POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Approches intégrées d'analyse de performance des systèmes distribués asynchrones par trace d'exécution système

HERVÉ MBIKAYI KABAMBA
Département de génie informatique et génie logiciel

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

Décembre 2023

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Cette thèse intitulée :

Approches intégrées d'analyse de performance des systèmes distribués asynchrones par trace d'exécution système

présentée par **Hervé Mbikayi KABAMBA**
en vue de l'obtention du diplôme de *Philosophiæ Doctor*
a été dûment acceptée par le jury d'examen constitué de :

François GUIBAULT, président
Michel DAGENAIS, membre et directeur de recherche
Heng LI, membre
Ferhat KHENDEK, membre externe

DÉDICACE

À ma très chère épouse et mes enfants...

REMERCIEMENTS

En premier lieu, j'aimerais porter mes remerciements à mon directeur de recherche, Michel Dagenais, qui a su croire en moi et qui a manifesté beaucoup d'intérêt à mon égard en acceptant d'encadrer cette thèse. Présent dans toutes nos difficultés, il a été un guide hors pair, un véritable pilote disposé durant tout notre cheminement à fournir encouragements et conseils constructifs, à partager connaissances et expériences qui ont été les éléments motivateurs de la poursuite et la réalisation de ce travail.

Je tiens à exprimer ma profonde gratitude à tous les associés de recherche, en particulier Geneviève Bastien et Arnaud Fiorini pour leur aide tout au long de ce cheminement qui arrive à son terme.

Mes remerciements s'adressent aussi à tous les membres du laboratoire de recherche DORSAL, et en particulier à Adel Belkhiri, Pierre, Iman, Quentin, Sébastien, Benjamin et Mariam.

Je remercie aussi le CRSNG (Conseil de Recherches en Sciences Naturelles et en Génie du Canada), le Conseil Régional du Nord-Pas de Calais-Picardie, le Ministère de l'Enseignement Supérieur et de la Recherche, et tous les partenaires du laboratoire, Ericsson, Ciena, AMD, EfficiOS et Prompt, pour leur soutien financier.

J'exprime une profonde gratitude à tous mes enseignant(e)s et à tous les employé(e)s de Polytechnique Montréal.

Mes pensées vont aussi vers mes nombreux amis et frères qui ont été une source d'encouragement et de motivation pour la réalisation de ce travail : Glory Mali, Michel Bokolo, Serge Mpwate, Nono. Je vous en suis infiniment reconnaissant.

Mes remerciements s'adressent aussi à tous mes collègues de services dans mon pays d'origine et aux différentes autorités académiques au sein de la Haute École de Commerce de Kinshasa, qui m'ont beaucoup soutenu et encouragé.

Enfin, je remercie de manière particulière ma famille, qui a toujours été là pour moi : mon père et ma mère, qui ont beaucoup attendu de voir la réalisation de cette œuvre, sont les personnes les plus heureuses sur cette terre à présent. Puissent mes frères et sœurs : Carine, Ornella, Linda, Cynthia, Daniel, Jonathan, Benjamin et Divine, trouver ici l'expression de mon grand amour pour eux.

RÉSUMÉ

Les avancées technologiques récentes ont entraîné l'émergence de nouveaux paradigmes dans le développement d'applications, la surveillance de leurs performances et la gestion des ressources nécessaires. Les systèmes distribués ont résolu de nombreux problèmes qui étaient impossibles à aborder avec les architectures monolithiques, offrant des avantages tels que la résilience et l'élasticité pour gérer la charge utilisateur de manière dynamique.

Cependant, malgré les avantages indéniables de ces architectures, elles posent également des défis en termes de débogage des performances. Les systèmes distribués utilisent des composants autonomes qui interagissent de manière coordonnée pour atteindre un résultat souhaité. Ils fonctionnent généralement de manière asynchrone, ce qui signifie que le système peut passer à d'autres tâches après avoir envoyé un message ou mis une tâche en attente, sans bloquer le processus.

L'asynchronisme peut être étudié selon deux approches d'implémentation à savoir : les systèmes asynchrones multicouches et les systèmes asynchrones unicouche. Dans le premier cas, hormis le module d'orchestration d'événements, le système est constitué de plusieurs couches interdépendantes qui interagissent dans le but d'exécuter des tâches. Ces systèmes sont extrêmement complexes et leur débogage nécessite une prise en compte de cette dimension. Dans le second cas, le système est implémenté simplement avec une couche d'orchestration qui assure l'asynchronisme. Le débogage et l'analyse de performance dans ce cas, appellent à des méthodes permettant de discriminer les contextes d'exécution des différents événements entrants.

Ce mode de fonctionnement complexifie l'architecture et rend l'analyse des performances difficile avec les méthodes actuelles. Les systèmes asynchrones complexes tels que `Node.js` intègrent plusieurs couches internes qui coopèrent pour exécuter des tâches. Par exemple, la couche supérieure traite le code JavaScript, le transmet à une couche d'orchestration appelée Libuv, qui coordonne l'exécution, puis la tâche est soumise à la machine virtuelle pour exécution.

Le suivi des performances dans un tel système devient difficile car il est bati sur une architecture mono-thread, ce qui complique la distinction des contextes d'exécution lors du suivi d'une tâche spécifique. Les développeurs ont proposé une API expérimentale pour suivre le cycle de vie des ressources asynchrones, mais son utilisation entraîne un surcoût considérable pour le système.

La recherche présentée dans cette thèse se concentre sur de nouvelles méthodes de gestion des contextes de la trace dans les systèmes asynchrones. Nous proposons des méthodes d'instrumentation efficaces avec un faible surcoût, et des méthodes d'analyse permettant le débogage des performances avec un niveau de granularité élevé.

Pour mettre en œuvre nos approches, nous avons instrumenté deux plates-formes populaires et matures, à savoir `Node.js` et `Redis`. Nous avons d'abord instrumenté Node.js avec un traceur particulier pour contourner les surcoûts induits par son API `Asynch Hooks`. Les différentes couches internes ont été instrumentées pour permettre l'identification de modèles d'orchestration et de coopération internes récurrents à partir de la trace collectée, afin de guider le développement de nos analyses. Le développement de vues nous permet d'interroger les analyses pour visualiser les informations pertinentes sur le fonctionnement du système, et identifier les problèmes de performance.

Dans le but de minimiser l'effort d'instrumentation des applications asynchrones, la deuxième partie du travail se penche sur des méthodes de collecte transparente de données télémétriques, sans intervention de l'utilisateur. Nous avons introduit une technique de reconstruction du contexte des requêtes en interne, appelée ITTCR, permettant de reconstruire le flux d'interactions entre microservices de manière transparente, et de produire des visualisations pour comprendre les performances du système.

Enfin, la troisième étape de ce travail s'est concentrée sur l'analyse des performances de `Redis`, qui apporte la composante de stockage au système et implémente un modèle d'asynchronisme différent de celui de Node.js. Nous avons proposé une approche basée sur un modèle d'abstraction pour analyser les performances de `Redis`, identifiant avec précision de nombreux problèmes de performances, voire des bogues inhérents à ce système.

Une approche d'intégration des analyses hétérogènes a été proposée pour créer un cadre unifié permettant de définir la granularité souhaitée pour l'analyse. Pour ce faire, nous avons développé une technique permettant l'intégration transparente des analyses développées pour `Node.js` et `Redis` dans un cadre unifié. Ce cadre offre une vue globale sur l'ensemble du système distribué pour identifier les problèmes de performance de haut niveau, et utilise les analyses intégrées pour approfondir l'étude et remonter à la source du problème.

Les approches sont proposées de manière à faciliter leur extension sur d'autres systèmes asynchrones disponibles. L'étude du fonctionnement des modules d'orchestration relatif à chaque système permet d'adapter et d'affiner les modèles proposés, ainsi que l'implémentation de nos approches pour l'analyse des performances sur différents systèmes. L'orchestration des événements étant la fonctionnalité clé étudiée et adressée par nos approches.

ABSTRACT

The significant advancements in technology have triggered various new paradigms in application development, performance monitoring, and resource management. Distributed systems have resolved many problems that were unfeasible with monolithic architectures, offering benefits such as system resilience, scalability, and elasticity for dynamic user load management.

However, despite the undeniable advantages of these architectures, they also pose parallel concerns regarding performance debugging. Distributed systems employ autonomous components that interact cohesively to achieve the desired outcomes. They typically operate asynchronously, allowing the system to proceed with other tasks after sending a message or queuing a task without blocking the process.

Asynchronousness can be studied through two implementation approaches, namely: multi-layer asynchronous systems and single-layer asynchronous systems. In the former case, apart from the event orchestration module, the system consists of several interdependent layers that interact with the aim of executing tasks. These systems are highly complex, and debugging them necessitates consideration of this dimension. In the latter case, the system is implemented with only one orchestration layer that ensures asynchronousness. Debugging and performance analysis in this case require methods that allow for the discrimination of execution contexts for various incoming events.

This mode of operation increases the complexity of the architecture and complicates performance analysis with current methods. Complex asynchronous systems like Node.js incorporate multiple internal layers that cooperate to execute tasks. For instance, the uppermost layer processes JavaScript code, delegates it to a lower-level orchestration layer called Libuv, which coordinates execution, and subsequently submits the task to the virtual machine for execution.

Tracking performance in such a system becomes challenging as it is typically single-threaded, making it difficult to distinguish execution contexts for monitoring a specific task. Developers have proposed an experimental API to trace the lifecycle of asynchronous resources; however, its use incurs a substantial overhead on the system.

The research presented in this thesis focuses on novel methods for managing trace contexts in asynchronous systems. We propose efficient instrumentation methods with minimal overhead and analyses for high-granularity performance debugging.

To implement our approaches, we instrumented two popular and mature platforms, namely Node.js and Redis. We initially instrumented Node.js with a specific tracer to circumvent the overhead induced by its Asynch Hooks API. Different internal layers were instrumented to identify recurrent patterns of orchestration and internal cooperation from the collected trace data, guiding the development of our analyses. The developed views allow querying the analyses to visualize relevant information about the system operation, and identify performance issues.

In an effort to minimize the instrumentation effort for asynchronous applications, the second part of the work explores methods for transparently collecting telemetry data without user intervention. We introduced an internal request context reconstruction technique called ITTCR, enabling seamless reconstruction of the interaction flow between microservices and producing visualizations to comprehend system performance.

Finally, the third stage of this work focused on the performance analysis of Redis, which provides the storage component to the system and implements a different asynchronous model than Node.js. We proposed an approach based on an abstraction model to analyze Redis performance, accurately identifying numerous performance issues and even revealing inherent bugs in the system.

An approach for integrating heterogeneous analyses was suggested to create a unified framework that allows defining the desired level of granularity for analysis. To achieve this, we developed a technique for seamless integration of analyses developed for Node.js and Redis into a unified framework. This framework offers a holistic view of the entire distributed system, to identify high-level performance issues, and leverages the integrated analyses to conduct in-depth analysis and trace the root causes of the problems.

The approaches are designed to facilitate their extension to other available asynchronous systems. Studying the operation of orchestration modules for each system allows for adapting and refining the proposed models as well as implementing our approaches for performance analysis on different systems. Event orchestration is the key functionality examined and addressed by our approaches.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

IETF	Internet Engineering Task Force
OSI	Open Systems Interconnection
API	Application Programming Interface
ITTCR	Internal Trace Context Reconstruction
CTF	Common Trace File
I/O	Input/Output
E/S	Entrée/sortie
VSM	Vertical Span Model
TC	Trace Compass
OS	Operating System
ITTR	Internal Trace Context Reconstruction

CHAPITRE 1 INTRODUCTION

Les grandes avancées qu'a connues la technologie ont conduit à de nouveaux paradigmes dans la construction des systèmes et des architectures informatiques. Des systèmes qui autrefois adoptaient le monolithisme comme approche de fonctionnement ont connu un changement drastique de paradigme, en empruntant des architectures de construction beaucoup plus résilientes, évolutives, tolérantes aux pannes et sécuritaires.

Cette évolution a talonné la prolifération des environnements infonuagiques, qui ont assoupli le déploiement des applications distribuées. Cette avancée permet d'optimiser l'expérience utilisateur et assurer la disponibilité, la performance et surtout la résilience des applications.

Les systèmes distribués répondent aujourd'hui à un éventail de besoins, que ce soit en termes de développement, de mise à jour ou de mise à l'échelle. Ils offrent élasticité et robustesse, et changent le paradigme de déploiement en mettant l'accent sur l'automatisation.

Un système distribué est appelé à assurer que ses composants soient capables de coopérer et de se coordonner pour produire le résultat attendu. Les interactions entre les composants se font par l'échange de messages légers, via un bus de communication réseau, de manière synchrone ou asynchrone. Dans les opérations asynchrones, un composant revient à son état initial après l'envoi d'un message, sans interrompre son processus lors de l'attente de la réponse. On parle de systèmes asynchrones distribués.

Ces systèmes, de par leur architecture et leur fonctionnement, augmentent la complexité du suivi de leur performance, et appellent à des outils spécifiques permettant d'extraire les données télémétriques, devant être soumises à des analyses pour comprendre leur fonctionnement. À la différence des autres environnements, ces derniers intègrent généralement une couche d'orchestration qui complexifie l'analyse de performance, et son degré dépend du modèle employé pour la conception du système.

Ce projet de recherche se focalise sur l'analyse des performances des systèmes distribués asynchrones. L'objectif est de déterminer si les méthodes et outils actuellement disponibles, sont aptes à réaliser des analyses efficaces dans ces types d'architectures. Le rôle du chercheur est d'identifier ces lacunes, de proposer de nouvelles approches et de nouveaux paradigmes permettant de pallier ces insuffisances.

Dans la première partie du travail, l'état de l'art en rapport avec ce type d'architecture sera abordé. Un parcours sur les différents outils d'analyse de performance qui s'embarquent dans l'écosystème sera effectué. Étant donné que l'asynchronisme dans un contexte distribué

complexifie les approches d'analyse de performances granulaires, les méthodes d'instrumentation efficaces et peu intrusives seront étudiées. Ces méthodes permettront de collecter l'information nécessaire aux analyses post-exécution, par le biais du traçage du système. Le développeur pourra visualiser l'information nécessaire à l'interprétation du fonctionnement de son système.

Dans la seconde partie, les approches visant à assurer la transparence dans la collecte des informations dans ces architectures seront particulièrement investiguées. L'étude visera à proposer de nouveaux paradigmes de traçage transparent, limitant les efforts fournis par les opérateurs et développeurs dans les environnements distribués asynchrones complexes. La seconde étape visera à étudier et proposer des mécanismes d'intégration des approches d'analyse de performance hétérogènes. L'objectif visé est de définir un cadre intégral, permettant d'assembler les différentes analyses définies, pour construire un outil unifié pour l'analyse de performance des systèmes distribués asynchrones.

Ce travail vise à améliorer la qualité et la polyvalence des outils d'analyse de performance appliqués à des contextes distribués asynchrones, eu égard des insuffisances relevées.

De façon générale, cette recherche tente de répondre aux questions suivantes : (1) Les méthodes et outils d'analyse des performances des systèmes distribués existants sont-ils capables de permettre une analyse granulaire dans le contexte des systèmes distribués asynchrones ? (2) Comment apporter des améliorations aux méthodes actuelles de traçage dans les environnements distribués asynchrones complexes, dans le but de limiter les efforts du développeur ? (3) Comment produire un cadre d'analyse de performance qui rassemble les différentes analyses des systèmes asynchrones hétérogènes, dans un outil unifié et intégré, offrant observabilité et granularité à l'analyse de performance conduite ?

1.1 Objectifs de la recherche

Les systèmes distribués asynchrones sont implémentés sur la base de modèles de conception divers. Leurs architectures sont d'autant plus complexes qu'elles intègrent généralement des couches additives d'orchestration internes. Le débogage de performance dans ces architectures demeure une tâche extrêmement complexe, de par leur nature. Le traçage demeure une solution très efficace pour avoir une compréhension approfondie du fonctionnement du système. Les causalités liées aux interactions entre les différents composants dans ces architectures, sont généralement établies par l'emploi des outils existants. Cependant, ces derniers fournissent une vue de niveau très élévé sur les services invoqués dans les différents nœuds de l'infrastructure.

Pour optimiser la consommation des ressources, les systèmes asynchrones sont généralement construits sur la base des modèles mono-threads, qui permettent d'implémenter une couche d'orchestration responsable de la gestion des événements. Du côté du système d'exploitation, ce système est vu comme une boîte noire effectuant des opérations nécessitant des changements de contexte. Il est alors difficile de discriminer les opérations sur la base de leurs contextes de causalité, et de les lier aux fonctions ou requêtes qui les ont initiées dans un contexte distribué. Le débogage des problèmes de performance dans ces systèmes, appelle à des analyses granulaires permettant de visualiser les interactions et leur lien de causalité de manière détaillée, en vue d'être capable de remonter à la cause d'un problème particulier.

Une revue exhaustive de la littérature en rapport avec le contexte de la recherche, combinée à des expérimentations avec les outils existants, nous ont permis d'identifier les limites des approches actuelles, en vue de proposer les améliorations qui s'adaptent pleinement au contexte des systèmes distribués asynchrones.

Cette thèse vise à fournir des solutions efficaces pour l'analyse de performance des systèmes distribués asynchrones, en proposant de nouvelles approches, en les comparant à celles existantes sur la base des cas d'utilisation pratiques et réels, et en établissant une discussion critique sur les résultats obtenus en précédence à la proposition des améliorations futures.

Elle s'articule autour de trois projets principapux :

1. **Étude approfondie, revue des outils de traçage, et développement des analyses adaptées aux architectures distribuées asynchrones** : Les systèmes asynchrones sont des systèmes complexes de par leur architecture de fonctionnement. Ils implémentent des mécanismes qui leur permettent de gérer des nombres élevés d'événements concurrents, en optimisant la consommation des ressources internes. Pour atteindre ce niveau de performance, ils intègrent dans leur architecture une couche d'orchestration, permettant de gérer les différents événements sans pour autant obstruer le fonctionnement du processus principal. Pour le cas de systèmes asynchrones très complexes, la complexité est d'autant plus accrue par le contexte multicouches de leur architecture, qui impose une coordination verticale et une coopération interne des différents composants du système, dans le but de renvoyer un résultat. Cette complexité peut premièrement être vue comme atomique, du fait qu'elle se rapporte au fonctionnement du système isolé dans un nœud particulier de l'infrastructure. La complexité s'accroît de nouveau lorsque ce nœud est un composant d'une architecture distribuée. Dans ce cas, le débogage de problèmes de performance doit considérer l'état du système dans sa globalité.

Ce premier axe de recherche caractérise les modèles de communication et d'interaction interne des composants dans un système asynchrone multicouche, et propose de nouvelles méthodes d'analyse de performance. La première démarche propose une méthode d'instrumentation directe des fonctions de haut-niveau de `Node.js` en proposant l'utilisation des modules natifs. Cette première étape résulte en l'obtention d'une trace d'exécution permettant de visualiser le fonctionnement global du système distribué, après exécution des analyses. Ces visualisations obtenues sont comparables à celles produites par les outils de traçage distribué.

L'objectif de ces visualisations est de permettre d'identifier les symptômes de dégradation des performances, qui sont une étape nécessaire pour remonter à la source du problème. Dans ce contexte, la seconde étape consiste à effectuer une étude approfondie sur les modèles de fonctionnement interne, dans un environnement multicouche. Les résultats permettent de proposer des méthodes d'analyse de traces multiniveaux, capables d'identifier les corrélations entre événements provenant des couches différentes du système, les relier aux fonctions responsables, et les connecter aux requêtes du système distribué global.

Ceci permet d'effectuer des analyses de performances du système avec un niveau de granularité très élevé. Par conséquent, les débogages des problèmes peuvent être réalisés avec précision dans ces systèmes asynchrones complexes.

2. **Architecture de collecte et d'analyse des performances transparentes des systèmes distribués asynchrones :** La collecte des données télémétriques nécessaires à la surveillance ou à l'analyse des performances des systèmes distribués passe par une phase d'instrumentation du système préalable. Elle consiste à placer des points de trace qui sont activés lorsque le système emprunte leur chemin d'exécution. Cette étape impose des efforts aux développeurs ou opérateurs dans la compréhension du code source de l'application, afin de l'instrumenter de façon adéquate.

   L'hétérogénéité des systèmes distribués appelle à des technologies et des équipes diverses, responsables de la maintenance et de la gestion de composants particuliers du système. Des coûts non négligeables sont liés à ces efforts, et n'épargnent pas le système d'une altération considérable après insertion des points de trace. Des approches permettant de déployer des infrastructures de collecte pour assurer le traçage transparent ont été proposées dans le contexte des systèmes distribués. Cependant elles soulevent deux problèmes fondamentaux dans le contexte des systèmes asynchrones distribués. Premièrement, elles impliquent des efforts de configuration de l'architec-

ture de collecte, et en même temps elles permettent de collecter de l'information de haut-niveau.

En outre, l'information de haut niveau est importante pour observer les goulots d'étranglement ou les symptômes de la dégradation de la performance du système, mais ne peut cependant pas permettre d'identifier la source du problème. Dans les architectures asynchrones distribuées complexes, l'information de haut niveau, telle que la forte latence liée à l'invocation d'une fonction particulière, ne renseigne pas sur la source réelle du problème de performance. Des méthodes qui se basent non pas seulement sur la transparence dans la collecte de la trace d'exécution, mais capables d'étendre le niveau de transparence aux analyses exécutées, sont nécessaires.

Dans ce contexte, l'identification transparente des causalités multicouches entre les événements de la trace, et leur corrélation avec les fonctions de haut niveau à l'origine de leur génération, permet le débogage précis des problèmes de performance dans les systèmes distribués asynchrones complexes. Ce second projet étudie de manière approfondie le problème en l'appliquant sur l'environnement Node.js. Premièrement, les méthodes actuelles de traçage transparent sont employées sur des systèmes distribués asynchrones complexes issus de cet environnement, afin de pouvoir identifier leurs insuffisances. Des nouvelles approches permettant de combler ces manques sont proposées et validées sur la base de cas d'utilisation réels.

3. **Architectures unifiées d'analyse de performance des systèmes distribués asynchrones :** Les systèmes asynchrones sont implémentés selon différents modèles de conception pour remplir des objectifs différents. Dans les architectures distribuées, les composants atomiques peuvent être construits sur la base de modèles asynchrones pour différentes fins, apportant ainsi de l'hétérogénéité au système dans sa globalité. La composante d'un système distribué peut être un système de messagerie pour lequel un mécanisme de publication et de souscription de messages est mis en place. Elle peut constituer une base de données en mémoire, dans le but d'optimiser les performances du système. Elle peut aussi être l'implémentation d'un microservice, responsable d'une partie de la logique métier de l'entreprise.

   Dans tous les cas, partant des composants atomiques au système distribué dans sa globalité, ces derniers peuvent être construits sur des modèles asynchrones. La granularité de l'analyse des performances de ces derniers nécessite l'exécution des analyses efficaces sur chaque composant atomique du sytème, en vue de comprendre son fonctionnement de façon détaillée, pour ainsi déboguer avec précision des problèmes de

performance. Pourtant, une architecture distribuée se compose de l'ensemble de ces sous-systèmes atomiques, qui opèrent en coordination pour fournir les résultats attendus.

Dans ce contexte, une analyse de performance du système se voulant granulaire, doit prendre en compte la dimension d'analyse de performance intégrée. L'intégration est vue comme un assemblage des différentes analyses liées à chaque composant hétérogène du système vers un cadre unifié. Un tel cadre permet d'effectuer une analyse de performance globale sur le système distribué, alors que pour chaque composant du système, la granularité nécessaire au débogage précis des problèmes de performance est définie par l'analyse qui lui est rattachée.

Dans ce troisième projet, une approche d'intégration des analyses hétérogènes des systèmes asynchrones est proposée dans le but de fournir un cadre d'étude de performance unifié pour les systèmes distribués asynchrones.

L'objectif visé dans la première phase de ce projet est de proposer des nouvelles méthodes d'instrumentation et d'analyse de performance efficaces pour les bases de données en mémoire. Nous avons porté notre choix sur le système Redis, qui implémente un modèle différent d'asynchronisme et qui renferme les fonctionnalités de stockage, de cache et de systèmes de messagerie en lui seul. Dans ce contexte, la démarche peut être généralisée aux autres systèmes asynchrones offrant ces fonctionnalités. La validation des approches d'analyses proposées est effectuée sur la base de la résolution de cas réels de problèmes d'entreprises, et des situations de problèmes évoquées par la communauté de développement.

La seconde phase du projet consiste à proposer une approche d'intégration des différentes analyses développées précédemment (`Node.js`, `Redis`), vers un cadre unifié, permettant une analyse globale des performances du système distribué asynchrone. Les résultats obtenus permettent de définir la profondeur de la granularité dans l'analyse de performance de chaque système intégré, en vue de déboguer avec précision les problèmes de performance identifiés au niveau le plus élevé.

1.2 Contributions originales

En se référant aux objectifs décrits pour cette thèse, nous présentons les contributions originales suivantes dans le domaine de l'étude des performances des applications distribuées asynchrones.

1. Une instrumentation du code source des applications dont nous avons analysé les per-

formances est proposée pour une intégration dans les versions futures de ces dernières.

2. Des analyses permettant de diagnostiquer les problèmes de performance avec un degré de précision très élevé. Ces analyses reposent sur l'identification des modèles internes récurrents d'orchestration asynchrone, en vue d'identifier les contextes des exécutions internes.
3. Une méthode transparente de collecte des traces d'exécution des microservices. Cette méthode épargne aux développeurs le besoin d'instrumenter leurs systèmes pour collecter les données télémétriques. De ce fait, elle permet d'obtenir une trace d'exécution sans intervention particulière dans l'infrastructure de collecte, d'exécuter des analyses automatisées permettant de reconstituer les flux d'exécution des différents composants distribués.
4. Un modèle d'abstraction pouvant être étendu à d'autres systèmes asynchrones. Ce dernier permet de faire abstraction des entités qui sont susceptibles d'impacter la performance du système Redis. Ce modèle une fois instancié et rempli, les visualisations développées peuvent puiser l'information pertinente à l'analyse de performance, et par conséquent, il permet de détecter avec précision les problèmes dans l'application.
5. Des interfaces interactives sont développées dans un cadriciel d'analyse de performance à sources ouvertes. Ces visualisations résultant de nos analyses permettent de faciliter l'étude des performances du système.

1.3 Plan de la thèse

Dans le Chapitre 2, une revue de la littérature est présentée. Dans ce dernier, les concepts importants et fondamentaux nécessaires à la compréhension de ce manuscrit sont introduits. Les différents travaux de recherche se rapportant à notre thématique sont passés en revue. Dans le Chapitre 3, la méthodologie que nous avons suivie pour atteindre les objectifs de notre recherche est détaillée entièrement. Les Chapitres 4, 5, 6 présentent les résultats obtenus dans le cadre de notre recherche, sous forme d'articles soumis dans différentes revues scientifiques. Dans le Chapitre 7, les résultats obtenus sont contrastés avec nos objectifs initiaux. Enfin, le chapitre 8 parachève cette thèse en tirant une conclusion et en présentant les axes potentiels pour des améliorations futures.

1.4 Publications

1. Hervé M. Kabamba, Matthew Khouzam, Michel R. Dagenais, "Node Compass : Multilevel Tracing and Debugging in JavaScript-Based Web Servers", soumis à Future

Generation Computer Systems-Elsvier

2. Hervé M. Kabamba, Matthew Khouzam, Michel R. Dagenais, “Vnode : Low-overhead Transparent Tracing of Node.js-based Microservices Architectures”, soumis à Future Internet-MDPI

3. Hervé M. Kabamba, Matthew Khouzam, Michel R. Dagenais, “Advanced Strategies for Precise and Transparent Debugging of Performance Issues in In-Memory Data Store-Based Microservices”, soumis aux Transactions on Cloud Computing-IEEE

CHAPITRE 2 REVUE DE LA LITTERATURE

L'évolution rapide de la technologie a donné lieu à divers modèles de construction des systèmes informatiques. Plusieurs architectures ont vu le jour, et chacune d'elles s'adapte à un contexte particulier. L'objectif visé est généralement la satisfaction de l'utilisateur, ce qui implique un fonctionnement optimal du système.

Par ailleurs, la performance d'un système peut se dégrader en raison de nombreux facteurs et affecter l'expérience utilisateur. Des méthodes d'analyse de la performance des systèmes informatiques sont requises, pour comprendre leur fonctionnement, et identifier des problèmes à la base de la dégradation des performances.

Ce chapitre présente une revue de littérature en rapport avec le sujet, dans le but de placer les fondations nécessaires à la compréhension de notre recherche. Il présente l'ensemble des outils et des méthodes employés dans le jargon du cadre étudié, afin de situer notre recherche dans l'écosystème du sujet.

2.1 Les Systèmes asynchrones

Les systèmes informatiques ou électroniques qui fonctionnent indépendamment, sans être synchronisés par une horloge globale, sont appelés systèmes asynchrones. Ces derniers n'ont pas de rythme ou d'ordre de fonctionnement préétablis. L'expansion des réseaux informatiques a conduit à leur émergence, et leur adoption comme les véritables moteurs de révolution de la communication entre les composants distribués. Leur non dépendance à une horloge centralisée permet de garantir une communication efficace, en se basant sur des protocoles établis [2]. L'emphase de leur conception a été mise sur la tolérance aux pannes, car leur prédisposition garantit un fonctionnement continu même en cas de composant défaillant dans le système. Ils sont donc mieux adaptés pour les environnements critiques [3]. Les systèmes asynchrones nécessitent parfois de parvenir à des consensus, ce qui implique de s'accorder sur des références communes dans des contextes de pannes. Ce défi a conduit au développement des algorithmes de consensus tel que Paxos [3].

Dans les environnements distribués, ces systèmes permettent de gérer correctement le flux de requêtes et d'optimiser la consommation des ressources. Les Entrées/Sorties asynchrones sont implémentées de manière à retourner le thread qui les gère dans le bassin de thread, sans passer à un état bloquant. Dans cet état, le thread peut continuer à recevoir des requêtes enfilées pour leur exécution. Ce principe de fonctionnement a été implémenté par plusieurs

systèmes utilisés tels que Redis [4] , Nodejs [5], Firefox [6], etc.

L'utilisation des systèmes asynchrones est aussi reconnue dans les réseaux ad-hoc, afin de permettre aux nœuds de communiquer sans avoir besoin d'une infrastructure centralisée. La recherche dans ce domaine se focalise principalement sur l'optimisation du routage, la préservation de l'énergie et la garantie de sécurité de ces réseaux [7].

