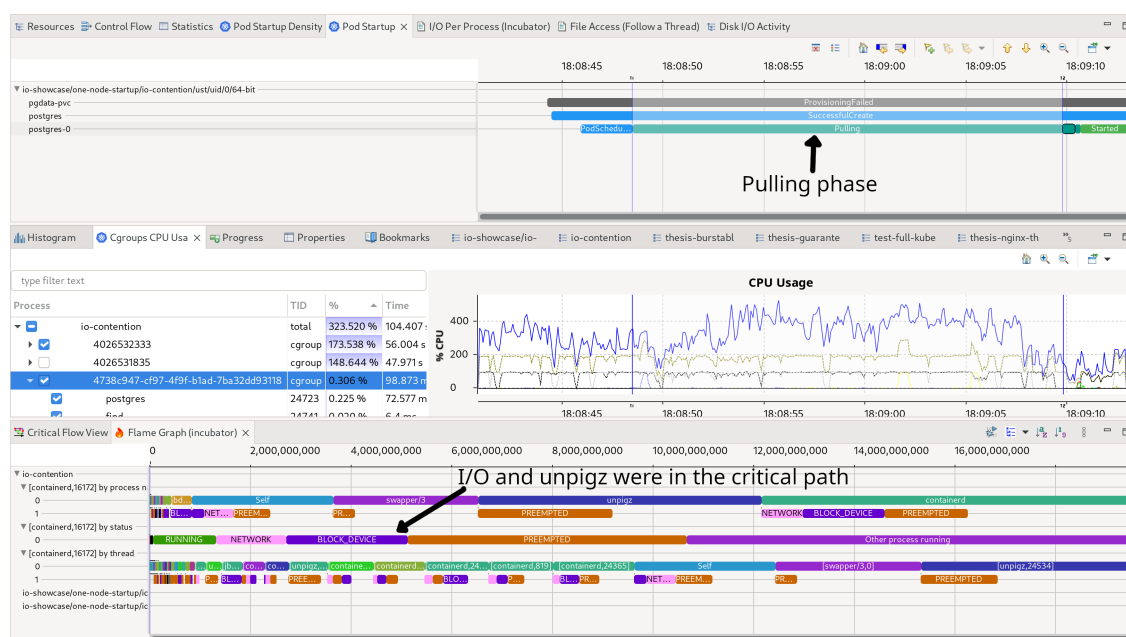


Figure 4.8 Slow Pod Startup Caused by I/O Contention

A specific concern arises from slow startups due to block I/O contention. One limitation with Kubernetes is its lack of support for the block I/O Cgroup [7]. This becomes problematic as the scheduler, by default, prioritizes CPU and memory, when determining a suitable node. To illustrate this, a PostgreSQL deployed container was used. Given that this container processes around 500 MB and is I/O-bound, it becomes notably sensitive to I/O slowdowns. To exacerbate the condition, an I/O-heavy process using the 'stress' program was deployed,

targeting the disruption of the PostgreSQL deployment. Under typical circumstances, this Pod starts in about 16 seconds. However, due to the I/O contention, the startup time stretched to 21 seconds. Employing our instrumentation and Trace Compass, we can delve deeper into the reasons for this delay. As depicted in Figure 4.8, a significant portion of the delay arises from pulling the container image. Upon closer examination of the critical path of the container runtime, it becomes evident that the process experienced longer I/O wait times, compared to network waits.

4.6 Discussion and Future Work

Bridging the gap between the orchestration layer and the operating system layer is an interesting topic. The limitations with this approach is the availability of tracepoints.

There is some current work from EfficOS on supporting more languages using libraries like *libside* and an ABI for the languages runtime. For instance, a trace would be written from GO statements, instead of going through the CGO interface. This will reduce the overhead, and support more language natively.

The methodology used to associate cgroups to system calls can be reused for analysing network-related system calls. Kubernetes is the favourite choice for microservices and they rely heavily on the network. Combining this work with Open Telemetry and network traces will be interesting to diagnose performance bottlenecks based on the network.

