Université de Lille
Faculté des Sciences et Technologies

Comprendre et Améliorer la Résilience des Applications Conteneurisées par l'Observabilité et la Réduction des Appels Système

Zakaria ELKHAYARI

Master Informatique
Master mention Informatique

Département d'Informatique
Faculté des Sciences et Technologies

juin, 2024

Ce mémoire satisfait partiellement les pré-requis du module de Mémoire de Master, pour la 2e année du Master mention Informatique.

Candidat: Zakaria ELKHAYARI, Nº 41916263,
`zakaria.elkhayari.etu@univ-lille.fr`

Encadrant(e): Adrien Luxey, `adrien.luxey@inria.fr`

Département d'Informatique
Faculté des Sciences et Technologies
Campus Cité Scientifique, Bât. M3 extension, 59655 Villeneuve-d'Ascq

juin, 2024

Résumé

Les architectures basées sur des microservices, déployées à l'aide de conteneurs, offrent flexibilité et efficacité, mais introduisent des défis en termes de résilience et de performance. Ce mémoire vise à comprendre l'impact des appels système sur la résilience des environnements conteneurisés et à proposer des solutions pour améliorer cette résilience. L'étude s'appuie sur l'application du chaos engineering pour tester la robustesse des systèmes et sur l'analyse des travaux de Simonsson et al. pour classifier et atténuer les défaillances des appels système. Les objectifs incluent l'identification des défaillances critiques, l'analyse des approches actuelles de détection et d'analyse pour améliorer la résilience des applications conteneurisées. En combinant ces approches, cette étude vise à fournir une compréhension accrue des solutions pratiques pour minimiser les interruptions de service et assurer des performances fiables même en cas de défaillances en s'appuyant sur l'observabilité des appels système.

Mots-clés : Microservices, Conteneurisation, Résilience, Appels système, Observabilité, Kubernetes, Docker, eBPF, Chaos engineering, ChaosOrca.

Abstract

Microservice-based architectures, deployed using containers, offer flexibility and efficiency, but introduce challenges in terms of resilience and performance. In this master thesis we aim to understand the impact of system calls on the resilience of containerized environments, and to propose solutions for improving this resilience. The study is based on the application of chaos engineering to to test the robustness of systems, and on the analysis of the work of Simonsson et al. to classify and mitigate system call failures. Objectives include identification of critical failures, analysis of current approaches to detec- and analysis approaches to improve the resilience of containerized applications. By combining these approaches, this study aims to provide a better understanding of practical solutions for minimizing service interruptions and ensuring- reliable performance even in the event of failures, based on the observability of system system calls.

Keywords: Microservices, Containerization, Resilience, System calls, Observability, Kubernetes, Docker, eBPF, Chaos engineering, ChaosOrca.

Remerciements

Je tiens à exprimer ma profonde gratitude à toutes les personnes qui ont contribué à la réalisation de ce mémoire.

Tout d'abord, je remercie chaleureusement mon tuteur, M. Adrien Luxey, pour son encadrement, sa disponibilité, et ses conseils avisés tout au long de ce projet. Son expertise et son soutien ont été essentiels à la réalisation de ce travail. Merci pour avoir toujours été disponible pour répondre à mes questions et m'avoir guidé avec patience et persévérance.

Je tiens également à remercier les auteurs de l'étude principale sur laquelle se base une grande partie de ce mémoire, en particulier Simonsson et al. Leur recherche approfondie et leurs contributions sur l'observabilité et le chaos engineering ont été d'une grande valeur pour mes travaux. Leur étude a fourni une base solide et un cadre théorique indispensable à l'élaboration de ce mémoire.

Enfin, je souhaite exprimer ma gratitude à l'ensemble des professeurs et des membres du département d'informatique de l'Université de Lille pour leurs enseignements et leur soutien tout au long de ma formation.

Indice

1 Introduction **5**

2 Fondements Théoriques et Contexte **9**

3 Importance et Analyse des Appels Système **13**

3.1 Rôle et Définitions 13

3.2 Défis liés aux Appels système 15

3.3 Impact sur les Applications Conteneurisées 16

4 Résilience et Évaluation de la Robustesse dans les Environnements Conteneurisés **19**

4.1 Observabilité et Gestion des Incidents 19

4.2 Application du chaos engineering 21

4.3 Etude de cas : Observabilité et Injection de fautes via ChaosOrca . . 23

4.4 Automatisation et Auto-guérison pour la Résilience 29

5 Réduction des Appels Système pour Améliorer la Résilience **33**

5.1 Principes de la Réduction des Appels Système 33

5.2 Méthodes de Réduction des Appels Système 35

5.3 eBPF pour la Sécurité 38

6 Conclusion **43**

Références **46**

Chapitre 1

Introduction

Dans le paysage technologique actuel, les applications conteneurisées jouent un rôle crucial en offrant flexibilité et efficacité, tout en isolant les applications de leur environnement sous-jacent [1]. Cette isolation permet une meilleure portabilité et une gestion plus efficace des ressources [2], mais elle ne vient pas sans défis. En particulier, la résilience de ces systèmes est une préoccupation majeure, car ils sont vulnérables aux défaillances et aux retards des appels système, ce qui peut gravement compromettre leur performance [3] [4]. Ces vulnérabilités nécessitent une analyse approfondie pour développer des solutions qui améliorent la stabilité et la fiabilité des applications conteneurisées. L'incident d'*OVH-Cloud* à Strasbourg en 2021, où un incendie a révélé des faiblesses majeures en termes de redondance et de récupération d'urgence, a dramatiquement souligné la nécessité de renforcer les stratégies de résilience [5]. Cet événement a mis en lumière les risques liés à la dépendance aux infrastructures centralisées et a incité à une réévaluation critique des pratiques de résilience parmi les fournisseurs de services cloud [5].

Les applications modernes évoluent rapidement vers des architectures distribuées pour mieux répondre aux exigences de scalabilité, de résilience, et de flexibilité. Parmi ces architectures, les microservices, notamment lorsqu'ils sont déployés dans des systèmes conteneurisés, se distinguent par leur capacité à transformer des applications monolithiques en ensembles de services plus petits et indépendants [6]. Cette approche, bien que prometteuse, introduit également une complexité accrue dans la gestion des communications inter-services, la cohérence des données et la résilience aux défaillances [3]. Les systèmes conteneurisés ajoutent une couche supplémentaire

de gestion des ressources, de surveillance, et d'automatisation [4]. Ces technologies offrent des solutions puissantes pour la gestion des applications à grande échelle, mais elles introduisent également des défis uniques liés à la performance et à la sécurité [3].

Dans ce contexte, l'observabilité des systèmes conteneurisés devient cruciale. Une visibilité granulaire sur les appels système et les interactions réseau est essentielle pour détecter et résoudre les goulots d'étranglement ainsi que les anomalies potentielles [7]. L'importance de l'observabilité est renforcée par l'utilisation de techniques avancées comme le Chaos Engineering, qui permet de tester la résilience des systèmes en simulant des défaillances réelles [8]. Ces tests permettent d'observer le comportement des systèmes dans des scénarios similaires à ceux rencontrés en production, notamment face à des perturbations telles que des retards et des erreurs [9].

Ce mémoire est motivé par la nécessité croissante de garantir des applications plus robustes et performantes, capables de résister aux perturbations et d'optimiser l'utilisation des ressources. En apportant une meilleure compréhension et des outils efficaces pour la gestion des appels système, ce mémoire vise à contribuer à l'avancement des pratiques en matière de développement et d'orchestration des microservices [4].

Objectifs de l'étude Les objectifs de cette étude sont structurés autour de trois axes principaux :

1. **Identifier et classer les défaillances des appels système :** Cette première étape consiste à recenser et à catégoriser les différents types de défaillances et de retards des appels système pouvant survenir dans des environnements conteneurisés. Une analyse approfondie des erreurs de permissions, des fichiers ou répertoires introuvables, des défaillances matérielles, des interruptions de processus, des fonctionnalités non implémentées, ainsi que des défis de latence et de retards sera réalisée. Pour cela, nous nous appuierons sur l'étude de Simonsson et al.[7], qui fournit un cadre détaillé pour comprendre l'impact de ces défaillances sur la résilience des systèmes conteneurisés.
2. **Analyser les approches existantes pour la détection et l'analyse des défaillances :** La deuxième étape consiste à examiner les techniques actuelles utilisées pour détecter et analyser ces défaillances et retards. Cela inclut l'évaluation des méthodes de surveillance et d'observabilité des systèmes, l'application du chaos engineering pour tester la résilience des systèmes, et l'utilisation d'outils comme **eBPF** pour une surveillance granulaire des appels système. L'application du chaos engineering, en particulier, permet d'introduire des perturbations contrôlées afin de tester la capacité des systèmes à résister aux défaillances en simulant des scénarios réalistes de pannes.

3. **Développer et évaluer des meilleures pratiques pour améliorer la résilience :** Enfin, cette étude vise à développer et à évaluer des recommandations et des pratiques optimales pour atténuer l'impact des défaillances des appels système et améliorer la résilience globale des applications conteneurisées. Cela inclut la réduction des appels système, l'implémentation de méthodes d'auto-guérison, et des recommandations spécifiques basées sur les résultats des tests de **chaos engineering**. En s'appuyant sur les travaux de Simonsson et al.[7], nous allons démontrer comment la surveillance et l'injection de fautes contrôlées via des outils comme **ChaosOrca** peuvent significativement renforcer la robustesse des systèmes.

En combinant ces objectifs, cette étude vise à fournir des solutions pratiques et concrètes pour améliorer la résilience des applications conteneurisées, minimisant ainsi les interruptions de service et assurant des performances fiables, même en présence de défaillances système.

Chapitre 2

Fondements Théoriques et Contexte

Les applications modernes basées sur des architectures de **microservices** bénéficient d'une méthode efficace pour le déploiement et la gestion des services. Pour aborder ces défis, la **conteneurisation**, avec des outils comme **Docker**(`docker.com`) ou **Kubernetes**(`kubernetes.io`) qui permettent l'orchestration, se présente comme une solution efficace. Les infrastructures de microservices isolent chaque service, ce qui améliore la portabilité et la sécurité des applications, mais introduisent de la complexité pour la communication entre ces inter-services, ce qui soulève des questions sur la **résilience** de ces systèmes.

Pour assurer cette **résilience**, des techniques spécifiques sont adoptées, telles que l'**observabilité**, utilisant des outils basées sur **eBPF**(`ebpf.io`) pour analyser le comportement des **appels système**, ou des méthodes plus modernes comme le **chaos engineering**, garantissant fiabilité et robustesse face aux perturbations imprévues.

Il est important de noter que la **résilience** ne se limite pas seulement à la communication entre les microservices. Le cycle de vie du **DevOps**, avec ses phases d'intégration continue et de déploiement continu, peut également être affecté par des défis de **résilience** [10].

En résumé, l'amélioration de la **résilience** des **environnements conteneurisés** repose sur une compréhension approfondie des interactions système et des

techniques d'**observabilité**. En adoptant des approches basées sur l'analyse proactive et les tests de **résilience**, il est possible de minimiser les interruptions de service et de garantir des performances fiables, même en cas de défaillances [7]. Cette démarche est essentielle pour préparer nos systèmes aux exigences futures et assurer leur robustesse face aux défis technologiques continuellement évolutifs.

Pour mieux comprendre ces concepts et leur application, il est utile de définir les termes clés utilisés dans ce contexte.