2.1.1 Les systèmes basés sur la programmation réactive

Le modèle de programmation réactive repose sur l'observation des flux de données et la diffusion des modifications. Il est idéal pour les applications nécessitant une mise à jour dynamique des données, garantissant que tout changement est immédiatement visible pour l'utilisateur, ou intégré dans d'autres composants du système. Cette méthode est particulièrement bénéfique pour la création des systèmes réactifs, maintenables et robustes, surtout dans des environnements comme l'infonuagique et les architectures distribuées, où la capacité de réponse rapide du système est cruciale [8].

Au cœur de la programmation réactive se trouve la gestion des flux de données, qui sont des séquences d'événements pouvant être transformées et observées via des opérations asynchrones. Les traitements de ces données sont structurés en chaînes d'opérations, ou pipelines, réagissant aux entrées pour générer des résultats adéquats. Ces pipelines sont souvent conçus à l'aide d'opérateurs fonctionnels qui permettent une manipulation déclarative, comme le filtrage, la création, la transformation et la combinaison de flux [9].

L'un des atouts majeurs de ce modèle est qu'il autorise les opérations d'entrée/sortie et les calculs intensifs sans immobiliser le thread principal. L'asynchronisme inhérent à la programmation réactive favorise une composition fluide et une orchestration de séquences d'opérations complexes. Des librairies comme RXJava [10], et Project Reactor [11], fournissent les primitives nécessaires pour faciliter ce processus [12].

Les applications réactives sont conçues pour une meilleure gestion des erreurs et offrent une résilience accrue, permettant la construction d'applications performantes et solides, grâce à l'isolation des pannes, et la reprise rapide des opérations après un incident. Elles sont également capables de s'ajuster aux variations de charge en allouant ou en libérant des ressources, un avantage particulièrement utile en infonuagique [13].

Les concepts fondamentaux des systèmes réactifs ont été décrits dans le Manifeste Réactif de 2013, qui établit les caractéristiques essentielles telles que la réactivité, la résilience, l'élasticité et l'orientation messages de ces systèmes. Ces qualités sont vitales pour l'architecture de nombreux systèmes distribués contemporains.

Cependant, bien que la programmation réactive comporte de nombreux avantages, elle présente également des défis tels qu'une courbe d'apprentissage abrupte et une complexité inhérente lors du débogage des flux de données asynchrones. Le raisonnement sur le code peut devenir ardu, en particulier lors de la gestion de la concurrence et des états partagés [14].

Avec l'intégration de la programmation réactive dans des cadriciels et langages modernes comme l'API Stream de Java, le cadriciel Akka utilisé par Scala, RxJs pour JavaScript et les observables pour ECMAScript, cette approche est devenue plus accessible pour les développeurs, et un choix robuste pour développer des systèmes distribués et des applications interactives modernes.

2.1.2 les systèmes asynchrones basés sur la boucle d'événement

Les systèmes modernes à haute performance s'appuient fréquemment sur le paradigme des systèmes asynchrones à boucle d'événement. Cette évolution architecturale est dictée par la nécessité de performances accrues et de réactivité élevée, qui constituent des prérequis essentiels au bon fonctionnement des applications modernes. Le principe d'orchestration des tâches ou d'événements est mieux adapté pour les systèmes traitant des nombres très élevés de connexions, ou des tâches sans interruption du thread principal [15]..

L'idée derrière le modèle à boucle d'événements est de permettre au système de ne pas bloquer lors d'une opération d'E/S bloquante, et aussi de ne pas attendre la fin de l'exécution d'une tâche pour poursuivre son cours. Ces modèles sont généralement implémentés avec une architecture système mono-thread, contribuant à l'optimisation de la gestion des ressources [16]. Dans les systèmes basés sur la boucle d'événements, cette dernière constitue le socle dédié à l'orchestration des exécution des tâches. Un événement perçu par la boucle d'événements peut être une interaction de l'utilisateur à partir de son navigateur, la finalisation de la lecture d'un fichier, ou encore une connexion entrante [17].

Les fonctions de rappel (callback) sont un élément important dans ce type de système, car elles permettent d'être appelées par la boucle d'événements en vue de déléguer les opérations.

L'asynchronisme se fonde sur l'implémentation de fonctionnalités qui autorisent le système à initier une tâche et à s'inscrire pour recevoir une notification dès l'achèvement de cette dernière, permettant ainsi à la boucle d'événements de gérer simultanément d'autres activités entrantes.

Ces systèmes sont optimisés pour supporter une charge élevée de connexions simultanées tout en conservant une empreinte mémoire modeste, avantage mis en lumière par [18], lors d'une comparaison de performance entre les modèles asynchrones et synchrones dans le domaine

des applications Web.

La classification des systèmes asynchrones peut s'articuler autour de leur mécanique opérationnelle ou de l'environnement d'exécution. Certains, comme Redis [4] ou RabbitMQ [19], incarnent une approche où l'asynchronisme est intrinsèque au modèle de conception et de développement. Certains environnements sont par contre intrinsèquement asynchrones, par conséquent ils imposent deja au départ ce mode de fonctionnement aux applications dévoloppées sur leur base, on peut citer Node.js [5].

Bien que les bénéfices des systèmes asynchrones soient manifestes, leur programmation peut s'avérer complexe, souvent en raison de la difficulté à suivre le flux de contrôle et à la tendance à une imbrication excessive de fonctions de rappel, ce qui peut nuire à la lisibilité et à la maintenance du code [20]. Pour surmonter ces défis, des abstractions telles que les promesses et les fonctions async/await ont été introduites, rendant le code asynchrone plus clair et plus gérable.

L'adoption de ces systèmes et de leur mode de fonctionnement nécessite de s'assurer d'éviter d'obstruer la boucle d'événements, car l'efficacité de ce dernier en dépend [21].

2.2 Les systèmes informatiques modulaires

Une architecture modulaire est un modèle de conception de systèmes informatiques mettant l'accent sur la décomposition des applications complexes en composants plus petits, pouvant être facilement gérés et maintenus. Ces composants, appelés modules, possèdent chacun une fonctionnalité bien définie. Ils interviennent avec un rôle principal lors de la conception et du développement des logiciels, et offrent de nombreux avantages dont la maintenabilité.

La décomposition du système en plusieurs petits modules rend facile l'identification et la prise en charge des fonctionnalités nécessitant des mises à jour. Dans ce contexte, le système dans son entièreté n'est pas affecté. Un autre des avantages est le principe de responsabilité unique, qui constitue un concept fondamental en génie logiciel. La productivité est améliorée par ces architectures qui garantissent une meilleure collaboration entre les équipes de développement. Des modules spécifiques peuvent être affectés à chacune d'elles, alors que le développement intervient en parallèle. Par conséquent, ces architectures permettent de simplifier la réutilisation du code, et de favoriser la mise à l'échelle en intégrant avec facilité des modules additifs en vue de s'adapter aux besoins évolutifs, sans repenser le système dans son entièreté. La décomposition du système en modules permet l'isolation des problèmes et facilite leur identification ainsi que le débogage.

2.3 Les systèmes distribués

Un système distribué est un ensemble de composants interconnectés, répartis sur plusieurs nœuds et partageant un état commun. Ces derniers coopèrent ensemble pour un but précis et apparaissent à l'utilisateur comme une seule entité cohérente. Bien que la gestion et la conception de ces systèmes introduisent un niveau élevé de complexité, elles demeurent cependant cruciales dans le but d'optimiser leur performance, assurer une disponibilité constante et les rendre résilients aux pannes.

Les éléments constitutifs d'un système distribué fonctionnent de manière autonome et sont interconnectés par un réseau, ce qui leur permet d'échanger des messages et d'offrir aux utilisateurs une expérience unifiée et cohérente [22]. Ces systèmes offrent de substantiels bénéfices, notamment en termes de distribution des charges, propulsant ainsi l'amélioration des performances systémiques. Ils renforcent la tolérance aux pannes, circonscrivant les incidents à un nœud spécifique, sans impacter l'ensemble du système. De plus, ils sont conçus pour être évolutifs, gérant l'augmentation des capacités pour s'adapter aux besoins croissants [23].

Néanmoins, la complexité de ces systèmes implique des défis significatifs. La gestion de la cohérence des données est primordiale ; il est impératif que les données répliquées restent en parfaite synchronisation à travers les nœuds. La latence du réseau doit être gérée avec précision, car elle peut influencer négativement l'expérience utilisateur. En outre, la sécurité prend une dimension critique, les risques étant amplifiés dans un environnement distribué avec plusieurs vecteurs d'attaque potentiels [24].

La recherche dans ce domaine progresse rapidement, avec des avancées notables dans les technologies telles que l'infonuagique, les systèmes de fichiers distribués, les chaînes de blocs et l'internet des objets, qui s'appuient sur les principes des systèmes distribués pour leur fonctionnement efficace à grande échelle [25].

2.3.1 L' architecture microservice

L'architecture microservice se positionne comme une méthode de développement d'applications, qui se base sur la subdivision de ces dernières en un ensemble de composants faiblement couplés. Sa popularité est due au fait de la flexibilité qu'elle offre en termes de simplicité dans la livraison continue, et des capacités d'adaptation qu'elle apporte aux applications en termes de mise à l'échelle [26].

Dans une architecture microservice, les composants sont autonomes et responsables d'une fonctionnalité particulière de l'application. Les composants communiquent par des protocoles légers, bien définis par le biais des APIs. L'un des éléments importants offerts par cette

architecture est l'indépendance des composants, qui assure la souplesse aux équipes de développement dans la mise à jour et le déploiement des différents services, sans perturbation de l'application. Le cycle de développement se voit raccourci et une plus grande agilité est assurée [27].

L'architecture microservice présente de nombreux avantages en termes de modularité, élément fondamental pour assurer la maintenance et l'évolution du code [28]. Mieux, l'indépendance est assurée lors du déploiement des services, réduisant de surcroît les probables conflits entre les différents composants difficilement gérables.

Les microservices ont la particularité d'offrir de l'élasticité de par leur architecture, étant donné leur capacité à être mis à l'échelle ou redimensionnés de manière indépendante, sans nécessiter de s'attaquer au système dans sa globalité [29].

Les grands avantages que ces architectures apportent peuvent cependant être balancés par les nombreux défis qui les accompagnent, car ces derniers introduisent une complexité au niveau conceptuel et surtout au niveau du débogage. Cela inclut la capacité de pouvoir gérer un afflux de transactions distribuées [30].

Dans une architecture microservice, la complexité appelle parfois à des outils d'orchestration et de gestion tels que des conteneurs, permettant de les emballer et de les déployer avec facilité [31].

2.3.2 Conteneurisation

La conteneurisation permet de développer des applications en incluant toutes les dépendances et les différents fichiers de configuration nécessaires à leur exécution, dans différents environnements, depuis une machine de développement jusqu'à plusieurs serveurs. Elle repose sur le principe de construire, expédier et exécuter n'importe quelle application, dans différents environnements. Cela évite le problème lié à l'exécution du code de la machine du développeur à un serveur de production.

Kubernetes [32], et Docker [33], sont les technologies les plus utilisées. Docker permet de créer des conteneurs d'applications, tandis que Kubernetes est utilisé pour leur déploiement. Ce dernier fournit également des fonctionnalités de mise à l'échelle et d'équilibrage de charge.

Une application déployée nécessite une surveillance constante, pour s'assurer qu'elle maintient un niveau élevé de performance en production. Une gestion et une surveillance proactives aident à prévenir les défaillances du système. Logstash, ELK, Nagios, Prometheus et Grafana sont des exemples de systèmes de surveillance [34–38].

2.4 Analyse de performance des systèmes distribués

Les systèmes distribués sont constitués de plusieurs composants inter-communicants, afin de fournir des services aux utilisateurs. Ils sont également très exigeants en termes de performance. Il est essentiel de mettre en œuvre des mesures de surveillance et de collecte d'information, pour fournir des détails sur la santé du système en production dans un tel contexte. Lorsque ces applications sont déployées en infonuagique, la tâche devient d'autant plus difficile, qu'il faut prendre en compte et calibrer des mécanismes appropriés, pour obtenir une vision globale de l'état du système, ou de son fonctionnement, sur une période donnée.

Plusieurs techniques informatiques existantes permettent d'investiguer en profondeur, les problèmes de performance observés dans le système. Cela nécessite des connaissances approfondies sur la manière d'interpréter les informations collectées, et les corrélations qui peuvent exister avec les problèmes apparaissant [39].

Les architectures distribuées sont soumises à des exigences de performances strictes, qui nécessitent une surveillance constante du système afin d'avoir une vue sur son fonctionnement. Dans une telle architecture, l'un des problèmes importants étudiés par la recherche, est la mise en œuvre de solutions adéquates permettant d'avoir une vue globale de l'ensemble des performances du système.

Dans cette section, différentes approches et outils d'analyse de performance sont présentés. Nous commençons premièrement par définir les concepts importants employés dans le jargon de l'analyse de performance, afin d'éloigner toute ambiguité, et progressivement nous présenterons l'état de l'art en rapport avec l'analyse de performance. Nous clôturons en abordant les outils existants, nécessaires à l'entreprise d'une analyse de performance.

2.4.1 Instrumentation

L'instrumentation d'un système ou d'une application implique l'ajout de portions de code à des endroits spécifiques du système cible, pour collecter des informations sur l'application en cours d'exécution. Il cible l'analyse des performances du système, le diagnostic de problèmes spécifiques et le traçage de ce dernier [40]. L'instrumentation peut être de deux types : l'instrumentation du code source et l'instrumentation dynamique. Dans le cas de l'instrumentation du code source de l'application, le temps de recompilation représente un coût supplémentaire, car le programme doit être redémarré pour s'exécuter avec ses nouvelles propriétés. De plus, l'accès au code source de l'application est requis[41].

En revanche, pour l'instrumentation dynamique, le code est injecté éventuellement sans aucune connaissance du code source de l'application au moment de l'exécution [42].

Chacune de ces méthodes d'instrumentation possède des propriétés intéressantes, selon le contexte de leur application. La majorité des outils d'analyse des performances basés sur l'infonuagique utilisent une instrumentation statique, car elle peut améliorer la qualité du logiciel [43]. La méthode d'instrumentation dynamique permet d'obtenir une vision raisonnablement claire du flux des opérations et du chemin d'exécution du code logiciel. Elle fournit une grande quantité d'informations, et permet de cibler les modules qui ont un rôle déterminant lors de l'exécution[44][45].

2.4.2 Traceur

La complexité et les exigences des applications monolithiques et distribuées dictent des mécanismes pour collecter des informations pertinentes sur leur exécution, afin d'évaluer et de comparer le niveau de performance requis. La simple constatation d'une dégradation des performances par l'utilisateur ne renseigne pas sur l'origine du problème. Il est crucial de collecter une certaine quantité d'information sur l'exécution du système en vue d'effectuer une analyse permettant d'identifier la source du problème.

Un traceur est un outil qui aide à comprendre le comportement du système. Il collecte des informations sur son exécution et produit une trace. Les traceurs induisent une perturbation qu'on espère minimale du système, afin de réduire les surcoût d'exécution dans son ensemble. Principalement, il collectera des informations au niveau de l'espace utilisateur et de l'espace noyau. Le choix d'un traceur dépend de l'approche utilisée pour réaliser l'analyse de performance. L'instrumentation du code ou du système est alors nécessaire pour placer des points de trace permettant de capturer des événements spécifiques lors de l'exécution du système.

2.4.3 Débogueur

Un débogueur est utilisé pour inspecter l'exécution d'un programme qui ne se comporte pas comme prévu. Cela permet d'isoler les points défectueux. Il fige l'exécution d'un programme à un moment bien choisi, et inspecte l'état dans lequel se trouve ce dernier. La meilleure utilisation d'un débogueur consiste à choisir ces points d'arrêt en fonction de la structure du programme. Cela peut être considéré comme un processus utilisé pour supprimer les bogues du code[46][47].

Il utilise également des simulateurs de jeux d'instructions, au lieu d'exécuter un programme directement sur le processeur, pour obtenir un niveau de contrôle plus élevé sur son exécution. Il existe de nombreuses façons d'y parvenir. Par exemple, le débogage peut être effectué en analysant le code pour détecter les fautes de frappe, ou en utilisant un analyseur de code. Le

débogage du code se fait également à l'aide d'un profileur de performances. Un débogueur est un outil de développement très spécialisé qui s'attache à une application en cours d'exécution, pour permettre l'inspection du code et de la mémoire.

2.4.4 Traçage distribué

Magpie [48], un outil efficace sous Windows NT, est un précurseur et implémente plusieurs fonctionnalités du traçage distribué. Il offre la possibilité d'instrumenter plusieurs sous-systèmes du noyau Windows NT. Il récupère les données de trace des différentes machines. La synchronisation est appliquée pour résoudre le problème du délai d'horloge. Magpie aide également à identifier les événements liés de manière causale, pour exposer une vue cohérente de chaque demande d'utilisateur. De plus, il prend en compte l'induction d'une charge minimale lors du traçage. Il est basé sur l'infrastructure Event Tracing for Windows (EWT) disponible dans Windows NT. Magpie utilise le traçage de bas niveau pour exposer les événements qui permettent de séquencer une requête RPC lorsqu'elle traverse les différents composants du système distribué, lors de son envoi ou de sa réception.

L'infrastructure embarquée d'EWT permet également d'extraire des informations sur les accès au disque et la consommation du processeur à l'aide d'événements de bas niveau. La charge du système est influencée par les différentes requêtes exécutées, empruntant différents chemins à travers le système et consommant les ressources différemment. Magpie permet d'enregistrer l'avancement fonctionnel des requêtes, et la consommation de ressources par ces dernières à chaque étape. Comparativement à la version utilisée dans [48], elle est moins intrusive, car elle évite d'injecter le contexte via un identifiant, comme présenté dans[49][50].

Mike Y. Che et al. ont proposé une approche basée sur le chemin, pour gérer les défaillances et l'évolution des systèmes distribués. Ils se sont concentrés sur les interactions entre les composants du système. Un chemin est une collection d'observations. Il peut s'agir de mesures locales discrètes du système, à différentes étapes d'une requête. Les observations peuvent inclure l'horodatage, le nom du composant ou le nom d'hôte. La mise en œuvre de cette approche utilise le traçage, comme indiqué dans[48][49], et associe chaque requête à un identifiant. Cependant, contrairement à ce dernier, il enregistre des informations supplémentaires telles que les dépendances des ressources et les numéros de version pour permettre d'effectuer une analyse de détection des dysfonctionnements du système.

L'identifiant est ainsi accessible à tous les niveaux du système dans lequel se déroule l'observation. La propagation du contexte utilise trois méthodes : (i) Le contexte est propagé en utilisant la structure de stockage locale du thread chargé de stocker l'identifiant, si les requêtes sont traitées les unes après les autres. (ii) La deuxième approche consiste à injecter

directement le contexte dans les entêtes des protocoles HTTP et SOAP. (iii) La troisième approche consiste carrément à modifier les protocoles, interfaces ou membres de classe existants. Cette approche est assez intrusive, car elle augmente la complexité de l'instrumentation.

Un autre traceur, Webmon[51], injecte le contexte sous la forme d'un identifiant et le propage tout au long du cycle de vie des requêtes, qui émanent du navigateur, vers le serveur Web et le serveur d'applications à l'aide de cookies HTTP. Ce dernier permet de tracer la requête pour en créer un profil en fonction du chemin.

Le principe de l'identifiant unique a été introduit avec X-Trace par Fonseca et al. [52]. Cet identifiant doit suivre le même chemin que la requête. Ainsi les informations nécessaires au traçage sont définies, et sur la base de l'identifiant unique, une trace de bout en bout est obtenue. De plus, pour minimiser la consommation de la bande passante allouée aux requêtes, les données de trace sont agrégées en empruntant différents chemins. La dimension sécurité est également prise en compte, car les informations collectées par le traceur ne sont accessibles qu'aux utilisateurs concernés.

Sigelman et al.[39] ont présenté Dapper. Dapper est un traceur distribué développé pour fonctionner sur l'infrastructure de Google, afin de permettre l'analyse de performance des applications typiques. Il base principalement son fonctionnement sur les concepts de X-Trace[52]. Son apparition a assuré le développement d'applications distribuées intégrant la dimension d'analyse de performance. Il s'appuie sur l'instrumentation de bibliothèques partagées, comme celles qui gèrent les appels de procédures distantes, les threads d'exécution et les tâches asynchrones.

Il permet de sauvegarder l'état des requêtes traitées, dans la mémoire du thread d'exécution, par le processus en cours. Dapper permet aux développeurs d'annoter la trace distribuée, bien que l'instrumentation permette de suivre les requêtes sans intervention du développeur. L'intérêt est cependant de permettre à la trace de contenir des informations spécifiques et nécessaires au débogage. Une interface utilisateur dédiée permet de visualiser la trace collectée. La fonctionnalité d'échantillonnage probabiliste des requêtes Dapper facilite la mise à l'échelle et la gestion du coût supplémentaire généré par l'instrumentation.

Une évolution qui en a résulté a depuis été observée dans l'industrie du logiciel, inspirée par Dapper, et donnant naissance à plusieurs autres outils comme Zipkin [53], développé par Twitter, et qui fut ensuite relâché comme logiciel libre. Ce développement a rendu nécessaire la normalisation de l'API et l'orientation de l'instrumentation du traçage distribué. Ainsi, le projet OpenTracing est apparu et propose une API standard pour tracer les applications distribuées, non liée à un fournisseur particulier.

En fonction de la manière dont les données collectées sont envoyées au système, deux types de traceurs distribués seront considérés :

Transfert direct de trace

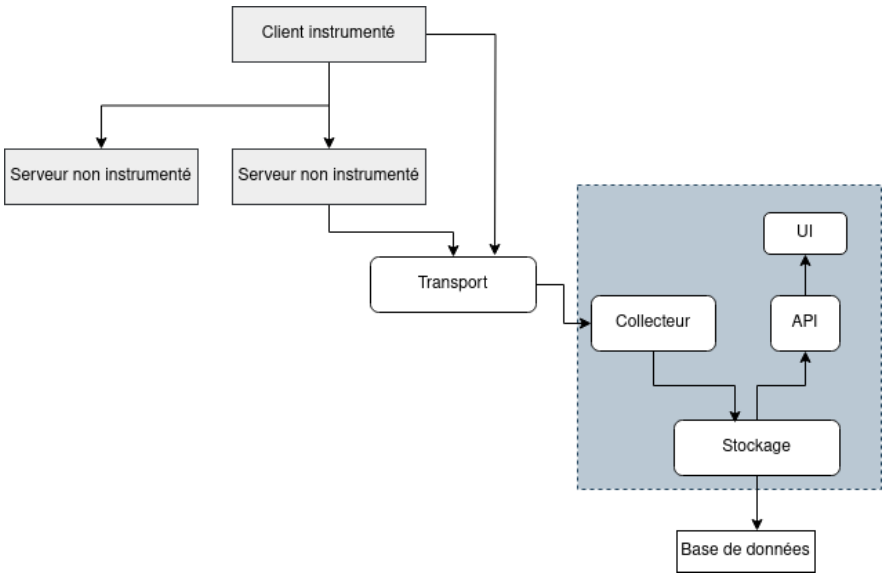


FIGURE 2.1 Architecture de fonctionnement de Zipkin

Twitter a développé Zipkin, un traceur que Dapper a finalement inspiré. Il prend en charge les protocoles HTTP Kafka et Scribe pour envoyer des informations au collecteur. Son stockage est compatible avec Elasticsearch et Cassandra, et dispose d'une interface utilisateur. Il prend en charge l'échantillonnage pour limiter le volume de données envoyées au collecteur et maintenir un haut niveau de performance. Il prend aussi en charge l'instrumentation et l'envoi de données asynchrones, par plusieurs langages de programmation. La Figure 2.1 présente son architecture et les interactions entre les différents composants.

Transfert basé sur un agent installé

Développé par Uber, Jaeger s'est inspiré de Dapper mais aussi de Zipkin. La différence avec Zipkin réside dans l'architecture et les fonctionnalités ajoutées. Jaeger utilise un agent, qui est un démon réseau, déployé sur n'importe quel hôte, en tant qu'agent d'infrastructure. Il permet de faire abstraction du routage et de la découverte du jaeger-collector du client. Il utilise des tampons pour recevoir les traces UDP envoyées par les hôtes locaux et permet le contrôle de flux pour assurer une transmission efficace des traces. Il aide à identifier la stratégie d'échantillonnage correcte, grâce à son infrastructure, et prend en charge un large

éventail de langages de programmation. La figure 2.7 représente l'architecture opérationnelle de Jaeger.

2.4.5 Normes de traçage distribué

La nécessité de surveiller les performances des applications distribuées a donné naissance au paradigme du traçage distribué. Ce dernier est particulièrement adapté aux applications microservices, et permet de collecter des mesures de performances d'applications largement distribuées, qui se basent sur des concepts et spécifications particuliers. Ce paradigme a guidé la conception et le développement de plusieurs outils de traçage distribués commerciaux et en logiciel libre.

Le besoin de disposer d'outils de collecte de mesures de performances étant élevé, il est apparu nécessaire de normaliser ces approches pour assurer une instrumentation des applications, indépendamment d'une bibliothèque ou d'un cadriciel particulier, laissant ainsi au développeur le soin d'utiliser les outils qu'il juge adaptés à un contexte particulier. Cette motivation a donné naissance à deux standards principaux : OpenTracing et OpenTelemetry.

OpenTracing

OpenTracing[54] est une API indépendante du fournisseur, qui aide les développeurs à instrumenter le code de leurs applications pour les tracer. De plus, elle permet à de nombreuses entreprises produisant des outils de traçage de l'adopter de manière standardisée, pour instrumenter des applications. L'objectif est de parler un langage commun sur ce qui est considéré comme une trace et une instrumentation de code, pour effectuer du traçage distribué.

OpenCensus

OpenCensus est une collection de bibliothèques de traçage. Il a été lancé chez Google en tant que bibliothèque de recensement interne, mais n'a été publié en tant que projet en logiciel libre qu'en janvier 2018. Contrairement à OpenTracing, qui ne traite que des spécifications de traçage distribué, OpenCensus est une bibliothèque permettant de collecter des mesures de traçage distribué et de séries chronologiques à partir d'applications. De plus, l'outil dorsal d'observabilité choisi par le développeur offre une vue plus intégrée et complète des performances de l'application, puisque les métriques et les traces qu'il collecte partagent les mêmes métadonnées et les mêmes balises de propagation de contexte.

OpenTelemetry

Le besoin intense de trouver la meilleure façon de générer, collecter et analyser les données de télémétrie des applications, a conduit les développeurs depuis plusieurs années à réfléchir à la meilleure façon d'y parvenir. Dapper, apparu en 2010, a mis en œuvre une approche de télémétrie et de traçage qui a donné naissance au projet open source OpenCensus. Il a fusionné plus tard avec OpenTracing pour devenir OpenTelemetry. OpenTelemetry propose un standard unique, ajouté à diverses technologies, pour capturer et exporter des traces, des métriques, des journaux d'applications et des infrastructures instrumentées.

— **Collecteur :** Le collecteur est un composant qui s'exécute à proximité des composants d'application utilisateur et reçoit des étendues de trace et des métriques des récepteurs pris en charge. Il y parvient grâce à des modules instrumentés, permettant aux clients d'envoyer des métriques de span directement ou via des agents spécifiques. Le collecteur fournit un point de sortie central pour exporter des traces et des métriques, vers un ou plusieurs consommateurs de traces ou de métriques, tout en fournissant une mise en mémoire tampon et une agrégation, un filtrage, une annotation et un échantillonnage intelligent avancés. Le rôle du collecteur est d'écrire les données de trace dans le stockage, qui seront ensuite interrogées pour analyse [55].

— **Contexte :** Pour que l'arborescence de trace soit construite avec toutes ses relations, chaque intervalle propage son contexte à ses fils (requêtes imbriquées). Le contexte de trace contient différentes informations transmises entre les fonctions au sein d'un processus, ou entre les processus via des appels de procédure distante. Le contexte indique l'ID de l'intervalle parent et l'ID de trace. Dans ce processus, l'intervalle fils génère son ID, puis le propage en tant qu'ID parent et ID de trace, dans le contexte de son intervalle fils.

  Plusieurs autres informations peuvent être présentes dans le contexte, mais l'ID d'intervalle parent et l'ID de trace permettent de créer l'arborescence de trace.

— **Durée :** Un intervalle est une vue des différentes étapes par lesquelles passe une requête dans un système distribué. Il est annoté d'un identifiant, ainsi que de son temps d'exécution. Une trace du système peut être obtenue en assemblant tous les intervalles collectés par le traceur. La trace apparaît comme une hiérarchie d'intervalles, associés à leur temps d'exécution ainsi qu'à leurs relations. La trace commence par un intervalle racine qui peut avoir plusieurs intervalles fils. L'ensemble de la trace forme un arbre hiérarchique.

2.4.6 Service en maillage

La complexité des architectures des services modernes soulève des défis en termes d'orchestration et de gestion dans les environements distribués, particulièrement dans un contexte d'utilisation des conteneurs et des microservices. Une des solutions pour pallier ce problème est l'adoption des services en maillage.

Un service en maillage [56], est une infrastructure configurable de bas-niveau, permettant de gérer la communication entre les différents composants d'une application, particulièrement des infrastructures microservices. Il permet de faciliter les interactions complexes entre les services, telles que la répartition des charges, la tolérance aux pannes, l'observabilité et la sécurité des communications.

Dans le cas des architectures microservice, la découverte des services est cruciale, étant donnée la complexité liée au nombre des interactions qui y surviennent. Les services en maillage assurent une gestion convenable en maintenant un registre de tous les instances et services disponibles. Un service peut alors interroger le maillage pour avoir l'adresse d'un autre service, en vue de communiquer avec ce dernier. De cette façon, la communication est simplifiée, épargnant aux développeurs le besoin d'intégrer les adresses IP et les numéros de port dans le code des applications.

Les maillages permettent de gérer le trafic, le déploiement progressif (canary release), ainsi que la répartition de charge dynamique, des éléments cruciaux dans la fluidité de la production et l'optimisation des risques associés aux déploiements de nouvelles versions de l'application.

Du point de vue de la sécurité, les services en maillage permettent de sécuriser les communications entre services par le biais du chiffrement de bout en bout et de l'authentification mutuelle. Cela assure un cryptage des données dans les échanges, qui repose sur une authentification préalable avant toute communication entre services.

Les services en maillage peuvent offrir de la résilience au système en fournissant des modèles de résilience permettant de gérer les défaillances localisées du système, sans pour autant affecter l'expérience utilisateur finale.

Dans un tel environnement, l'observabilité constitue un élément important pour assurer le fonctionnement optimal du système. Les services en maillage permettent de collecter et d'analyser les métriques des journaux et des traces, pour permettre aux opérateurs d'avoir une vue globale sur le fonctionnement du système. Le diagnostic et la détection de problèmes peuvent être effectués de manière rapide, grâce à des outils tels que Prometheus [57], dans le cas du suivi des métriques, et Jaeger [58], pour le traçage distribué dans des opérations dans le système.