Our work mainly focuses on the critical path of object scheduling in Kubernetes, and their impact on the CPU. It will be interesting to trace more the control plane, to give a more in-depth image of the scheduling. This will be helpful for the developers of Kubernetes.

Modifying the Linux kernel is usually not something accepted by the cloud provider. This is problematic, since the approach relies on kernel traces to identify throttling. This is not an issue for our industrial partners, since they host Kubernetes themselves.

4.7 Conclusion

The challenge of managing application performance in a Kubernetes environment is multifaceted, with factors spanning across the orchestration and operating system layers. The research presented in this paper has taken a significant step towards bridging this gap, proposing a methodology that provides a comprehensive and deeper understanding of these complex performance issues.

The cornerstone of our approach is an integrated approach to observe Kubernetes from the

perspective of the kernel. This combination allows us to not only identify the phases that slow down Pod deployment, but also to understand the causes of such delays, be it CPU throttling, resource contention, or lock contention. These insights are crucial for both developers and system administrators, enabling them to optimize their deployments more effectively and efficiently.

Furthermore, our approach demonstrates the potential of investigating performance from both the orchestration layer (Kubernetes) and the operating system layer (kernel). This combined perspective offers a more holistic view of the performance of the system, providing an opportunity to identify and address bottlenecks, that may be overlooked when these layers are considered in isolation.

The approach and findings detailed in this paper contribute to the broader understanding of Kubernetes performance, and serve as a stepping stone towards more advanced and integrated tools for Kubernetes performance diagnosis. Furthermore, the proposed methodology underscores the value of a combined view of the orchestration and operating system layers, for effective and efficient Kubernetes resource management, as shown by the real-world use cases shown earlier.

In the future, we believe that the methodology developed in this study will have broad applicability beyond the specific use cases we have considered. The potential for investigating other aspects of system performance, such as networking and storage, using a similar approach, is strong. With the increasing adoption of Kubernetes in the industry, tools and methodologies such as those proposed in this paper will be instrumental in ensuring the reliable, efficient, and scalable operation of distributed applications.

CHAPITRE 5 DISCUSSION GÉNÉRALE

Dans ce chapitre, nous discuterons de nos contributions, des résultats obtenus, et des perspectives pour des travaux futurs.

Notre travail a permis d'introduire une instrumentation supplémentaire et des points de trace dans Kubernetes, facilitant ainsi l'observation du cycle de vie des objets au sein de Kubernetes. Cela permet également de diagnostiquer les phases qui ont nécessité le plus de temps. L'instrumentation a un impact négligeable sur la performance. Cela s'intègre harmonieusement avec un traceur de noyau comme LTTng, ce qui permet de synchroniser les événements de l'orchestrateur jusqu'au niveau du noyau.

Nous avons proposé une méthodologie permettant de calculer l'utilisation des ressources par chaque *cgroup*. Cette technique fournit un éclairage précis sur la consommation des ressources CPU, se révélant utile pour déceler les restrictions de ressources qui pourraient compromettre la qualité de service.

Enfin, nous avons créé deux visualisations distinctes pour améliorer la compréhension du cycle de vie des conteneurs et de leur consommation de ressources CPU. La première visualisation offre une vue d'ensemble du cycle de vie des conteneurs, mettant en lumière les différentes phases par lesquelles ils passent depuis leur création jusqu'à leur destruction. Elle permet également d'identifier rapidement les phases qui ont eu un impact significatif sur le déploiement, contribuant ainsi à une identification efficace des goulots d'étranglement potentiels. La seconde visualisation se concentre sur l'utilisation des ressources CPU par les conteneurs, en fournissant des détails granulaires sur la consommation du CPU au fil du temps. Elle révèle notamment les limitations de ressources et les conteneurs les plus gourmands, permettant ainsi aux opérateurs et aux développeurs de mieux comprendre et optimiser la performance des applications hébergées sur Kubernetes.

Nous avons démontré que ces outils sont capables de diagnostiquer des problèmes de performance couramment observés dans l'industrie, tels que les déploiements lents et la gestion inefficace des ressources.