Microservices Les microservices représentent une architecture où une application est divisée en plusieurs petits services autonomes et modulaires, chacun gérant une fonctionnalité spécifique [11]. Ce modèle facilite la compréhension, le développement et la maintenance en offrant modularité, déploiement indépendant et une meilleure scalabilité [12]. Chaque service est développé indépendamment des autres et communique à travers des interfaces bien définies, souvent via des requêtes HTTP ou des messages asynchrones.

Docker et la Conteneurisation Docker est un outil de conteneurisation qui encapsule une application et ses dépendances dans un conteneur virtuel, permettant l'exécution sur tout serveur Linux [6]. Les conteneurs fournissent une portabilité accrue, réduisent l'utilisation des ressources par rapport aux machines virtuelles traditionnelles, et améliorent la sécurité grâce à l'isolation entre conteneurs [13].

Kubernetes : Orchestration des Conteneurs Kubernetes est un système d'orchestration de conteneurs qui supporte le déploiement, la mise à échelle, et la gestion des applications conteneurisées [1]. Il fournit des fonctionnalités d'auto-guérison, de *load-balancing* intelligent et d'automatisation des déploiements, facilitant la gestion à grande échelle des applications [14].

Observabilité L'observabilité est la mesure dans laquelle l'état interne d'un système peut être inféré à partir de ses sorties externes. Dans le contexte des architectures de microservices, l'observabilité devient cruciale pour diagnostiquer et résoudre les problèmes de performance et de résilience [15].Cette approche permet de collecter, mesurer et analyser les données provenant des appels système, des transactions réseau, et des performances des conteneurs, aidant ainsi à identifier rapidement les anomalies et à diagnostiquer les problèmes avant qu'ils n'affectent gravement les applications.

Résilience La résilience dans le contexte des systèmes informatiques se réfère à la capacité d'un système à se préparer, répondre et se rétablir rapidement après un incident qui perturbe le service normal [16]. Cela inclut la capacité à opérer sous des conditions dégradées et à se rétablir après un changement.

Appels Système Les appels système sont des interfaces fournies par le noyau de l'OS permettant aux programmes d'effectuer des opérations de bas niveau qui ne pourraient pas être réalisées en mode utilisateur. Ces opérations incluent la gestion des fichiers, des processus, de la mémoire, et des communications réseau [17].

Chaos Engineering Le Chaos Engineering est une technique de fiabilité des systèmes qui consiste à introduire délibérément des conditions de stress ou des anomalies pour tester la capacité des systèmes à y résister. Cette pratique vise à identifier et corriger les problèmes avant qu'ils ne deviennent des défaillances critiques, en particulier dans les environnements de production [9]. Dans le contexte des applications conteneurisées et des microservices, le Chaos Engineering permet de vérifier la robustesse de l'orchestration, les mécanismes de d'auto-gestion/auto-guérison, et de l'infrastructure sous-jacente, comme Kubernetes [18].

DevOps DevOps est une approche de l'ingénierie logicielle qui vise à unifier le développement logiciel (Dev) et l'opération logicielle (Ops), favorisant une culture et un environnement où la construction, les tests et la libération du logiciel peuvent se faire rapidement, fréquemment, et de manière fiable [19].

eBPF L'Extended Berkeley Packet Filter (eBPF) est un outil puissant pour surveiller et manipuler les appels système au niveau du noyau, offrant des capacités d'observation granulaire sans nécessiter de modifications du code source des applications [20]. eBPF permet de capturer des métriques détaillées sur les appels système, telles que la latence, le taux de réussite et les erreurs, fournissant ainsi des informations précieuses pour l'analyse des défaillances [21].

Chapitre 3

Importance et Analyse des Appels Système

3.1 Rôle et Définitions

Les appels système constituent des interfaces essentielles entre les applications fonctionnant en espace utilisateur et le noyau du système d'exploitation. Ces interfaces permettent aux applications de demander des services du noyau, tels que la création de processus, la gestion des fichiers, et les communications réseau. Dans les architectures de microservices, où les applications sont souvent conteneurisées pour isoler et gérer efficacement les ressources, observer le comportement des appels système est crucial pour garantir la performance et la sécurité [22]. Les appels système influencent directement la performance des conteneurs, permettant une surveillance et une optimisation ciblée pour améliorer les performances globales [23]. Avant de discuter de l'influence des appels système et leur impact sur la résilience des applications conteneurisées, il est important de comprendre les rôles de ces différents appels. Le tableau 3.1 présente les appels système clés, classés par leur fonction principale. Par exemple, les appels `open` et `write` sont essentiels pour la gestion des fichiers, `socket` et `bind` facilitent la communication réseau, tandis que `futex` est utilisé pour la synchronisation. Cette classification permet de mieux appréhender comment chaque type d'appel système contribue au fonctionnement global des applications conteneurisées et met en lumière les aspects critiques à surveiller pour améliorer leur performance et leur sécurité.

Appel Système	**Définition**
`open`	Ouvre un fichier ou un périphérique et renvoie un descripteur de fichier. Peut créer un fichier s'il n'existe pas, avec les permissions spécifiées.
`write`	Écrit des données depuis un tampon vers un fichier référencé par un descripteur de fichier, en commençant à la position actuelle du fichier.
`writev`	Écrit des vecteurs de données depuis des tampons multiples vers un fichier en une seule opération atomique.
`read`	Lit des données d'un fichier dans un tampon, à partir de la position actuelle du fichier, et met à jour cette position.
`readv`	Lit des vecteurs de données depuis un fichier dans plusieurs tampons en une seule opération atomique.
`sendfile`	Transfère des données d'un fichier à un socket sans copier les données entre l'espace utilisateur et le noyau, optimisant ainsi la performance.
`sendfile64`	Fonction identique à `sendfile`, mais avec le support des fichiers de grande taille (64 bits).
`accept`	Accepte une connexion entrante sur un socket, créant un nouveau socket pour gérer la communication avec le client.
`bind`	Associe une adresse locale à un socket, nécessaire pour les sockets de serveur qui attendent des connexions entrantes.
`socket`	Crée un nouveau point de communication (socket) pour la communication réseau, spécifiant la famille d'adresses, le type de socket et le protocole.
`setsockopt`	Configure les options d'un socket, telles que les options de niveau protocole, les tailles de tampons et les délais de timeout.
`poll`	Surveille plusieurs descripteurs de fichiers pour déterminer s'ils sont prêts pour des opérations d'E/S, avec une attente optionnelle.
`select`	Surveille un ensemble de descripteurs de fichiers pour vérifier leur état (prêt pour lecture, écriture ou erreur) avec une attente jusqu'à un certain délai.
`futex`	Fournit un mécanisme de synchronisation rapide basé sur les mutexes, principalement utilisé pour la gestion des threads dans un espace d'adressage partagé.

TABLE 3.1 – Définition des Appels Système par Classification[17]. Les lignes rouges représentent les appels système de Gestion des Fichiers, en bleu, la Communication Réseau et en vert, la Synchronisation.

Cette classification permet de mieux comprendre comment chaque type d'appel système contribue au fonctionnement global des applications conteneurisées. En ayant une vue d'ensemble de ces appels, nous pouvons mieux appréhender les défis associés à leur utilisation.

3.2 Défis liés aux Appels système

Les appels système, bien qu'essentiels pour le fonctionnement des applications, peuvent également poser divers défis. Ces défis incluent des problèmes de permissions et d'autorisations, des erreurs de fichiers ou répertoires introuvables, des défaillances matérielles et des erreurs d'entrée/sortie, des interruptions de processus, et des fonctionnalités non implémentées. Dans les sections suivantes, nous examinerons ces défis en détail.

Problèmes de Permissions et Autorisations Les problèmes de permissions surviennent lorsque des applications tentent d'accéder à des fichiers ou des répertoires sans disposer des autorisations nécessaires. Ces problèmes peuvent résulter de configurations incorrectes des permissions ou de restrictions imposées par les politiques de sécurité du système. Un utilisateur non autorisé rencontrera une erreur lorsqu'il tente de lire ou d'écrire dans un fichier protégé. Certaines opérations nécessitent des privilèges élevés, tels que ceux accordés aux utilisateurs root. L'exécution de ces opérations sans les privilèges nécessaires entraînera une erreur, signalée par les codes d'erreur `EACCES` (Permission refusée) et `EPERM` (Opération non permise) [24, 25, 17].

Erreurs de Fichiers ou Répertoires Introuvables Les erreurs de fichiers ou répertoires introuvables se produisent lorsque le chemin spécifié est incorrect ou si le fichier a été supprimé ou déplacé. Les liens symboliques brisés peuvent également causer ce type de problème. Une application échoue lorsqu'elle tente d'accéder à un fichier inexistant, ce qui entraîne des interruptions de service ou des pertes de données. Ce problème est signalé par le code d'erreur `ENOENT` [25, 17].

Défaillances Matérielles et Erreurs d'Entrée/Sortie Les défaillances matérielles, la corruption de données ou les problèmes de communication avec des périphériques entraînent des erreurs d'entrée/sortie. Ces erreurs surviennent fréquemment lors des opérations de lecture ou d'écriture sur des périphériques [26]. Un disque dur défaillant empêche une application de lire ou d'écrire des données correctement, provoquant ainsi une perte de données et des interruptions de service. Ce problème est signalé par le code d'erreur `EIO` [24, 17].

Interruptions de Processus Dans des environnements où des signaux sont utilisés pour gérer des événements asynchrones, les appels système peuvent être interrompus avant d'être complétés. Cela se produit lorsque le système reçoit un signal pendant l'exécution d'un appel système, interrompant ainsi l'opération en cours [27]. Une opération de lecture interrompue par un signal d'alarme nécessite une relance de l'opération. Ce problème est signalé par le code d'erreur `EINTR` [25, 17].

Fonctionnalités Non Implémentées Lorsqu'une application utilise une fonctionnalité qui n'est pas supportée par le noyau du système, une erreur se produit. Cela se produit si l'application appelle une fonction non implémentée dans la version actuelle du noyau. Des appels à des fonctionnalités non disponibles dans le noyau Linux entraînent une erreur, limitant les capacités de l'application ou nécessitant des modifications du code pour utiliser des alternatives. Ce problème est signalé par le code d'erreur `ENOSYS` [17].

Défis de Latence et Retards La virtualisation introduit des couches supplémentaires qui augmentent la latence réseau. L'empilement des couches de virtualisation et le traitement nécessaire à l'acheminement des paquets à travers ces couches prolongent les délais de transmission, compromettant ainsi les performances des applications. Les augmentations de la latence sont particulièrement préjudiciables pour les applications nécessitant des réponses en temps réel. Les erreurs couramment associées aux défis de latence et de retards incluent :

— `ETIMEDOUT` : L'opération a expiré avant de pouvoir être complétée.

— `EAGAIN` ou `EWOULDBLOCK` : L'opération aurait bloqué si elle avait été effectuée en mode bloquant, souvent rencontré dans les opérations d'I/O non bloquantes.

— `ECONNREFUSED` : La connexion a été refusée par le serveur cible, souvent en raison de délais de réponse excessifs.

— `ENETUNREACH` : Le réseau est injoignable, souvent lié à des problèmes de latence élevés ou de congestion du réseau.

— `EHOSTUNREACH` : L'hôte cible est injoignable, ce qui peut être dû à des délais de transmission élevés ou des problèmes de routage.

La latence accrue peut significativement affecter les performances des applications distribuées et conteneurisées [28, 29].