Parmi les solutions de service en maillage les plus populaires, Istio [59], et Linkerd [60], se distinguent. Istio, intégré étroitement avec Kubernetes, offre une plateforme complète pour gérer les microservices avec une attention particulière sur la sécurité et la robustesse. Linkerd, quant à lui, est reconnu pour sa simplicité et sa légèreté, offrant des fonctionnalités de base pour la plupart des besoins, sans l'encombrement de fonctionnalités avancées qui peuvent être superflues pour certaines équipes.

Bien qu'il offre de nombreux avantages, l'implémentation d'un service maillage apporte en contrepartie des défis à considérer avec beaucoup d'attention. Sa complexité opérationnelle est un des éléments à prendre en compte. En outre, l'implémentation de ces solutions implique l'ajout d'une couche additionnelle au niveau de l'infrastructure, ce qui nécessite une expertise et augmente les coûts liés à la disponibilité des ressources additives. En sus, la dimension performance doit être prise sérieusement en compte, car la gestion du trafic et des fonctionnalités de sécurité peuvent introduire une latence supplémentaire.

Malgré ces avantages, l'implémentation d'un service en maillage n'est pas sans défis. La complexité opérationnelle est l'une des préoccupations majeures. L'ajout d'une couche supplémentaire d'infrastructure nécessite une expertise et des ressources pour la gestion et le maintien en condition opérationnelle. De plus, il faut tenir compte de la performance, car la gestion du trafic et les fonctionnalités de sécurité peuvent introduire une latence supplémentaire.

2.4.7 Traçage Distribué Multi-couches

Le traçage permet d'obtenir des informations détaillées, sur l'exécution d'un système particulier, pour connaître les changements d'état des ressources. Dans le cas du traçage distribué, les latences des requêtes peuvent être observées. De plus, il est possible de comprendre la logique interne du système [61]. Bien que cela aboutisse à la détection de bugs et à l'optimisation des performances, les informations sont généralement de haut niveau [61]. La dégradation des performances du système peut avoir plusieurs origines.

Elle peut résulter d'un problème matériel imprévu, de la contention ou de la saturation de la bande passante, pour n'en nommer que quelques-uns. De telles informations ne peuvent pas être obtenues par un traçage de haut niveau. Plusieurs travaux sur les approches de traçage et les analyses multicouches ont été menés afin de pouvoir corréler les événements et identifier les causes profondes des problèmes dans les deux espaces (niveaux utilisateur et noyau).

Bationo et al. [61] ont étudié l'analyse de performance d'OpenStack, en utilisant l'approche multicouches de traçage. Les traces sont collectées aux niveaux de l'espace utilisateur et du

noyau. Bien qu'un problème de performance soit observé au niveau utilisateur, il peut être le résultat d'un problème sur une autre couche. Puisque les machines virtuelles sont considérées au niveau le plus bas comme des ressources partagées, les processus respectifs seront en concurrence avec d'autres processus dans le système pour accéder au CPU. Lorsqu'un autre processus monopolise ce dernier pendant une période prolongée, l'impact se ressent au niveau des couches virtuelles supérieures.

Il est difficile de détecter de tels problèmes sans utiliser la combinaison d'informations à plusieurs niveaux. Ce cas, parmi tant d'autres, démontre suffisamment la valeur d'adopter des approches de traçage hybrides pour analyser la performance des systèmes distribués. Cette approche implique l'instrumentation de méthodes spécifiques, impliquées dans le cycle de vie de la machine virtuelle du projet OpenStack Nova. L'instrumentation est basée sur l'activité de journalisation de Nova, ce qui simplifie le placement des points de trace.

Le traçage se fait sur trois couches, au niveau de l'application OpenStack, de la couche de virtualisation et du noyau. La combinaison de ces informations aide à identifier les corrélations entre elles et à détecter d'éventuels problèmes de performance. Yu Gan et al. [62] ont développé Seer, un débogueur en ligne capable de détecter et de prédire les problèmes de performance à l'avance, et d'empêcher leur propagation aux services dépendants dans l'infrastructure infonuagique. Le système est basé sur l'apprentissage profond. Il utilise d'énormes données de trace collectées au fil du temps, pour former des réseaux neuronaux artificiels, basés sur des motifs temporels et spatiaux qui résultent en violations de la qualité de service.

L'approche combine le traçage de haut niveau des requêtes RPC avec des données de bas niveau pour détecter les violations potentielles de QoS et prédire la propagation imminente du problème de performance. Seer utilise un système de traçage distribué, pour collecter des informations d'exécution de requête de bout en bout qui incluent la latence, et il associe les RPC qui appartiennent à la même requête pour les agréger dans une base de données Cassandra centralisée. Ces traces sont ensuite utilisées pour former Seer à reconnaître, dans l'espace et le temps, les motifs liés à la violation de la qualité de service.

Après avoir détecté une violation, le traçage de bas niveau est utilisé pour accéder aux compteurs de performance afin d'identifier les raisons derrière la violation. Un traceur distribué, comparable à Dapper [39] et Zipkin [53] dans l'enregistrement de la latence des micro-services, est développé sur la base de l'interface de mesure de temps Thrift. Seer suit le nombre de requêtes en file d'attente pour chaque micro-service, puisqu'il existe une corrélation entre les longues files d'attente et la performance des micro-services, comme discuté dans [63] [64] [65] [66]. Deux types d'instrumentation sont appliqués. Pour connaître les latences des différentes requêtes, l'instrumentation de l'application se fait au niveau RPC. La deuxième couche

d'instrumentation est dans l'application Memcached, afin d'accéder à des informations de bas niveau.

Dan Ardelean et al. [67] se sont penchés sur la collecte des données nécessaires à l'analyse de la performance des applications infonuagiques. Leurs expériences ont été menées sur des données Gmail contenant plus de 1 million de comptes utilisateurs. Il décrit deux techniques pour collecter des données exploitables à partir de systèmes en production. En raison de la difficulté de réaliser leur expérimentation sur le système de production, des modèles statistiques sont utilisés pour prédire les résultats de l'expérimentation avant de la réaliser sur le système de production.

Les deux techniques employées dans leur approche pour la collecte d'informations sont (i) le traçage par rafales coordonné, qui permet la collecte de rafales de traces à travers toutes les couches logicielles, sans nécessiter de coordination explicite. Le début et la fin de la collecte sont basés sur le temps pour la coordination. La collecte de cet ensemble d'informations multicouches permet d'obtenir un échantillonnage aléatoire pour conclure l'analyse de performance. (ii) L'injection verticale du contexte : plusieurs problèmes peuvent résulter des interactions entre différentes couches logicielles, d'où l'importance de corréler les événements d'une couche particulière à une autre couche.

L'injection de contexte vertical a été adoptée pour résoudre le problème de séquençage des événements hybrides dans la trace. Contrairement à [52] et [39], leur approche ne nécessite pas la propagation explicite du contexte de trace à travers les couches, mais repose sur le fait qu'une opération RPC peut générer des événements au niveau du noyau. L'analyse des données collectées au niveau des différentes couches permet l'annotation de la trace obtenue, avec des informations permettant d'associer les événements RPC de haut niveau avec les appels système dans l'espace noyau. Cela repose sur le fait que faire une séquence stylisée d'appels système inoffensifs à des couches supérieures peut injecter des informations dans la trace du noyau.

2.4.8 Outils de traçage

LTTng

LTTng [68] est un traceur qui s'est imposé sous Linux comme étant robuste, riche en fonctionnalités et à faible surcoût. Il repose sur des points de trace statiques aux niveaux du noyau et de l'application. Il permet aux développeurs d'instrumenter leurs applications au niveau de l'espace utilisateur. Il offre des fonctionnalités de traçage au niveau du noyau et de l'utilisateur avec une faible surcharge. Il est en effet l'outil de traçage par défaut sous Linux pour les

applications temps réel, en raison de sa capacité à s'échelonner avec le nombre de cœurs, tout en maintenant un surcoût minimal proche de zéro. Au-delà du fait qu'il permet l'insertion de points de trace statiques, il peut également ajouter des points de trace dynamiques en utilisant les kprobes. Les points de trace statiques sont ajoutés au niveau du noyau et au

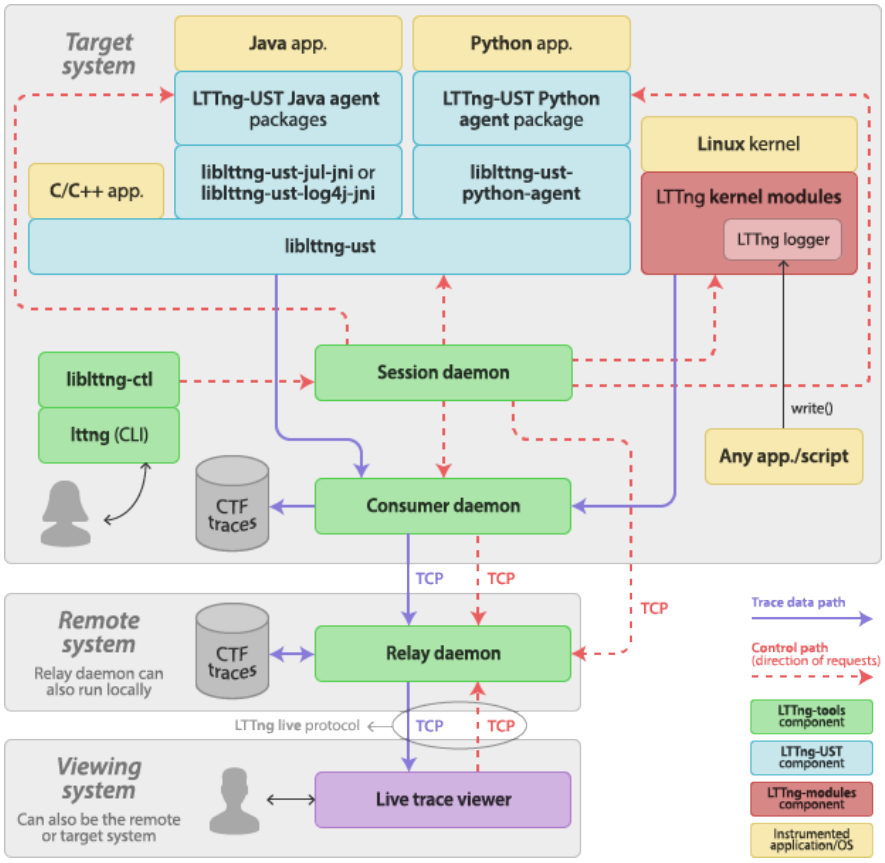


FIGURE 2.2 LTTng components and their tracing path [1]

niveau du programme utilisateur en utilisant UST (User Space Tracer). Il utilise le Common Trace Format (CTF) pour exporter efficacement les traces sur disque. Les composants inclus dans LTTng, et leurs interactions avec l'application et le noyau Linux, peuvent être vus dans la figure 2.8. Le démon de session est le composant le plus crucial, car il contrôle et gère les autres composants.

Les modules du noyau LTTng consistent en des sondes attachées aux points de trace du noyau Linux, ainsi qu'aux points d'entrée et de sortie des fonctions système. Il utilise un tampon circulaire pour communiquer les événements tracés au démon consommateur. C'est la boîte à outils de traçage la plus rapide sous Linux, grâce à son implémentation de tampon sans verrou.

2.4.9 Ftrace

Ftrace permet de tracer des événements dans le noyau [69]. Il fait partie du noyau Linux principal et utilise, comme LTTng, des points de trace statiques insérés à cet effet dans le noyau Linux. Il offre la possibilité d'insérer dynamiquement de nouveaux points de trace grâce aux kprobes. Les points de trace insérés dans le noyau se présentent sous la forme d'une macro TRACE_EVENT [70] que partagent d'autres traceurs au niveau du noyau. Le système de fichiers tracefs [71] permet de gérer les points de trace de ftrace. Cependant, il n'offre pas de traçage au niveau utilisateur.

eBPF

L'un des outils les plus récents est le Filtre de Paquets de Berkeley Étendu (eBPF) [72]. Bien qu'il ne soit pas considéré comme un traceur en soi, eBPF permet l'agrégation, l'analyse et le stockage d'événements. eBPF est une amélioration du BPF, disponible dans les versions antérieures de Linux. eBPF fait plus que le simple filtrage de paquets. Il est disponible dans la série de noyaux Linux 4.x et permet à l'utilisateur d'écrire de petites sections de code et de les insérer n'importe où dans le noyau en utilisant Kprobes. Ces programmes écrits par l'utilisateur peuvent accéder à certaines structures à l'intérieur du noyau et effectuer des opérations simples suivies par l'utilisateur [73]. Le surcoût en temps d'exécution peut être comparé à celui de **perf**, tout en étant assez flexible, car il permet d'exécuter du code arbitraire dans le noyau à des fins de surveillance.

Le code eBPF est compilé à l'exécution et inséré dans une petite machine virtuelle BPF pour être exécuté. Les événements sont stockés en mémoire, grâce à un tampon **perf** et peuvent être exportés sous forme d'événements au format CTF et écrits sur disque. eBPF est un outil efficace pour l'agrégation et la surveillance en ligne. La figure 2.9 montre le programme compilé par eBPF avec LLVM, qui est ensuite injecté dans le noyau par l'appel système bpf. Le vérificateur du noyau assurera ensuite la sécurité, attachera l'interface de contrôle du trafic et exécutera le programme dans une petite machine virtuelle.

Perf

Perf est un outil intégré dans le noyau Linux qui peut utiliser les points de trace présents [74]. Sa principale force est sa capacité à échantillonner les compteurs de performance matériels pour les processus. Il peut aider à déterminer l'optimisation d'une application sur le matériel en enregistrant l'évolution du nombre de fautes de cache ou de TLB [75]. Il intègre plusieurs outils, permettant le profilage ou la détection des contentions de verrouillage. Grâce aux

kprobes, il permet d'insérer dynamiquement des points de trace dans le noyau, et utilise les uprobes pour les insérer dans les applications au niveau de l'espace utilisateur.

SystemTap

SystemTap est un traceur permettant de réaliser une instrumentation dynamique et de collecter des traces en utilisant la macro TRACE_EVENT. Il utilise des scripts pour écrire le code de traçage. Son format est similaire au langage C. Pour être insérés dans le noyau Linux sous forme de modules, les scripts sont convertis en C. L'instrumentation dynamique est réalisée à l'aide de Kprobes. Cependant, il pose des défis spécifiques en termes de performance pour la collecte de traces [76].

2.4.10 BabelTrace

BabelTrace s'ajoute à la liste des outils de traçage comme référence pour la production du format CTF. Ce dernier est un format binaire pour les systèmes logiciels. C'est un format de trace compact permettant d'écrire des événements de traces de manière optimisée sur le disque. BabelTrace intègre les capacités de conversion des traces en différents formats, afin d'effectuer le débogage et l'analyse de performance des systèmes modernes, incluant les systèmes embarqués et les systèmes d'exploitation. Le format CTF se constitue d'un entête, d'un contexte, et d'une liste variable d'arguments pour chaque événement. Les événements sont classés en ordre chronologique en se basant sur l'estampille de temps.

Le contexte contient des informations associées à un événement particulier. Elles peuvent inclure l'identifiant du processus à la base de la génération de ce dernier, et le numéro de processeur (CPU) qui l'a sauvegardé en mémoire. La liste d'arguments inclut les données de l'utilisateur nécessitant d'être exportées par le biais de l'événement.

L'outil BabelTrace a été développé dans le cadre du projet Diamon (Open Source Diagnostics and Monitoring Tools) et est présentement maintenu par EfficiOS Inc. Babel Trace occupe une part importante dans l'écosystème des outils de traçage et ses utilisateurs tirent profit de sa flexibilité et de ses capacités de permettre des analyses granulaires des systèmes informatiques. La lecture des traces au format CTF sur terminal peut être effectuée par un outil portant aussi le nom de BabelTrace et faisant partie de la suite d'outils et des bibliothèques qu'il apporte.

Au-delà de ses capacités de s'intégrer facilement dans les processus de développement des applications, il est capable de traiter les traces en ligne et ainsi offrir aux développeurs la possibilité de filtrer et de transformer les données de trace en temps réel [77].

2.4.11 Trace Compass

Trace Compass est un puissant analyseur de traces et de logs open source permettant la visualisation et l'analyse de performance des applications informatiques. Dans l'écosystème de traçage, Trace Compass s'utilise de manière complémentaire à d'autres technologies telles que LTTng, Babeltrace, et Eclipse. Sa particularité réside dans ses capacités à permettre l'analyse détaillée des traces d'exécution des applications, et des systèmes d'exploitation. Il offre une interface interactive et intuitive, ainsi que de nombreuses vues et analyses pour interpréter les données présentes dans la trace.

Il vise à offrir aux développeurs et analystes de systèmes une meilleure vue et une compréhension fluide du comportement des applications. Il a la particularité de gérer des de gros volumes de données de trace, un élément crucial dans les environnements de production complexes.

Les traces peuvent être vues comme des empreintes digitales des événements système, qui enregistrent les actions et les différentes interactions survenant au sein de l'infrastructure informatique. Trace Compass est capable de supporter de multiples formats de trace tels que CTF, et est largement adopté en industrie en raison de sa compatibilité et de sa flexibilité.

Les traces peuvent être facilement chargées, organisées efficacement et parcourues par les utilisateurs, à l'aide de divers filtres et recherches. Sa suite d'outils de visualisation est l'une de ses forces. Les utilisateurs peuvent choisir des graphiques chronologiques pour visualiser les événements à travers le temps, afin de comprendre l'ordonnancement des opérations, et déboguer des problèmes de concurrence ou d'interblocage. L'utilisation des ressources peut être visualisée grace à des vues intégrées, permettant ainsi au développeur d'identifier les goulots d'étranglements et optimiser la performance.

Trace Compass est conçu pour s'intégrer de manière simple avec d'autres outils. Il peut être utilisé en tant que plug-in dans l'environnement de développement intégré (IDE) Eclipse. Dans ce cas, il permet aux développeurs de travailler dans un environnement qui leur est familier, en vue de d'accélérer le processus de développement et d'analyse. Il est aussi extensible, par conséquent les développeurs ont la possibilité de créer leurs propres analyses et des vues particulières pour répondre à des besoins uniques.

2.5 Synthèse

Un ensemble d'outils, d'approches et de méthodes se rapportant à notre travail a été abordé dans ce Chapitre.

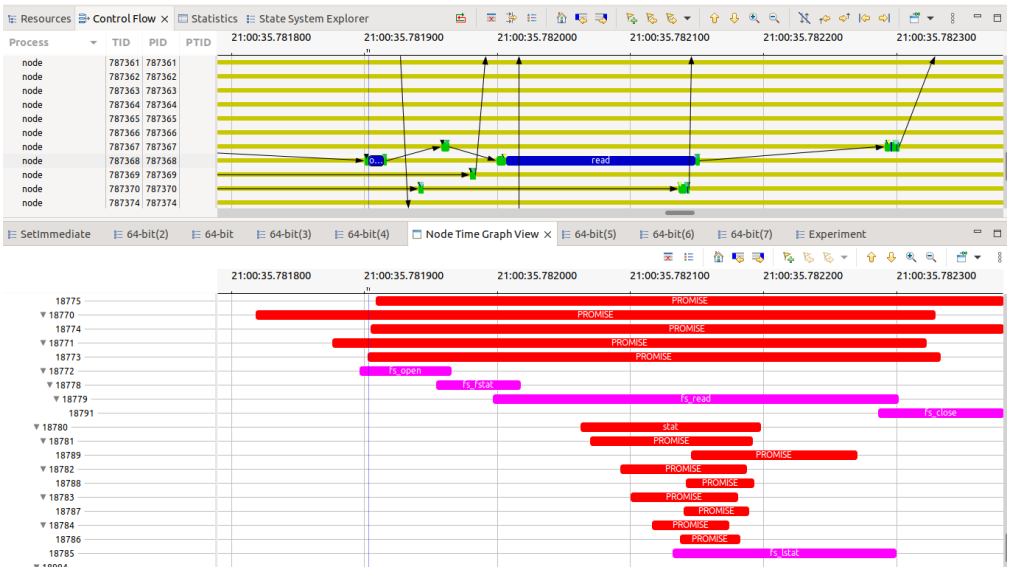


FIGURE 2.3 Apperçu des vues d'analyse dans Trace Compass

Les aspects convergents de l'étude identifient en particulier les systèmes distribués asynchrones et se penchent vers les méthodes et les outils efficaces pour analyser leur performance. Les systèmes distribués apportent plusieurs avantages, car ces modèles de fonctionnement des systèmes introduisent une dimension de résilience, d'indépendance et d'élasticité. Par ailleurs, la complexité de mise en place nécessite l'emploi des modèles éprouvés pour gérer la cohérence des données, et de leurs copies, à travers les différents nœuds du système. Autant la complexité induite par le fonctionnement de ces derniers est un élément à prendre sérieusement en compte, autant il est nécessaire de choisir les outils spécifiques pour répondre aux besoins adéquats des fonctionnalités du système.

Un système distribué nécessite un monitoring permanent afin de collecter les données télémétriques permettant de diagnostiquer des problèmes qui pourraient survenir. Aussi, dans leur fonctionnement, ces systèmes étant bâtis sur des technologies diverses, l'hybridation liée à la composition hétérogène fait appel à des connaissances diversifiées, comme cela se constate dans le contexte des architectures microservice.

Dans de telles architectures, les technologies utilisées peuvent être de différents ordres, généralement des technologies bâties sur les modèles asynchrones. En général, de tels systèmes permettent à leurs composants de communiquer ou de traiter des événements, sans pour autant bloquer le processus principal. Les systèmes peuvent alors continuer à effectuer différentes autres tâches en revenant à leurs états initiaux. Le contexte d'exécution de la tâche ou de la requête en cours de traitement est alors préservé par plusieurs mécanismes, dépendamment de la technologie utilisée. Par exemple, dans le cas des systèmes de bases de données

en mémoire, tels que Redis, le numéro de descripteur de fichier est utilisé pour conserver le contexte pendant l'exécution de la requête. Node.js, en revanche, utilise un moteur interne pour gérer le contexte des ressources asynchrones, ainsi que leur cycle de vie. La complexité de tels systèmes, qui joignent à eux seuls la distribution des composants et l'asynchronisme, constitue un véritable défi pour l'analyse de performance, et en particulier le débogage de problèmes de performance.

Un résumé des outils a aussi été effectué, ces derniers sont parfois adaptés à des analyses dans les systèmes distribués, certains sont idéaux pour des fonctionnements atomiques sur des composants individuels. Ces outils peuvent par contre être employés ensemble pour produire des approches d'analyse de performance granulaires et précises, en termes d'identification des problèmes dans les systèmes distribués.

La recherche s'est beaucoup penchée sur les méthodes de traçage distribué, qui offrent la possibilité de suivre le parcours d'une requête à travers les différents nœuds qu'elle traverse, en vue d'identifier des goulots d'étranglement et d'avoir un diagnostic sur la performance du système. De même, de nombreux outils permettent de surveiller la santé du système en temps réel, grâce à des métriques prédéfinies et fournies par des API proposées par plusieurs systèmes.

Le traçage permet de collecter un ensemble d'informations sur l'exécution du système en vue de comprendre son comportement, et le cas échéant remonter à la source d'un problème. Dans les environnements distribués, des traceurs distribués sont utilisés à cet effet. Ils nécessitent une étape préalable d'instrumentation, ce qui implique la modification du code source de l'application en vue d'assurer la collecte des informations nécessaires. Ces opérations apportent un surcoût au niveau du système, et la quantification de ce dernier est un élément important concernant la manière de procéder ou la technologie à utiliser. Cela inclut le type de traceurs, les méthodes d'échantillonnage, etc. Hormis le coût lié à l'instrumentation, la compréhension du code et sa modification nécessitent des efforts considérables en termes de ressources humaines.

Plusieurs recherches ont été effectuées pour éliminer le coût engendré par la phase d'instrumentation en proposant des méthodes transparentes de traçage. Ce chapitre a abordé l'état de l'art en la matière en décrivant ces approches et leurs limites. L'une des approches consiste à l'utilisation des services en maillage qui sont une manière d'insérer des couches additionnelles de bas-niveau pour gérer la communication entre microservices. Elles sont configurables et leurs interactions peuvent être interceptées pour extraire des informations nécessaires au traçage.

Des méthodes telles que l'utilisation des mandataires pour intercepter les appels système

responsables de la communication entre composantes, dans une architecture distribuée, ont été proposées. Le point commun entre ces deux approches est la nécessité de configurer l'environnement de collecte associé au système pour son déploiement, en vue d'assurer la transparence dans le traçage du système. Cette étape nécessite l'intervention de l'opérateur dans la configuration de l'infrastructure qui est parfois complexe, modifiant ainsi les procédures de déploiement standards dans le pipeline de l'opérateur-développeur.

Cependant, ces approches offrent de la transparence dans la collecte de données, mais ces dernières étant basées sur les interactions réseaux, ne peuvent reconstruire le contexte vertical d'exécution d'une requête pour identifier, par exemple, un problème de performance lié à l'exécution d'une fonction particulière, un bogue, ou des erreurs logiques dans le système. Des outils basés sur des approches de traçages totalement transparents et qui poussent l'analyse vers plus de granularité sont cruciaux pour assurer le débogage des problèmes de performance dans les systèmes modernes actuels.

À la différence des approches existantes, cette recherche vise à proposer de nouvelles approches d'analyse de performance qui affinent la démarche en élevant le niveau de granularité dans l'analyse. Dans le contexte des environnements asynchrones, ces méthodes permettent d'étudier en interne les interactions entre les couches associées à l'exécution d'une tâche dans le cas des systèmes asynchrones multicouches, ainsi que le module d'orchestration dans le cas des systèmes asynchrones en général.

Ceci permet de comprendre et d'exposer avec précision les problèmes de performance et leurs causes lorsqu'ils interviennent. Ainsi, cette recherche propose des méthodes qui emploient à la fois les approches basées sur les traceurs distribués et les traceurs logiciels dans une démarche intégrée pour développer des analyses précises et adaptées aux systèmes asynchrones.

CHAPITRE 3 METHODOLOGIE

Ce chapitre traite de la méthodologie employée par cette recherche pour atteindre ses objectifs. D'une façon générale, l'analyse de performance impose deux principales démarches préalables. La collecte des données dans une première phase, et une seconde phase consistant en leur analyse. Les sections suivantes mettent en lumière ces démarches, ainsi que le contexte technique dans lequel elles ont été appliquées.

3.1 Collecte des données

Le débogage des problèmes de performance dans les systèmes distribués requiert généralement d'entreprendre une phase de collecte des données. L'architecture de collecte se base sur le traçage qui permet de suivre les états du système, et d'extraire l'information qui les renseigne en vue de la soumettre à une analyse. Le traçage nécessite une phase d'instrumentation préalable, dans laquelle des points de trace sont insérés à des endroits précis du système ou de son environnement, en vue de générer des événements lorsque le système emprunte leur chemin.

Pour assurer le traçage du système, nous avons choisi d'utiliser LTTng, qui est un traceur connu pour le faible surcoût qu'il induit au système tracé. Il est donc pratique pour les environnements en production.

La technique d'instrumentation employée est l'instrumentation statique, qui consiste à insérer les points de traces LTTng directement dans le code source de l'application. Le traçage s'est effectué à deux niveaux : au niveau de l'espace utilisateur et au niveau de l'espace noyau du système d'exploitation. LTTng a l'avantage de permettre le traçage combiné dans les deux espaces, ce qui évite de gérer la synchronisation des événements, lorsque des analyses doivent être effectuées sur les deux espaces.

L'impact de l'instrumentation sur le comportement du système, ou sur ses performances, est un élément à prendre sérieusement en compte car, si cette dernière n'est pas correctement effectuée, le surcoût induit peut devenir difficilement supportable pour le système, et pire le comportement du système peut être sérieusement altéré.

Pour minimiser l'impact sur les performances du système, nous avons optimisé l'instrumentation en faisant des compromis sur la granularité requise par les différentes analyses. Des compromis ont été faits sur le type et la quantité des données à extraire, versus le niveau de performance minimal attendu du système.

Cependant, lorsque l'accès au code source de l'application n'est pas possible, l'instrumentation dynamique est recommandée, bien qu'elle soit prédisposée à induire un surcoût plus élevé au système. Cette dernière consiste en l'injection des points de trace dans les fichiers binaires de l'application. Elle peut intervenir lors de la compilation ou lors de l'édition de liens. Les points de trace peuvent aussi être insérés directement dans l'espace mémoire de l'application, au cours de son exécution.

3.2 Analyses des données

L'objectif du traçage d'une application est de collecter les données qui renseignent sur les différents états du système. Ces données appelées événements sont généralement produites dans un fichier de trace de taille variable. Cependant, dépendamment de la granularité de l'instrumentation, et surtout de la durée du traçage, le volume du fichier peut rapidement augmenter et atteindre des tailles très grandes en contenant des millions, voire des milliards d'événements. Il devient alors difficile, voire impossible, d'en extraire humainement de l'information pour comprendre le fonctionnement du système. De plus, les événements capturés lors du traçage nécessitent généralement d'être corrélés à d'autres événements pour établir l'état réel du système, et comprendre les détails de son exécution à travers une période de temps. Il est alors crucial de développer des analyses efficaces, permettant de simplifier l'interprétation des données incluses dans la trace en vue d'en extraire les informations pertinentes.

3.3 Méthodes d'analyse

L'analyse repose sur des méthodes permettant de faire abstraction du comportement du système tracé. Cela permet de comprendre le fonctionnement du système à un niveau plus élevé, en évitant de naviguer sur les nombreux détails de la trace, et ainsi faciliter le débogage de problèmes de performance.

Une des approches que nous employons consiste à modéliser le système sous forme d'un automate à états finis, construit sur la base des événements de la trace collectée. Les événements sont les déclencheurs des transitions, incitant le système à changer d'état. Dans un tel contexte, les diagnostics des problèmes de performance peuvent être réalisés en analysant la cohérence des états de ce dernier, à la lumière de ses interactions avec les différentes entités avec lesquelles il interagit, et aussi en analysant l'état des ressources qui lui sont attachées.

La seconde approche d'analyse que nous avons employée consiste à définir des métriques qui caractérisent le système et qui permettent de quantifier le degré de ses performances dans des contextes particuliers.

Ces deux techniques d'analyse se basent sur l'extraction des informations de la trace. De manière générale, les attributs des événements sont extraits selon un modèle préalablement défini et sont sauvegardés dans une structure de données particulière appelée SHT (State History Tree) [78]. Cette structure de données est optimisée pour des accès logarithmiques, elle est évolutive tout en prenant en charge de nombreux formats, et supporte de très grandes quantités de données.