Ces travaux ont le potentiel d'assister les fournisseurs de services Kubernetes dans le diagnostic approfondi des problèmes de performance. De plus, ils pourraient s'avérer utiles pour les développeurs de Kubernetes cherchant à identifier des problèmes de performance spécifiques lors des tests à grande échelle.

La revue de littérature révèle que les outils existants se limitent souvent à une perspective

fragmentée des couches système. En contraste, notre approche transcende les normes établies pour la collecte de métriques, offrant une analyse à la fois plus affinée et détaillée de l'utilisation des ressources par les conteneurs. Cette précision accrue dévoile des aspects du système souvent omis ou négligés, permettant une compréhension plus profonde et une gestion plus efficace des performances au sein des environnements de conteneurs.

Créer un environnement de test complet, allant de l'instrumentation au déploiement, en passant par la charge de travail et l'analyse des résultats, a représenté un défi majeur. Ce travail, en plus de contribuer à l'effort d'instrumentation, de construction et de déploiement de Kubernetes, marque une étape fondamentale dans l'analyse des orchestrateurs distribués au sein des environnements LTTng et Trace Compass. Il ouvre la voie à des investigations plus approfondies sur ce sujet ou d'autres.

CHAPITRE 6 CONCLUSION

Dans ce dernier chapitre, nous revisiterons les buts que nous nous étions fixés au début de ce travail, discuterons des limitations de notre étude, et réfléchirons sur la pertinence et l'efficacité de la solution proposée. De plus, des perspectives de travaux futurs seront évoquées, afin d'approfondir et d'enrichir davantage la valeur et l'intérêt de notre projet.

6.1 Synthèse des travaux

L'objectif principal de nos efforts était d'instrumenter un orchestrateur, en l'occurrence Kubernetes, en utilisant des outils de traçage logiciel tels que LTTng-UST et LTTng-Modules. Cette instrumentation avait pour finalité de générer des visualisations sur une plateforme comme Trace Compass. Le but ultime de cette démarche étant de diagnostiquer des problèmes de performance surgissant à différents niveaux, du noyau jusqu'à l'orchestrateur lui-même.

Dans la revue de la littérature, nous avons discuté des approches prises en industrie pour collecter et analyser l'information, et avons conclu qu'il n'y avait pas encore de solution qui combinait l'information de bas niveau et de haut niveau avec un aussi grand niveau de granularité et de précision que nous avons réalisé avec des outils comme Trace Compass et LTTng.

Dans le Chapitre 3 nous avons discuté de notre environnement de test et comment nous avons expérimenté avec des grappes de serveurs sur un seul nœud, et des nœuds avec un contrôleur ou un travailleur afin de donner une utilisation réaliste de notre projet.

Finalement dans l'article au chapitre 4, nous avons discuté de notre approche et de nos résultats. Nous avons démontré comment notre outil peut résoudre des problèmes vus en industrie et rencontrés par la communauté Kubernetes.

En somme, ce travail contribue significativement à approfondir l'observabilité des grappes de serveurs Kubernetes, offrant ainsi une vision plus précise et détaillée de leur fonctionnement et performance. Cette avancée est cruciale pour le diagnostic précis et l'optimisation des performances dans des environnements complexes et distribués.

6.2 Limitations de la solution proposée

La solution proposée dans ce travail de recherche présente certaines limitations importantes à prendre en considération.

La première limitation notable concerne la librairie de traçage utilisée, LTTng-UST. Malgré les avantages indéniables qu'elle offre, surtout lorsqu'elle est combinée à LTTng-modules, LTTng-UST n'a pas été initialement conçue pour s'intégrer avec un langage récent tel que Go. Cette contrainte représente un obstacle à l'intégration de notre instrumentation directement dans le code source de Kubernetes. Par contre, du travail intéressant est en cours de réalisation chez EfficiOS, avec la librairie *libside* qui permettra de créer un traceur qui s'intègre mieux avec différents langages.