3.3 Impact sur les Applications Conteneurisées

Les erreurs d'accès peuvent provoquer des interruptions de service, tandis que les retards et erreurs d'écriture/lecture peuvent entraîner une perte de données ou

une corruption des fichiers. Il est donc important de surveiller les appels système critiques pour identifier et atténuer les défaillances potentielles.

La surveillance de tous les appels système peut être contre-productive, l'étude de Simonsson et al. [7] a montré que le ciblage des appels système les plus critiques permet de mieux identifier les défaillances sans surcharger le système de surveillance [7]. En utilisant l'outil *ChaosOrca* [30], ils ont évalué la résilience de trois systèmes différents (un client de transfert de fichiers, un serveur proxy inversé et une application web basée sur des microservices) dans des environnements conteneurisés *Docker*.

Le tableau 3.2 classifie les différents types de défaillances des appels système et leur impact sur la performance et la résilience s'appuyant sur l'étude de Simonsson et al. [7].

Appel Système	**Défaillance**	**Impact sur les Performances et la Résilience**
`open`	EACCES,ENOENT	Erreurs d'ouverture de fichier peuvent causer des échecs d'application.
`write`	EIO,EINTR	Échecs d'écriture peuvent mener à une perte de données ou une corruption.
`writev`	EAGAIN	Dégradations de performance si les écritures doivent être réessayées.
`read`	ENOENT,EIO	Échecs de lecture peuvent empêcher l'accès aux données nécessaires.
`readv`	EAGAIN	Latences accrues si les lectures doivent être réessayées.
`sendfile`	EINTR,EIO	Perturbations dans le transfert de données réseau, impactant les performances.
`sendfile64`	EINTR,EIO	Similaire à `sendfile`, avec des implications pour les systèmes 64 bits.
`poll`	EINTR,ENOMEM	Problèmes de latence et de réactivité du système.
`select`	EINTR,ENOMEM	Impact sur la gestion des I/O et la performance des applications.

Table 3.2 – Tableau des appels système, types de défaillance et impacts[7].

En se concentrant sur les impacts des défaillances des appels système, nous pouvons développer des stratégies pour améliorer la résilience des applications conteneurisées. Dans les sections suivantes, nous examinerons les méthodes de détection et d'analyse des défaillances ainsi que les approches pour les atténuer.

Chapitre 4

Résilience et Évaluation de la Robustesse dans les Environnements Conteneurisés

4.1 Observabilité et Gestion des Incidents

L'observabilité est devenue une composante essentielle pour la gestion efficace des architectures de microservices, permettant de comprendre l'état des systèmes complexes en temps réel. L'implémentation de l'observabilité aide à identifier et résoudre rapidement les incidents, assurant ainsi la résilience et la performance [31]. Contrairement à la surveillance, qui gère que les *known unknowns* en collectant des données prédéfinies, l'observabilité permet de répondre à des questions inconnues sur l'état du système, en se basant sur une large gamme de données provenant de diverses sources, en capturant le bon niveau de détail, l'observabilité permet de traiter les *unknown unknowns* [32].

Transition de la Surveillance à l'Observabilité

L'évolution de la surveillance traditionnelle vers une observabilité complète implique l'intégration de multiples sources de données et la capacité de corréler ces données pour obtenir une vue holistique de l'état du système [20]. Les systèmes modernes utilisent des métriques, des traces, des logs, et des événements pour créer une image complète de la santé des applications. Cette transition est essentielle pour identifier

les *unknown unknowns*, des problèmes qui ne seraient pas détectés par une simple surveillance des métriques [20].

L'outil *eBPF* [33] joue un rôle majeur dans cette transition en fournissant des capacités avancées de traçage et de surveillance au niveau du noyau, permettant de collecter des données granulaires sur les appels système, y compris les appels réseau.

Appels Système dans l'Observabilité

L'analyse des appels système, notamment les appels système réseau, offre une vision approfondie du fonctionnement interne des microservices et de leurs interactions. En analysant les appels système, on peut identifier les goulots d'étranglement, les erreurs de communication, les problèmes de latence, et bien d'autres problèmes qui peuvent affecter la résilience globale du système [20, 28].

La figure 4.1 illustre comment *eBPF* peut être utilisé pour surveiller les appels système dans une architecture de microservices. Ce schéma montre le flux de données depuis la collecte des métriques au niveau des conteneurs de microservices, jusqu'à l'analyse centralisée et la visualisation des résultats.

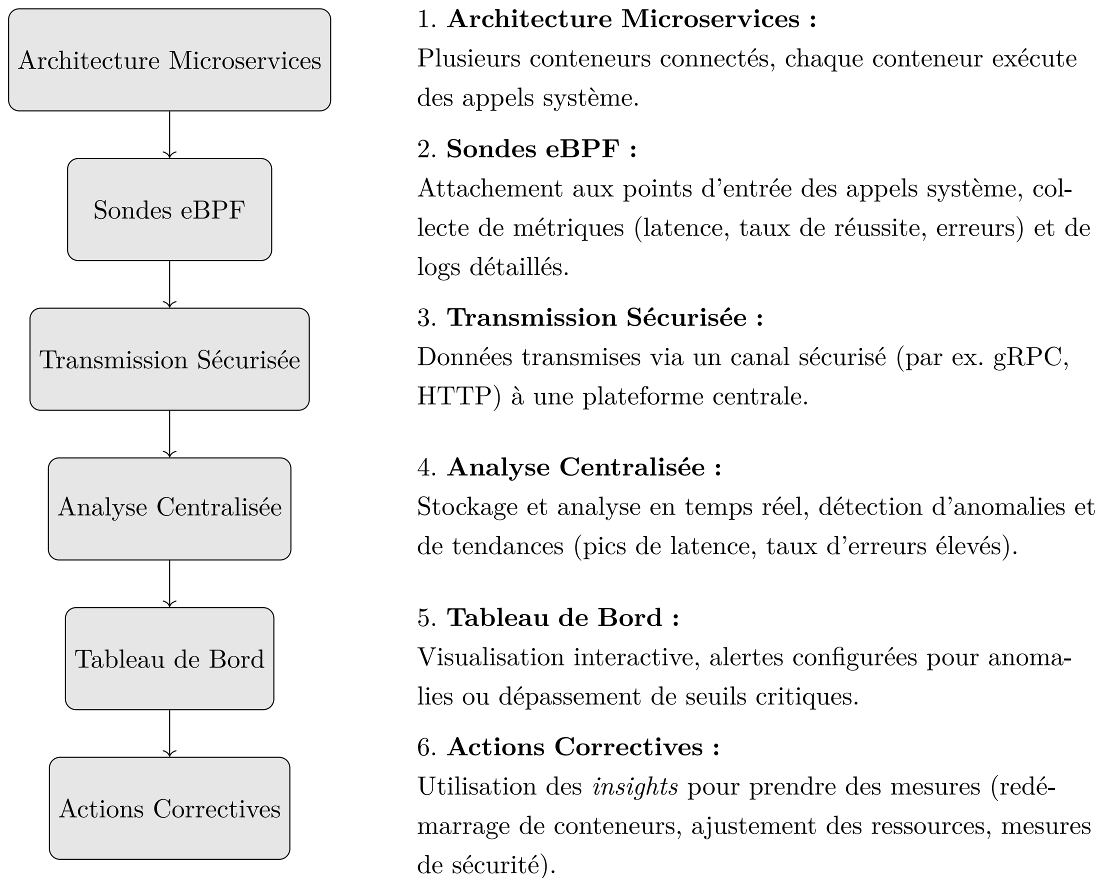

FIGURE 4.1 – Schéma d'Utilisation d'*eBPF* pour la Surveillance des Appels Système

4.2 Application du chaos engineering

Le *Chaos Engineering* permet d'identifier les faiblesses potentielles et de renforcer la capacité des systèmes à gérer des défaillances imprévues. En se concentrant sur les appels système réseau, le Chaos Engineering peut révéler des vulnérabilités dans les communications interservices, améliorant ainsi la stabilité et la performance globales des microservices [20, 21].

Méthodologie du Chaos Engineering La figure 4.2 présente la méthodologie du Chaos Engineering sous la forme d'un diagramme de flux, mettant en évidence les étapes essentielles et leur séquence logique.

1. **Définir des Conditions Stables** : Cette étape consiste à identifier les métriques de performance clés, telles que le temps de réponse et la disponibilité, qui définissent un état stable pour le système. Cela fournit une base de référence pour évaluer les impacts des perturbations.

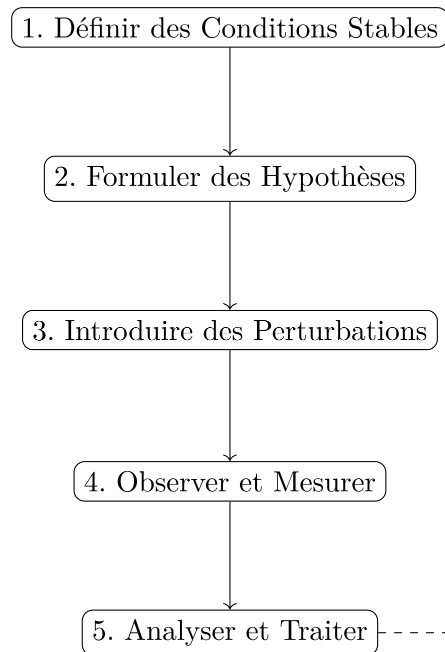

FIGURE 4.2 – Méthodologie du Chaos Engineering [9]. Ce diagramme de flux illustre les étapes clés de la méthodologie du Chaos Engineering, de la définition des conditions stables à l'analyse et au traitement des résultats.

2. **Formuler des Hypothèses** : Une fois les conditions stables définies, des hypothèses sur le comportement attendu du système en cas de perturbation sont établies. Ces hypothèses servent de guide pour les tests et les observations.
3. **Introduire des Perturbations** : Cette étape implique l'introduction de perturbations contrôlées, telles que la coupure de services ou l'injection de latence, pour simuler des scénarios de défaillance et tester la résilience du système.
4. **Observer et Mesurer** : Pendant les tests, les réactions du système sont surveillées à l'aide de métriques et de *logs*. Les données collectées permettent d'analyser comment le système réagit aux perturbations et d'identifier les points de défaillance.
5. **Analyser et Traiter** : Enfin, les résultats des observations sont analysés pour identifier les faiblesses et les points de défaillance du système. Les informations recueillies sont utilisées pour améliorer les systèmes et mettre en place des mesures correctives, bouclant ainsi le processus d'amélioration continue.

Cette méthodologie scientifique permet de valider les hypothèses de stabilité des systèmes et de tester leur capacité à maintenir des performances optimales sous des conditions adverses [9].

4.3 Etude de cas : Observabilité et Injection de fautes via ChaosOrca

Pour illustrer l'application pratique de la méthodologie du *chaos engineering*, nous allons nous référer à l'expérience de Simonsson et al.[7] sur l'application *BookInfo*. Leur étude se concentre sur l'utilisation du Chaos Engineering pour tester la résilience des systèmes conteneurisés en injectant des défaillances au niveau des appels système critiques.

Definition 1 *BookInfo est une application d'exemple de microservices utilisée principalement pour démontrer et tester les technologies de service mesh comme Istio. L'application est composée de plusieurs microservices, chacun responsable d'une fonction différente, qui ensemble simulent un service d'informations sur les livres. Les microservices incluent typiquement :*

— ***Service de Page Produit*** *: Récupère les détails d'un livre (par exemple, titre, description) et les affiche.*

— ***Service de Détails*** *: Fournit des détails supplémentaires sur le livre, tels que l'éditeur, la date de publication, etc.*

— ***Service de Critiques*** *: Gère les critiques de livres. Ce service peut appeler d'autres microservices pour compléter les informations de critique, comme :*

  — ***Service de Notations*** *: Fournit des notations pour les critiques de livres.*

L'application BookInfo sert d'exemple pour montrer les avantages et les fonctionnalités des architectures de microservices, les services mesh, la gestion du trafic, l'observabilité et la sécurité. [34].