3.4 Évaluation des coûts

L'analyse de performance d'un système engendre un surcoût associé à la collecte des données. Pourtant, dans le cas des systèmes en production, il est impératif de répondre à des exigences strictes de performance. Lorsque le surcoût passe au-delà d'un certain seuil, le comportement de l'application devient altéré, ses performances se dégradent très rapidement, et entachent parfois celles du système qui l'exécute.

Dépendamment des applications tracées, le seuil du surcoût peut être différent et tolérable pour certaines, et insupportable pour d'autres. Dans le cas des applications temps réel, le temps étant la contrainte principale, toute instrumentation doit veiller tant soit peu à préserver les coûts dans une marge qui ne compromet pas les caractéristiques de l'application. Cela appelle à l'emploi des méthodes d'analyse du surcoût induit par l'instrumentation de l'application, en vue d'évaluer son impact sur le système. Cette phase se déroule en deux étapes principales. La première consiste à évaluer les coûts liés à la collecte des données sur l'exécution du système. La seconde phase quant à elle mesure les coûts liés à l'analyse de la trace collectée. Pour ces deux phases d'évaluation, nous employons la mesure du temps d'exécution de l'application et la mesure de la taille des traces générées.

Dans ce contexte, la mesure du temps d'exécution de l'application, pour traiter une charge particulière, est calculée selon le scénario avec l'activation des points de traces et celui de la désactivation de ces derniers. Nous proposons d'utiliser des générateurs de charge tels que JMeter, pour générer la charge utilisateur sur l'application.

En ce qui concerne la seconde phase, consistant à calculer la mesure du temps d'exécution pour l'analyse de la trace, nous nous basons sur le temps nécessaire au chargement de la trace et à l'exécution des différents algorithmes d'analyse pour créer les structures de données nécessaires sur le disque. Le déterminisme de l'environnement est assuré par l'isolation des processeurs afin d'affiner les mesures de performance. Cette démarche vise à réduire les interférences lors de l'affectation des fils à des cœurs différents.

CHAPITRE 4 ARTICLE 1 : NODE COMPASS : MULTILEVEL TRACING AND DEBUGGING OF REQUEST EXECUTIONS IN JAVASCRIPT-BASED WEB-SERVERS

Auteurs : Hervé Mbikayi Kabamba, Matthew Khouzam et Michel Dagenais.

Soumis à Future Generation Computer Systems

Date : le 20 Novembre 2023

4.1 Abstract

Adequate consideration is crucial to ensure that services in a distributed application context are running optimally with the resources available. Due to the asynchronous nature of tasks and the need to work with multiple layers that deliver coordinated results in a single-threaded context, analysing performance is a challenging task in event-loop-based systems. The existing performance analysis methods for environments such as `Node.js` rely on higher-level instrumentation but lack precision, as they can't capture the relevant underlying application flow.

As a solution, we propose a streamlined method for recovering the asynchronous execution path of requests called the Nested Bounded Context Algorithm (NBCA). The proposed technique tracks the application execution flow through multiple layers and showcases it on an interactive interface for further assessment. Furthermore, we introduce the vertical span concept.

This representation of a span as a multidimensional object (horizontal and vertical) with a start and end of execution, along with its sub-layers and triggered operations, enables the granular identification and diagnosis of performance issues. We proposed a new technique called the Bounded Context Tracking Algorithm (BCTA) for event matching and request reassembling in a multi-layer trace . By using developed analyses, the resulting information can be visualised in an interactive tool for further assessment.

4.2 Introduction

The rapid growth of distributed applications, combined with recent advancements in cloud-based application development tools and technologies, have created a significant need for

performance analysis and optimization of those applications. In order to ensure the alignment with their tools and application development pipelines, developers leverage specialized tools to provide metrics and evaluate performance, while coding their modules at different stages of the development process. Depending on the assessment and the metrics to be collected, the appropriate technologies are used to generate a runtime profile of the application and, in particular, to help identify any performance issues that may occur in specific execution conditions.

The complexity of debugging is one of the primary issues that arise in distributed systems, as explained by Michels [79]. Indeed, distributed applications often rely on many abstraction layers to ease communication, deployment, scaling, performance, and resilience. Furthermore, the separation of services into independent components, the containerization, the choice of communication protocols amongst micro-services, and various additional factors complicate the monitoring and debugging of these systems. Performance analysis remains a significant challenge, generating considerable interest in the topic. Although numerous solutions and tools are available, the environment in which the application runs, and the technologies employed, generally dictate the methodology for evaluating the performance of these applications.

Distributed applications typically use distributed tracers, to capture data on the application specific interactions, at the expense of instrumentation. A collector then aggregates the distributed traces to make them available for the user interface modules [80]. For instance, the collector can gather and hierarchically aggregate the information about the request life-cycle, and the correlations between the sub-request interactions with other system components. For `JavaScript` applications, particularly those written in the `Node.js` environment, the simple use of distributed tracers to monitor and pinpoint potential performance issues creates a significant concern. The `Node.js` infrastructure comprises multiple layers, each performing a distinct function in processing and executing various tasks.

Moreover, the single-threaded nature of `Node.js` requires the main thread to instantiate an event-loop (EL), which orchestrates the execution of tasks by passing through several phases [81], with the main thread responsible for callbacks invocation on the one hand. On the other hand, it dispatches lengthy (possibly blocking) tasks to a thread pool to expedite their execution and avoid blocking while waiting for lengthy operations to complete. This intricacy is all the more noteworthy, because relying solely on a distributed tracer to instrument higher-level functions does not allow for an objective assessment of application performance in this context. Such environments require a greater level of granularity to precisely determine the underlying cause of any performance issue, since submitting asynchronous requests to the

lower layer does not guarantee sequential and instantaneous execution. The latter must first be queued with other concurrent tasks and awaits orchestration by the EL.

The time delay of a single request, or its nested requests, is not a reliable source of information for identifying the issue in this specific situation. For instance, slow code execution can keep the EL in a specific phase for an extended period of time, preventing it from moving to another phase to execute the many pending callbacks. In turn, the issue propagates to other pending requests, increasing the delay of other spans in the distributed trace. Since `Node.js` applications load and interact with many other modules, non-instrumented functions are unable to give performance metrics for diagnosing a specific performance issue, resulting in fault propagation to the pending requests.

Furthermore, the inner workings of `Node.js` prioritize a set of tasks en-queued into specific data structures, and ensure that they are exhausted before moving to the next phase. If the executing code is slow in this context, the transition latency between phases will increase, and the application performance will degrade. Consequently, the latency of pending tasks will increase rapidly, even though they are not the root cause of the problem.

This paper presents an efficient method for analyzing the performance of `Node.js` applications based on metrics monitoring and root cause analysis. We use the *Nested Bounded Context Algorithm* (NBCA), that exposes the request pathways, and the *Bounded Context Tracking Algorithm* (BCTA), that reconstructs the request vertical execution sequences into a span, allowing for fine-grained diagnosis and pinpointing of performance issues.

To the best of our knowledge, there is no pre-existing efficient technique that allows a global performance analysis with such level of granularity for `Node.js` applications.

The following are the primary contributions of this paper : (1) We present a novel trace span representation model, named the *Vertical Span Model* (VSM), designed to describe the operations of a single request across several layers and sub-layers, along with its metadata. (2) We present a fine-grained method for analyzing the performance of `Node.js` applications using kernel and user traces. We employ a four-layer paradigm to collect data at the `JavaScript` interpreter, the `virtual machine`, the `libuv`, and `kernel` levels. (3) We introduce a new crucial performance metric named the *Atomic Task Time Latency* (ATTL) that measures the internal atomic tasks execution time. (4) We build a state system model that captures the traces events attributes and metadata. (5) A multi-level traces events correlation technique is presented. (6) We demonstrate the relevance of our tool and the effectiveness of our technique with realistic use cases. (7) We implemented the following graphical views : (a) a graphical view of the nested request flows that exposes the request pathways, (b) a graphical view that exposes inter-process(IPC) communications, (c) a graphical view that tracks the

pass of the garbage collector based on our metrics, (d) a graphical perspective that depicts the condition of the EL and the various phases it goes through on a timeline.

The remainder of the paper is organized as follows : Section 4.3 discusses related work about the subject. Section 4.4 presents the methodology followed by our approach. In section 4.4.4, we present the system architecture. Section 4.5 presents some use cases to show our work pertinence and relevance. In Section 4.6, we discuss the results obtained while in Section 4.7 a conclusion on the work is drawn.

4.3 Related Work

The rapid transition of programs from monolithic architectures to distributed systems, particularly in Cloud Computing environments, has created a significant need for performance management [82]. Performance management, in this case, is analyzing the behavior of the program in order to determine if it deviates from its typical behavior and, if so, to identify the root cause of the problem and thereby resolve it. Various works related to `Node.js` have been proposed in the literature. The run-time monitoring was applied to `Node.js` in [83], to verify the correctness of an asynchronous nested callback in the context of the Internet of Things (IoT). They employed the parametric trace expression to monitor running applications developed in `Node.js` and Node-KED, which is a tool built on top of `Node.js` for flow-based IoT programming. The correctness of the application is verified at run-time through application tracing mechanisms, to check the compliance of the system behavior with the expected behavior defined by a specification formalism.

Chang et al. [84], investigated the process for detecting atomicity violations in event groups while studying the performance of `Node.js`. A task is a collection of discrete, deterministic events. They addressed event-race conditions within the context of many events. Tracing was utilized to gather information about the execution of the analyzed system, in order to predict and discover atomicity violations within a set of events. The identification of event pairings was based on the Happens-before relationship present in the execution trace. Atomicity violations were identified using predefined patterns.

Different approaches to the performance analysis of asynchronous `JavaScript` applications have been presented in the literature. Some approaches have been confined to detecting race conditions using client-side web application code. Predictive approaches are based on the recording of a part of the application execution by performing an analysis of the different memory accesses and the causal dependencies between the executions of the event handlers, and the data of the various execution scenarios, which form the basis for the generation of

race conditions.

Jensen et al.[85], introduced a browser-based data flow analysis for `JavaScript` applications named TAJS. The authors focused on representing the Document Object Model (DOM) instead of considering the events in depth. The approach assumes at the same time that any event generated by the browser can be executed at any time but cannot be executed in the context of separation of data flow, based on which listeners are registered. GATEKEEPER was introduced in [86], as a tool for enforcing specific `JavaScript` widget security policies. In particular, GATEKEEPER allows for the detection of unauthorized data flows. The concurrency problem on the client side was addressed in [87]. It presented WAVE, which is a framework for detecting concurrency errors. It is based on a model emanating from the execution. Test cases are generated and allow to swap the order of operations on the original execution flow. Any exception generated as a result of the test, or any result contrary to the original execution flow, constitutes an error reported by the system.

Under `Node.js`, few server-side race detection approaches have been utilized. NodeAV [84], is only capable of detecting atomicity violations. Davis et al. [88], applied fuzzing to the `Node.js` scheduler for race detection. Internal `Node.js` queues were used to randomize the association of inputs in order to maximize the probability of discovering faulty executions during testing. This technique enables the identification of several race conditions but is not applicable to issues that manifest during the delay of specific events. This method also makes it challenging to identify the underlying cause of the performance issue. With NodeRacer [89], investigations on alternate schedules by selectively delaying events, without affecting `Node.js` core queues, was conducted. The work was predominantly influenced by [90] and [91]. The analysis is based on the formalism of the happens-before relationship, which enables the identification of causal relationships between events. The initial running of the application enables the inference of the happens-before relationship based on either a regression test or manual interaction with the application.

The network performance of `Node.js` was investigated by Nkenyereye and Jang [92], as they evaluated the performance of a Healthcare Hub Server for a Remote Monitoring System. For this latter, emphasis was placed on concurrency, because `Node.js` contains all the necessary technology to effectively manage its issues. Throughput and response time were measured using `Apache Jmeter`[93]. Madsen et al. [94] used static analysis on `Node.js` to identify problems in the events processing. According to their observation, ordinary call graphs are unsuitable because they do not accurately depict the control flow implied by the registration of event listeners and the emission of events. Therefore, they proposed an extension of the Call Graph and implemented some analyses using the static analysis tool RADAR.

Although few studies have focused on analysing the specific problems of `JavaScript` server applications, they all share a commonality : they are founded on the `JavaScript` upper layer. According to our knowledge, multilayer performance analysis under `Node.js` is unexplored in the literature.

4.4 Methodology

4.4.1 The Vertical Span Representation Model

The vast majority of `Node.js` performance analysis approaches rely on distributed tracing to collect the necessary data. This information, combined in a single trace, enables the observation of the request execution latency in the form of spans. The metric is the duration of the execution of each request component, and it contains very high-level information. The `Node.js` architecture comprises a collection of components that may be classified into three distinct layers. The top layer is the `JavaScript` interpreter, followed by the internal layers, namely the `V8` engine, `libuv`, and the operating system kernel. Instrumentation using distributed tracers necessitates a degree of technological homogeneity to allow the collector to connect the events and reconstruct the trace spans properly. Such approaches present a significant challenge as they expose only symptoms of the poorly performing application.

In general, the instrumentation will be performed at the upper layer level, which can only provide an overview of a performance issue, without allowing it to be precisely pinpointed, due to the complexity of the environment. For instance, calling a `JavaScript` file `read` function may incur considerable lag, suggesting a performance issue. However, this information cannot be used to establish the root cause of a problem. Indeed, the latency could be caused by a blocking state in the EL or by the exhaustion of the threads in the pool, to mention a few. As a result, it is crucial to employ alternative performance analysis techniques, offering a higher degree of granularity, in order to isolate the cause of the problem. The VSM provides the vertical representation of a span. In the latter model, the various sub-layers of the system involved in the execution of the request are used to reconstruct the sequences of execution and granularly display the span for a precise pinpointing of the performance problem.

The span in Figure 4.1 was obtained by instrumenting a `JavaScript readFile` function call. The task execution time (300 ms) is denoted by the span-shaped label “A”. Label “B” shows the VSM, which depicts the synchronous and asynchronous operations of the different sub-layers that make up the span. As seen in Figure 4.1, calling the `readFile` method, in the higher layer, triggers a set of asynchronous function calls that the `libuv` layer will execute. At this level, the `fs_read` function may be transparently invoked several times. The execution of

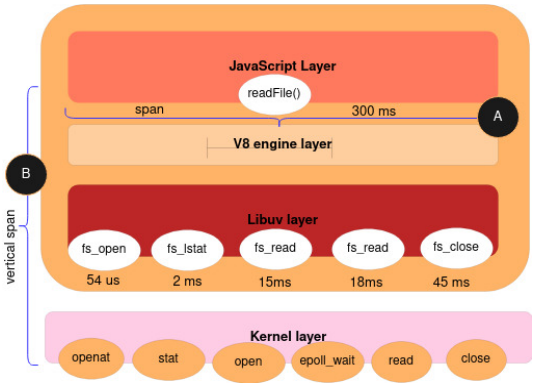


FIGURE 4.1 Vertical Span Model (VSM) representation. The label "A" depicts a span as it appears in distributed tracers. The label "B" depicts a vertical span representing the flow of the request sequences in all layers.

I/O functions triggers related system calls. Since function calls are not sequentially executed in the different layers, but are instead coordinated by the EL, it is challenging to figure out the root cause of the performance problem. The VSM enables the definitive identification of the nature of issues at hand.

4.4.2 Asynchronous Operations in `Node.js`

The execution of asynchronous operations is a core feature of `Node.js` that allows the execution of other instructions without waiting for blocking and slow tasks. To do this, an internal orchestration system in `Node.js` allows the execution of tasks, and the storage of various related references. An asynchronous operation in `Node.js` embeds a callback function that is invoked at the end of its execution.

Definition : an atomic operation is the smallest component of a high-level function that triggers system calls or that can be executed directly by the Node.js runtime.

Formally, the asynchronous operation can be seen as a monad [95]. In Figure 4.1, the `Node.js` `readFile` function triggers a set of I/O function calls at the `libuv` layer, which can be interpreted as follows : the `fs_open` function is called while the `fs_stat` is passed to it as a callback. The invocation of the callback function embeds the function `fs_read` as a callback, and likewise, the latter is invoked with the callback function `fs_close` passed to it as an argument.

This sequence of asynchronous operations can be represented formally as a monad. Consider R as the result of executing the I/O function f at the `JavaScript` interpreter layer. By projecting the execution of the latter on the bottom `libuv` layer, as shown in Figure 4.1, the result $R^{'}$ obtained there must be equivalent to R. Based on Janin work [96] , we can derive the result R by :

$$R = f_1 \gg= f_2 \gg= f_3 \gg= f_4 \gg= f_5 \tag{4.1}$$

where $R \equiv R^{'}$.

The chained execution of these I/O functions makes a monad reference memory that the parent function can use to get the chain result. In this case, the monad reference precludes the execution of higher-level functions. This reference points to an active task. Under `Node.js`, it will often be a memory address used to access the value returned by the referenced task.

Let m be a monad of type T, referenced by a pointer to its forked action P_m, as modeled in [96].

$$P_m :: * \rightarrow * \tag{4.2}$$

$$\texttt{fork} :: m\ x \rightarrow m(P_m\ x) \tag{4.3}$$

$$\texttt{read} :: P_m\ x \rightarrow m\ x \tag{4.4}$$

P_m is a monad reference bound to an active action of type $m\ x$, `fork` m is the activator of the execution of the monadic action x and returns a reference bound to that action, and `read` p is an action that waits for and accesses the memory reference, to return the value returned by the running action bound by the monad reference p.

In Equation 4.1, the chained execution of functions requires the passing of arguments of objects references bound to these functions, and to events that provide information on the state of the function execution. The triggering of these events allows access to the addresses containing the returned values. A value indicating the `ATTL` is attached to each atomic function $f1, \ldots, f5$. The execution time $\mathbb{T}$ can therefore be obtained by :

$$\mathbb{T} = \Sigma_{i=1}^{n} t(f_i) \tag{4.5}$$

$$\mathbb{L} = \mathbb{T} + \Sigma_{i=1}^{n-1} \Delta(t_{i,i+1}) \tag{4.6}$$

The overhead latency $\mathbb{L}$ of the `JavaScript` function is computed by summing the waiting times $\Delta(t_{i-1,i})$, for each function f_i to be executed from the completed execution of the

predecessor f_{i-1}. In this case, $\Sigma_{i=1}^{n-1}\Delta(t_{i-1,i})$, constitutes the overhead of the request execution, and a threshold must be fixed to its value to declare a performance issue. The VSM allows the visualization of each atomic operation, along with its latency throughout the involved layers. The reader is referred to [96] and [97] for additional information on monad functions.

4.4.3 Task Transitions and Atomic States

The interpreter is the entry point for a `JavaScript` function execution. This is the request S_0 state, as depicted in Figure 4.2. The latter is subsequently sent to the bottom layer, which queues it. Then it moves to state S_1. The transition between the two states is captured by the `uv_send` event in the trace. At this point, the request is waiting to be removed from the queue in the order in which it was received. The EL conducts continual checks and executes the different queued jobs in a First In, First Out (FIFO) pattern. The task is then removed from the queue and again queued for the corresponding EL phase. Then it moves to state S_2. Function `uv_dequeue` is used to instrument this transition.

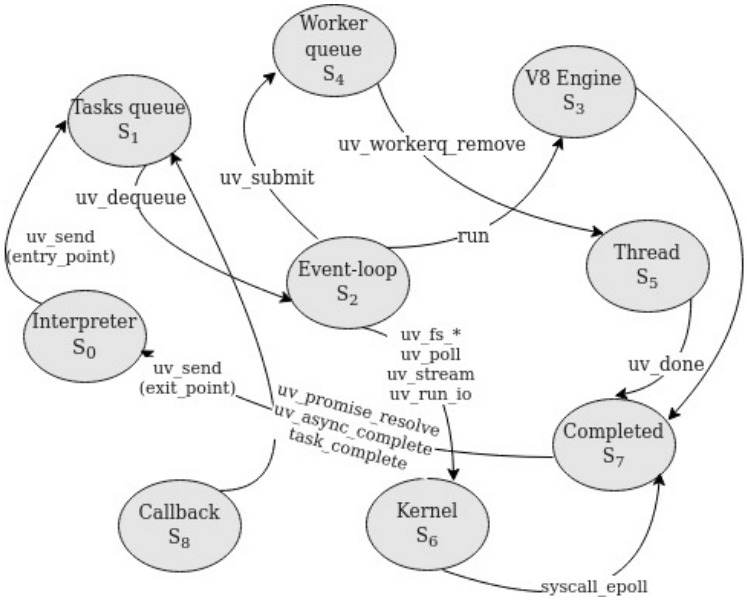


FIGURE 4.2 Request states during runtime. Nodes represent the state of tasks as seen by the Node.js runtime and edges represent the transitions labeled with the triggered events.

The event loop will see the request and manage its execution by the `Node.js` runtime, as it moves through different stages. At this level, three paths are possible : If the request is a `Node.js` operation, the runtime executes it instantly. It then transitions to state S_3, which is instrumented by the `run` event. If the task involves an I/O operation, it is pushed into a worker queue and transitions to S_4 state. The latter, `uv_submit`, is instrumented. It will then

be pulled from the worker queue by a thread from the pool, and the associated I/O operations will be performed. In this case, it transitions to state S_5. Function `uv_workerq_remove` is used to instrument the transition. The third possible path for the task is to move to state S_6. In this case, the operating system runs the task and notifies the upper layer through the built-in asynchronous communication mechanisms.

Functions `uv_fs_*` and `uv_socketRead` are used for many input and output operations. For example, `uv_socketRead` is used to read from a socket. The S_7 state is instrumented by `uv_done` and is used to indicate the completion of a task. This event returns an atomic operation unique ID. Recall that an atomic operation is one part of the chain of execution of a request made by the interpreter, but there are many more. The result can be sent back directly to the interpreter from S_7 state. The transition is instrumented by `uv_send` as the exit point. It can also return to state S_1 from state S_7, if a monad reference was given to the superior function.

In the case of the `readFile` function, the `fs_open` atomic task returns a file descriptor that is provided as an argument to subsequent atomic file access operations.

4.4.4 System Architecture

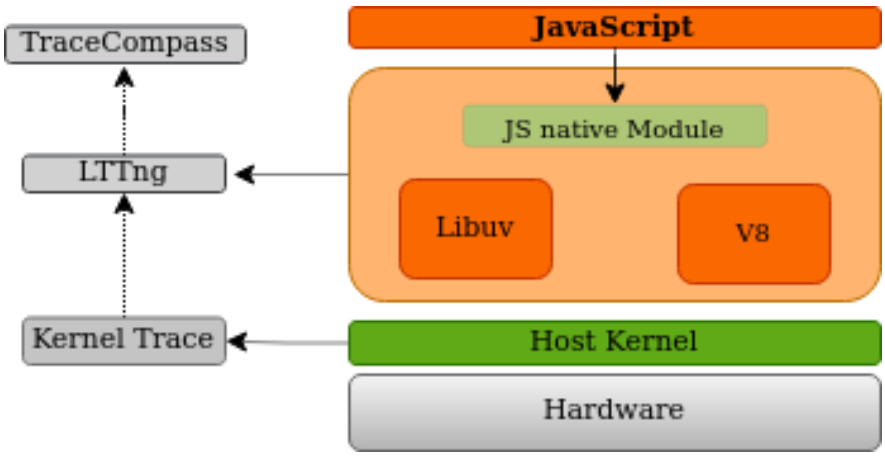


FIGURE 4.3 Architecture of the collection and analysis system.

The system architecture is shown in Figure 4.3. The data is collected through tracepoints inserted into the system. The `LTTng` (Linux Tracing Toolkits Next Generation) tracer is used to perform the static instrumentation. A native module called `calculate` allows tracepoints injected into `JavaScript` functions to invoke the `LTTng` tracer libraries at the highest level. Static tracepoints are also added to the `libuv` layer to collect information on atomic tasks

orchestration, and in the `V8` engine to collect information on the asynchronous resources life-cycle. Kernel traces are obtained with `LTTng`.

The system instrumentation allows two synchronized traces to be loaded into Trace Compass[98] as an experiment. In the latter, precise analyses and algorithms are run, metrics are extracted, and views are built to observe the system performance. Because of its low overhead on the system [68], the `LTTng` tracer was chosen from a set of tracers. `LTTng` is considered today as the fastest Linux tracer [99]. An extension of the tool enables the analyses to be run for Windows `Node.js` versions.

The process for conducting the analysis is as follows :

1. Start the tracer.
2. Launch the concerned `Node.js` application.
3. Stop the tracer on the host.
4. Run the analysis.
5. Use an interactive tool (Trace Compass) to visualize the results.

`LTTng` generates and collects events, which are then loaded into Trace Compass. On the latter, an event analyzer was developed to handle our trace events. Trace Compass is a free and open-source program that analyzes traces and logs. Its extensibility enables the creation of views and graphs, as well as the extraction of metrics.

4.4.5 Nested Asynchronous Operations Detection

The execution of a `JavaScript` function consists of a group of atomic operations that happen across the system layers. The `Node.js` environment handles requests made at the interpreter level in a specific execution context. It is feasible to obtain information about the context *id* by inserting tracepoints that are activated during the initialization of asynchronous resources, and immediately following their execution.

```
app.get('/compute-with-promises', function computePromise(req, res) {
  const as=async_hooks.executionAsyncId();
  const ass= async_hooks.triggerAsyncId();
  calculate.send_event(as, ass, 'js_open_computePromise');
  const hash = readData();
  const updatePromise = () => new Promise((resolve) =>{
     calculate.send_event(async_hooks.executionAsyncId(), async_hooks.
   triggerAsyncId(), 'jsx_updatePromise');
     hash.update();
     resolve(); });
  const loop = () => {
```

```
11      updatePromise().then(loop).then(calculate.send_event(as, ass, '
    js_exit_computePromise'));
12     } }
13  loop();  });
```

Listing 4.1 JavaScript instrumented code snipet. The `calculate` module is the `C++` addon that calls the LTTng functions activating the tracepoints.

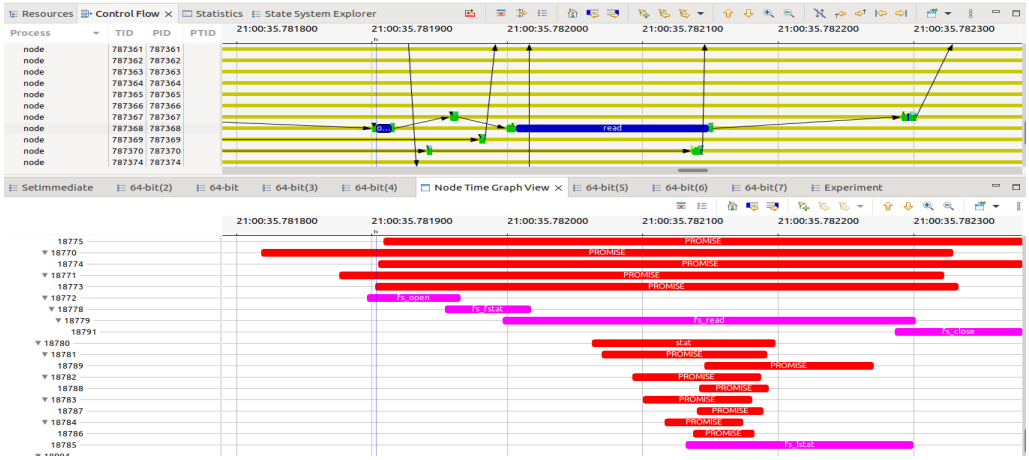


FIGURE 4.4 Execution Flow and Nested Asynchronous Operations. The Control Flow perspective shows a timeline of the multi-layer events. At the bottom, the Node TimeGraph view depicts atomic operations flow in the middle and their executions contexts numbers in the left.

The `uv_send` event is used to instrument `JavaScript` functions and to track the asynchronous objects life-cycle. In Listing 4.1, the `AsyncHooks` API is used along with the `calculate` module. The latter accepts an *id*, a `channel`, and a `method` as arguments. When an `HTTP` request is received, function `ComputePromise` is called. A tracepoint is then triggered embedding the information about the execution context *id* as well as the *id* of the resource that triggered the current asynchronous object. At line 4, the function entry point is instrumented. Information about the request context identification number can be found in the *id* field, while information about the *id* of the resource that triggered the current context can be found in the *channel* field. There is a field called `method` that has the name of the function that was instrumented. In this instance, it is `computePromise`. The *channel* field can also contain the name of the asynchronous resource, like `promise` or `FSReqCallback`, among many others. It is obtained by getting the value of the `type` variable.

At the end of the function execution, `uv_send` from the `calculate` module is used again to signal the completion of the request. It can be observed at line 11 in the Listing 4.1. In this instance, the *id* field includes the same unique identifier as the initial request, the

channel field is *null*, and the field `method` has the "`js_exit`" value. In this case, it signifies the end of the `JavaScript` function execution. At line 6 a new asynchronous resource is triggered creating a new execution context bound for the asynchronous resource. The latter is instrumented at line 7. In this case, the execution context *id* is new. The *id* of the resource that triggered the new object, is the execution context *id* of the `computePromise` function. We do not need to instrument the completion of asynchronous resources, since this will be obtained from the `V8` engine directly. The NCBA algorithm uses the `uv_send` event to figure out the nested execution order of the atomic operations. Instrumentation of the `JavaScript` function entry and exit points is needed to get back the time it took to run.

However, the goal of the algorithm is to figure out the nested flow of operations. In response to the `uv_send` event, the latter will look at the value of the `channel` field, which contains the *id* of the resource execution context, and will query the State History Tree (SHT). The SHT is a tree-shaped data structure that efficiently stores and indexes the attributes and the states extracted from the trace. It will get the *id* of the parent resource that created the current resource execution context. This SHT will be queried again using the obtained parent resource execution context *id* to obtain the *quarkParent*, which is the value specifying the position in the SHT at which a new child entry will be created.

The algorithm will store the new entry position once more, and will update its status with the name of the method in the SHT. It will then monitor the "`run`" or "`resolve`" values in the `method` field of the `uv_send` event, to later alter the entry status, initially set to *null*, based on the *id*. This signals the end of the asynchronous resource execution. As it traverses the trace, the algorithm repeats the same operations. Finally, the algorithm outputs an SHT comprising all the operations, in hierarchical order, along with their execution time.

In Figure 4.4, on the left panel of the Node TimeGraph view, the execution context IDs of the different atomic operations are identified and displayed in a hierarchical order, as computed by the algorithm. For instance, the `readFile` function, executed at the top layer, triggered the execution of the `fs_open` function on the `18772` execution context, at the `libuv` layer. The latter triggered the execution of the `fs_stat` function on the execution context `18778`, which again triggered the execution of the `fs_read` function on the execution context `18779`, and finally the latter triggered the execution of the `fs_close` function on the execution context `18791`. On the right panel, the computed nested execution flow can be seen. Some operations may trigger promises, and the latter may trigger atomic operations within their execution contexts. Exposing the execution flow allows a granular identification of the root cause of performance issues.