La deuxième limitation de notre travail est liée à l'emploi de LTTng-modules. Bien que ces modules s'intègrent efficacement avec des outils d'analyse comme Trace Compass, ils se présentent sous forme de modules du noyau. Ceci peut être un inconvénient lorsque les modules noyau sont désactivés. Il faut donc avoir accès à chaque nœud Kubernetes afin d'instrumenter l'orchestrateur.

6.3 Améliorations futures

Plusieurs pistes d'améliorations futures peuvent être envisagées pour approfondir et optimiser le travail réalisé dans ce domaine de recherche.

Des travaux complémentaires pourraient se concentrer sur le calcul du chemin critique lié à différentes opérations, de l'orchestrateur jusqu'au noyau. Cette démarche impliquerait l'utilisation des points de trace rajoutés dans Kubernetes. L'objectif serait d'obtenir une compréhension plus approfondie des interactions et dépendances critiques de l'orchestrateur jusqu'au sein des opérations du noyau.

Il serait également pertinent d'explorer comment les informations obtenues du noyau peuvent guider et optimiser les décisions d'ordonnancement et de mise à l'échelle automatique. Des stratégies d'ordonnancement plus éclairées pourraient améliorer significativement l'efficacité et la réactivité des déploiements Kubernetes, en réponse aux besoins fluctuants des systèmes et des applications.

Une extension naturelle de notre recherche concerne l'analyse approfondie des différentes ressources contrôlées par les cgroups. En adaptant la méthodologie présentée, on pourrait étendre l'observation au-delà de la consommation de CPU, incluant des ressources telles que la mémoire, les entrées-sorties disque, et le réseau. Cette analyse enrichie, exploitant les points de trace statiques du noyau, pourrait déceler des éléments clés pour l'optimisation des performances dans les déploiements Kubernetes. Ceci permettrait de révéler des opportunités d'amélioration souvent négligées dans les stratégies actuelles de gestion des ressources.

Ces perspectives d'améliorations futures visent non seulement à renforcer la compréhension

des mécanismes internes de Kubernetes et du noyau, mais aussi à développer des outils et méthodologies permettant de gérer et optimiser de manière proactive les environnements orchestrés complexes.

RÉFÉRENCES

- [1] L. Screenshots, “grafana dashboard,” janv. 2016. [En ligne]. Disponible : <https://www.flickr.com/photos/xmodulo/24311604930/>
- [2] S. Corbett et W. Reed, “KubeCon + CloudNative North America 2022 : 73,000 Pods a Day, Lessons From Misadven...” oct. 2022. [En ligne]. Disponible : <https://kccncna2022.sched.com/event/182D5/73000-pods-a-day-lessons-from-misadventures-in-multi-tenant-shane-corbett-amazon-web-services-wil-reed-acquia?iframe=no&w=100%&sidebar=yes&bg=no>
- [3] D. Mackey et D. Luu, “The container throttling problem,” avr. 2019. [En ligne]. Disponible : <https://danluu.com/cgroup-throttling/>
- [4] “Case Studies.” [En ligne]. Disponible : <https://kubernetes.io/case-studies/>
- [5] A. M. Potdar, N. D. G. S. Kengond et M. M. Mulla, “Performance Evaluation of Docker Container and Virtual Machine,” *Procedia Computer Science*, vol. 171, p. 1419–1428, janv. 2020. [En ligne]. Disponible : <https://www.sciencedirect.com/science/article/pii/S1877050920311315>
- [6] Z. Zhuang, C. Tran, J. Weng, H. Ramachandra et B. Sridharan, “Taming memory related performance pitfalls in linux Cgroups,” dans *2017 International Conference on Computing, Networking and Communications (ICNC)*, janv. 2017, p. 531–535.
- [7] B. Gregg, *Systems Performance*, 2^e éd. Boston : Pearson, déc. 2020.
- [8] “Chapter 17. Jails.” [En ligne]. Disponible : <https://docs.freebsd.org/en/books/handbook/jails/>
- [9] O. Flauzac, F. Mauhourat et F. Nolot, “A review of native container security for running applications,” *Procedia Computer Science*, vol. 175, p. 157–164, janv. 2020. [En ligne]. Disponible : <https://www.sciencedirect.com/science/article/pii/S187705092031704X>
- [10] R. Pike, D. Presotto, K. Thompson, H. Trickey et P. Winterbottom, “The use of name spaces in plan 9,” dans *Proceedings of the 5th workshop on ACM SIGOPS European workshop : Models and paradigms for distributed systems structuring*, ser. EW 5. New York, NY, USA : Association for Computing Machinery, sept. 1992, p. 1–5. [En ligne]. Disponible : <https://dl.acm.org/doi/10.1145/506378.506413>
- [11] E. W. Biederman et L. Network, “Multiple instances of the global linux namespaces,” dans *Proceedings of the Linux Symposium*, vol. 1. Citeseer, 2006, p. 101–112.
- [12] “namespaces(7) - Linux manual page.” [En ligne]. Disponible : <https://man7.org/linux/man-pages/man7/namespaces.7.html>