Méthodologie de l'Expérience

Conditions Stables et Hypothèse Avant l'injection de perturbations, les conditions stables de l'application sont définies en surveillant les métriques suivantes :

— Latence des services.

— Taux de succès des requêtes HTTP.

— Ratio de paquets envoyés/reçus.

— Utilisation CPU et mémoire des conteneurs.

Simonsson et al.[7] enregistrent et observent ces métriques pour déterminer le comportement typique de *BookInfo* en conditions normales d'utilisation. L'hypothèse principale est que l'application doit maintenir un temps de réponse acceptable même en présence de perturbations au niveau des appels système critiques.

Injection de Fautes avec ChaosOrca *ChaosOrca* est utilisé pour injecter des erreurs et des délais d'entrée sur les appels système `open`, `write`, `read`, `select`, `sendfile` et `poll`. Ces pertubations injectées incluent :

— `EACCES` (permission refusée)

— `EPERM` (opération non permise)

— `ENOENT` (fichier ou répertoire inexistant)

— `EIO` (erreur d'entrée/sortie)

— `EINTR` (interruption par un signal)

— `ENOSYS` (fonction non implémentée)

— Des délais de 1s et 5s

Surveillance des Métriques Les métriques sont surveillées en utilisant des outils tels que *cAdvisor*[35] pour les ressources des conteneurs, *Bpftrace* pour les appels système, et *PyShark*(`pypi.org/project/pyshark`) pour les requêtes HTTP et l'analyse des paquets. Ces métriques sont ensuite visualisées via *Grafana*(`grafana.com`).

Résultats et Analyse Nous nous concentrerons principalement sur les résultats sur la latence HTTP, car elle est un indicateur critique de la réactivité et de la performance des architectures de micro-services, affectant directement l'expérience utilisateur et la capacité du système à répondre en temps réel.

Les données utilisées pour ces expériences sont disponibles dans le dépôt GitHub de ChaosOrca (`https://github.com/KTH/royal-chaos/tree/master/chaosorca`). Les graphiques présentés dans la Figure 4.3 ont été générés à partir des données collectées lors de ces expériences. Les données sont organisées par appel système, type d'erreur et métrique (latence, utilisation du CPU, utilisation de la mémoire, etc.). Deux scripts Python ont été développés pour traiter ces données : le premier script extrait et regroupe les données pour chaque métrique, tandis que le second script génère les graphiques correspondants. Ces scripts sont publiquement disponibles sur `https://github.com/zaka59/chaosorca-results`.

Pour garantir une comparaison cohérente entre les différentes expériences, plusieurs ajustements ont été nécessaires :

— **Axe des y** : Les échelles des axes des y ne sont pas uniformes entre les différents graphiques, car certaines expériences n'ont pas produit suffisamment de données pour couvrir toute la plage de valeurs. Par conséquent, les échelles ont été ajustées individuellement pour chaque graphique afin de maximiser la clarté et la lisibilité des résultats.

— **Synchronisation des Données** : Les erreurs ont été injectées à des moments différents au cours des expériences. Pour permettre une analyse comparative,

les données ont été synchronisées et normalisées en utilisant un temps généralisé, noté Time(s), au lieu des timestamps originaux. Une ligne rouge verticale à t(10) indique le début de l'injection d'erreurs dans chaque graphique.

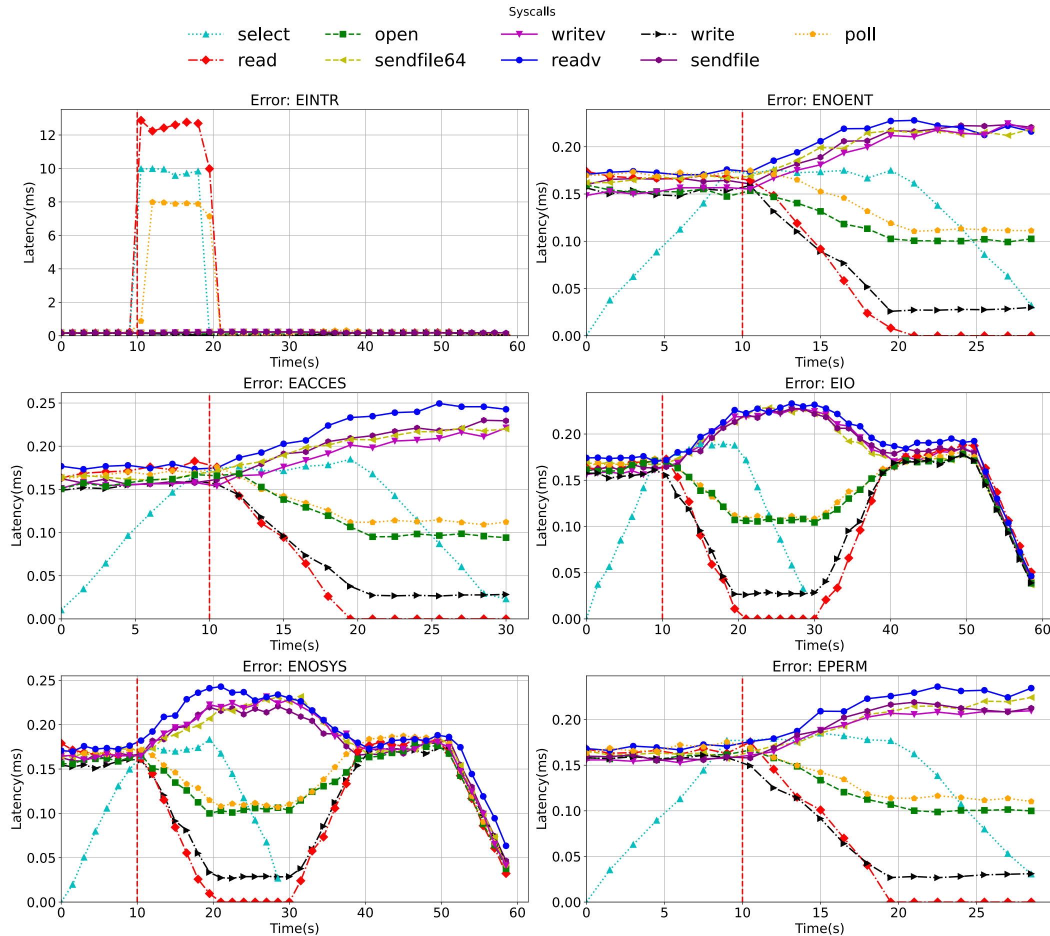

FIGURE 4.3 – Résultats des expériences d'injection d'erreur sur divers appels système et l'impact de ces injections sur la latence du service de Page Produit de BookInfo. Chaque sous-figure illustre l'évolution de la latence en fonction du temps, avec l'injection d'erreurs commençant à environ t(10), marquée par une ligne rouge verticale.

— Erreur `EINTR` : On observe une augmentation brutale de la latence pour les appels `select`, `read`, et `sendfile64` dès le début de l'injection. Cette augmentation significative, atteignant un pic autour de 12 ms pour `read`, montre que l'interruption de ces appels système affecte immédiatement la performance de l'application. Cette réaction est typique car `EINTR` signifie qu'une opération a été interrompue par un signal, nécessitant souvent une gestion supplémentaire pour recommencer l'opération [36, 37].

— Erreur `ENOENT` : L'injection de l'erreur `ENOENT`, qui indique qu'un fichier ou répertoire est introuvable, entraîne une augmentation progressive de la latence pour les appels `open`, `write`, `writev`, et `sendfile`. La latence pour ces appels augmente lentement mais de manière constante, atteignant des valeurs maximales après environ 15 secondes. Cela suggère que l'absence de fichier ou de répertoire perturbe les opérations normales de l'application, provoquant des ralentissements continus. Ce phénomène peut être attribué à des tentatives répétées d'accès à des fichiers inexistants, entraînant des boucles d'erreur qui consomment des ressources [38].

— Erreur `EACCES` : Avec l'erreur `EACCES`, indiquant un accès refusé, la latence des appels `read`, `write`, et `readv` augmente également, mais de manière moins prononcée que pour `EINTR`. Pour `read` et `write`, la latence atteint environ 0,25 ms. Ces résultats montrent que les erreurs de permission perturbent les opérations de fichier, mais pas aussi sévèrement que les interruptions ou les absences de fichiers [36].

— Erreur `EIO` : L'erreur `EIO`, liée aux erreurs d'I/O, montre des augmentations de latence variées et instables pour la plupart des appels système. Les fluctuations observées, notamment pour `write` et `sendfile` indiquent que l'application n'a pas de mécanismes de reprise robustes pour gérer les erreurs d'I/O de manière uniforme [38, 36].

— Erreur `ENOSYS` : Pour l'erreur `ENOSYS`, qui signifie que l'appel système n'est pas implémenté, on observe des augmentations de latence similaires à celles de l'erreur `EIO`. Notamment, les appels `write` et `sendfile` voient leur latence augmenter de manière significative, mettent en évidence la nécessité de mécanismes de secours lorsque certaines fonctionnalités ne sont pas compilées ou disponibles dans le noyau du système d'exploitation [37].

— Erreur `EPERM` : L'injection de l'erreur `EPERM`, qui indique une opération interdite, entraîne des augmentations de latence pour les appels `write`, `writev`, et `poll`. Cette latence accrue, bien que moins prononcée que pour d'autres erreurs, montre que les restrictions de permission affectent également la performance de l'application. Les erreurs `EPERM` peuvent être causées par des restrictions de sécurité au niveau du noyau ou par des politiques de contrôle d'accès strictes [36].

Analyse du Cas de l'Appel Système `select` Les résultats de l'appel système `select` sont particulièrement intéressants et quelque peu ambigus. Contrairement aux autres appels système, la latence pour `select` montre des variations moins prévisibles et plus dispersées après l'injection d'erreurs. Cela pourrait être dû à la nature même de `select`, qui est utilisé pour surveiller plusieurs descripteurs de fichier et peut être affecté de manière inégale par les erreurs injectées. En ce qui

concerne les perturbations de l'appel système `select`, *BookInfo* se bloque directement si un code d'erreur est injecté. Cela indique que l'application est sensible aux erreurs de l'appel `select`, et ses stratégies d'auto-protection ne sont pas capables de ramener *BookInfo* à un état normal lorsque cette perturbation se produit [7].