4.4.6 Multi-layer Events Matching

In the context of multi-layer event matching, our focus is directed towards two primary concepts : vertical profiling and calling-context profiling. The first has been investigated in [100] and [101]. It entails profiling various metrics from various layers of the system stack. Analysis of application performance is complicated by the addition of multiple abstraction layers to the execution environment. The methodologies investigated and suggested for vertical profiling predominantly rely on multi-layered systems that enable multi-threading. Nevertheless, these methodologies are not congruent with event-driven systems such as Node.js.

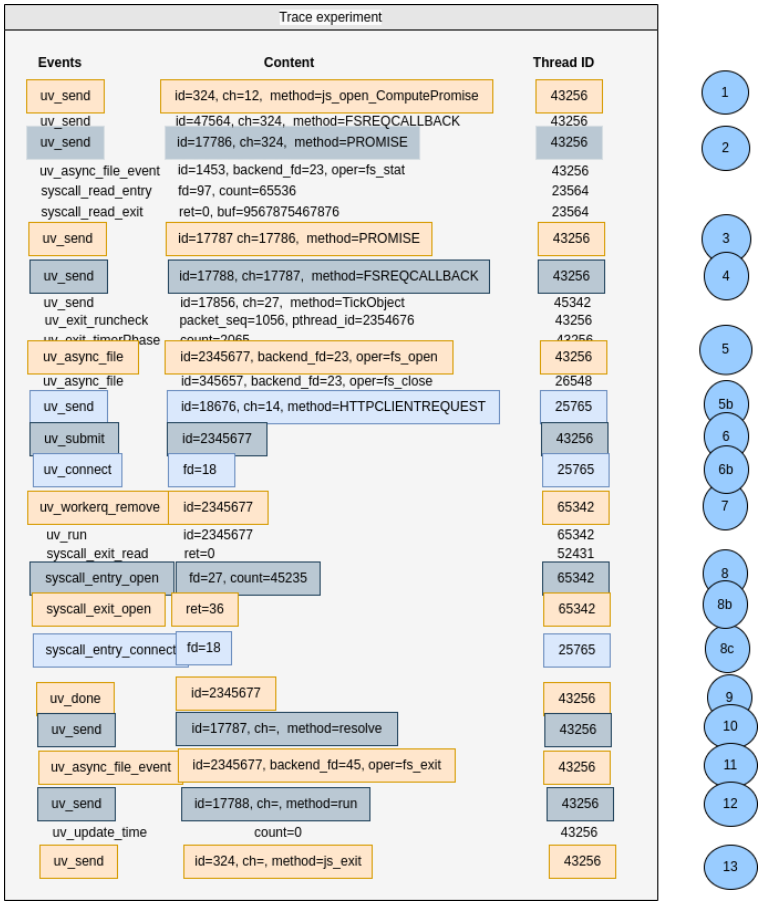


FIGURE 4.5 Multi-layer trace. Event from different layers are aggregated and loaded as an experiment. For clarity some contents, fields and timestamps have been omitted

The operating system views these systems as black-box devices whose operations are not executable in a deterministic manner. An EL orchestrating their execution introduces a horizontal overhead. This EL also adds an extra layer of abstraction, complicating any performance

analysis. Calling-context profiling [102, 103], uses a tree data structure to provide dynamic metrics for each calling-context.

We propose a new approach that uses a state system created from trace attributes data. It employs a state history tree (SHT), a data structure that is optimised during the construction of our model while traversing the trace, to store the attributes extracted from the latter. The platform for modelling and developing such data structures is provided by Trace Compass. Their optimisation enables access to the model with a logarithmic complexity. Taking into account the two proposed concepts, we developed a profile of the analysed system based on the upstream trace. This is a crucial and difficult step, particularly considering there is virtually no literature on the subject.

It is obvious that building a calling-context profile within the same layer is a straightforward approach. However, in such a setting, it is quite difficult to correlate the same information, arriving from various layers, while also connecting them to the functions from higher layers. To the best of our knowledge, there is no documented method for accomplishing this using Node.js. Our Node Compass approach instruments, aggregates, and constructs a model utilising the state system and SHT data structures. The latter is constructed by looking for patterns in the multi-layer trace that enable the reconstruction, of the vertical and horizontal execution sequences, of the system under inspection. The end result is a visual and interactive instrument that enables the user to identify problems.

Basically, two types of traces are obtained from the system instrumentation, as depicted in Figure 4.5 : userspace and kernel-space traces. The BCTA is used to look for patterns in the multi-layer traces and allows the correlation of events. In Figure 4.2, after being submitted in the upper layer, the request moves to state S_2, where the intermediary layers manage its execution. It can be observed that the request can take the following state paths : $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_7 \rightarrow S_0$, $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_6 \rightarrow S_7 \rightarrow S_0$, and $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_4 \rightarrow S_5 \rightarrow S_7 \rightarrow S_0$. In the first case, it is a `Node.js` non-computational intensive task. Il is executed right away by the runtime. In this case, the higher-layer function is instrumented by the `uv_send` event, which identifies the entry and exit points of the function Figure 4.5 (labels 1 and 13).

In the second case, the task execution is delegated to the operating system (OS). The interpreter, the kernel, and the `libuv` layers will all be involved in the event matching process, Figure 4.5 (labels 5*b*, 6*b* and 8*c*). At this level, events that cause the OS to run system calls are captured. These latter are sequentially preceded by a resource initialization process in the `virtual machine`, Figure 4.5, label 5*b*. Directly inserted tracepoints enable to obtain the information on the execution context, and the name of the resource being executed. The

BCTA will first look for the closest `uv_send` event that precedes the `libuv` atomic task (label 6*b*) based on its thread ID, since both events are sequential. This `uv_send` event, obtained from the `VM` instrumentation, provides the context data required to match the appropriate `JavaScript` function, instrumented with the `AsyncHooks` API. Consequently, it is possible to reconnect operations from these three layers, as Figure 4.5 shows. The upper layer and

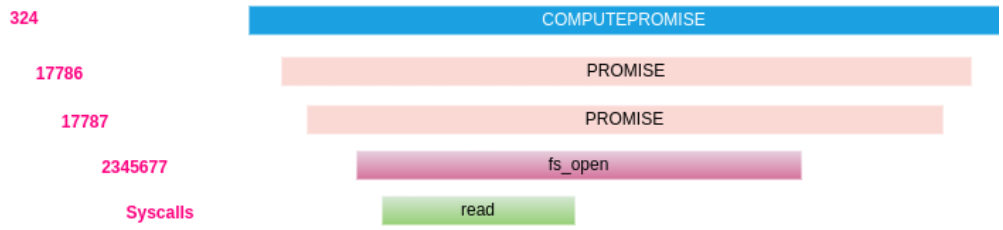


FIGURE 4.6 Resulting vertical span from the NBCA and BCTA execution of the trace in Figure 4.5

the `libuv` events are linked by creating a new child entry at the position of the `JavaScript` initial function in the SHT. To match the kernel events, the BCTA tracks all system calls executed by the same thread. It creates child entries at the corresponding `libuv` atomic task position in the SHT for each one, as long as the request execution is not completed. This information is obtained by watching the `uv_send` events, which signal the `JavaScript` functions exit point. The algorithm uses this information as a constraint, before searching for patterns. These execution bounds are named the *Bounded Execution Context* (BEC).

For the third case, consider the execution of a JavaScript function that opens a file. The `libuv` layer utilises a thread pool, which simulates the asynchronous execution for blocking processes. Thus, it can continue its execution and move through the various phases, invoking the callbacks that are awaiting execution. In the trace presented in Figure 4.5, the `ComputePromise` function that opens a file is called. The instrumentation of the beginning of the function execution triggered the event uv_send with the id=324. The BCTA will track the resource id=324 and reach label 2. A `promise` is created from the execution context of the higher level function `ComputePromise`. The BCTA will keep tracking the id which is now 17786 and reach label 3, where a new promise has been created from the execution context of the preceding `promise`. It will then track the id=17787 which captures the beginning of a file system operation. Constrained by the BEC, the BCTA relies on the events `uv_async_file`, `uv_submit`, and `uv_workerq_remove`, which respectively indicate, the beginning of a file system atomic task, the queuing of the task by the main thread executing the EL, and its removal from the queue by an available thread in the pool. These events are obtained from the `libuv` instrumentation. It can be observed in Figure 4.5, label 7, how a thread from the

pool is removing the task from the queue (different thread ID). The algorithm seeks for the `uv_send` event from the `VM` instrumentation that immediately precedes the `uv_async_file` event, as depicted in Figure 4.5, label 4. In this way, the execution context information is obtained by the algorithm.

These events provide the BCTA with sufficient information to establish a connection between the task being executed by the thread and the main process that runs the EL, as well as the initial `JavaScript` function. The atomic operation *id* is determined by the sequential `uv_async_file` event that precedes the `uv_submit` event. Both events are always sequential(label 5). The thread ID is used to match kernel events(labels 8, 8*b*). The SHT is then updated by creating the child entries corresponding to the detected kernel events. Figure 4.6 depicts the resulting vertical span from the trace in Figure 4.5 as outputted by the analysis.

4.4.7 Data Abstraction

Tracers are employed to capture information about the system behavior, during its execution for a specific period. Consequently, they enable the inspection of the system and the identification of inherent or operation-specific issues. In addition, the size of the obtained trace might rapidly expand as a function of the number of generated events, resulting in a significant increase in the processing cost.

Hence, it becomes necessary to devise mechanisms and strategies for efficiently managing this issue. Trace Compass offers a framework and several features for organizing trace data to conduct effective analyses. It accomplishes this through multi-level abstraction, highlighting, and filtering. To construct our trace storage and analysis model, we built a data structure based on the time interval, which allowed us to build a state system capable of storing the attributes and the specified execution states of the monitored `Node.js` applications.

The events contained in the trace allow deriving the attributes and states of the system. Trace Compass extracts the crucial and necessary information to incrementally build the SHT when the multi-layer trace is loaded. This tree-shaped data structure (see Figure 4.7) provides access to its data in logarithmic time. Thus, it allows defining the granularity of the desired performance measurements. It also allows applying analysis within a clearly defined time limit.

4.5 Use-Cases

This section investigates five realistic issues to illustrate the range of applications of the proposed new algorithms and methodology. Additionally, the use-cases demonstrate the ef-

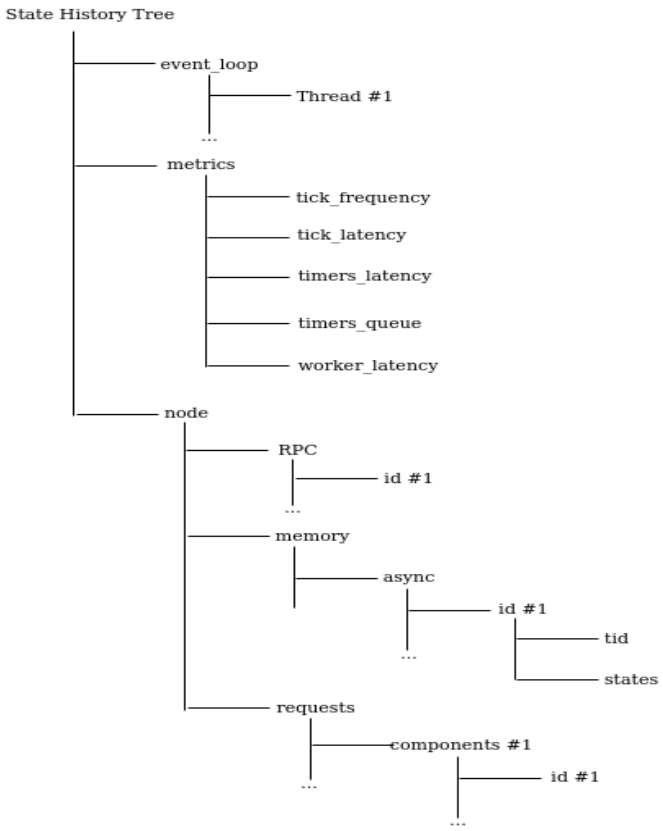


FIGURE 4.7 State History Tree. Attributes are extracted from the trace and stored hierarchically in this data structure. State values are assigned to each attribute with a timestamp.

fectiveness of the methodology in identifying and addressing performance issues for various problem types and contexts.

Use case 1 represents a common scenario encountered by node.js application developers. We demonstrate the efficacy of Node Compass in detecting Regular Expressions Denial of Service (ReDOS) in application modules. Such a vulnerability permits malicious code to obstruct the EL in a particular phase, thereby delaying the execution of pending callbacks[104] [105]. Use case 2 represents one particular characteristic of our tool. It demonstrates its distinctiveness by revealing to what extent garbage collection operations can affect the execution time of an atomic operation in `Node.js`. Our tool is subjected to a realistic issue raised by developers on `Github`.

The third use case illustrates how our tool can be used to detect memory leaks in Node.js applications. The performance is monitored by the utilisation of offline profiling of memory consumption, based on the collected trace. Use case 4 demonstrates how our tool can track Inter-process Communication (IPC) to conduct a performance analysis.

The last use case is a particular instance on how a root cause analysis can be driven by Node Compass.

The experiments were conducted with Node.js versions 12.22.3 and 16.1.0, on an I7 Core running Ubuntu 20.4 with 16 GB RAM.

4.5.1 Detecting REDOS Vulnerabilty in Node.js

Our tool capability to spot ReDOS vulnerabilities in Node.js is highlighted in the first use case. It is not uncommon for certain modules loaded in applications to have this type of vulnerability. They have been investigated in [106]. Executing vulnerable code in the application can cause the event loop to block and errors to propagate to requests awaiting execution. Collected high-level information from distributed tracers cannot expose the source of the problem. Accurately identifying the layer responsible for latency propagation requires a multilevel analysis approach.

In order to demonstrate the effectiveness of our technique in identifying ReDOS vulnerabilities, we employed the benchmark resulting from [106]. The latter encompasses numerous `JavaScript` RedOS vulnerabilities and exploits, that have been documented and detected across multiple accessible modules. Node Compass was able to identify all performance issues caused by them. To demonstrate how our tool operates, we have selected one vulnerable module from the benchmark as a showcase.

Figure 4.8 illustrates a performance issue that is triggered by calling the `resolve` `JavaScript` function of the Domain Name Service (DNS) `Node.js` module. It takes approximately 3.9 seconds to execute. Our tool permits the correlation of high-level information with lower layers, such as the `libuv` and `V8` layers, in order to identify the underlying cause of the problem. Examining Figure 4.8 reveals that the EL is halted in the polling phase, following the initiation of high-level `JavaScript` function execution. As a result of the developed analyses, the correlation between distinct layers is obvious. The activation of the tracepoint at the VM, by invoking the `exec` function , confirms the suspicion of a ReDOS exploit. The latter function processes regular expressions. The correlation of those informations, from different layers, results in the visual alignment of the events and the reconstruction of their internal operations.

4.5.2 Correlating Performance issues with Garbage Collector Operations

The `Node.js` runtime executes submitted `JavaScript` code at the interpreter level. An executed `JavaScript` function may exhibit latency that deviates from its normal behavior.

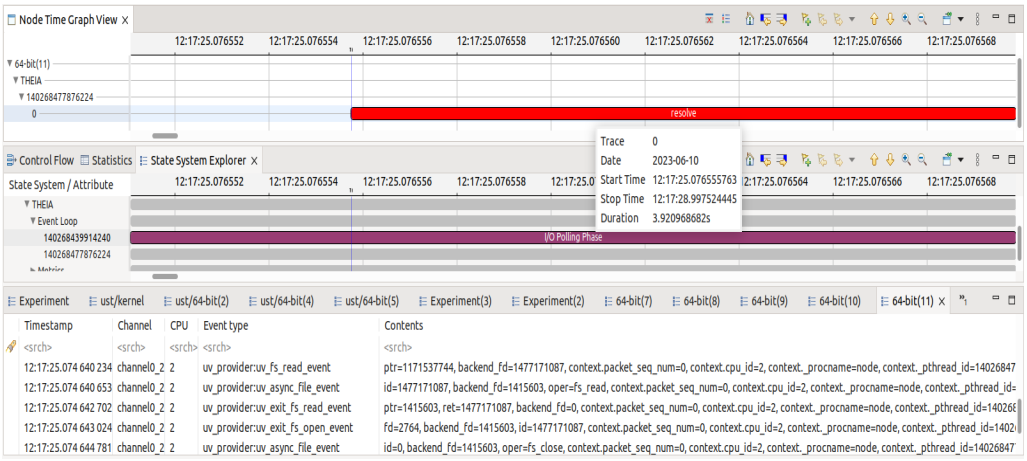


FIGURE 4.8 Calling the resolve function from the DNS module. The event loop is stalled for 4 seconds, due to a REDOS vulnerabilty exploited. It takes 3.92 seconds for the resolve function to complete.

Although the performance issue can be diagnosed in this case, the root cause cannot be identified. The vertical reconstruction of the execution sequences of a function enables the association of the different atomic operations with their execution context. To this extent, it allows the identification of atomic operations at the root of the performance problem, as well as the layer in which the issue is located.

Our approach effectiveness, and level of granularity, in exposing performance issues is demonstrated in this use case. `Node.js` developers reported issue #37583 on GitHub. A performance degradation was observed by using the `fs.promise.readFile` function, in comparison to the `fs.readFile`.

The performance issue was replicated in `Node.js` version 12 by executing the suggested benchmark, which measures the performance of fs.readFileSync, fs.readFile, and fs.promises.readFile. Calling fs.promise.readFile to read a 1 MB file was slower than fs.readFile.

Our tool could exposed the execution flow of those two distinct reading functions. The `fs.readFile` function reads the entire file in a single function call. However, in the case of the `fs.promise.readFile`, the latter is split into many `fs_read` calls at the `libuv` layer. In Figure 4.9, we can see how memory operations from the garbage collector (GC) impact atomic read operations, increasing their latency by destroying memory resources linearly. Atomic read operations are blocked as a result. The general execution time for an atomic `fs_read` operation in Figure 4.9 is approximately 50 microseconds. The memory pass of the garbage collector disrupts certain `fs_read` read operations that require between $3ms$ and $11ms$ to complete. This has the overall consequence of increasing the execution time of the

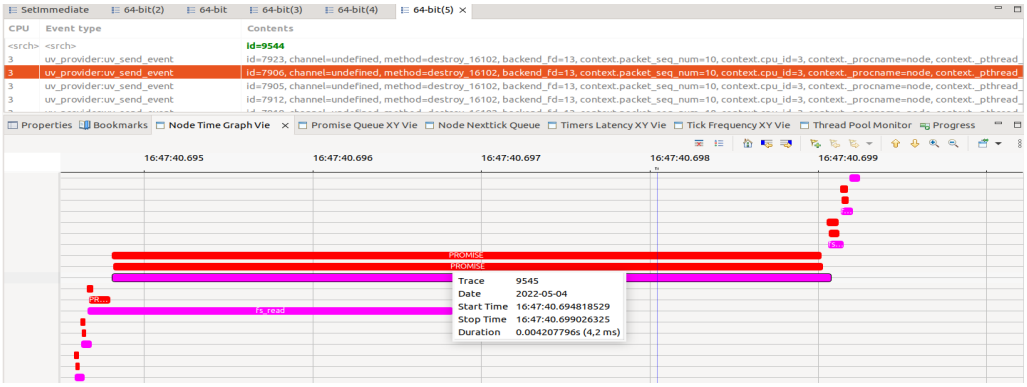


FIGURE 4.9 VSM of the "fs.promise.readFile" execution. Atomic operations at `libuv` layer can been seen in the bottom perspective. The read operation is split into many atomic `fs_read`. The top perspective show linear events triggered as result of the GC operation.

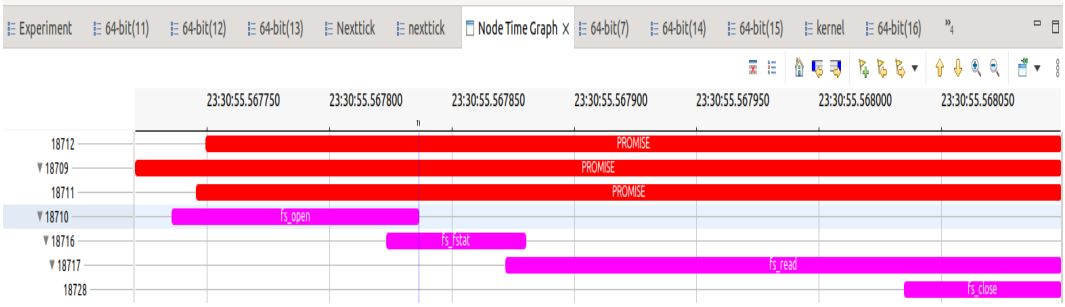


FIGURE 4.10 VSM of the "fs.readFile" execution. Atomic operations at `libuv` layer can been seen. they are triggered by promises. To read a file, the atomic functions fs_open, fs_stat, fs_read, fs_close are called respectivly.

higher level function, because promise type resources are created in the memory in large numbers, and must be destroyed by the GC to optimize memory usage.

`Node Compass` stands out for its high level of granularity, which enables the identification of performance issues with exceptional precision. Our introduced metric `ATTL` measures the execution time of an atomic operation. To the best of our knowledge, there is no performance analysis approach in the literature for accomplishing this in `Node.js`.

4.5.3 Memory leak Detection

In this use case, we demonstrate the effectiveness of our approach in profiling resources based on defined metrics. Node Compass analyzes GC behavior using 2 specific metrics. The Time Spent in the Garbage Collection (TIGC) and the Time Between 2 GC Operations (TBGC). We utilized the `Node.js` memory leaks benchmark [1], which encompasses 4 types : (Global, promise, cache, enclosures).

Node Compass exposed memory leak-related performance issues suspicions in all 4 cases using the defined metrics. We select one type of leaks (cache type) to discuss the Node Compass uncovering mechanisms, and demonstrate how our tool works. The same methodology applies to all types. The absence of memory leaks is reflected in Figure 4.11, which depicts the regular execution profile of the Node.js process. The time spent between two garbage collections is constantly more than the time spent in garbage collection (red).

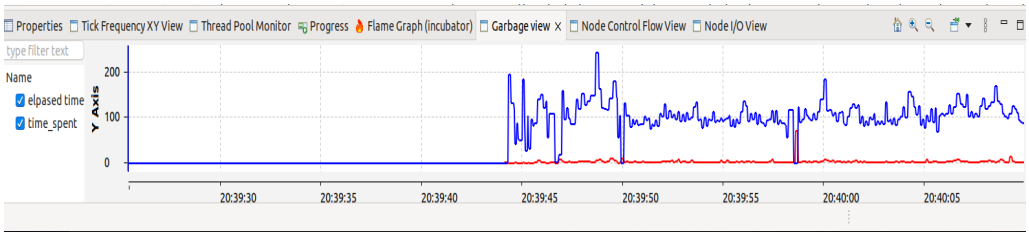


FIGURE 4.11 Execution of the Cache type memory leak benchmark. Normal profile of the execution with no memory leak

An optimal execution profile for the process application is achieved in such a situation. A difference can be observed in Figure 4.12 that shows the profile of the Node.js process exposing a memory leak. As seen, both metrics balance each other, raising a suspicion on heap problem. We observed this pattern in each memory leaks context.

1. `https://github.com/Joker666/nodejs-memory-leak`

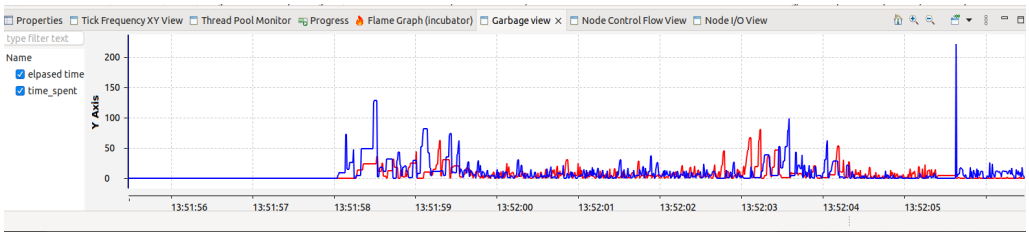


FIGURE 4.12 Execution of the Cache type memory leak benchmark with memory leaks

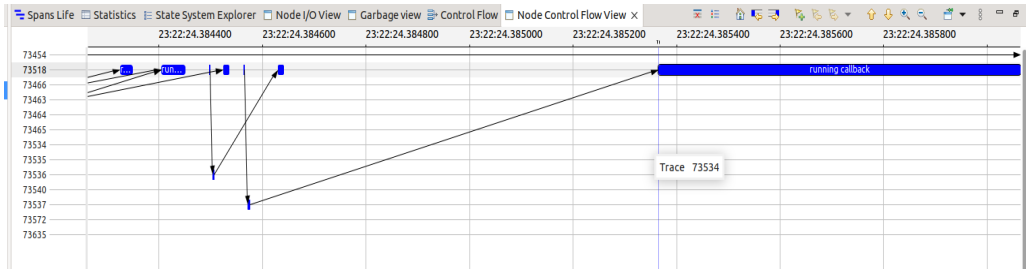


FIGURE 4.13 Processes and threads interactions in Node.js. The arrows show the direction of the communications. The numbers are the threads and processes IDs

4.5.4 Exposing Inter-process Communication issues

We tested our performance analysis approach on a use case that involves inter-process communication (IPC). The latter was raised by the developers at `Ericson Canada`, the main committers of `Theia`. Our approach was successful in revealing the nature of the error. The `Node.js GitHub` site highlighted an issue with reference to #43936, explaining that data on Windows gets "clogged" in extra pipes when forking a child_process.

The distinguishing factor of this use case is its connection to the Windows NT environment. Our tool global application for multi-level performance analysis in Node.js is demonstrated in this context. Node Compass has the capability to trace, aggregate, and analyze events from the Windows OS without modifying its approach. Our instrumented `Node.js` versions can be compiled with a flag indicating the use of the "barectf" library in its trace collection architecture. The tracepoints within Node.js remain unchanged, with modifications made only at the tracer level. The tracer that was defined with `barectf` is responsible for collecting these events. `Barectf` produces a file in the Common Trace Format (CTF), the same format used by `LTTng`.

The use case pertains to a performance issue that occurs when a primary process (`main.js`) forks a child process (fork.js) with an additional open pipe. In normal operation, the child process automatically responds to pings from the parent process with pongs. In a similar

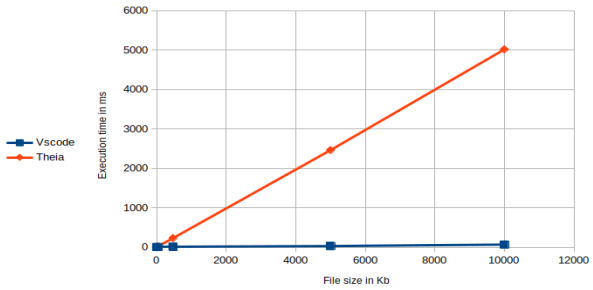


FIGURE 4.14 VSCode and Theia file reading average time.

manner, the parent process responds with a ping after reading the pong. This type of interaction makes process monitoring possible. Therefore, a rapid message exchange between the two processes is required.

During the execution of this procedure on Windows 10, the communication was hindered to a certain extent. Node Compass was used to identify the IPC communication flow within the ecosystem, to identify the problematic process, among others. Based on Figure 4.13 "Node Control Flow" view, the processes interactions can be observed, and the blocking processes can be identified. It is of utmost importance to explicitly indicate that the pipe is "overlapped", particularly in the context of `Windows`, to adhere to the Windows API requirements. The pipe for the various OS is not abstracted by Libuv. Therefore, the overlapped communication was denied, causing the parent process to remain blocked while it awaits the pong response from the child process.

4.5.5 Root Cause Analysis

In this subsection, we apply our technique to a major performance issue highlighted by `Theia` developers [107]. It relates to issue 9514 [2] :
The performance of vscode.workspace.fs.readDirectory(Uri) is a hundred times slower than VS Code, and issue 10684 [3] : Improve JSON-RPC communication performance. This specific issue has drawn much attention and interest, and has been the subject of intensive efforts. We were able to reproduce the performance issue locally, in order to evaluate our technique and show its usefulness and effectiveness.

To replicate the performance issue, we developed a VS Code plugin that calls the `fs.readFile` function to read a file. The plugin was explicitly tested in VS Code multiple times, with files of various sizes, and the average execution time was gathered. Then, we injected the plugin

2. `https://github.com/eclipse-theia/theia/issues/9514`
3. `https://github.com/eclipse-theia/theia/issues/10684`

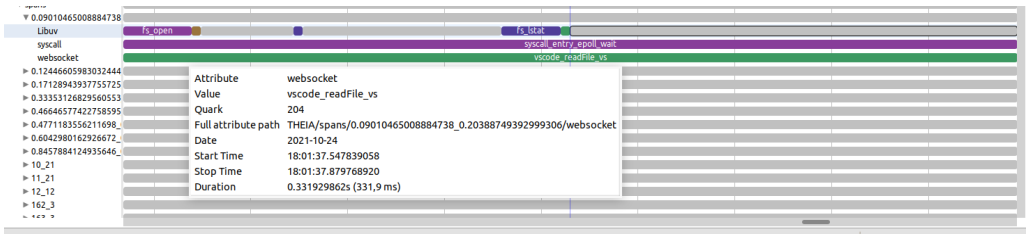


FIGURE 4.15 Plugin triggered vertical sequences. The sequences of `libuv` operations at the backend are presented along with the websocket span obtained from the execution of the function in the plugin. Syscall represents the sequences of system calls triggered by the plugin.

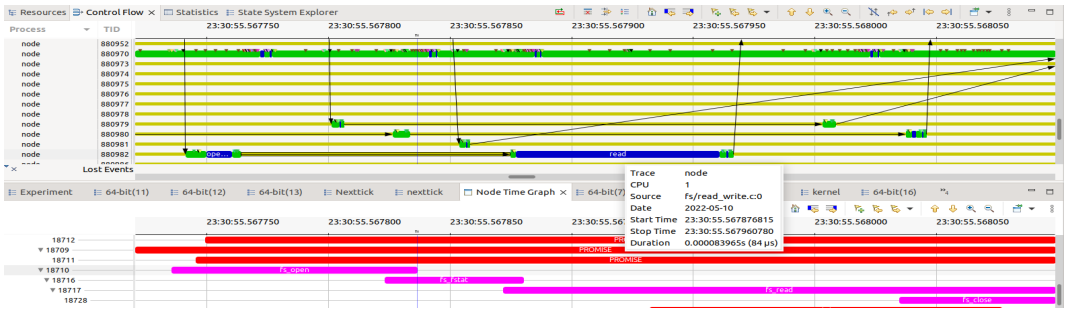


FIGURE 4.16 Vertical reading sequences of the Vscode plugin.

into `Theia` and ran the same tests. A significant performance issue was caused in `Theia` when the file size exceeded 450 Kb. The compatibility of `Eclipse Theia`, with VS Code plugins and extensions, makes it a robust and adaptable framework, chosen by numerous developers. Eclipse `Theia` is at the core of several tools popular in industry.