- [13] T. Heo, “Control Group v2 — The Linux Kernel documentation,” oct. 2015. [En ligne]. Disponible : <https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html#resource-distribution-models>
- [14] M. Moravcik, P. Segec, M. Kontsek, J. Uramova et J. Papan, “Comparison of LXC and Docker Technologies,” dans *2020 18th International Conference on Emerging eLearning Technologies and Applications (ICETA)*, nov. 2020, p. 481–486.
- [15] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah et P. Merle, “Elasticity in Cloud Computing : State of the Art and Research Challenges,” *IEEE Transactions on Services Computing*, vol. 11, n^o. 2, p. 430–447, mars 2018.
- [16] X. Wang, J. Du et H. Liu, “Performance and isolation analysis of RunC, gVisor and Kata Containers runtimes,” *Cluster Computing*, vol. 25, n^o. 2, p. 1497–1513, avr. 2022. [En ligne]. Disponible : <https://doi.org/10.1007/s10586-021-03517-8>
- [17] B. Burns, B. Grant, D. Oppenheimer, E. Brewer et J. Wilkes, “Borg, Omega, and Kubernetes,” *ACM Queue*, vol. 14, p. 70–93, 2016. [En ligne]. Disponible : <http://queue.acm.org/detail.cfm?id=2898444>
- [18] “Kubernetes Components,” juill. 2023, section : docs. [En ligne]. Disponible : <https://kubernetes.io/docs/concepts/overview/components/>
- [19] “Control CPU Management Policies on the Node,” mai 2023. [En ligne]. Disponible : <https://kubernetes.io/docs/tasks/administer-cluster/cpu-management-policies/>
- [20] D. Ongaro et J. Ousterhout, “In search of an understandable consensus algorithm (extended version),” dans *Proceeding of USENIX annual technical conference, USENIX ATC*, 2014, p. 19–20.
- [21] “How nodes work,” 0200. [En ligne]. Disponible : <https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/>
- [22] A. Bhardwaj et C. R. Krishna, “Virtualization in Cloud Computing : Moving from Hypervisor to Containerization—A Survey,” *Arabian Journal for Science and Engineering*, vol. 46, n^o. 9, p. 8585–8601, sept. 2021. [En ligne]. Disponible : <https://doi.org/10.1007/s13369-021-05553-3>
- [23] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker et I. Stoica, “Mesos : A platform for {Fine-Grained} resource sharing in the data center,” dans *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, 2011.
- [24] “etcd versus other key-value stores,” août 2021, section : docs. [En ligne]. Disponible : <https://etcd.io/docs/v3.3/learning/why/>