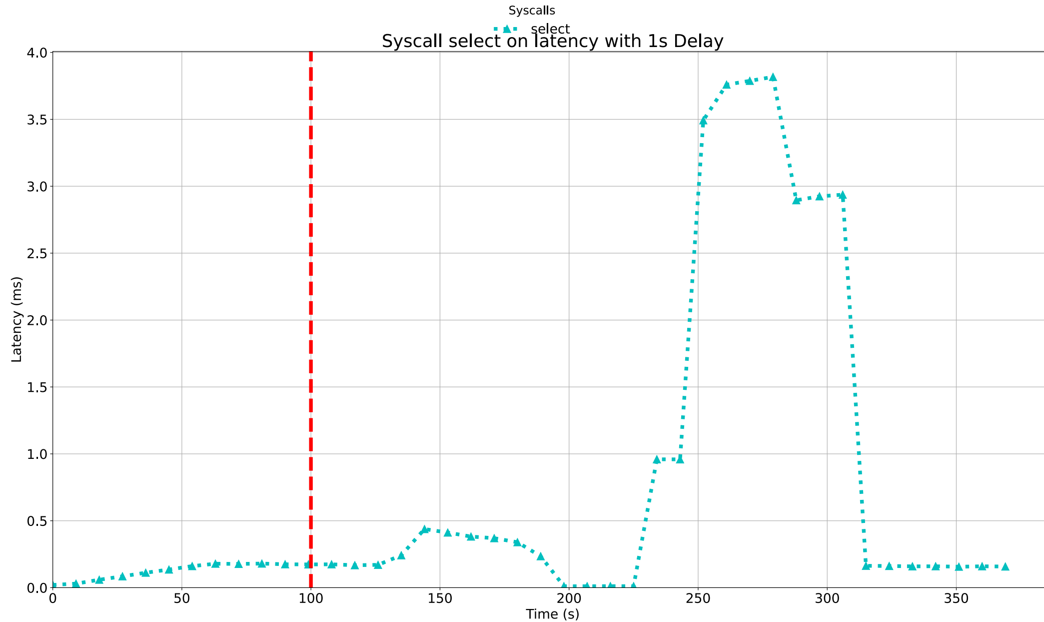

FIGURE 4.4 – Impact des délais de 1s sur l'appel système `select`, montrant comment les interruptions affectent les performances de l'application.

Les injections de fautes 4.3 et les délais 4.4 sur un appel système spécifique n'ont pas seulement un impact sur la latence, les ressources de la machine et les paquets envoyés/reçus, mais affectent également d'autres appels système. Dans ce contexte, `select` avec une erreur `EINTR` a montré des impacts significatifs sur la latence. En observant le comportement des autres appels systèmes face à ces perturbations, on constate un impact significatif sur le nombre d'appels des autres appels systèmes.

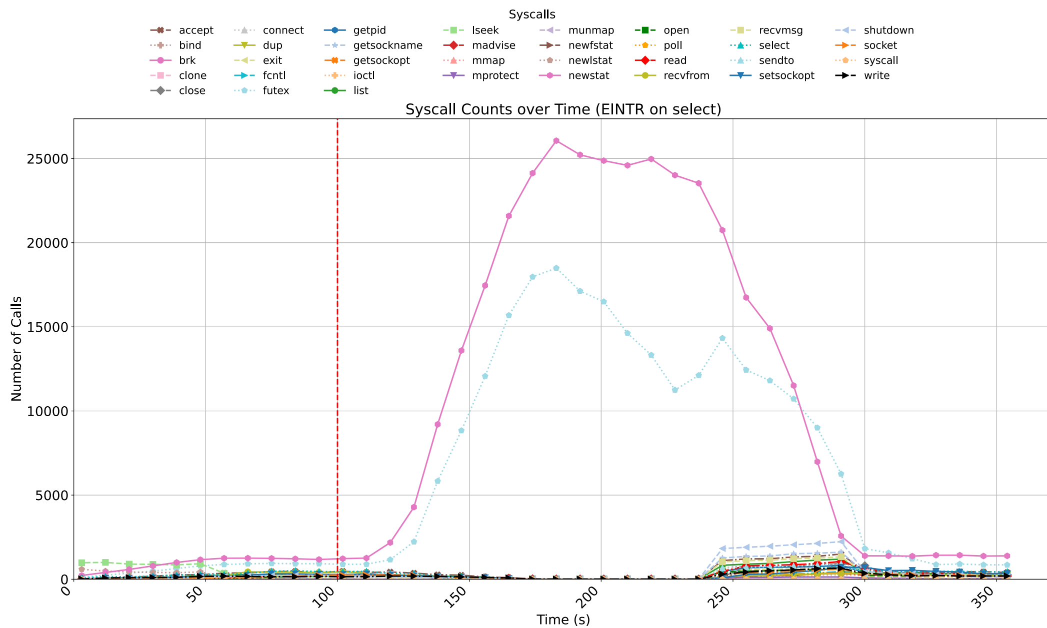

FIGURE 4.5 – Impact de l'injection d'erreurs sur l'appel système `select`. Le graphique montre que les perturbations commencent à environ t(100) et se terminent vers t(230).

La figure 4.5 montre que les perturbations causent une augmentations conséquente de certains appels systèmes, et une diminution d'autres :

— L'augmentation du nombre d'appels `newstat` peut s'expliquer par la nécessité de réinitialiser les états des fichiers en réponse aux perturbations. Les composants de l'application semblent vérifier ou actualiser l'état des fichiers plus fréquemment pour tenter de récupérer de l'erreur [7]. L'augmentation des appels `futex` indique une gestion plus active des verrous et des conditions de concurrence. Ces appels sont souvent utilisés pour la synchronisation entre threads, suggérant des tentatives répétées de synchronisation en réponse aux erreurs injectées [38].

— La diminution drastique des appels comme `read`, `write`, et `poll` pourrait indiquer que l'application échoue à gérer les erreurs `EINTR`. Simonsson et al. notent que les erreurs `EINTR` provoquent des exceptions de *timeout* dans les composants de *BookInfo*, entraînant des redirections vers des pages initiales et finalement une réduction de l'activité des appels système [7].

Les erreurs `EINTR` interrompent fréquemment les appels `select`, provoquant des délais et des exceptions. En réponse, les composants de l'application initient davantage d'appels `newstat` et `futex` pour tenter de récupérer les états et synchronisations perdus. La réduction des autres appels pourrait indiquer que les mécanismes de protection automatique de l'application ne sont pas capables de ramener l'application à un état stable après des interruptions fréquentes.

Cette étude de cas sur l'utilisation de *ChaosOrca* pour tester la résilience des systèmes conteneurisés met en lumière l'impact significatif de diverses erreurs sur la latence HTTP. Les résultats montrent que certaines erreurs, comme `EINTR` et `ENOENT`, perturbent fortement les opérations de l'application, tandis que d'autres, comme `EACCES` et `EPERM`, ont des effets moins prononcés. Les injections de délais révèlent également des faiblesses spécifiques dans la gestion des défaillances par l'application. Ansi, cette approche centrée sur l'observabilité des appels système permet de mieux comprendre la sources des défaillances et d'améliorer les performances des applications conteneurisées.

4.4 Automatisation et Auto-guérison pour la Résilience

L'expériences de chaos via *ChaosOrca* a révélé que les systèmes conteneurisés sont significativement affectés par les erreurs des appels système, mettant en évidence les points critiques nécessitant un renforcement. Les résultats obtenus fournissent des indications précieuses sur les modifications nécessaires pour améliorer la résilience. En complément de l'observabilité et des injections de fautes, des techniques plus globales peuvent être envisagées. Nous allons donc explorer des approches complémentaires, tels que l'automatisation et l'auto-guérison pour faire face aux perturbations imprévues.

Les systèmes auto-adaptatifs de microservices peuvent surveiller leur comportement et ajuster leur configuration ou leur architecture en temps réel pour préserver et améliorer leurs attributs de qualité tels que la performance, la fiabilité et la sécurité, même sous des conditions d'exploitation incertaines [39].

Automatisation des déploiements et autoscaling *Kubernetes* peut ajuster automatiquement le nombre d'instances d'un service en fonction de la charge et redémarrer automatiquement les instances de service défaillantes. Les systèmes auto-guérissants utilisent des contrôleurs de santé qui surveillent en continu les instances de service pour détecter et corriger automatiquement les défaillances. Cette capacité d'auto-guérison minimise les temps d'arrêt assurant donc une disponibilité continue des services [39].

Déploiement continu (CI/CD) Les outils de gestion de CD comme *Spinnaker* [40] permettent la création de pipelines de CD indépendants pour chaque microservice, facilitant ainsi des déploiements fréquents et indépendants. Cela améliore la capacité de réponse aux incidents et la résilience globale du système [39].

Un exemple typique d'application conteneurisée est un cluster de conteneurs auto-adaptatif, tel que décrit par Sliem et al. [41].

L'application consiste en un ensemble de conteneurs orchestrés par des plateformes comme *Docker Swarm* ou *Kubernetes*. La structure typique d'un cluster de conteneurs comprend un nœud de gestion et des nœuds de travail. Les noeuds de travail exécutent les conteneurs avec les charges de travail soumises par les utilisateurs, tandis que le noeud de gestion orchestre le déploiement des conteneurs sur les nœuds de travail et maintient l'état du cluster en vérifiant continuellement l'état des nœuds.

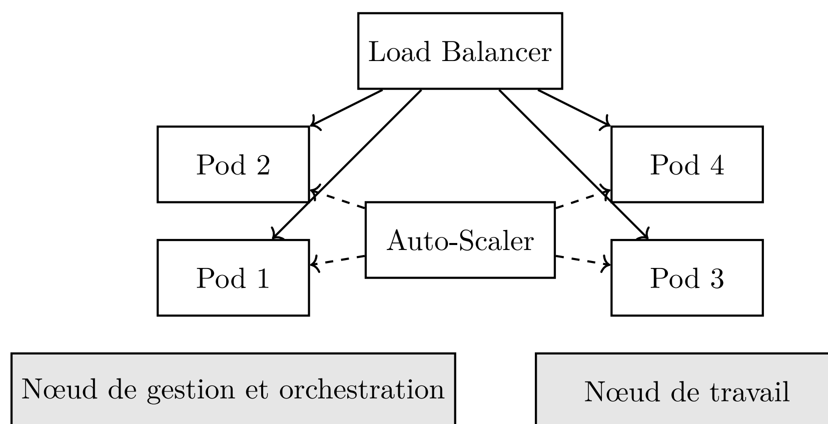

FIGURE 4.6 – Cette figure illustre les concepts clés de l'architecture d'un cluster de conteneurs : Les **Pods**, représentés par les rectangles bleus, sont des unités d'exécution qui encapsulent un ou plusieurs conteneurs. Ils exécutent les applications et services dans un environnement isolé. Le **Load Balancer**, illustré par le rectangle vert, est responsable de répartir la charge de travail entre les différents nœuds disponibles dans le cluster, assurant une distribution équilibrée des requêtes. L'**Auto-Scaler**, représenté par le rectangle rouge, surveille l'utilisation des ressources des pods et ajuste dynamiquement le nombre de pods en fonction de la charge de travail, en ajoutant ou supprimant des pods selon la stratégie de configuration définie.

L'efficacité de l'auto-scaler est évaluée en fonction de la capacité à traiter les requêtes et à maintenir les performances sous différents scénarios de charge de travail. Par exemple, la simulation de différentes intensités de charge de travail et de périodicité de surveillance de l'auto-scaler permet d'identifier les configurations optimales pour éviter la sur-provision et maintenir un débit de traitement élevé [41].

Bien que cette approche soit prometteuse et puisse offrir une grande flexibilité et robustesse, elle nécessite davantage d'évaluations pour confirmer son efficacité par rapport à d'autres stratégies de résilience plus établies. Mais, cela n'empêche pas son utilisation en complément des autres stratégies de résilience. L'intégration des systèmes d'auto-guérison avec des méthodes plus traditionnelles peut en fait

renforcer la résilience globale du système, offrant une couche supplémentaire de protection [41].