Figure 4.14 depicts the outcomes of our experiments with multiple files of different sizes. We perceive that the processing time is acceptable for extremely small files, but it begins to deviate from those of VS Code when the file size exceeds 420 Kb. Beyond this, the performance issue is obvious. Node Compass was used to identify the root cause of the problem, and demonstrate its efficacy in resolving such challenges.

Under Theia, reading a 500 Kb file required approximately 331 ms. Figure 4.15 depicts a part of the Theia vertical execution flow. The latter includes information regarding the initial function `vscode_readFile_vs`, executed by the plugin to send the service information in the payload to the backend, via websocket and JSON-RPC. When the relevant Remote Procedure Call (RPC) service is invoked at the backend level, atomic operations are generated at the `libuv` level. Each I/O action in `libuv` triggers a system call. This information is displayed in Figure 4.15, where we can also note the `vscode_readFile_vs readFile` method latency

at around 331 milliseconds.

Clearly depicted in Figure 4.15, the node process that runs the plugin enters the `epoll_wait` mode after sending the request. Nonetheless, this does not account for the execution delay of the request. Digging further into the execution flow of the request in the backend, as depicted by the `Control Flow` perspective of Figure 4.16 on the top, it can be observed that the `read` system call requires only 84 μs to perform. After executing the method, the backend process returns the response to the browser through Websocket. By narrowing the instrumentation to the Theia functions managing the communication layer, we observe that most of the request time is spent at the `JSON-RPC` communication layer. In this case, encoding binary data into a string is extremely inefficient in terms of performance. Mechanisms that allow sending direct binary data through Websockets should be implemented to handle such communications, instead of encoding it. Such an approach would improve the performance of the communication.

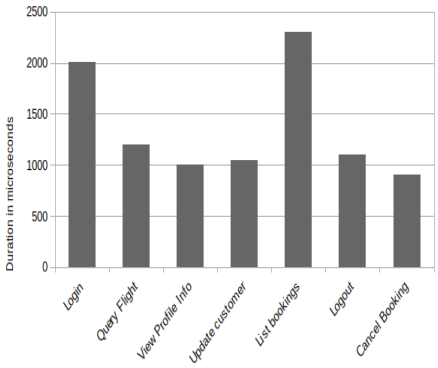
4.6 Evaluation

In this part, we measure the overhead associated with collecting the traces, and the time necessary to run our analyses. We also compare the overhead introduced by Node Compass with other `Node.js` tools. The primary purpose is to demonstrate a priori that our approach incurs a low overhead, when collecting data, in comparison to other `Node.js` analysis tools.

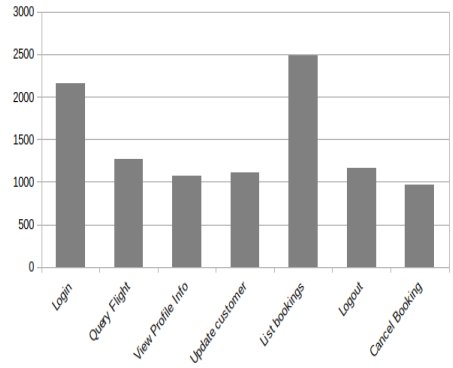
Our tool overhead is compared to that of `NodeRacer` [89] and `NodeAV` [84]. Although these two tools are used to solve specific problems, we will concentrate on their user trace collection architectures, which share with Node Compass the use of the `Async Hooks` API to monitor the lifecycle of asynchronous resources.

Figure 4.17(c) depicts the outcomes of the various micro-benchmarks we conducted to assess the overhead induced by the activation of `Async Hooks`. Observably, the activation of the `Init` callback imposes an very high cost on the performance. When a new asynchronous resource is created, this callback is initiated. As a result, it is possible to obtain information regarding the execution context of the resource. As can be observed, the situation deteriorates when all callbacks are activated, because it becomes necessary to monitor the resource initialization, execution start, execution end, and the destruction. The challenge lies in the tricky nature of resource tracking at the top layer of `JavaScript`. Clearly, for each resource, the `JavaScript` and `C++` barrier must be crossed. This results in a significant increase in costs.

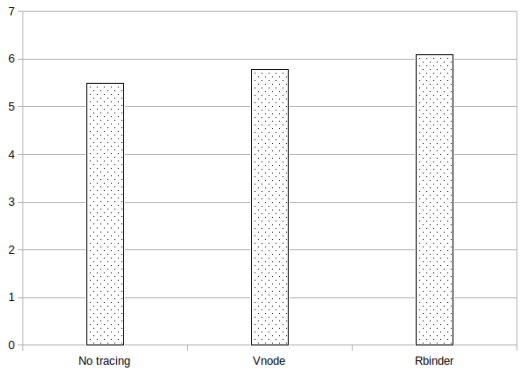
Unlike `Node Compass`, `NodeRacer` and `NodeAv` utilise `Async Hooks` in this manner. Unlike the



(a) Services duration (without instrumentation)

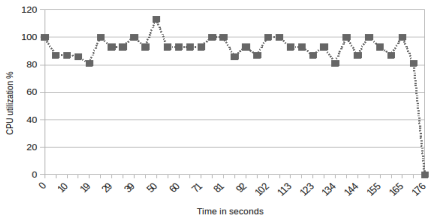


(b) Services duration (with instrumentation)

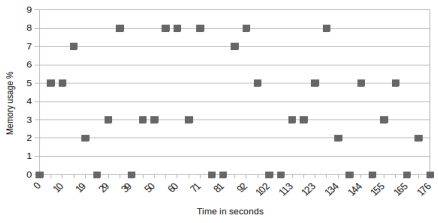


(c) Async Hooks performances benchmarks.)

FIGURE 4.17 Acmeair benchmark API paths. Time duration (Instrumented vs no instrumentation).

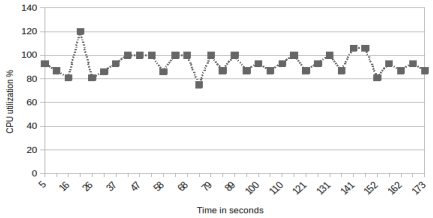


(a) Acmeair benchmark CPU usage without instrumentation

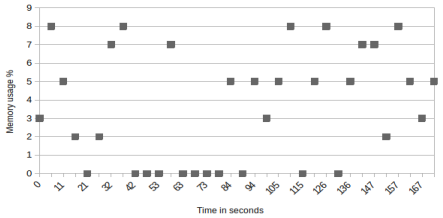


(b) Acmeair benchmark memory usage without instrumentation

FIGURE 4.18 Acmeair benchmark cpu and memory usage.



(a) CPU usage with instrumented Acmeair benchmark



(b) Memory consumption with instrumented Acmeair benchmark

FIGURE 4.19 Instrumented Acmeair benchmark cpu and memory usage.

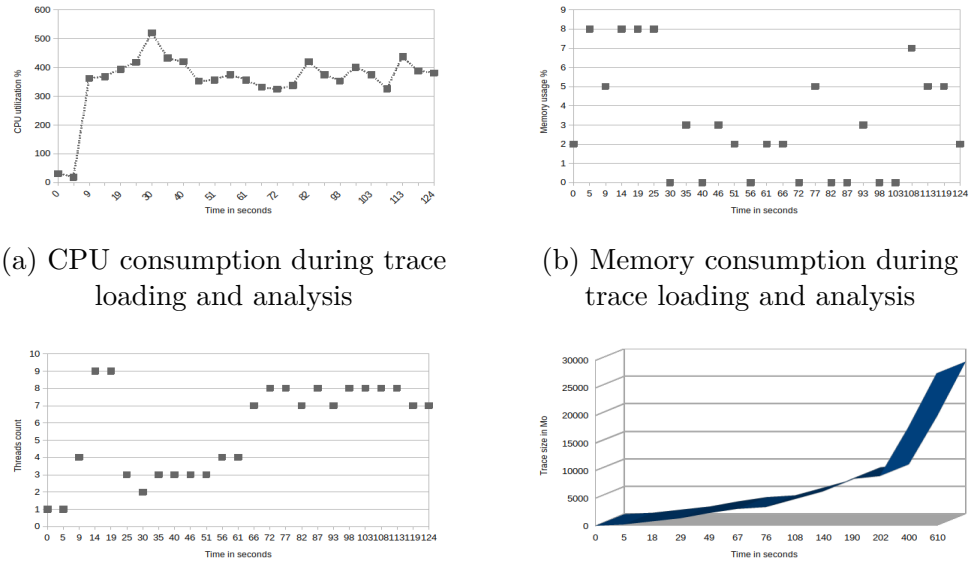
other tools, Node Compass does not activate the `JavaScript` layer callbacks. Using the `Async Hooks` module, information about the execution context is obtained, whereas monitoring is performed at the `VM` level. The instrumentation of the latter with `LTTng` enables the recovery of the execution context of the resources at the lowest level, resulting in a cost on the order of 8% for our method. To the best of our knowledge, there is no multi-layered performance analysis approach for `Node.js` that incurs a comparable low cost. The experimental version of Node Compass is available for free download on GitHub [4].

`Acmeair` [5], a well-known benchmark for flight management applications, has also been used to test Node Compass. `Apache Jmeter` was used to produce the user load. The results of 2000 requests executed with and without Node Compass instrumentation are displayed in Figure 4.17. As can be seen, the additional cost incurred is acceptable.

Figure 4.18 and Figure 4.19 illustrate resources consumption during the benchmark execution. Respectively, with no instrumentation and with instrumentation. We observe that the differences between the two cases are not discernible in terms of memory and especially processor usage.

We evaluated the cost of conducting our analyses for various trace sizes. Figure 4.20 depicts the utilisation of the CPU (a) and memory (b) during the loading and execution of the various trace-related analyses. We can observe the distribution of intervening threads over time, given the concurrent nature of the analysis execution. As the execution time of the analysis depends on the size of the collected trace, Figure 4.20 (d) displays the results of the latter in the context of different sizes. Instrumenting a `Node.js` application with Node Compass allows to collect a trace over time. The ratings provided represent the worst-case scenario. It is not necessary to obtain a kernel trace for each analysis. Some analyses do not

4. `https://github.com/dorsal-lab/tracecompass_nodejs_support`
5. `https://github.com/acmeair/acmeair-nodejs`



(a) CPU consumption during trace loading and analysis

(b) Memory consumption during trace loading and analysis

(c) Number of threads over running time

(d) Trace size vs analysis running time

FIGURE 4.20 Trace loading and analysis with Node Compass.

require their use and are limited to collecting a user trace from the internal `Node.js` layers. Analyses such as I/O processing rely on kernel events. The size of a user trace is approximately 450 times smaller, in comparison to a kernel trace, when all events are collected.

4.7 Conclusion

`JavaScript` is an environment with great potential and is the basis for thousands of applications currently in production worldwide. It enjoys widespread popularity among developers [108–110]. Its flexibility and continuously expanding ecosystem make it one of the most used distributed application environments. Understanding the performance of `JavaScript` applications is crucial for developers. Therefore, tools and adequate approaches are necessary to help detect the various problems at the basis of performance degradations.

We introduced a novel technique based on the vertical reconstruction of the execution flow of `Node.js` applications. Vertical profiling and calling-context profiling are effectively attained through the application of two distinct techniques. Through the identification of execution patterns in the trace, our method correlates events between the various layers. The end result is an interactive interface for identifying potential performance issues.

Our technique enables performance analysis from several angles. It is a global approach

that sees the system as being complex and dynamic, responding to a certain number of expectations. Therefore, unlike most tools that address specific problems, such as atomicity violations, detection of race issues, or that rely on higher level operations latency, to diagnose performance problems, Node Compass adopts a multi-level approach. This enables the reconstruction of the execution sequences of a task. In this context, depending on the conducted analysis, the vast majority of performance issues can be pinpointed, each in the responsible layer. The reconstruction of the application execution flow is an essential element in the granularity of the approach, in order to detect the problem root cause. One exciting area to consider for future work is to perform a critical path analysis of the different requests under `JavaScript`, in a distributed system context. It would allow a defined execution model, and help reveal bottlenecks related to specific requests processing and I/O threads management.

CHAPITRE 5 ARTICLE 2 : VNODE : LOW-OVERHEAD TRANSPARENT TRACING OF NODE.JS-BASED MICROSERVICE ARCHITECTURES

Auteurs : Hervé Mbikayi Kabamba, Michel Dagenais et Matthew Khouzam.

Soumis à Future Internet

Date : le 18 Novembre 2023

5.1 Abstract

Tracing serves as a key method for evaluating the performance of microservices-based architectures, which are renowned for their scalability, resource efficiency, and high availability. Despite their advantages, these architectures often pose unique debugging challenges that necessitate trade-offs, including the burden of instrumentation overhead. With Node.js emerging as a leading development environment, recognized for its rapidly growing ecosystem, there is a pressing need for innovative approaches that reduce the telemetry data collection efforts, and the overhead incurred by the environment instrumentation. In response, we introduce a new approach designed for transparent tracing and seamless deployment of microservices in cloud settings. This approach is centered around our newly developed Internal Transparent Tracing and Context Reconstruction (ITTCR) algorithm. ITTCR is adept at correlating internal metrics from various distributed trace files, to reconstruct the intricate execution contexts of microservices operating in a Node.js environment. Our method achieves transparency by directly instrumenting the Node.js virtual machine, enabling the collection and analysis of trace events in a transparent manner. This process facilitates the creation of visualization tools, enhancing the understanding and analysis of microservice performance in cloud environments.

5.2 Introduction

The swift advancement of technology has propelled the widespread adoption of microservice architectures, which highlights key aspects like availability, resilience, fault tolerance, and enhanced collaboration among teams. In such an architecture, each component operates independently and communicates through efficient, lightweight protocols. These architectures are highly favored due to their facilitation of collaborative efforts and their capability to meet

modern challenges in application design, development, maintenance, and deployment [26]. They bolster the system resilience, effectively manage failures, and seamlessly adapt to scaling requirements. Nonetheless, these advantages come with certain challenges, particularly due to the heterogeneity of the components, issues in service allocation, and notably, concerns regarding the overall system performance [111].

Debugging issues in microservices architectures poses significant difficulties. System malfunctions can compromise user experience, and identifying the root cause of these issues can be elusive based on the information at hand. Even with error indications from protocol headers like HTTP, the process of issue diagnosis can be challenging. Additionally, the arrangement of the components in a way that maintains system attributes such as availability and low latency raise intricacy.

The challenges of debugging [39, 112–114] and component arrangement [115], have been tackled through the implementation of distributed tracing techniques [52, 116, 117]. However, applying these methods necessitates instrumenting the application source code, which brings additional overhead and the risk of altering the application behavior. Other strategies extend instrumentation to dependencies to minimize changes at the application level.

Transparent tracing, in contrast to distributed tracing, allows for the collection of system telemetry data without the need for a prior instrumentation phase. Therefore, developers are not required to modify the application's source code to insert trace points. On the other hand, tracing a system incurs an additional cost, which must be considered by any collection approach. Developing tracing methods that have a minimal impact on the system is necessary for applications where performance is a primary requirement. Although overhead issues have partly been addressed through sampling methods [118, 119], both approaches to instrumentation present challenges. Instrumenting at the dependency level may fail to capture internal application logic issues like bugs, whereas instrumenting the application's source code could potentially alter its functioning. Both methods entail modifications to the application, necessitating human effort and incurring extra costs.

The burden of instrumentation efforts has led to the development of strategies aimed at reducing these costs across several research domains [41, 120]. In the context of microservices, this issue has recently been addressed by advocating for the use of proxies, [121], as an intermediary layer for transparent tracing. However, while this spares the application from source code modifications, it shifts the burden to setting up and configuring the proxies. Additionally, their operation involves intercepting system calls to insert trace contexts into requests, which can be problematic in public clouds where kernel access might be restricted.

We propose an innovative approach for tracing and debugging microservices in the `Node.js`

environment, which is designed to address the limitations of existing methods while ensuring full transparency.

This approach distinguishes itself in two key areas : (1) it obviates the need for developers to invest effort in establishing a collection infrastructure that includes instrumentation, and (2) it transparently analyzes collected traces, leading to visualization tools that map the interactions among microservices, thereby enabling the debugging of performance issues in a completely transparent manner. The focus on `Node.js` stems from its complex, asynchronous environment. However, this approach could potentially be extended to other environments such as Java, Golang, and more.

The remainder of the paper is organized as follows : Section 5.3 introduces the basic concepts of the study. Section 5.4 discusses related work about the subject. It is followed by Section 5.5 that presents the proposed approach for tracing and analysis. Section 5.6 presents the results of our work leveraging some use cases to show its pertinence and relevance. An evaluation of our tool is conducted in Section 5.7. In Section 5.8, we discuss the results obtained while in Section 5.9 a conclusion on the work is drawn.

5.3 Basic concepts

5.3.1 Microservice architecture

The microservices architecture is recognized as an application development strategy that involves breaking down applications into a series of loosely interconnected components. Its growing popularity can be attributed to the ease it brings to continuous delivery and its ability to enhance the scalability of applications [22].

Within this architectural framework, each component operates independently, managing a distinct function of the application. Communication between these components is facilitated through clearly established, lightweight protocols, typically using APIs. A key advantage of microservices architecture is the autonomy of its components, which grants development teams the flexibility to update and deploy individual services without impacting the overall application. This approach leads to a more streamlined development process and greater agility in software development [23].

While microservices offer numerous benefits, they also pose challenges in debugging. As an architecture comprising multiple heterogeneous components, performance monitoring tools must take this diversity into account. Typically, instrumentation is carried out on each component using tracing libraries specific to the component environment. To gain a comprehensive view of the system health, it is crucial to use distributed tracers. A benefit of microservice

architectures is that the instrumentation phase can be treated as an application update, allowing for its flexible integration into the application deployment pipelines.

The health of microservices can also be monitored using logs, but the cost of this strategy becomes prohibitively high when the application consists of multiple nodes. The challenge lies not only in interpreting the inter-causalities among nodes but also in managing the large volume of data, which quickly becomes unmanageable. Tracing remains the best way to address this problem, as it allows for understanding the system operation as a whole and, if necessary, identifying bottlenecks.

5.3.2 Distributed Tracing

Distributed tracing is a strategy for collecting execution data from modern systems, particularly microservices architectures. It traces the lifecycle of a request as it passes through all the nodes in the system. Distributed tracing provides a hierarchical view of the trace, in which one can observe the time a request spends on each service [39].

Distributed tracing was introduced to address the complexities of distributed architectures, which are not suited to traditional debugging and tracing methods. In microservices environments, components need to interact to produce a result. In other words, when a request is issued from a particular node, it may need to interact with several other services in the infrastructure to return the result. In this context, when an issue arises along its path, it is necessary to identify where the bottleneck occurs to resolve the problem.

Traditional tracing techniques are not suitable because they are generally used for debugging monolithic applications. In the case of microservices, tracing involves injecting a trace context into the request headers to identify it throughout its lifecycle. In this way, the request can be properly aligned and hierarchized according to the service level it invokes over time. The various calls made by microservices during their interactions can thus be traced to understand the overall performance of the system [113].

Distributed tracers such as Google Dapper [39], Zipkin, and Jaeger have revolutionized technology by offering the ability to collect data from each request, their execution times, and the causalities between services in the infrastructure. Furthermore, analyzing the collected trace data can be complex. Distributed tracing systems generate a large amount of data, and interpreting them often requires advanced data analysis skills and a deep understanding of the system architecture [117]. Distributed tracing continues to evolve, with new improvements and integration into increasingly sophisticated tools and platforms. Therefore, it is necessary to propose tools that address shortcomings and bring more flexibility to the ecosystem.

5.3.3 Node.js environment

Node.js, an open-source environment, originates from the JavaScript V8 engine it is built on. It has brought the versatility of server-side JavaScript programming since its beginning in 2009. Node.js revolutionized the web ecosystem by enabling developers to use JavaScript for server-side application development.

Distinctive for its non-blocking, event-driven nature, Node.js is well-suited for high-performance applications. Its architecture is asynchronous and built on a single-thread model. In a Node.js application, a primary process known as the event loop orchestrates the execution of various events in different execution phases.

Tasks or events submitted for execution are first queued in a specific queue based on the event nature. The event loop traverses these phases, déqueues the events in each queue, and executes them. For blocking events, such as I/O operations, a thread pool is used to delegate execution and avoid blocking the event loop. Node.js is a multi-layered system. Internally, it comprises several components that work together to yield a result.

However, at the lower layers of the operating system, such as in the kernel space, the Node.js process appears as a black box making system calls or generating context switch events. From this perspective, it is challenging to discriminate events being executed. A significant unresolved issue is linking high-level information, such as requests and invoked JavaScript functions, to actions performed in the V8 engine and the Libuv orchestration layer.

Current tools only allow visualization of high-level information, such as a request duration or the execution time of a service or JavaScript function. However, when such information is provided by distributed tracing tools, debugging is necessary to trace back to the cause. Debugging such issues in Node.js is extremely challenging, since, as a single-thread system, all concurrent request executions are conflated into the same process. Distinguishing which request is causing performance issues is difficult because, even if a service shows high latency, it does not necessarily mean the service itself is at fault. For instance, the event loop might have been blocked by the execution of an non-optimized function, thus propagating the error to other pending requests. Therefore, it's important to develop methods suitable to these environments.

5.3.4 Transparent Tracing of Node.js-based Microservices

A fundamental principle of the microservices architecture is the use of tools that are adequate for the task at hand. It enables the utilisation of various development environments, tools, and libraries. Due to the performance-restrictive architecture of microservices, effective and

established methods are required to track and monitor their behaviour and, if necessary, address bugs. Distributed tracers are among the most efficient tools available to developers for observing and monitoring their systems. Through the use of collection modules, they are able to aggregate traces and enable their presentation through graphic user interfaces in the form of spans.

However, the adoption of these tools in the context of distributed systems, such as microservices architectures, necessitates a solid understanding of the source code of the application. Worse, modification of the latter is essential to incorporate tracepoints needed for information collection. On the one hand, the application behaviour and programming structure are altered, and on the other hand, these tracers incur a substantial additional cost, which varies based on the execution environment and the technology employed.

Accelerating the performance analysis process requires methods that add little overhead on the system, and yet spare the developer from having to modify the program during the instrumentation phase. It simplifies the task by reducing the amount of work required and, in general, the associated costs.

The proposed method uses a transparent tracing approach that does not need any intervention from the practitioner, it uses the ITTR, which is a new technique we introduce that leverages the internal asynchronous mechanism of Node.js for context reconstruction. In this way, request internal execution sequences can be reconstructed in a fully transparent manner.

5.4 Related work

Recent research has demonstrated that distributed tracers are crucial for enabling the monitoring of interactions between microservices as studied in [115]. This experience demonstrates the need for a more lightweight technique to trace such fine-grained architectures.

Santana et al. [121], suggested a novel transparent tracing methodology that leverages the kernel of the operating system to intercept system calls associated with communication among microservices. They proposed using a proxy that adds a neutral layer to the microservice, to intercept its interactions and correlate the information to deduce the causalities associated with the various requests. The interception of system calls ensures application tracing transparency, but the developer is responsible for configuring the infrastructure.

Statistically extracted dependency structures from documentation was used in service discovery in [122], while fault detection was addressed in [123], through middleware instrumentation to log the respective components that process a particular request. A degree of transparency could be achieved, but such an approach requires the developer to dedicate

much time to library instrumentation.

Distributed tracers Dapper and X-trace proposed respectively in [39], and [52], are able to trace the whole request lifecycle, expose its flow, and help diagnose issues throughout their execution. They rely on trace context injection mechanisms to reconstruct the context of the trace. A prior instrumentation phase of the application is required to activate the collection mechanism. In contrast to those approaches, our method does not inject the trace context into the request. It leverages the internal asynchronous mechanism of `Node.js` to reconstruct the request path.

Tracing request path strategies was also tackled in [124], using heuristics. Request causality diagnosing algorithms were proposed in [125]. Both approaches offer a degree of transparency but rely on middleware instrumentation.

Gan et al. [62], introduced Seer, an online debugger designed to foresee quality of service (QoS) violations in cloud-based applications. Their research was conducted on a microservices framework, employing Memcached, which shares functional similarities with Redis as an in-memory database.

The process of debugging performance issues using Seer requires an instrumentation phase for microservices. This step is also implemented in the Memcached data store, particularly focusing on polling functions and various network interface queues. Seer has the ability to predict QoS violations using a model developed from deep learning techniques applied to upstream traces. By instrumenting the functions responsible for managing packet queuing at the data store level, Seer can effectively identify potential bottlenecks, especially those involving Memcached.

5.5 Proposed solution

5.5.1 System Architecture

The depicted diagram in Figure 5.1 illustrates the operational framework of the system. Given that each microservice is deployed in a container, `LTTng` [68] (Linux Tracing Toolkits Next Generation) is enabled on each of them to capture traces. This produces several CTF (Common Trace File)-formatted local trace files. A file aggregator retrieves the aforementioned files for importing as an experiment under Trace Compass (TC) [98], where analyses will be executed. The running of the analyses allows building an execution model based on the state system technology. It is constructed from the trace extracted attributes.

The State History Tree (SHT) [78], is a highly efficient data structure employed in the creation

of the model, while reading the trace. It is used to store the attributes extracted from the trace, as well as various analysis-related data. TC provides the framework for modelling and developing such data structures. The optimisation technique employed enables the model to be queried in logarithmic time. It provides multiple features for the organisation of traces and the objectivity of analyses. This is achieved by employing multi-abstraction, highlighting, and filtering. TC enables the definition of the desired granularity of performance measurements and the application of analyses within the desired time limits.

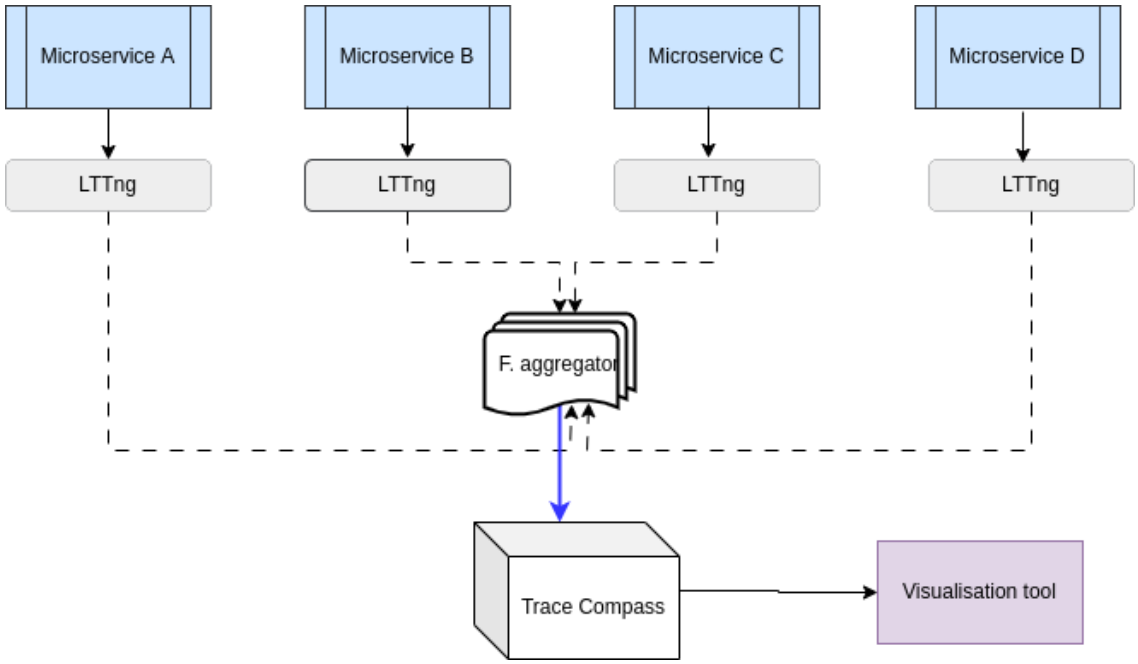


FIGURE 5.1 Example of a trace collection architecture used by our approach.

The trace is collected by inserting tracepoints into the internal `Node.js` layers. LTTng is used to conduct static instrumentation. The tracepoints are initially inserted at the level of the `C++` bindings that interface with the native `JavaScript` modules, specifically at the functions that handle the socket communications in `Node.js`. Then, it is possible to extract the necessary information pertaining to the various sockets and their attributes for the correlation and context reconstruction stages.

Information on the request attributes (methods, addresses, and ports) is transmitted directly from native `JavaScript` modules to `C++` bindings. In this manner, the expense of parsing `HTTP` requests is avoided, and an `LTTng` probe is inserted at that juncture. The engine that generates asynchronous resource identification numbers is probed during the second instrumentation step. This technique allows accessing, at the origin, the identifiers of the execution contexts associated with asynchronous resources, as well as the identifiers of the execution contexts of

the resources that generated them. In addition, it enables monitoring of their entire life-cycle, from creation to destruction.

Therefore, the costly `Node.js Async Hooks` API is unneeded for the monitoring of asynchronous resources. Tracing is performed directly within the VM, which significantly reduces the overhead in this context. `LTTng` has become known as the fastest tracer in the world and incurs minimal system overhead. It permits the creation and collection of events, which are then loaded into TC. Developed extensions in the latter facilitate the creation of event analyzers and handlers. It is a free, open-source tool that allows for the analysis of traces and logs. The extensibility of the system enables the creation of graphs and views, as well as the extraction of metrics.

5.5.2 Patterns-based context reconstruction formalization

When conducting the analysis using TC, a state system is built. The states encompass all the events that are deemed acceptable by the analyzer for the purpose of instantiating and activation of the different system transitions. Our approach for reconstructing the execution contexts of all the requests relies on identifying specific sequences of system transitions. These sequences are retrieved from the global state system and serve as the detected patterns. By traversing the trace to construct the state system, concurrent state subsystems belonging to the global system are identified and correlated with the concurrent requests to which they are bound. The obtained state system can be regarded as a finite state machine consisting of six components.

Let M be a finite state machine :

$$M =< P, F, B, G, s, tr > \tag{5.1}$$

where :
P represents the state space
F represents the event space
B represents the action space
system's G represents a subset of P
s represents an element of P, the initial state
$tr : P \times F \rightarrow P \times A$, represents the state transition function

$$tr(p, f) = (q, b) \tag{5.2}$$

$$TR : E^* \rightarrow P^* \times B^* \tag{5.3}$$

where X^* represents all the sequences of members belonging to X. In other words, the patterns that model the concurrent requests execution in the state system are sequences of members of the global state system.