- [25] X. Li, Y. Jiang, Y. Ding, D. Wei, X. Ma et W. Li, “Application Research of Docker Based on Mesos Application Container Cluster,” dans *2020 International Conference on Computer Vision, Image and Deep Learning (CVIDL)*, juill. 2020, p. 476–479. [En ligne]. Disponible : https://ieeexplore.ieee.org/abstract/document/9270483?casa_token=0Tvr_PsXwNkAAAAA:G_jTbD32aFa5dj7NBfkar44rKZUmPKJeQhauQtXJlkS9ugKFwGehe4ZZ_zuLi5wsLEJ3TletqRSh
- [26] C. Sridharan, “4. The Three Pillars of Observability - Distributed Systems Observability [Book],” juill. 2018, iSNB : 9781492033424. [En ligne]. Disponible : <https://www.oreilly.com/library/view/distributed-systems-observability/9781492033431/ch04.html>
- [27] M. Tirmazi, A. Barker, N. Deng, M. E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter et J. Wilkes, “Borg : the Next Generation,” dans *EuroSys’20*, 2020, place : Heraklion, Crete.
- [28] J. Wilkes, “Google cluster-usage traces v3,” août 2020. [En ligne]. Disponible : https://drive.google.com/file/d/10r6cnJ5cJ89fPWCgj7j4LtLBqYN9RiI9/view?usp=embed_facebook
- [29] K. Rzađca, P. Findeisen, J. Świdorski, P. Zych, P. Broniek, J. Kusmierrek, P. K. Nowak, B. Strack, P. Witusowski, S. Hand et J. Wilkes, “Autopilot : Workload Autoscaling at Google Scale,” dans *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020. [En ligne]. Disponible : <https://dl.acm.org/doi/10.1145/3342195.3387524>
- [30] “google/cadvisor,” sept. 2023. [En ligne]. Disponible : <https://github.com/google/cadvisor>
- [31] E. Casalicchio et V. Perciballi, “Auto-Scaling of Containers : The Impact of Relative and Absolute Metrics,” dans *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, sept. 2017, p. 207–214.
- [32] E. Casalicchio, “A study on performance measures for auto-scaling CPU-intensive containerized applications,” *Cluster Computing*, vol. 22, n^o. 3, p. 995–1006, sept. 2019. [En ligne]. Disponible : <https://doi.org/10.1007/s10586-018-02890-1>
- [33] “What is the meaning of housekeeping_interval? · Issue #2496 · google/cadvisor.” [En ligne]. Disponible : <https://github.com/google/cadvisor/issues/2496>
- [34] “Overview,” juill. 2023. [En ligne]. Disponible : <https://github.com/kubernetes/kube-state-metrics>
- [35] C. Fontaine, C. Lamoriniere et A. Mezghani, “Our Journey Taking Kubernetes State Metrics to the Next Level,” oct. 2021. [En ligne]. Disponible : <https://www.datadoghq.com/blog/engineering/our-journey-taking-kubernetes-state-metrics-to-the-next-level/>