Chapitre 5

Réduction des Appels Système pour Améliorer la Résilience

Dans les chapitres précédents, nous avons démontré comment l'observabilité des appels système et le Chaos Engineering permettent de prédire et de mitiger les problèmes de performance et de sécurité dans les environnements conteneurisés. Les expériences de Simonsson et al. [7] ont mis en évidence que les retards et les erreurs des appels système, en particulier les appels système réseau, ont un impact significatif sur la résilience des applications. L'observabilité révèle que chaque retard ou erreur peut poser des problèmes conséquents, soulignant l'importance des appels système dans la stabilité et la sécurité des applications conteneurisées.

Etant donné que la majorité des interactions des applications conteneurisés passent par des appels système, il devient essentiel de trouver des moyens d'améliorer la résilience de ces appels. Une approche efficace pour atteindre cet objectif est de réduire et limiter le nombre d'appels système autorisés.

5.1 Principes de la Réduction des Appels Système

La réduction des appels système vise à identifier et à limiter les appels nécessaires au fonctionnement des applications. Cette approche permet de minimiser les appels superflus qui pourraient causer des instabilités. En restreignant les appels système à une *whitelist*, il est possible de réduire le nombre de points de défaillance potentiels, améliorant ainsi la robustesse des applications [42, 43, 44].

Avant de se pencher sur les différentes méthodes de réduction des appels système, il est essentiel de comprendre les principes fondamentaux qui soutiennent cette approche.

Mécanisme Seccomp *Seccomp* (*Secure Computing Mode)* est un mécanisme de sécurité intégré au noyau Linux qui permet de restreindre les appels système qu'un processus peut exécuter. Introduit initialement pour offrir une protection supplémentaire aux processus, *seccomp* permet de mettre un processus en mode "strict" où il ne peut effectuer que des appels système jugés sûrs. Cela est réalisé en filtrant les appels système via un profil défini par l'utilisateur, souvent appelé "profil seccomp" [45].

Depuis *Docker 1.10*, un profil *seccomp* par défaut est fourni, désactivant environ 44 appels système sur plus de 300 possibles [43]. Ces appels désactivés sont généralement ceux qui sont jugés inutiles ou dangereux pour la majorité des applications conteneurisées. Toutefois, ce profil est générique et ne prend pas en compte les besoins spécifiques de chaque conteneur. Par exemple, un conteneur exécutant une base de données *MySQL* pourrait avoir des besoins en appels système différents d'un conteneur exécutant une application *Node.js* [45].

Seccomp utilise des règles de filtrage basées sur *BPF*, un mécanisme de filtrage de paquets réseau, pour décider quels appels système sont autorisés ou bloqués. Les règles peuvent être définies pour autoriser ou bloquer des appels système spécifiques, ainsi que pour vérifier les arguments des appels système afin de bloquer des comportements spécifiques [43].

Whitelist d'appels système Pour un conteneur particulier, de nombreux appels système peuvent être inutiles, et leur désactivation peut réduire la surface d'attaque. Par exemple, un conteneur *MySQL* nécessite les appels système `read` et `write` pour accéder aux fichiers, mais pas `sched_setparam` pour modifier les paramètres de planification des processus [43].

Une *whitelist* d'appels système est une liste des appels système autorisés qu'un processus peut effectuer. L'utilisation d'une *whitelist* permet de réduire considérablement la surface d'attaque en limitant les capacités d'un processus à ce qui est strictement nécessaire pour son fonctionnement. Cela signifie que même si un attaquant parvient à exécuter du code malveillant dans un conteneur, ses capacités seront limitées par les appels système autorisés [46].

Par exemple, *Docker* fournit un profil *seccomp* par défaut qui interdit certains appels système jugés dangereux. Cependant, ce profil est conçu pour être largement compatible avec la majorité des applications et peut ne pas être suffisamment strict pour des cas d'utilisation spécifiques [45]. Ainsi, pour un conteneur exécutant une base de données *MySQL* (`mysql.com`), il peut être nécessaire de créer un profil

seccomp personnalisé qui n'autorise que les appels système nécessaires, tels que `read`, `write`, `open`, et `close`, tout en bloquant les autres appels qui ne sont pas requis, comme `ptrace` et `mknod` [42].

5.2 Méthodes de Réduction des Appels Système

Pour réduire efficacement les appels système, plusieurs méthodes peuvent être employées. Deux approches notables sont *SPEAKER* et *RSDS*, qui se distinguent par leur stratégie d'analyse et de gestion des appels système.

SPEAKER : Exécution en Phase Divisée des Conteneurs *SPEAKER (Split-Phase Execution of Application Containers)* propose une approche pour réduire les appels système en séparant l'exécution d'un conteneur en deux phases distinctes : démarrage et exécution. Cette séparation permet de restreindre les appels système à ceux nécessaires pour chaque phase, réduisant ainsi le nombre total d'appels système [42].

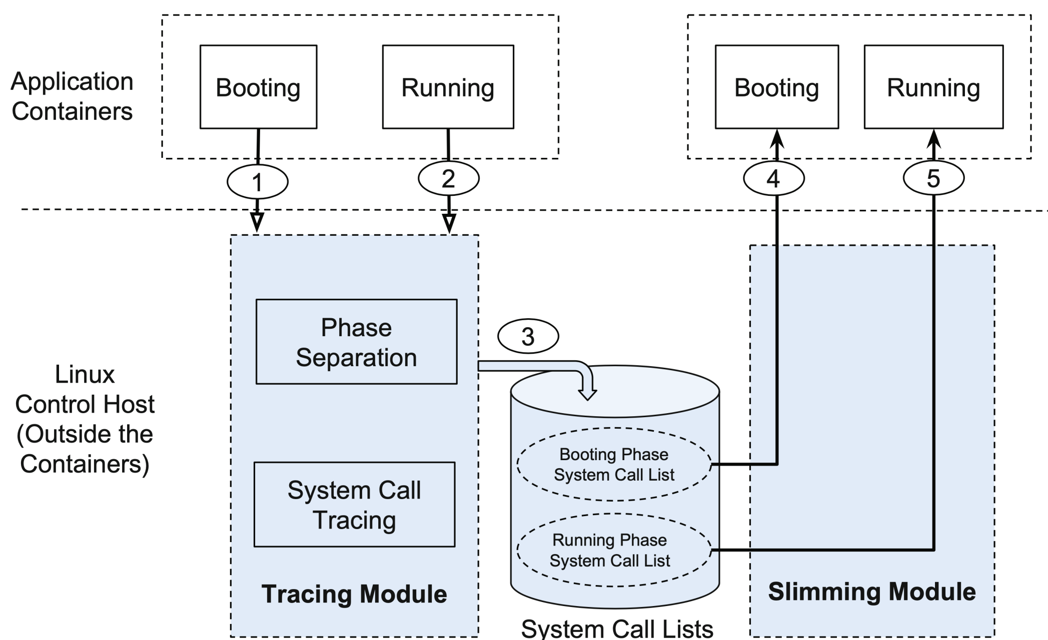

FIGURE 5.1 – Architecture de SPEAKER [42]

La figure 5.1 montre l'architecture de SPEAKER, qui se compose de deux modules principaux : le module de traçage et le module d'amincissement. Le module de traçage profile les appels système nécessaires pendant les phases de démarrage et d'exécution. Le module d'amincissement applique ensuite ces profils pour restreindre dynamiquement les appels système disponibles.

Phase de Démarrage Durant cette phase, le conteneur initialise l'environnement et démarre les services nécessaires. Un ensemble plus large d'appels système peut être nécessaire pour configurer l'environnement. Par exemple, le conteneur *MySQL* utilisé dans l'expérience de L. Lei et al.[42] nécessite environ 116 appels système durant cette phase.

Phase d'Exécution Une fois l'initialisation terminée, seuls les appels système nécessaires au fonctionnement continu des services sont autorisés. Cela permet de réduire considérablement le nombre d'appels système actifs pendant la majorité du cycle de vie du conteneur. Si nous reprenons l'exemple *MySQL*, le conteneur *MySQL* réduit à 58 le nombre d'appels système durant cette phase [42].

RSDS : Analyse Dynamique et Statique des Appels Système *RSDS (Dynamic and Static Analysis for System Call Whitelisting)* est une méthode qui combine l'analyse dynamique et statique pour déterminer les appels système nécessaires à une application conteneurisée spécifique.

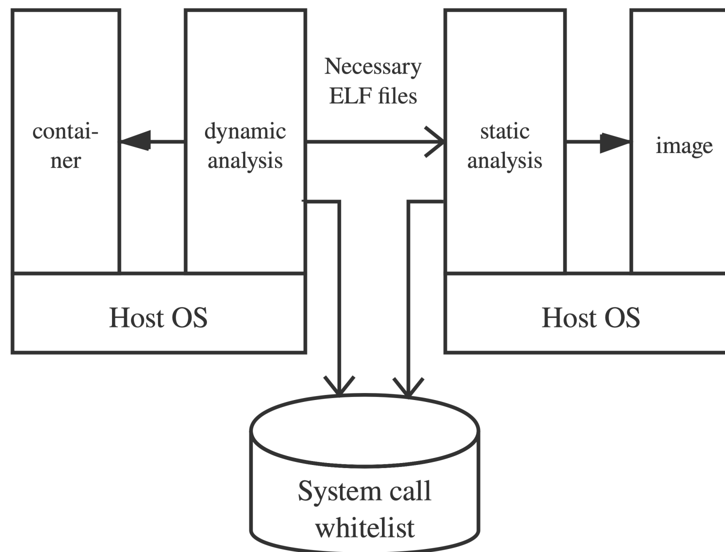

Figure 5.2 – Architecture de RSDS [43]

La figure 5.2 illustre l'architecture de *RSDS*. Le module d'analyse dynamique surveille les appels système pendant l'exécution de l'application, tandis que le module d'analyse statique examine les fichiers *ELF* pour identifier les appels système intégrés. Les résultats des deux analyses sont fusionnés pour créer une liste blanche complète des appels système.

Analyse Dynamique L'analyse dynamique vise à identifier les appels système nécessaires pendant le démarrage et l'exécution du conteneur. Lors du démarrage d'un conteneur, le *Docker daemon* initialise l'environnement d'exécution, ce qui inclut le montage des volumes et la configuration du réseau. Durant cette phase, tous les appels système sont autorisés. Après cette initialisation, le processus utilisateur spécifié est exécuté, et les appels système sont restreints par le profil *seccomp*. Pour cette phase, `strace` est utilisé pour tracer les appels système effectués après l'application du profil *seccomp* [43].

Analyse Statique L'analyse statique consiste à désassembler les fichiers exécutables *ELF* pour identifier les appels système utilisés par le conteneur à l'exécution en utilisant trois techniques differents pour déterminer les appels système (ces différentes techniques sont analysées et détaillés dans l'étude de Wang et al.[43]) :

- Assignation directe de l'ID de l'appel système au registre `eax` ou `rax`.
- Assignation indirecte de l'ID de l'appel système au registre `eax` ou `rax` via d'autres registres.
- Utilisation de la fonction `syscall()` pour les appels système sans fonction wrapper dans la bibliothèque libc.