The transition function can be extended as defined in Equation 5.2. In this case, if p is the active state of M, and if there is an occurrence of event f, then q becomes the new active state of the system, therefore action b is taken. The handling of events during the execution of the analysis make the system transition to multiple states. For each state, related actions are undertaken.

$$tr(s, p) = (k, a) \tag{5.4}$$

$$tr(k, t) = (w, b) \tag{5.5}$$

$$tr(w, h) = (n, c) \tag{5.6}$$

$$TR(pth) = (skwn, abc) \tag{5.7}$$

Consider the events p, t, h, the states s, k, w and the actions a, b, c defined in Equation 5.4, Equation 5.5 and Equation 5.6. Then `pth` is an event sequence in F^*. As defined by Equation 5.7, the systems should transition from its initial state, s, to k, then to w and n. For each transition, the system will perform the actions a,b and c. The inputs to the model are the different accepted events from the trace while the handler is running. Actions are taken for each of the accepted events to construct the global state system and build the SHT. The patterns are modelled as subset of the global state system identifying transition sequences.

5.5.3 Transparent trace collection

Microservice interactions occur through message passing across network sockets. In microservices using HTTP as the communication protocol, when one microservice wants to send a message to another, it first establishes a connection with the remote component. Internally in Node.js, native modules managing network sockets communicate with the Node.js virtual machine to request information about the created socket context. In a single-threaded environment like Node.js, multiple events are managed concurrently. Node.js implements an

internal mechanism to track different asynchronous resources by assigning them an identifier.

For instance, if a function with identifier "12" first creates a socket, this socket might receive identifier "13," implying that this new resource was created in the "12" context. Similarly, any new resource initiated within the socket's context will have "13" as its execution context, aside from its own resource identification number. Managing asynchronous resources in environments like Node.js is extremely complex and challenging. Async Hooks, an experimental API, was proposed to track the lifecycle of these asynchronous resources. However, this API is significantly costly in terms of system performance, sometimes adding overhead by up to 50%.

For most production applications, this overhead is not tolerable. This is because each call to the API requires crossing the JavaScript/C++ barrier of the V8 engine, which incurs a very high cost to the system. The proposed solution circumvents this by addressing and collecting information directly at the source, at the level of the engine responsible for generating asynchronous resources. In other words, our solution involves definitively instrumenting the Node.js virtual machine to transparently collect network communication data.

Figure 5.3 shows the internal process of execution context creation at the VM level when a network connection is established. It can be observed that when a microservice sends a message to another component, Node.js native network modules request the context from the asynchronous resource manager.

This manager initializes the new resource and assigns it a generated number. The number assigned to the created socket allows for the identification of the execution context of objects. For example, when a response to a request returns, Node.js checks the socket number (context) in the request header to route it to the waiting resource. This is Node.js multiplexing function, given its single-threaded nature.

To trace interactions between microservices, five main tracepoints were inserted into the native module and the Node.js virtual machine : `http_server_request`, `http_server_response`, `http_client_request`, `http_client_response`, and `async_context`.

The `http_server_request` tracepoint is activated when the server receives a request to process, such as when a microservice receives a specific request. The `http_server_response` tracepoint is activated when the server returns a response after processing the request. The `http_client_request` tracepoint is activated when a component emits a request, for example, when a microservice contacts another microservice. The `http_client_response` tracepoint is activated when the response is returned to the sender.

Figure 5.4 shows how this process unfolds when a request is issued by a microservice. First,

native network functions get the socket context information, containing the socket number and the number of the resource that created it, to preserve hierarchy and execution sequence. The creation and initialization of the new resource by the VM trigger an event captured by LTTng (the event is recorded as `async_context`).

This event exports various information in its attributes, including the identification number of the created resource, the identification number of the parent resource, and the resource type. After the request is sent by the native network functions, the `http_client_request` tracepoint is activated and captured by `LTTng`. It is exported with the attributes represented in Figure 5.3.

When the destination microservice receives the request, it can be observed in Figure 5.5 that the request is first decoded by Node.js network functions, and at the same time, the `http_server_receive` tracepoint is activated. The event is exported to LTTng with all the attributes present in Figure 5.5. At the end of the request processing, the request is returned, activating the `http_server_response` tracepoint, which is also exported with all its attributes.

Activating the different tracepoints produces a file in the CTF format at the end of the system tracing. This file containing the transparently collected information can be very large depending on several factors, such as the system load or the duration of its tracing. Therefore, it is crucial to employ automated methods to analyze the trace to extract relevant information. The next section addresses the new analysis approach we propose in this work.

5.5.4 Trace analysis technique

Performance debugging in microservices involves a phase of trace collection that necessitates system instrumentation. This step is crucial for gathering the necessary data to interpret the system operation. Analysis abstracts the system overall functioning to avoid delving into the minutiae of the data extracted from the system. However, effective, robust, and rapid methods are required to utilize the data collected in the initial phase, as the files can become exceedingly large, containing millions or even billions of events. This necessitates the use of tools capable of handling such vast quantities. Our approach conceptualizes system executions as finite state machines. A request is viewed as an automaton transitioning through states based on trigger events. The automaton sequence of transitions represents a series of events occurring during system execution.

Our method achieves its transparency in analysis by matching pattern sequences observed during request executions. An initial understanding of system activity based on collected

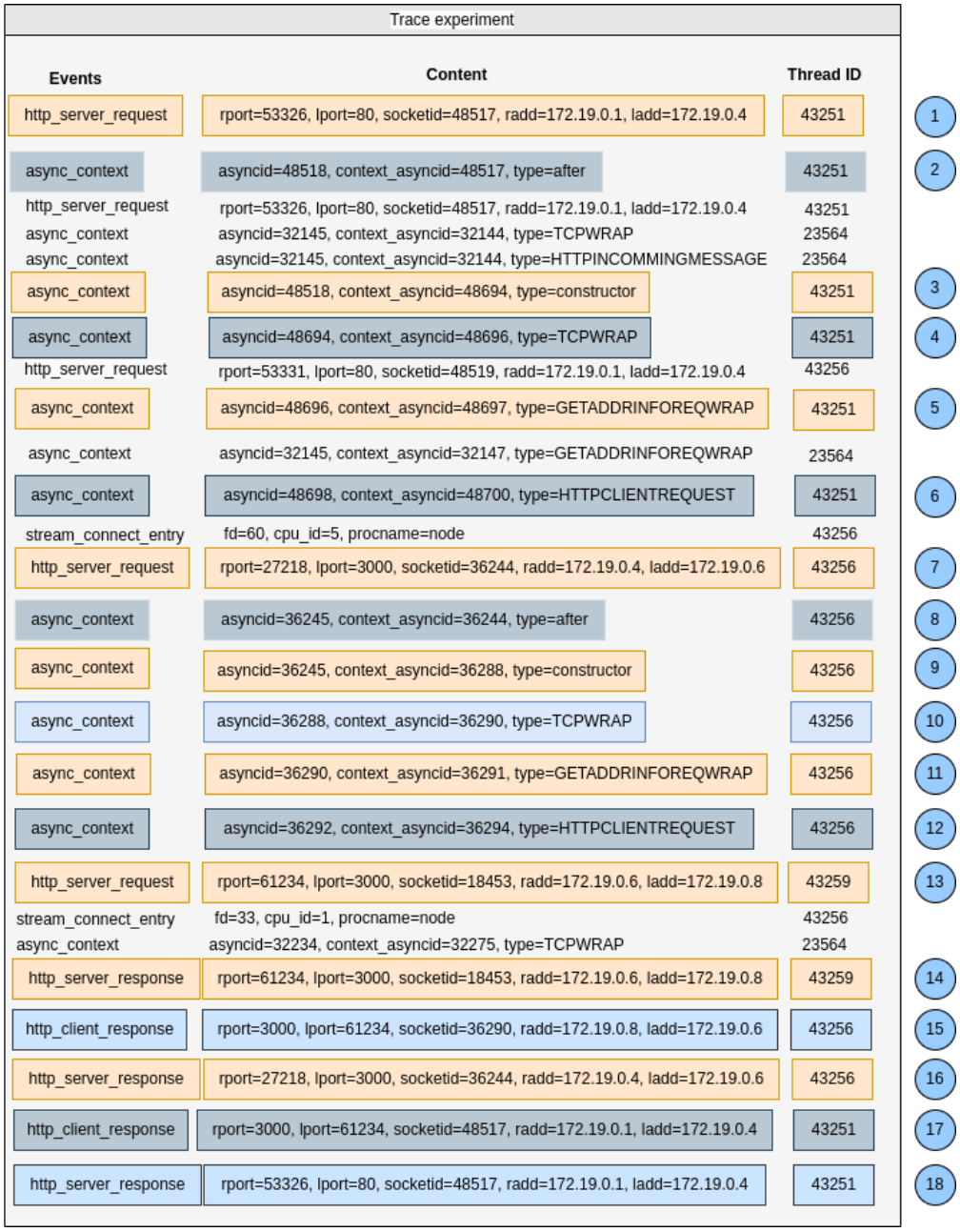


FIGURE 5.2 Example of a user trace experiment. Several trace files are aggregated and opened as experiment in TC

traces allowed the identification of recurring patterns in microservice interactions within Node.js. We then modeled these patterns as finite state machines to understand the system various states to debug its performance. We utilized TC to load the trace for our analyses. TC' extensibility enables the development of visualizations based on analyses, providing essential tools for system performance analysis. To preserve the automaton different states, we used

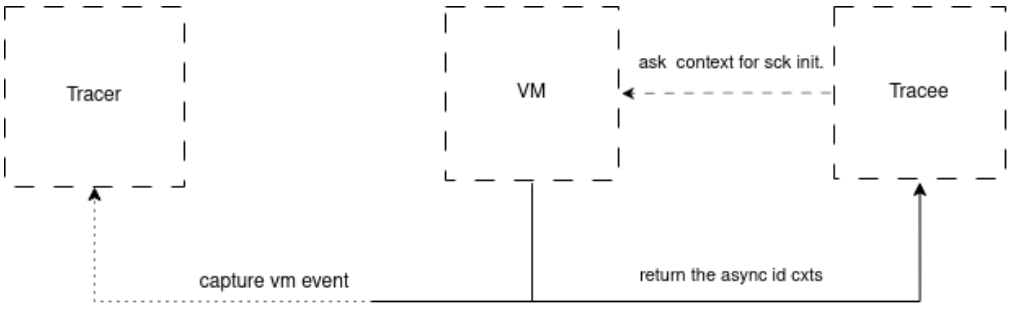


FIGURE 5.3 Internal resource context creation by the VM

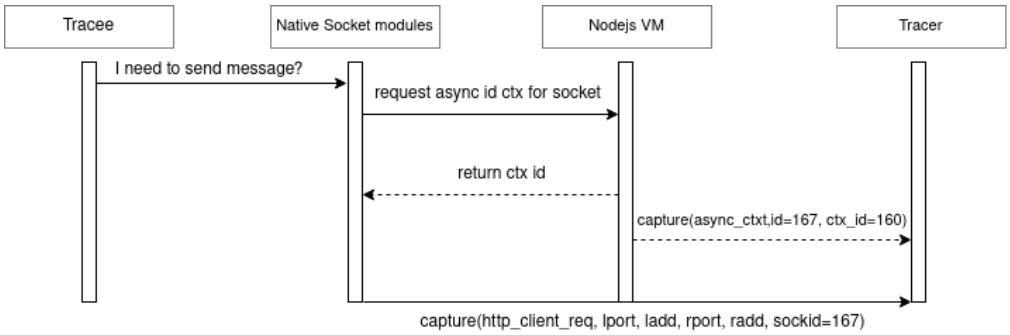


FIGURE 5.4 Internal communication process when sending a request. Tracepoints are activated

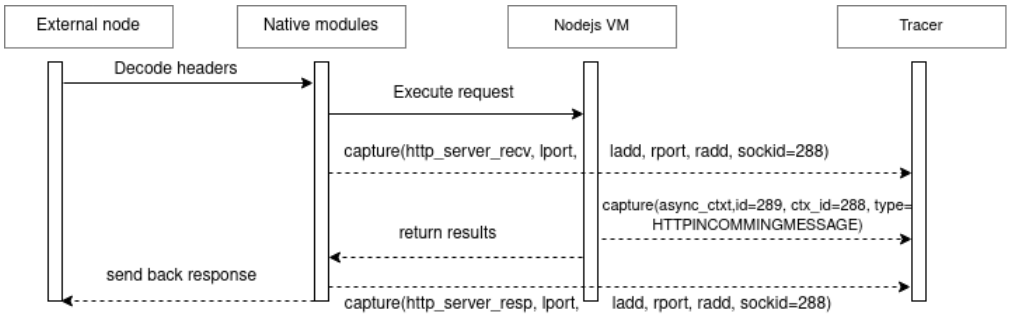


FIGURE 5.5 Processing incoming request. Tracepoints get triggered.

a particular, expandable data structure optimized for supporting very large file sizes, known as the SHT.

Observing Figure 5.6, a request arrives at the gateway microservice, which must redirect it to the relevant service microservice.

In this case, activating the `http_server_request` tracepoint initiates the state machine and sets it to the "receive request" state. To preserve this state, the highest level in the hierar-

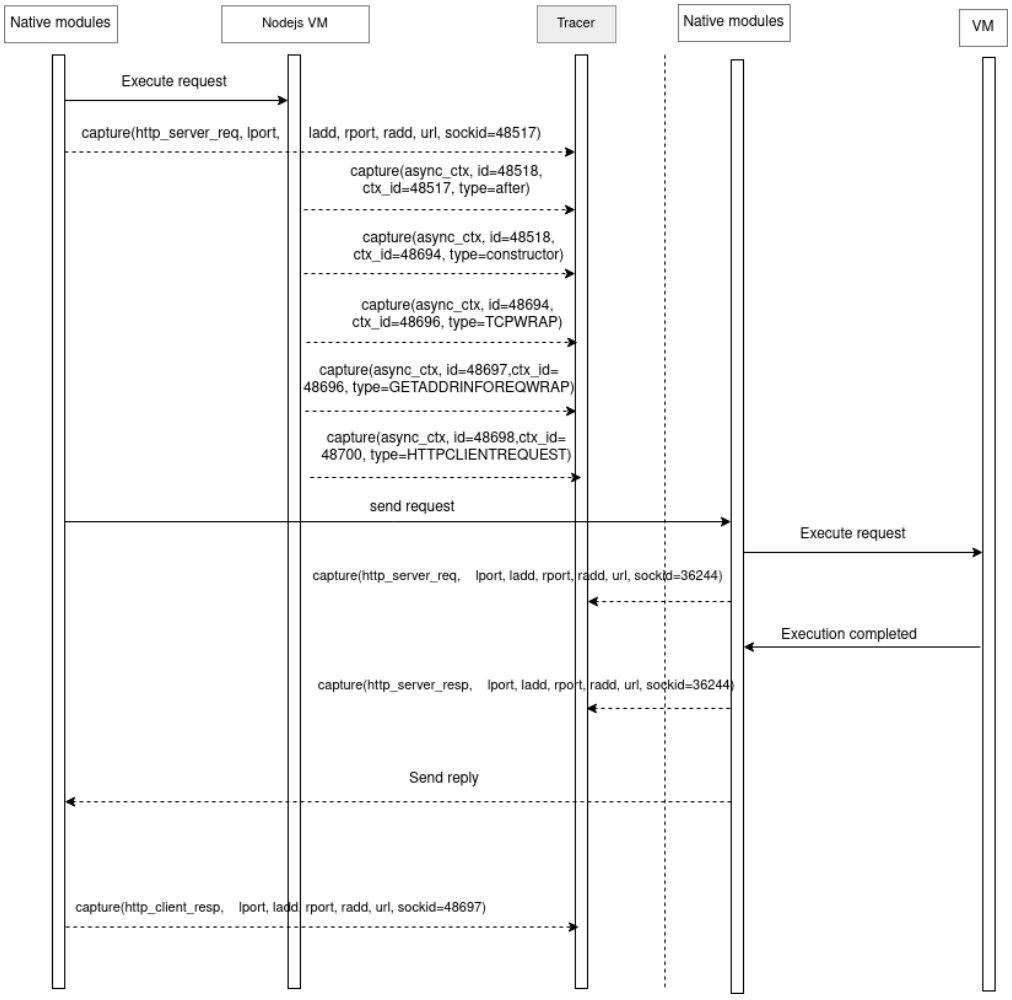


FIGURE 5.6 Communication between two microservices.

chy, the automaton state is recorded in the SHT with data extracted from its attributes. The automaton transitions to the next state when the `async_context` event is encountered in the trace, and its attribute type is "after". Here, the sockid value of the previous `http_server_request` event is matched with the `ctx_id` attribute of the `async_context` event, ensuring that the current event occurred within the context of the ongoing request.

The next state is activated when the `async_context` event's attribute type is "constructor". At this level, the id attribute value is matched with that of the previous state (48518 and 48518), ensuring that the reconstructed sequence is linked to the initial request. The automaton transitions to the next state when the `async_context` event type value equals "TCPWRAP".

To maintain the sequence context, the id attribute is matched with the `ctx_id` of the previous state (48694 and 48694). The next automaton transition occurs when the `async_context`

event type value is “GETADDRINFOREQWRAP”, with its `ctx_id` matched to the previous state (48696 and 48696). Finally, the system transitions to the next state when the `async_context` event type value is “HTTPCLIENTREQUEST”. Here, the id attribute value must be one order higher than the previous state (48698 and 48697).

At this point, the asynchronous sequences through which the request received by the gateway microservice passes to be sent to the concerned microservice are transparently reconstructed. This state sequence is the model followed by Node.js to communicate with microservices via HTTP (REST API). The type attribute values are obtained directly from the Node.js virtual machine, representing the different asynchronous resources created during request execution. Each system state is recorded with its attribute values in the SHT. The context allows hierarchically inserting states and attributes into the SHT tree to form a hierarchy defining the sequence of automaton execution, their start and end, thus enabling visualizations to extract information for studying the system performance. The outcome of the algorithm yields a structured and hierarchical representation of the diverse interactions across microservices, achieved through a fully transparent process.

5.6 Results

In this section, we demonstrate the capabilities of our tool using three use cases scenario. The objective is to effectively articulate the anticipated outcomes derived from the utilisation of `Vnode`. The `Nodejs-Restful-Microservices` [1] application is utilised for this purpose. This application is a complete microservice architecture developed with `Node.js` that uses `Redis` as an in-memory store.

5.6.1 Use case 1

In the initial use case, simultaneous requests are performed in order to retrieve specific information pertaining to an individual user. The requests made are of the `GET` type. Executing those requests after deploying the application generates multiple CTF-formatted trace files. As described previously, they are aggregated and imported as an experiment in TC. Figure 5.7 depicts the visual outcome of our analyses. The reconstruction of the alignment of request execution flows in accordance with their respective contexts can be observed.

After receiving the request, the server transmits it to the user microservice. The request is then forwarded to the Redis gateway since the data has been put into memory in the Redis data structure. The `Vnode` facilitates the transparent horizontal sequencing of requests and

1. `https://github.com/tudtude/MICROSERVICE-RESTful-Nodejs`

can be seamlessly integrated into the application development and operation pipelines.

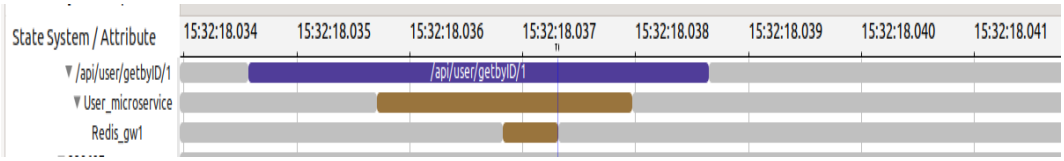


FIGURE 5.7 Execution of the get request reconstructed by the analysis

5.6.2 Use case 2

The second use case involves the sending of `POST` requests to place item orders. Figure 5.8 demonstrates the capability of `VNode` to smoothly rebuild request execution contexts. it depicts the output of the algorithm execution. In contrast to the initial use case, upon receipt of the request by the server, it is promptly forwarded to the microservice responsible for handling orders. As the operation entails the insertion of data to the database, the microservice executes the operation directly, bypassing the `Redis` gateway.

By analysing the two use cases, the unique feature of `Vnode` regarding the reconstruction of the communication architecture becomes evident. The strength of `Vnode` resides in its capacity to enable developers to comprehend and visualise the communication architecture of microservices systems implemented in `Node.js`. The developer does not need to grasp the application code or inner workings to comprehend how its components interact internally. `Vnode` reconstructs each API call execution sequence transparently and presents the result in a visual and interactive tool.

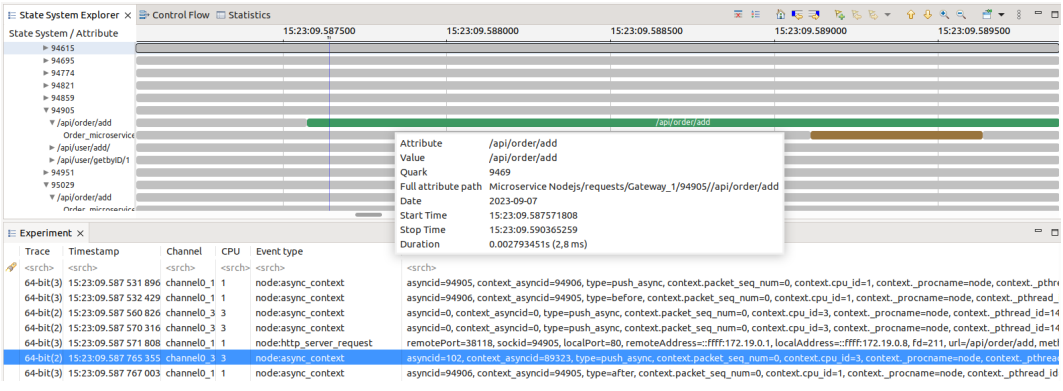


FIGURE 5.8 Execution of the POST request reconstructed by the analysis

5.6.3 Use case 3

In this third scenario, requests are sent to a user authentication service to obtain the tokens necessary for user session validity. As shown in Figure 5.9, in this case, once the request reaches the gateway microservice, it is automatically redirected to the authentication microservice. After processing the request, the latter redirects it to the microservice acting as a proxy for the user service, before being redirected again to the user microservice, which retrieves user information from the Redis database through the redis gateway microservice.

Aligning the spans allows the complete visualisation of the time the request spent in each microservice. By combining this analysis with the one presented the `Redis` one, shown on the bottom in Figure 5.9 and Figure 5.10, we can observe that the execution of the "get" command on the Redis server took an extremely short time, only 40 microseconds. However, the request took at least 18 milliseconds to complete. It is clear that the majority of the time was spent during the interactions between the microservices.

These results demonstrate a trace can be collected and analyzed in a completely transparent manner without requiring developer intervention. To the best of our knowledge, there is currently no approach that allows for this. In this context, the execution flow and operational architecture of the microservices can be visualized without prior knowledge of the system's implementation.

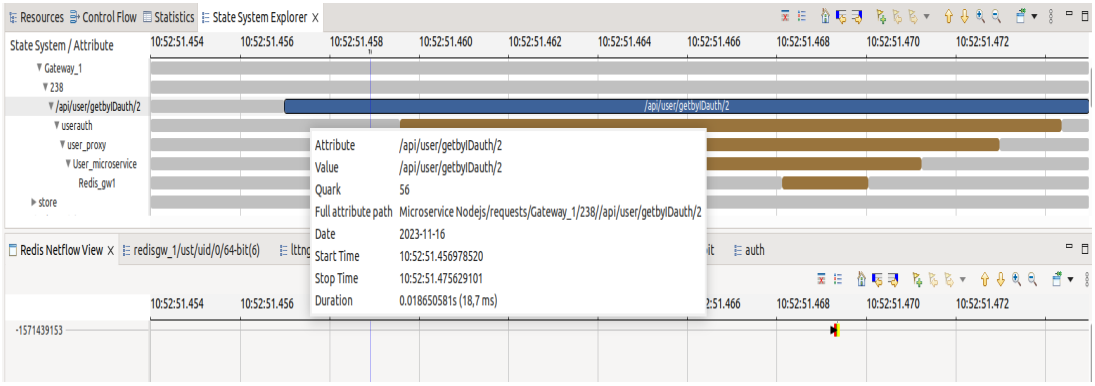


FIGURE 5.9 Request sent through the authentification service

5.7 Evaluation

This section presents an evaluation of our tracing approach. Experiments we carried out in different scenarios for the purpose of validating our strategy and comparison to state of the art approaches.

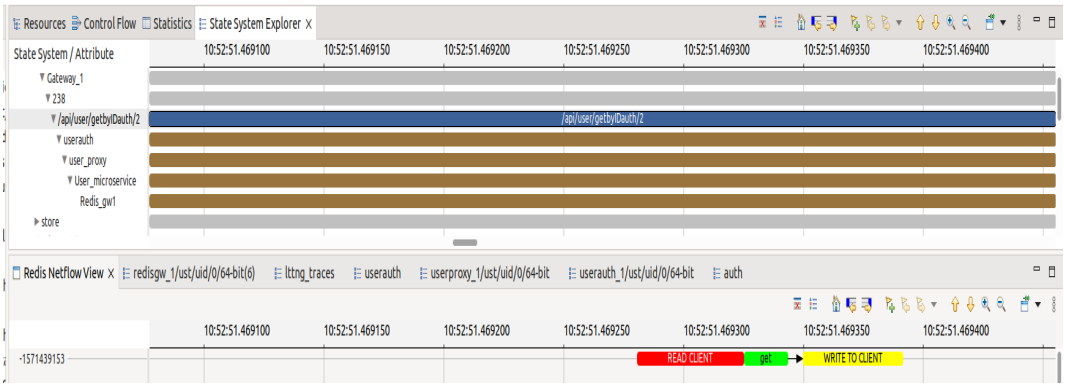


FIGURE 5.10 Zooming in into the Redis analysis that is joined to the Nodejs one.

5.7.1 Objectives

A comprehensive assessment was conducted to evaluate the performance of our solution with respect to the incremental cost incurred by the system being evaluated. The process encompasses three primary parts : i) an evaluation of the overall overhead incurred by the implementation of our tool ; ii) a comparison of the overhead associated with our tool in relation to other tracing methodologies ; and iii) the assessment of metrics that directly influence the development and operation of microservices.

5.7.2 Experiments

TABLEAU 5.1 Defined experiment parameters

Parameters	value
number of executed requests	1200
Time between requests	Randomly generated (Gaussian distribution)
Tracing configurations	No Tracing State-of-the-art Rbinder Our strategy

We utilized the Nodejs-Restful-Microservices application to generate the user traffic for the experiments. The response time was selected as the metric for evaluation because it accurately reflects the user experience, as highlighted in [126]. The parameters employed in our investigations are presented in Table 5.1. `Apache Jmeter` [93], was utilized to generate 1200 `HTTP POST` requests directed at the “add user” operation and 1200 `HTTP GET` requests directed at the “get user” operation, both with the longest critical paths.

We considered four evaluation scenarios when carrying out our experiments. First, the evaluation of the application was performed without the use of any instrumentation. In this particular scenario, the microservices were deployed with their original `Node.js` versions without any alterations, and the average response time was observed. `JMeter` was employed for the purpose of generating the user load. In the second scenario, the application was instrumented using Open Telemetry [127], and the average response time was also observed. In the third scenario, the application was deployed using the strategy proposed in [121], and the average response time was also observed. In the final scenario using our technique, the instrumented `Docker` images of `Node.js` were deployed alongside the application, and the response times were also observed.

In order to evaluate the effectiveness of the Rbinder technique [121], we implemented it in conjunction with our microservice application and proceeded to configure the different proxies accordingly. No system call activation was performed in our strategy ; only the `Node.js` Docker images were deployed alongside the application. The technique implemented did not necessitate changes to the application deployment procedure. One of the benefits of this approach is its ability to provide transparency in both the deployment and tracing processes. The analyses are characterized by transparency, as they can generate visual results without any involvement from the developer.

The process requires starting the `LTTng` tracer on every container to capture the application trace and then halting the tracer and automatically aggregating the traces. The microservices are then imported into the TC platform, where an analysis is executed and a visualization depicting the interactions among these microservices is generated.

5.8 Discussion

The outcomes of the experiments depicted in Figure 5.11a show that the response times of the application executed using our method incurred a low overhead on the system compared to the uninstrumented application. The average response time for the “get user” operation was 4.7 ms (standard deviation : 1.33) when no instrumentation was conducted, 5.1 ms (standard deviation : 1.18) when distributed tracing was employed, 5.35 ms (standard deviation : 1.05) when the Rbinder strategy was employed, and 5.07 ms (standard deviation : 1.23), when our strategy (Vnode) was employed.

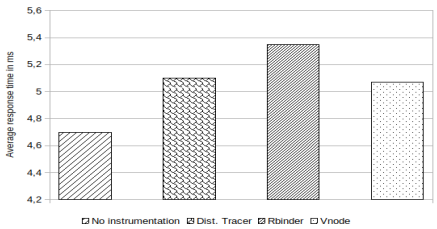
In the second scenario, experiments were conducted by initiating `POST` requests for the “user add” operation. The obtained response times are depicted in Figure 5.11b. It is clear that the supplementary cost associated with tracing, on average, was consistently of a comparable

magnitude, similar to the preceding scenario. The mean response time for the untraced program was 4.1 ms (standard deviation : 1.33), 4.29 ms (standard deviation : 1.37) when using distributed tracing, 4.6 ms (standard deviation : 1.46) when using Rbinder, and 4.3 ms with Vnode. An examination of the overhead resulting from tracing using our approach reveals that it is comparable to state-of-the-art approaches, incurring only a 5% cost on the traced system. Figure 5.11c illustrates the central processing unit (CPU) utilization in the scenario when the application was executed without any tracing. Comparing the aforementioned data with those in Figure 5.11d, which illustrates the CPU utilization while employing our proposed methodology, a slight increase in resource use was observed.

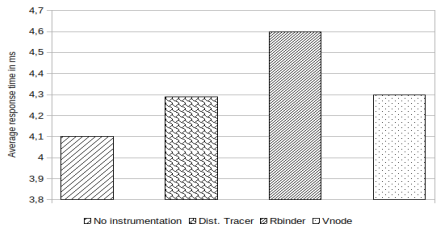
Given that the execution time of the analyses required to build the model is strongly dependent on the number of events contained in the trace, it is important to measure the time required on various trace file sizes. Figure 5.11e,f, respectively, depict the CPU utilization and time to run the analysis according to the trace sizes. For different sizes of the trace, the needed time is presented. The experiments show that our technique is optimized to run with very good times. This is due to the nature of the data structure (SHT) used, enabling access in logarithmic time.