- [36] P. Barham, A. Donnelly, R. Isaacs et R. Mortier, “Using magpie for request extraction and workload modelling,” dans *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6*, ser. OSDI’04. USA : USENIX Association, déc. 2004, p. 18.
- [37] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan et C. Shanbhag, “Dapper, a Large-Scale Distributed Systems Tracing Infrastructure,” Google, Inc., Rapport technique, 2010. [En ligne]. Disponible : <https://research.google.com/archive/papers/dapper-2010-1.pdf>
- [38] “OpenTelemetry.” [En ligne]. Disponible : <https://opentelemetry.io/>
- [39] “OpenZipkin · A distributed tracing system.” [En ligne]. Disponible : <https://zipkin.io/>
- [40] “Introduction.” [En ligne]. Disponible : <https://www.jaegertracing.io/docs/1.49/>
- [41] D. Ashpole, “Alpha in Kubernetes v1.22 : API Server Tracing,” sept. 2021. [En ligne]. Disponible : <https://kubernetes.io/blog/2021/09/03/api-server-tracing/>
- [42] “Traces For Kubernetes System Components,” mars 2023. [En ligne]. Disponible : <https://kubernetes.io/docs/concepts/cluster-administration/system-traces/>
- [43] “Distributed Tracing.” [En ligne]. Disponible : <https://istio.io/latest/docs/tasks/observability/distributed-tracing/>
- [44] Y.-T. Wang, S.-P. Ma, Y.-J. Lai et Y.-C. Liang, “Analyzing and Monitoring Kubernetes Microservices based on Distributed Tracing and Service Mesh,” dans *2022 29th Asia-Pacific Software Engineering Conference (APSEC)*, déc. 2022, p. 477–481, iISSN : 2640-0715. [En ligne]. Disponible : <https://ieeexplore.ieee.org/document/10043222>
- [45] B. Gregg, “The flame graph,” *Communications of the ACM*, vol. 59, n°. 6, p. 48–57, mai 2016. [En ligne]. Disponible : <https://dl.acm.org/doi/10.1145/2909476>
- [46] “Flame Graphs.” [En ligne]. Disponible : <https://www.brendangregg.com/flamegraphs.html>
- [47] “ddprof,” sept. 2023, original-date : 2022-01-18T14 :02 :54Z. [En ligne]. Disponible : <https://github.com/DataDog/ddprof>
- [48] M. Gebai et M. R. Dagenais, “Survey and Analysis of Kernel and Userspace Tracers on Linux : Design, Implementation, and Overhead,” *ACM Computing Surveys*, vol. 51, n°. 2, p. 26 :1–26 :33, mars 2018. [En ligne]. Disponible : <https://dl.acm.org/doi/10.1145/3158644>
- [49] M. Desnoyers et M. R. Dagenais, “The lttng tracer : A low impact performance and behavior monitor for gnu/linux,” dans *OLS (Ottawa Linux Symposium)*, vol. 2006. Citeseer, 2006, p. 209–224.

- [50] M. Desnoyers et M. Dagenais, “LTTng : Tracing across execution layers, from the hypervisor to user-space,” dans *Linux symposium*, vol. 101, 2008.
- [51] M. Jeanson, “A follow-up on LTTng container awareness,” févr. 2019. [En ligne]. Disponible : https://archive.fosdem.org/2019/schedule/event/containers_lttng/
- [52] B. Gregg, *BPF Performance Tools*, 1^{er} éd. Hoboken : Addison-Wesley Professional, déc. 2019.
- [53] “Cilium - Cloud Native, eBPF-based Networking, Observability, and Security.” [En ligne]. Disponible : <https://cilium.io>
- [54] N. Shirokov et D. Ranjeeth, “Open-sourcing Katran, a scalable network load balancer,” mai 2018. [En ligne]. Disponible : <https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer/>
- [55] D. Wendlandt, “Grafana and Cilium : Deep eBPF-powered observability for Kubernetes and cloud native infrastructure,” oct. 2022. [En ligne]. Disponible : <https://grafana.com/blog/2022/10/24/grafana-and-cilium-deep-ebpf-powered-observability-for-kubernetes-and-cloud-native-infrastructure/>
- [56] F. C. Eigler, “Problem Solving With Systemtap.”
- [57] “Choosing a Linux Tracer (2015).” [En ligne]. Disponible : <https://www.brendangregg.com/blog/2015-07-08/choosing-a-linux-tracer.html>
- [58] “Cloud Native Landscape.” [En ligne]. Disponible : <https://landscape.cncf.io/>
- [59] A. Bialecki, R. Muir, G. Ingersoll et L. Imagination, “Apache lucene 4,” dans *SIGIR 2012 workshop on open source information retrieval*, 2012, p. 17.
- [60] “Elasticsearch : What is it and why it’s important - AWS.” [En ligne]. Disponible : <https://aws.amazon.com/what-is/elasticsearch/>
- [61] “Elasticsearch Platform — Find real-time answers at scale.” [En ligne]. Disponible : <https://www.elastic.co/>
- [62] Prometheus, “Overview | Prometheus.” [En ligne]. Disponible : <https://prometheus.io/docs/introduction/overview/>
- [63] A. Huynh et I. Babrou, “Hidden Linux Metrics with Prometheus {eBPF} Exporter,” 2018. [En ligne]. Disponible : <https://www.usenix.org/conference/lisa18/presentation/huynh>
- [64] O. Mart, C. Negru, F. Pop et A. Castiglione, “Observability in Kubernetes Cluster : Automatic Anomalies Detection using Prometheus,” dans *2020 IEEE 22nd International*

Conference on High Performance Computing and Communications ; IEEE 18th International Conference on Smart City ; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), déc. 2020, p. 565–570.