Definition 2 *Le Docker daemon est le processus principal de Docker qui écoute les API Docker et gère les objets Docker tels que les images, les conteneurs, les réseaux et les volumes. Il automatise les tâches de déploiement, de gestion et de mise à l'échelle des conteneurs, permettant ainsi de simplifier la gestion des environnements applicatifs [47].*

Definition 3 *ELF (Executable and Linkable Format) est un format de fichier utilisé pour les exécutables, les bibliothèques partagées et les fichiers core dumps sous Unix et Unix-like. En examinant les fichiers ELF, il est possible d'identifier les appels système intégrés au code source d'une application, ce qui aide à créer des whitelists précises [48].*

Definition 4 *Une fonction wrapper est une fonction qui appelle une autre fonction, ajoutant une couche d'abstraction ou de contrôle autour de celle-ci. Elle est utilisée pour simplifier l'interface d'une fonction complexe, gérer des exceptions, ou ajouter des fonctionnalités supplémentaires comme la journalisation (logs), le suivi des performances, ou la sécurité [49].*

L'analyse comparative entre *SPEAKER* et *RSDS* montre que *RSDS* obtient des résultats plus complets en termes d'appels système nécessaires, car il combine les

phases d'analyse dynamique et statique. Par exemple, pendant la phase de démarrage (5.1), *RSDS* enregistre moins d'appels système que SPEAKER, car il prend en compte uniquement les appels après l'activation de *Seccomp* [43]. En revanche, lors de la phase d'exécution(5.2), *RSDS* capture plus d'appels système grâce à son analyse statique exhaustive, ce qui peut inclure des appels non utilisés immédiatement mais nécessaires pour des fonctions futures de l'application.

Container	**SPEAKER**	**RSDS**
MySQL	116	72
Postgres	116	72
MongoDB	110	73
Redis	92	53

TABLE 5.1 – Nombre d'appels système pendant la phase de démarrage

Container	**SPEAKER**	**RSDS**
MySQL	58	79
Postgres	52	85
MongoDB	55	73
Redis	42	46

TABLE 5.2 – Nombre d'appels système pendant la phase d'exécution

En termes de performances, les deux méthodes introduisent une légère augmentation du temps de réponse, de l'ordre de quelques millisecondes [42, 43], ce qui reste généralement acceptable dans la plupart des scénarios d'utilisation.

5.3 eBPF pour la Sécurité

Comme les deux méthodes précédemment abordées (*SPEAKER* et *RSDS*), *eBPF* peut également être utilisé pour créer des listes blanches des appels système. Cependant, *eBPF* offre des fonctionnalités supplémentaires qui en font un outil particulièrement puissant pour la sécurité des systèmes. Les programmes *eBPF* peuvent surveiller les appels système et déclencher des alertes si des modèles d'utilisation suspects sont détectés[44].

Nous avons expliquer dans la Figure 4.1 *eBPF* permet de charger des sondes dans le noyau qui peuvent être attachés à divers points d'ancrage, tels que les appels système. Ces sondes peuvent inspecter et manipuler les données avant ou après

l'exécution des appels système, permettant une flexibilité et une précision accrues dans le contrôle des comportements des processus [44].

Un exemple concret d'utilisation d'*eBPF* pour la détection en temps réel est la surveillance des appels système liés à l'accès aux fichiers. Un programme *eBPF* peut être configuré pour surveiller les appels `open`, `read`, et `write`. Si un modèle d'accès anormal est détecté, tel qu'un grand nombre de tentatives d'accès à des fichiers sensibles en peu de temps, une alerte peut être déclenchée pour informer les administrateurs [44].

Les méthodes *SPEAKER* et *RSDS* des filtres *BPF* et des fichier *ELF* pour implémenter des listes blanches d'appels système. Cependant, ils sont limités dans leur capacité à gérer des scénarios complexes. Les filtres *eBPF* permettent une logique de filtrage plus avancée. En utilisant des structures de données telles que les *maps*, les filtres eBPF peuvent maintenir des états entre les appels système. Cela permet d'implémenter des contrôles *stateful* et des politiques adaptatives qui s'ajustent en fonction des comportements observés. Pour mieux comprendre l'utilité d'eBPF dans ce contexte, nous allons expliquer les apports qui en font un outil puissant pour la résilience et la sécurité des applications conteneurisées [44] :

- Filtres *seccomp* utilisant *eBPF* peuvent créer des politiques qui vérifient non seulement les appels système mais aussi les arguments de ces appels.
- les filtres *seccomp* traditionnels sont *non-sleepable*, ce qui signifie qu'ils ne peuvent pas attendre des événements ou des conditions externes pour prendre une décision. Les *sleepable seccomp filters* avec *eBPF* permettent aux programmes de faire des appels système bloquants, d'attendre des événements, et de prendre des décisions basées sur des états ou des informations provenant de l'espace utilisateur.
- *eBPF* supporte *CRIU* en permettant la sauvegarde et la restauration des états des *maps* et des filtres *eBPF* associés. Cela permet des fonctionnalités telles que la migration en direct et les *snapshots*.

Definition 5 *CRIU (Checkpoint/Restore In Userspace) est une technologie permettant de sauvegarder et de restaurer l'état d'un processus en espace utilisateur, incluant la mémoire, les descripteurs de fichiers et les connexions réseau, facilitant ainsi la migration et la reprise après panne des conteneurs [50].*

Résultats Expérimentaux Pour démontrer l'efficacité d'*eBPF* dans la réduction et le contrôle des appels système, plusieurs expériences ont été menées dans l'étude de J. Jia et al[44]. Ces expériences visent à évaluer la capacité des filtres *eBPF* à améliorer la sécurité en simulant différents environnements d'application (*MySQL*(`mysql.com`), *PostgreSQL*(`postgresql.org`), et *Nginx*(`nginx.org`)). Chaque application utilise un conteneur *Docker*. Des programmes *eBPF* ont été attachés à ces conteneurs

pour surveiller ces actions, déclencher des alertes en cas de détection d'activités suspectes et filtrer les appels système.

L'objectif principal des expériences est de réduire le nombre d'appels système inutiles ou potentiellement dangereux en simulant des scénarios d'attaques tels que des tentatives d'accès non autorisé à des fichiers, modification de configurations critiques et exécution de commandes non autorisées. Le tableau 5.3 montrent la comparaison du nombre d'appels système avant et après l'application des filtres *eBPF* pour chaque application testée.

Application	Avant eBPF	Après eBPF
MySQL	300	120
PostgreSQL	250	110
Nginx	280	100

TABLE 5.3 – Réduction des appels système avec eBPF [44].

Les résultats montrent une réduction significative des appels système dans tous les cas, ce qui confirme l'efficacité des filtres *eBPF* pour limiter les appels système à ceux strictement nécessaires. Les résultats ont également montré que les filtres *eBPF* ont réussi à détecter et à bloquer toutes les tentatives d'attaque simulées, démontrant ainsi leur efficacité pour renforcer la sécurité des systèmes.

Impact sur les Performances Il est crucial que les filtres *eBPF* n'introduisent pas de surcharge significative qui pourrait dégrader les performances des applications. Les performances (5.3) ont été mesurées en termes de temps de réponse des applications avant et après l'application des filtres *eBPF*.

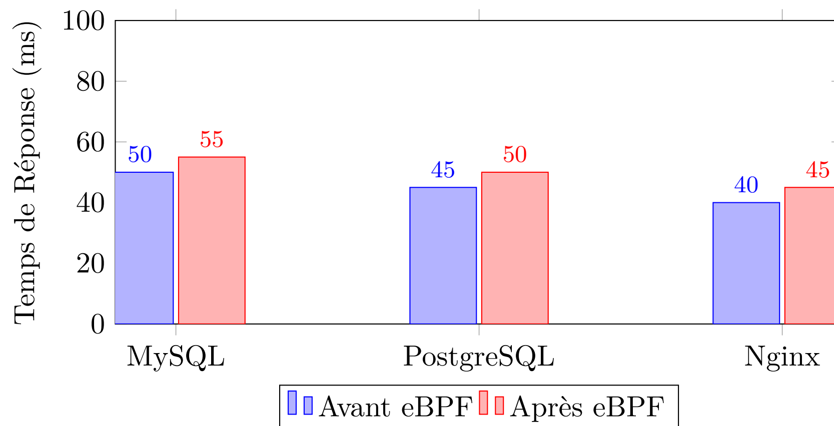

FIGURE 5.3 – Impact des programmes *eBPF* sur les performances des applications [44].

On peut observer que l'utilisation des filtres *eBPF* n'entraîne qu'une légère augmentation du temps de réponse, de l'ordre de quelques millisecondes, ce qui est généralement acceptable dans la plupart des scénarios d'utilisation.

Les méthodes *SPEAKER*, *RSDS* et *eBPF* peuvent améliorer la robustesse et sécurité des applications conteneurisées. Les expérimentations avec *SPEAKER* ont montré que la séparation des phases de démarrage et d'exécution permet une réduction significative des appels système, les résultats obtenus avec *MySQL*, *Nginx*, *Redis* et *PostgreSQL* montrent clairement les bénéfices de cette approche en termes de performances et d'efficacité. De même, les tests avec *RSDS* ont mis en évidence l'efficacité de la combinaison d'analyses dynamiques et statiques pour filtrer les appels système critiques en créant des listes blanches. L'utilisation des filtres *eBPF* à également prouvé une amélioration significative de la sécurité des appels système tout en introduisant une surcharge minimale sur les performances des applications. En réduisant le nombre d'appels système et en détectant les comportements suspects en temps réel, les filtres *eBPF* offrent une couche de sécurité supplémentaire essentielle pour les environnements conteneurisés.

Chapitre 6

Conclusion

Au cours de ce mémoire, nous avons exploré en profondeur le rôle crucial des appels système dans la sécurité et la résilience des applications conteneurisées. En comprenant les défis spécifiques liés à ces appels et en appliquant des stratégies de réduction et de surveillance, il est possible de renforcer considérablement la résilience de ces systèmes.

Nous pouvons tirer plusieurs conclusions clés :

Importance des Appels Système Les appels système sont essentiels pour les microservices et les applications conteneurisées car ils permettent la communication avec le noyau et les périphériques du système. Cependant, leur utilisation excessive ou inappropriée peut introduire des vulnérabilités et affecter la performance globale du système. Les conteneurs partagent le même noyau, ce qui signifie qu'une vulnérabilité dans un appel système peut potentiellement affecter tous les conteneurs exécutés sur le même hôte [43].

Stratégies de Réduction des Appels Système Les techniques comme basées sur Seccomp comme SPEAKER et RSDS se sont révélées efficaces pour réduire les appels système inutiles et potentiellement dangereux. En utilisant des listes blanches d'appels système et en séparant les phases de démarrage et d'exécution des conteneurs, il est possible de limiter les appels système à ceux strictement nécessaires,

réduisant ainsi la surface d'attaque et améliorant la sécurité. Les résultats expérimentaux ont montré que ces méthodes peuvent réduire significativement le nombre d'appels système autorisés sans affecter la fonctionnalité des applications. Par exemple, *RSDS* a permis de réduire le nombre d'appels système autorisés de 333 à 101 pour *MySQL*, soit une réduction de 69.67% [43].

Observabilité et Chaos Engineering Pour comprendre et améliorer la résilience des appels système, il est crucial d'adopter des pratiques comme l'observabilité et le Chaos Engineering. L'observabilité permet de surveiller les systèmes en temps réel et de comprendre leur comportement sous différentes conditions. Les outils basés sur *eBPF* sont particulièrement efficaces pour cette tâche, car ils offrent une visibilité approfondie sans impact significatif sur les performances du système [20]. Le Chaos Engineering, quant à lui, permet de simuler des perturbations dans des environnements contrôlés pour tester la résilience des systèmes. En injectant des erreurs et des délais dans les appels système, il est possible d'observer leur impact sur la latence, la consommation des ressources et le comportement général des applications [7].