- [65] “About Pixie \textbar How Pixie uses eBPF.” [En ligne]. Disponible : <https://docs.px.dev/>
- [66] “Babeltrace · A trace manipulation toolkit.” [En ligne]. Disponible : <https://babeltrace.org/>
- [67] F. Giraldeau et M. Dagenais, “Wait Analysis of Distributed Systems Using Kernel Tracing,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, n°. 8, p. 2450–2461, août 2016.
- [68] B. Poirier, R. Roy et M. Dagenais, “Accurate offline synchronization of distributed traces using kernel-level events,” *ACM SIGOPS Operating Systems Review*, vol. 44, n°. 3, p. 75–87, août 2010. [En ligne]. Disponible : <https://doi.org/10.1145/1842733.1842747>
- [69] A. Montplaisir-Gonçalves, N. Ezzati-Jivan, F. Wininger et M. Dagenais, “State History Tree : An Incremental Disk-Based Data Structure for Very Large Interval Data,” dans *2013 International Conference on Social Computing*, sept. 2013, p. 716–724.
- [70] “kelseyhightower/kubernetes-the-hard-way : Bootstrap Kubernetes the hard way on Google Cloud Platform. No scripts.” [En ligne]. Disponible : <https://github.com/kelseyhightower/kubernetes-the-hard-way/tree/master>
- [71] “kinvolk-archives/kubernetes-the-hard-way-vagrant : A port of Kelsey Hightower’s “Kubernetes the Hard Way” tutorial to Vagrant. – By the Kinvolk team.” [En ligne]. Disponible : <https://github.com/kinvolk-archives/kubernetes-the-hard-way-vagrant/tree/master>
- [72] “Features.” [En ligne]. Disponible : <https://ltnng.org/features/>
- [73] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune et J. Wilkes, “Large-scale cluster management at Google with Borg,” dans *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2015, place : Bordeaux, France.
- [74] C. Cassagnes, L. Trestioreanu, C. Joly et R. State, “The rise of eBPF for non-intrusive performance monitoring,” dans *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, avr. 2020, p. 1–7.
- [75] “Kubernetes Metrics Server,” juill. 2023. [En ligne]. Disponible : <https://github.com/kubernetes-sigs/metrics-server>
- [76] “Event,” avr. 2023. [En ligne]. Disponible : <https://kubernetes.io/docs/reference/kubernetes-api/cluster-resources/event-v1/>

- [77] A. K. Dubey et V. Mishra, “Performance analysis of cloud applications using cloud analyst,” dans *2017 7th International Conference on Communication Systems and Network Technologies (CSNT)*, nov. 2017, p. 79–84.
- [78] “Pod Lifecycle,” avr. 2023, section : docs. [En ligne]. Disponible : <https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/>
- [79] “CFS Bandwidth Control — The Linux Kernel documentation.” [En ligne]. Disponible : <https://docs.kernel.org/scheduler/sched-bwc.html>
- [80] “Bind mounts,” juill. 2023. [En ligne]. Disponible : <https://docs.docker.com/storage/bind-mounts/>
- [81] “Eliminate latency phase from density test · Issue #1311 · kubernetes/perf-tests.” [En ligne]. Disponible : <https://github.com/kubernetes/perf-tests/issues/1311>
- [82] Marcel Zięba, “2023-05-25 Kubernetes SIG Scalability Meeting,” juin 2023. [En ligne]. Disponible : <https://www.youtube.com/watch?v=tNMOgPgZ9jQ>
- [83] “testing package - testing - Go Packages.” [En ligne]. Disponible : <https://pkg.go.dev/testing#hdr-Benchmarks>