La gestion efficace des appels système est essentielle pour la sécurité et la résilience des applications conteneurisées. En combinant des techniques avancées de réduction des appels système, l'observabilité et le Chaos Engineering, il est possible de créer des systèmes plus robustes et sécurisés, prêts à faire face aux défis actuels et futurs. Les méthodes *SPEAKER*, *RSDS* et *eBPF* peuvent améliorer la robustesse et la sécurité des applications conteneurisées, les expérimentations de ceux-là montrent clairement les bénéfices de cette approche sans affecter les applications conteneurisées en termes de performances et d'efficacité.

Perspectives Futures En se basant sur les articles scientifiques utilisés dans ce mémoire, plusieurs pistes prometteuses se dégagent pour les futures recherches et applications dans le domaine de la sécurité et de la résilience des systèmes conteneurisés.

Élargissement de l'analyse des appels système Élargir l'analyse des codes d'erreur des appels système pour couvrir l'ensemble des plus de 100 codes d'erreur possibles. L'interaction entre un appel système spécifique et ses codes d'erreur n'a pas encore été analysée de manière exhaustive. De futures recherches pourraeint explorer cet espace de perturbation pour une meilleure compréhension des faiblesses potentielles des systèmes conteneurisés [7, 44].

Intégration et évaluation de nouveaux systèmes de conteneurs *ChaosOrca* a été spécifiquement conçu pour Docker mais pourrait être étendu à d'autres systèmes de conteneurs comme *Singularity*, en ajustant les commandes utilisées et en appliquant des outils de surveillance spécifiques. Cela permettrait d'évaluer l'extensibilité et la surcharge de *ChaosOrca* dans différents environnements de conteneurs [7].

Utilisation du chaos engineering pour évaluer la résilience L'application des principes du chaos engineering, tels que l'injection de défaillances au niveau des appels système, offre une méthode prometteuse pour identifier les faiblesses dans les mécanismes de protection et de résilience des applications conteneurisées. Cette approche pourrait être automatisée dans le contexte de *frameworks* d'orchestration de conteneurs comme *Docker Swarm* ou *Kubernetes* pour une évaluation continue de la résilience [7, 39].

Développement de politiques de sécurité adaptatives L'intégration de politiques de sécurité adaptatives, qui ajustent dynamiquement les listes blanches d'appels système en fonction des comportements observés, pourrait améliorer significativement la sécurité des systèmes conteneurisés. Cette approche combine les analyses dynamiques et statiques pour créer des profils de sécurité spécifiques à chaque conteneur, réduisant ainsi la surface d'attaque potentielle sans compromettre les fonctionnalités des applications [43, 41].

Application de nouvelles stratégies de test Les nouvelles approches de test, notamment celles intégrant des pipelines de livraison continue (CD) et des outils d'ingénierie des modèles pour microservices, pourraient être essentielles pour automatiser l'intégration des capacités d'auto-adaptation et de *DevOps*. En appliquant les principes du chaos engineering, ces stratégies pourraient aider à évaluer et à améliorer la résilience des systèmes auto-adaptatifs basés sur des microservices [39].

Références

[1] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, “Borg, Omega, and Kubernetes : Lessons learned from three container-management systems over a decade,” *Queue*, Jan. 2016. [Cité en pages 5 et 10]

[2] B. Bashari Rad, H. Bhatti, and M. Ahmadi, “An Introduction to Docker and Analysis of its Performance,” *IJCSNS International Journal of Computer Science and Network Security*, Mar. 2017. [Cité en page 5]

[3] A. Martin, S. Raponi, T. Combe, and R. Pietro, “Docker ecosystem – Vulnerability Analysis,” *Computer Communications*, Mar. 2018. [Cité en pages 5 et 6]

[4] L. Mariani, M. Pezzè, O. Riganelli, and R. Xin, “Predicting failures in multi-tier distributed systems,” *Journal of Systems and Software*, Mar. 2020. [Cité en pages 5 et 6]

[5] “Incendie SGB2 Strasbourg : OVH condamné à verser plus de 100 000 d’euros ; - Le Monde Informatique,” Feb. 2023. [Cité en page 5]

[6] D. Merkel, “Docker : lightweight Linux containers for consistent development and deployment,” *Linux Journal*, Mar. 2014. [Cité en pages 5 et 10]

[7] J. Simonsson, L. Zhang, B. Morin, B. Baudry, and M. Monperrus, “Observability and chaos engineering on system calls for containerized applications in Docker,” *Future Generation Computer Systems*, Sept. 2021. [Cité en pages 6, 7, 10, 17, 23, 27, 28, 33, 44 et 45]

[8] C. Rosenthal, L. Hochstein, A. Blohowiak, N. Jones, and A. Basiri, “Chaos Engineering,” [Cité en page 6]

[9] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, “Chaos Engineering,” *IEEE Software*, May 2016. [Cité en pages 6, 11 et 22]

[10] L. Leite, C. Rocha, F. Kon, D. Milojicic, and P. Meirelles, “A Survey of DevOps Concepts and Challenges,” *ACM Computing Surveys*, Nov. 2020. [Cité en page 9]

[11] S. Newman, *Building Microservices*. Beijing Cambridge Farnham Köln Sebastopol Tokyo : O’Reilly, 1er édition ed., 2015. [Cité en page 10]

[12] J. Thönes, “Microservices,” *IEEE Software*, Jan. 2015. [Cité en page 10]

[13] D. Bernstein, “Containers and Cloud : From LXC to Docker to Kubernetes,” *IEEE Cloud Computing*, Sept. 2014. [Cité en page 10]

[14] K. Hightower, B. Burns, and J. Beda, *Kubernetes : Up and Running : Dive into the Future of Infrastructure.* Sebastopol, CA : O'Reilly Media, 1st edition ed., Oct. 2017. [Cité en page 10]

[15] B. Sharma and D. Nadig, "eBPF-Enhanced Complete Observability Solution for Cloud-native Microservices," Apr. 2024. [Cité en page 10]

[16] S. Hosseini, K. Barker, and J. E. Ramirez-Marquez, "A review of definitions and measures of system resilience," *Reliability Engineering & System Safety*, Jan. 2016. [Cité en page 10]

[17] "syscalls(2) - Linux manual page." [Cité en pages 11, 14, 15 et 16]

[18] K. Veeraraghavan, J. Meza, D. Chou, W. Kim, S. Margulis, S. Michelson, R. Nishtala, D. Obenshain, D. Perelman, and Y. J. Song, "Kraken : leveraging live traffic tests to identify and resolve resource utilization bottlenecks in large scale web services," in *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation*, OSDI'16, (USA), USENIX Association, Nov. 2016. [Cité en page 11]

[19] H. Kang, M. Le, and S. Tao, "Container and Microservice Driven Design for Cloud Infrastructure DevOps," in *2016 IEEE International Conference on Cloud Engineering (IC2E)*, Apr. 2016. [Cité en page 11]

[20] M. Friedrich, "From Monitoring to Observability : eBPF Chaos." [Cité en pages 11, 19, 20, 21 et 44]

[21] S. Arora, *Cloud Native Network Slice Orchestration in 5G and Beyond.* phdthesis, Sorbonne Université, Oct. 2023. [Cité en pages 11 et 21]

[22] P. Munjal, D. Gangodkar, and Y. Lohumi, "Microservices based Ticket Booking Platform deployed on Cloud," in *2023 Second International Conference on Informatics (ICI)*, Nov. 2023. [Cité en page 13]

[23] "Analyzing a decade of Linux system calls | Proceedings of the 40th International Conference on Software Engineering." [Cité en page 13]

[24] "access(2) - Linux manual page." [Cité en page 15]

[25] "errno(3) - Linux manual page." [Cité en pages 15 et 16]

[26] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design : The Hardware/Software Interface.* Amsterdam Heidelberg : Morgan Kaufmann, 4th edition ed., Nov. 2011. [Cité en page 15]

[27] W. Stallings, *Operating Systems : Internals and Design Principles.* Boston : Pearson Education Limited, 8e édition ed., 2014. [Cité en page 16]

[28] J. Wu, Z. Wang, T. Ma, L. Kong, Y. Liu, Z. Song, J. Yu, Y. Yang, T. Ma, and G. Chen, "DySched : Relieving Large-Scale Incast for Cloud-Native RDMA Applications," in *2023 IEEE Intl Conf on Parallel & Distributed Processing with*

Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom), Dec. 2023. [Cité en pages 16 et 20]

[29] D. Popescu, N. Zilberman, and A. Moore, "Characterizing the impact of network latency on cloud-based applications' performance," [object Object], Nov. 2017. [Cité en page 16]

[30] "ASSERT-KTH/royal-chaos," May 2024. [Cité en page 17]

[31] M. Usman, S. Ferlin, A. Brunstrom, and J. Taheri, "A Survey on Observability of Distributed Edge & Container-Based Microservices," *IEEE Access*, 2022. [Cité en page 19]

[32] I. Gorton, L. Fong-Jones, and A. Larsson, "Observability Q&A," *IEEE Software*, Jan. 2024. [Cité en page 19]

[33] "eBPF - Introduction." [Cité en page 20]

[34] "Bookinfo Application." [Cité en page 23]

[35] "cadvisor." [Cité en page 24]

[36] R. Love, *Linux system programming*. Beijing ; Cambridge : O'Reilly, 1st ed ed., 2007. [Cité en pages 25 et 26]

[37] M. Kerrisk, *The Linux programming interface : a Linux and UNIX system programming handbook*. San Francisco : No Starch Press, 2010. [Cité en pages 25 et 26]

[38] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*. Hoboken, NJ : John Wiley & Sons Inc, 9e édition ed., 2012. [Cité en pages 26 et 28]

[39] N. C. Mendonca, P. Jamshidi, D. Garlan, and C. Pahl, "Developing Self-Adaptive Microservice Systems : Challenges and Directions," *IEEE Software*, Mar. 2021. [Cité en pages 29 et 45]

[40] "Spinnaker." [Cité en page 29]

[41] M. Sliem, N. Salmi, and M. Ioualalen, "Performance Analysis of Self-adaptive Policies in Containerized Microservices," in *2021 International Conference on Engineering and Emerging Technologies (ICEET)*, (Istanbul, Turkey), IEEE, Oct. 2021. [Cité en pages 30, 31 et 45]

[42] L. Lei, J. Sun, K. Sun, C. Shenefiel, R. Ma, Y. Wang, and Q. Li, "SPEAKER : Split-Phase Execution of Application Containers," *International Conference on Detection of intrusions and malware, and vulnerability assessment*, 2017. [Cité en pages 33, 35, 36 et 38]

[43] X. Wang, Q. Shen, W. Luo, and P. Wu, "RSDS : Getting System Call Whitelist for Container Through Dynamic and Static Analysis," in *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, Oct. 2020. [Cité en pages 33, 34, 36, 37, 38, 43, 44 et 45]

[44] J. Jia, Y. Zhu, D. Williams, A. Arcangeli, C. Canella, H. Franke, T. Feldman-Fitzthum, D. Skarlatos, D. Gruss, and T. Xu, "Programmable System Call Security with eBPF," 2023. [Cité en pages 33, 38, 39, 40 et 44]

[45] "Seccomp security profiles for Docker," 0200. [Cité en page 34]

[46] S. Barlev, Z. Basil, S. Kohanim, R. Peleg, S. Regev, and A. Shulman-Peleg, "Secure yet usable : Protecting servers and Linux containers," *IBM Journal of Research and Development*, July 2016. [Cité en page 34]

[47] "Docker daemon configuration overview," 0100. [Cité en page 37]

[48] "StuBS : Linux Insides : Executable and Linkable Format." [Cité en page 37]

[49] "G-Wrap." [Cité en page 37]

[50] "CRIU." [Cité en page 39]