The LTTng tracer had a minimal impact on the system, as the activation of a tracing point incurred a nanosecond-level overhead. LTTng is the fastest software tracer available [68]. The evaluation of the collection and analysis infrastructure using our approach demonstrates all the advantages it offers. Firstly, the induced overhead was relatively low on the system in terms of CPU and memory usage. In the first case, this can be justified by the tracer used and the optimization of the inserted tracepoints. In the second case, memory usage was not significantly impacted despite the size of the trace files, which can contain millions and billions of events, simply because the data structure constructed was disk-based. It depends primarily on disk write speed, especially during model construction.

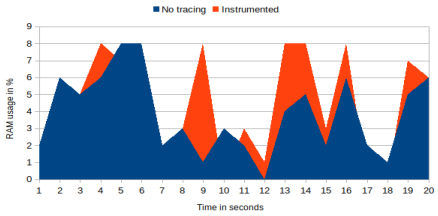
In the case of distributed tracing, the tracing overhead is acceptable, but using this strategy requires an understanding of the application's source code for instrumentation. This involves additional human costs and alters the application's behavior. To spare developers from these instrumentation efforts, an approach based on a "proxy envoy," called Rbinder, has been proposed. However, while it offers the advantage of transparently tracing the application, it still requires developers to invest significant time in setting up the collection's infrastructure, especially in configuring the proxies. In contrast to these approaches, our approach enables transparent tracing without any developer intervention. Developers are not required to establish the collection's infrastructure. Only the images of `Node.js` containers are replaced with instrumented images and deployed seamlessly, without modifying the development



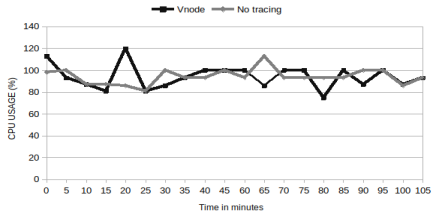
(a) Averaged Response time for microservice operation "user get"



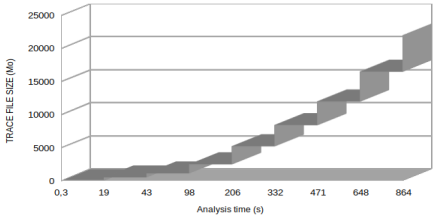
(b) Averaged Response time for microservice operation "user add".



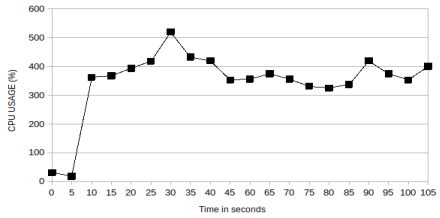
(c) RAM usage in no tracing and Vnode scenarios.



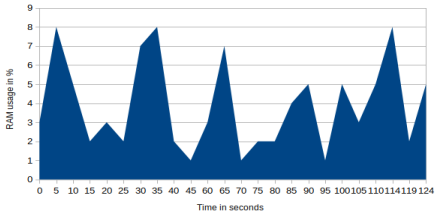
(d) CPU usage in no tracing and Vnode scenarios.



(e) Trace size versus analysis time.



(f) CPU usage when running analyses.



(g) RAM usage when running analyses.

FIGURE 5.11 Experiment results and impact on physical resources

pipeline procedures.

5.9 Conclusion

This work presented a new method for tracing Node.js microservice architectures. It emphasized the importance of tracing transparency so as to reduce the time spent on the performance analysis and validation phases of microservices. By adopting a tracing paradigm based on context reconstruction through `Node.js` virtual machine instrumentation, a specific algorithm has been developed for various multi-layer metrics correlations. In this case, the burden imposed by the instrumentation phase to developers can be avoided. The presented approach not only allows the transparent tracing of microservices, but it also provides a framework for uncovering the execution architecture of microservice.

Our approach could be improved by extending it to the support of other communication protocols, such as websockets.

Another potential avenue for further exploration involves doing an in-depth investigation into the root cause of performance issues, which our approach enables to pinpoint for the purpose of conducting a root cause analysis.

CHAPITRE 6 ARTICLE 3 : ADVANCED STRATEGIES FOR PRECISE AND TRANSPARENT DEBUGGING OF PERFORMANCE ISSUES IN IN-MEMORY DATA STORE-BASED MICROSERVICES

Auteurs : Hervé Mbikayi Kabamba, Matthew Khouzam et Michel Dagenais.

Soumis aux Transactions on Cloud Computing

Date : le 20 Novembre 2023

6.1 Abstract

The rise of microservice architectures has revolutionized application design, fostering adaptability and resilience. These architectures facilitate scaling and encourage collaborative efforts among specialized teams, streamlining deployment and maintenance. Critical to this ecosystem is the demand for low latency, prompting the adoption of cloud-based structures and in-memory data storage. This shift optimizes data access times, supplanting direct disk access and driving the adoption of non-relational databases.

Despite their benefits, microservice architectures present challenges in system performance and debugging, particularly as complexity grows. Performance issues can readily cascade through components, jeopardizing user satisfaction and service quality. Existing monitoring approaches often require code instrumentation, demanding extensive developer involvement. Recent strategies like proxies and service meshes aim to enhance tracing transparency, but introduce added configuration complexities.

Our innovative solution introduces a new framework that transparently integrates heterogeneous microservices, enabling the creation of tailored tools for fine-grained performance debugging, especially for in-memory data store-based microservices. This approach leverages transparent user-level tracing, employing a two-level abstraction analysis model to pinpoint key performance influencers. It harnesses system tracing and advanced analysis to provide visualization tools for identifying intricate performance issues. In a performance-centric landscape, this approach offers a promising solution to ensure peak efficiency and reliability for in-memory data store-based cloud applications.

6.2 Introduction

The advent of microservice architectures has changed the paradigm with which applications were once designed to adopt a much more resilient architecture. These architectures incorporate a simplistic scaling dimension that allows for adequate adjustments to the system, without disrupting its operation. They have encouraged collaboration among different teams responsible for specific components of the system, facilitating deployment, development, and maintenance.

A critical requirement, for the overwhelming majority of applications functioning within this ecosystem, is minimal latency [128]. Presently, the user experience has been substantially enhanced as a result of the swift progressions in technology. The utilization of cloud-based architectures has necessitated the placement of data storage in memory, due to the latency restrictions imposed by applications. This paradigm has significantly decreased data access times, due to the fact that data is now accessible at the memory level, rather than directly from the disk. In addition, the emergence of Big Data and Cloud Computing has prompted numerous organizations to forgo relational databases in favor of non-relational ones [129].

While these architectures have brought greater flexibility to the management and development of computer systems, they have also raised numerous challenges. Among them, system performance remains a crucial element, while performance debugging increases in complexity, as explained in [79].

Common approaches to performance issue debugging typically employ application instrumentation methods to collect execution traces. A notable challenge with these approaches is the necessity to modify the application source code to insert tracepoints at specific locations, often resulting in altered system behavior. This method has become widely prevalent in the microservices ecosystem, particularly with the use of distributed tracing.

Despite the advantages offered by these approaches, they have limitations in terms of granularity. Specifically, distributed tracing instrumentation allows for tracepoints placement only on microservice components, neglecting the backends connected to them. The storage component plays a crucial role in microservice architectures, as it facilitates data persistence or access, and can be a significant source of performance degradation.

High-performance architectures typically incorporate in-memory databases for rapid data access, messaging services, and even caching services. Usually, requests to these systems are instrumented at the microservice level, to measure latency. However, when these systems are the root cause of performance issues, distributed tracing renders them as black boxes, making it virtually impossible to trace back to the internal source of the problem.

Debugging such issues requires direct instrumentation of these systems, necessitating significant advancements in source code understanding and trace analysis development. Methods that enable developers to transparently trace microservices and various backends, particularly those providing the storage dimension to the architecture, are crucial. These methods help reduce costs and facilitate the debugging of performance issues in microservices.

Menasce [126], and Luk et al. [41], along with other scholars, have put forth methods to address the typically high costs associated with instrumentation efforts.

Similarly, recent approaches have employed proxies that establish an intermediary layer within the traced system [121]. This intermediary layer intercepts system calls generated during communications between subsystems at the lowest level.

We propose a novel approach to performance analysis in microservice architectures that incorporates the storage dimension. This approach aims to achieve two primary objectives : First, it involves instrumenting the in-memory data-store, Redis in our case, to develop an analysis that precisely debugs internal performance issues. Second, it offers a method for seamlessly integrating various analyses into a unified system, providing a comprehensive view of the whole architecture, and facilitating the identification of bottlenecks. Recognizing these bottlenecks enables the use of granular analyses previously developed to pinpoint these issues.

The integration is exemplified using two distinct analyses. The first, derived from previous work [130], involves the transparent tracing of microservices developed in `Node.js` [5]. This process requires no additional instrumentation effort from the developer. Instead, the developer merely deploys the instrumented image of the `Node.js` environment with the application in a container, then applies automated analysis to the obtained trace to visualize microservice interactions.

The second analysis results from the direct instrumentation of `Redis`, chosen for its popularity and its encompassing functionalities as a cache, database, and messaging service. The same method could similarly be applied to other in-memory data-store systems.

Our approach enables transparent tracing in these two environments, reconstructing interactions through automated trace analyses, and visualizing interactions between different components. In other words, it facilitates a comprehensive and granular performance analysis of the entire system.

Furthermore, this approach can be expanded to other environments, allowing the integration of environments beyond `Node.js` and `Redis`.

The following are the primary contributions of this paper : (1) We introduce an integrated

framework for performance debugging in microservice architectures. (2) We propose an abstraction model that captures actors likely to influence in-memory data-store performance, and demonstrate how such a model can lead to fine-grained performance debugging. This model can be applied to other systems. (3) We demonstrate the relevance and effectiveness of our framework through practical and complex use cases.

The remainder of the paper is organized as follows : Section 6.3 discusses related work on the subject. Section 6.4 describes the proposed framework. Section 6.5 describes the context of our experiments, and present some use cases to support the relevance of our tool. In Section 6.5.3, an evaluation of the solution is performed, while in Section 6.6 a conclusion on the work is drawn.

6.3 Related Work

Developers spend the majority of their time debugging problems in applications as explained in [131].

Seer, an online debugger that predicts quality of service violations in cloud applications was introduced in [62]. The study is based on a microservices infrastructure that uses Memcached, which has similarities to `Redis` as an in-memory database.

Debugging performance issues with Seer, involves an instrumentation step of microservices. It is also performed in the Memcached data store, mainly at the polling functions and various network interface queues. Seer is capable of predicting quality of service violations based on a model generated using deep learning on upstream traces. Instrumenting the functions that manage packet queuing, at the data store level, allows for identifying bottlenecks when they involve Memcached.

Merino and Otero [132] proposed a debugging approach that utilizes C/R (Checkpoint/Restore) techniques with CRIU (Checkpoint/Restore in Userspace). This method aims to replicate faulty environments, mitigating the iterative nature of the debugging process. Their approach extends the checkpoint capabilities of C/R engines based on ptrace, providing support for containerized processes attached to the debugger, all without the need for code instrumentation or custom operating system kernels.

OmniTable, a query model used as a debugging approach was proposed in[133]. It aims to reduce programming complexity and minimize the performance overhead associated with debugging, while preserving for developers an unrestricted access to the execution state. They demonstrated that this query model simplifies the debugging query formulation process, compared to existing tools. The approach was tested through a series of case studies involving

well-known open-source software such as `Redis`. Their approach produces a model of the system states that can be replayed to understand its behavior. However, an issue arises with the size of the tree structure that needs to be stored in memory. It relies on logging techniques to capture the execution history associated with each OmniTable, and then generates the instrumentation and traces needed for a new replay execution. It has been applied to a specific `Redis` use case to identify a performance issue. However, this approach is based on a model that requires significant resources.

Whittaker et al. [134], proposed Wat-provenance to leverage a data provenance approach to debug performance issues. It takes into account the causality dimension. A state machine is used to describe why a particular data is produced as an output. The approach was applied in a distributed systems context for root causes inference. The distributed system infrastructure included `Redis` servers in their experiments. However, such an approach can only infer external communication behavior and bottleneck, but cannot be applied to debug the internal `Redis` components functioning. Arafa et al. [135], proposed a dynamic instrumentation approach for performance debugging. Extraction of runtime information was applied to comprehend system behavior. The approach was tested on `Redis`.

Santana et al. [121], introduced an innovative method for achieving transparent tracing, which harnesses the kernel of the operating system to capture system calls related to communication among microservices. They advocated the use of a proxy that adds a neutral layer to the microservice, enabling the interception of its interactions and the correlation of information to infer causality associated with different requests. Intercepting system calls ensures transparency in application tracing, but it entails developer responsibility for configuring the infrastructure.

For service discovery, Wassermann and Emmerich [122], employed dependency structures extracted from documentation using statistical methods. In addressing fault detection, Chen et al. [123], used middleware instrumentation to log the specific components handling individual requests. While these approaches achieved a certain level of transparency, they demanded extensive dedication of the developer time for library instrumentation.

Distributed tracers such as Dapper [39] and X-trace [52], are capable of tracing the entire request lifecycle, revealing its flow, and assisting in diagnosing issues throughout execution. They rely on mechanisms for injecting a trace context to reconstruct the trace nesting structure. However, these approaches require an initial instrumentation phase, for the application to activate the collection mechanism.

Tracing strategies for request paths were also addressed in [124] using heuristics. Aguilera et al. [125] proposed algorithms for diagnosing request causality. Both of these approaches offer

a certain degree of transparency but rely on middleware instrumentation.

Unlike most of these approaches, our framework refrains from injecting trace context into the request ; instead, it capitalizes on the internal mechanism of host environments, to reconstruct the request path.

6.4 Proposed Framework

6.4.1 Integrated Framework Architecture

Figure 6.1 shows the overall architecture of the integration framework that our approach proposes. The various microservice environments, in their heterogeneity, are considered building blocks for constructing a performance analysis framework that takes into account the entirety of the system.

As shown in Figure 6.1, the internal `Node.js` virtual machine is instrumented with a tracer that produces traces in Common Tracing File (CTF) format. For each environment, the native contexts and network communication functions are instrumented according to a formal event definition specification. The goal is to ensure traceability and deployment transparency for the developer. In this specific case, microservices deployed with instrumented Java

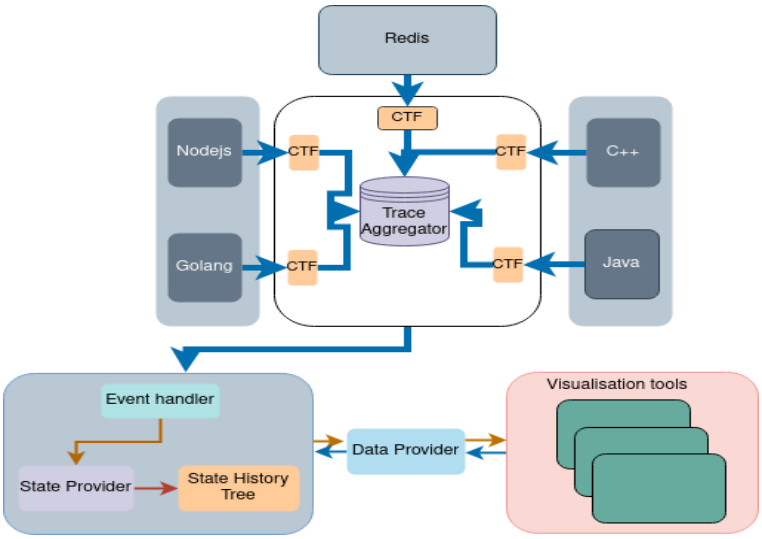


FIGURE 6.1 Performance Analysis Integration Framework architecture

or `Node.js` container images do not require any developer intervention to configure the collection infrastructure. The effort associated with altering application source code through instrumentation is not needed. Likewise, as depicted in Figure 6.1, an instrumented `Redis` image being part of the system is also included.

The proposed performance analysis framework aims to seamlessly integrate the various heterogeneous microservice components, enabling the developer to trace their system without changing the procedures in the application development and deployment pipeline.

The traces obtained in CTF format for each node are automatically sent to a trace aggregator. The synchronized traces obtained are loaded into Trace Compass (TC)[98] as an *experiment* (a collection of related traces). TC extensibility allows for the development of analysis modules, enabling the creation of interactive tools that facilitate the understanding of the system operation.

In Figure 6.1, under TC, the trace is read, and an event handler is used to extract various metrics and attributes from it. A trace provider constructs a powerful, efficient data structure to store event attributes. For each system state recorded by a particular trace event, its attributes and timestamp are used to build a data structure containing a history of the system states. Views, developed as extensions, allow querying the on-disk data structure through a data provider.

6.4.2 Redis Cluster Collection Architecture

Figure 6.2 illustrates the trace collection and analysis architecture. In this depiction of a `Redis`-based infrastructure, trace collection occurs through static instrumentation[136, 137], during the initial phase of performance analysis. Then, the Linux Trace Toolkit next generation (LTTng) tracer [68], is activated in each of the nodes. These nodes can be containers, virtual machines, or physical hosts. Trace points are inserted at specific locations in the `Redis` code to collect the necessary events for analysis, aimed at debugging and diagnosing performance issues. Within this collection infrastructure, the `LTTng` tracer generates events and stores them inside a CTF file on each node. Subsequently, these individual files are automatically aggregated by a trace aggregator, to obtain synchronized trace files. In the second step, which is the analysis phase, the traces are loaded as an experiment into Trace TC. In the latter, complex and automated analyses defined by algorithms are executed to identify specific events, correlate their metrics, and produce visualization tools focused on the ongoing performance analysis. Conducting the analyses involves the creation of an advanced execution model, leveraging state-system technology and relying on attributes extracted from trace data. When reading the trace, the State History Tree (SHT), a highly efficient data structure, comes into play for the storage of the extracted attributes, along with additional analysis-related data. The TC framework supports the development of these data structures, benefiting from an optimization technique that enables rapid model queries in logarithmic time. TC boasts a range of capabilities for organizing traces, such as filtering and abstraction.

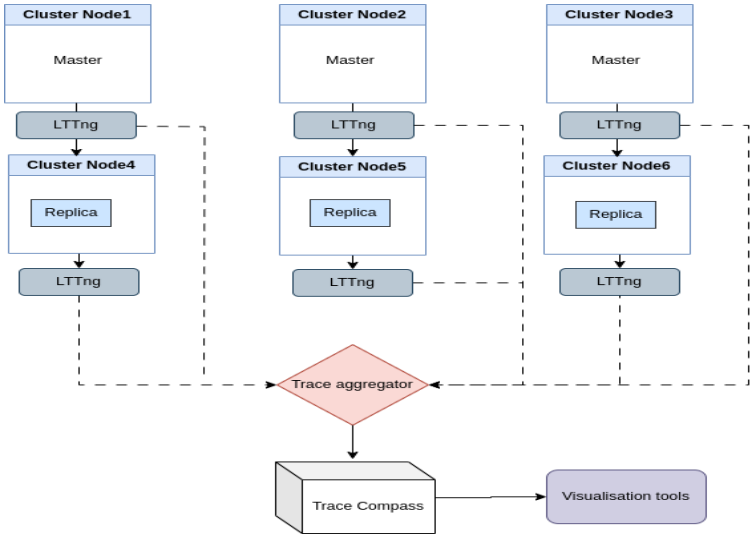


FIGURE 6.2 Performance Analysis Integration Framework architecture

6.4.3 Redis resources-driven modelling

The abstraction stage aims to develop a model that encapsulates the behavior of resources that could impact system performance. Specifically, five entities have been identified : `Connections, Requests, Bus, Threads, and Event-loop`. These entities serve as the entry points to the tree representing the model. The tree internal branches link to resources that define the states of different entities based on a timestamp. For instance, as depicted in Figure 6.3, the `Connections` entity is associated with three resources : `Memory, Data structure`, and `Objects`. These elements, linked to the `Connections` entity, abstract the resources potentially affecting system performance, within the context of network connections. For example, when a connection is received on a socket, `Redis` treats it as an incoming event, initially queued before being processed by the `Redis` event loop.

The queue length is directly correlated with system performance [63, 138, 139]. Memory-related attributes of the connection are attached to the `Memory` resource. A connection could be established by a client, be an internal link between two nodes in the cluster, or a link to a replica, and this information is attached to the `Type` resource state.

The `Requests` entity abstracts the resources potentially impacting performance in the context of requests execution. It includes three objects : `Data structure, type, and connection`. The `Data structure` object provides insights into the various data structures storing requests within the `Redis` infrastructure. A request might pass through several data structures, for example during message publication. The `type` object identifies the request type, which could be either *read* or *write*. The `Connection` resource links a `Connection` object to the request.

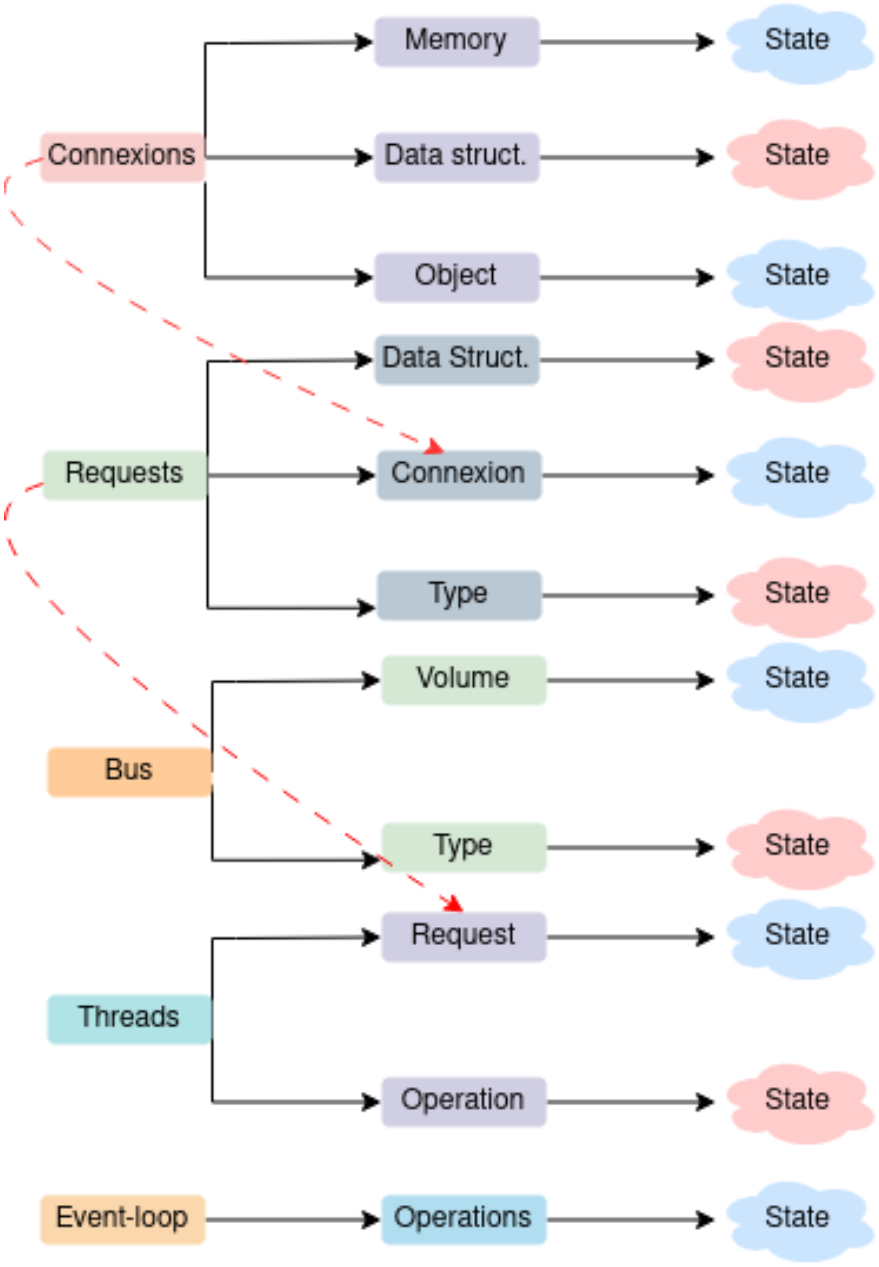


FIGURE 6.3 Model structure abstraction

The `Bus` encapsulates the states of the network link between nodes in a `Redis` cluster. The `Volume` resource abstracts the volume of data transmitted over the Bus per second or millisecond. `Type` discriminates the nature of communication within the Bus, which could be `Primary` to `Replica` or `Node` to `Node`.

Regarding the `Threads` entity, it abstracts the various active threads involved in executing a particular operation. The `Request` resource identifies the request associated with the

operation. The `Operation` resource attaches an ongoing operation executed by the thread. Therefore, information about the thread, the request, and the concerned operation can be obtained.

The abstraction of the event loop functioning is represented by the `Event-loop` entity, encompassing all operations necessary for monitoring its states. The model construction involves creating a tree-like data structure replicating the model depicted in Figure 6.3. However, this phase occurs during the analysis of collected traces, which can contain a vast amount of events, millions or even billions. Efficient tools are necessary to support such analyses.

Trace Compass [98], offers an ideal framework for building scalable, efficient data structures capable of handling large data volumes. This disk-based data structure, known as State History Tree (SHT) [78], facilitates constructing the referenced model in Figure 6.3, and querying it for data visualization with logarithmic complexity. It also allows abstraction of the system state at a specific moment, and understanding the system various states over a time interval.

The model is populated using event attributes, and queries on the SHT return the system state at a precise timestamp. The states are the different values assigned to the model resource statuses, as presented in Figure 6.3. In other terms, the model can be viewed as an automaton, and its various states are preserved in the SHT with their timestamps.

6.4.4 Trace analysis

This section outlines the various analyses conducted on the trace to construct the model and generate the necessary visualizations for debugging performance issues in microservice architectures. Two types of analyses have been developed. The first analysis aims to efficiently traverse the trace to populate the model by creating a high-performance and scalable tree data structure. This process relies on different events from the trace and their attributes. The resulting model can be queried to understand Redis operations and to debug performance issues in a granular manner.

The second analysis focuses on integrating microservice components based on well-defined events and their attributes. It facilitates linking the analysis, obtained during the model construction, with other previously developed analyses. In the context of this paper, we integrate analysis [130], developed under Node.js. This analysis enables the transparent tracing of microservice architectures built with Node.js, without the need for application instrumentation. The integration is conducted transparently, allowing the connection of microservice executions with the Redis analysis, resulting in a comprehensive debugging approach.

TABLEAU 6.1 A subset of tracepoints monitored by our tool

Tracepoints	Description
`start_read_client_query`	Triggered when `Redis` reads an incoming connection from the queue.
`end_read_client_query`	Triggered when `Redis` when `Redis` exits connection reading
`write_to_client_start`	Triggered when `Redis` starts writing response to a connected client
`write_to_end_start`	Triggered when `Redis` ends writing response to a connected client
`ssl_read`	Triggered when `Redis` reads SSL connection
`free_client`	Triggered when `Redis` frees client resources
`cluster_read`	Triggered when `Redis` reads message from cluster bus
`cluster_process_packet`	Triggered when `Redis` has finished reading packet from the bus
`call_command_start`	Triggered when `Redis` executes a command
`call_command_end`	Triggered when `Redis` has finished executing command
`add_file_event`	Triggered when `Redis` enqueues a new event
`delete_file_event`	Triggered when `Redis` remove an event after processing

Analysis model construction

This subsection describes how events are handled in a trace, to build a model as defined in Section 6.4.3.

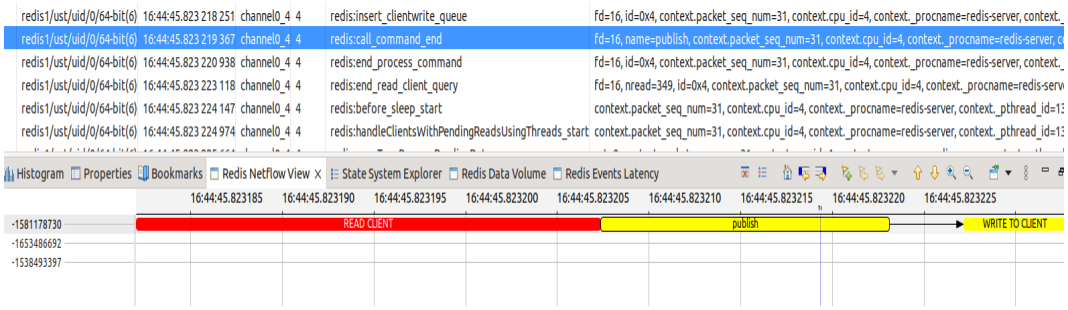


FIGURE 6.4 Partial view of the publish request life cycle. The top view shows the trace events. The bottom view shows the Netflow view used to track request life cycles. On the bottom left the TID are presented.

Consider a `Redis` client that transmits a request to publish particular data on a channel, to exemplify one facet on how the analysis operates. Figure 6.4 illustrates the outcomes of the analysis performed on the trace.

To run a command, the client establishes a connection with the server. The polling phase is executed on the Redis server side, during which the EL is notified of an incoming event on

the socket. Following this, the `start_read_client_query` tracepoint is triggered.

During the trace analysis, the algorithm extracts the FD, the timestamp, and the thread ID immediately upon encountering this event. Subsequently, it stores these values in the SHT. In the model, `Connection` and `Requests` branches are instantiated.

The FD is added to the `data structure` resource in the `Connexion` branch in the SHT, at that timestamp. Based on the FD, in the connection context, the `Memory` resource is populated with the value of the consumed memory, as determined by the event attributes.

The `Type` attribute of the corresponding resource is set to "client" in order to denote that the connection pertains to a client link. The FD is also added to the `Connection` resource within the `Requests` branch in the SHT, along with the timestamp extracted from the attributes.

Prior to encountering the `end_read_client_query` event, indicating that the incoming event has been read from the connection, the algorithm watches the `call_command_start` event. The latter enables the decoding of the pending command and the invocation of the `Redis` function that is accountable for carrying out the execution. The timestamp and the command name are obtained by extracting the attributes from this event.

The automaton then enters the *"publish"* state, as depicted in Figure 6.4. As the analysis progresses, it encounters the serial `add_file_event` event, prior to reaching the `call_command_end` event. The `add_file_event` tracepoint instruments the queuing process of the event, for its execution in the subsequent phase of the EL.

The analysis then updates the `Data structure` resource state in the `Requests` branch, by adding the new request with it timestamp. Given the specific context in which the command to be executed is *"publish*, the value *"write"* is appended to the `Type` resource, to signify that the request pertains to a write operation.

The `call_command_end` event, which is matched with the initial `call_command_start` event based on the FD, is used by the analysis to indicate the end of the command decoding and execution. The `end_read_client_query` event, which is matched with the preceding `start_read_client_query` based on the FD, marks the end of the connection reading.

Every alteration, made to the state of a resource in the model, is reflected as a state transition in the automaton. Throughout the process of the model construction, the system undergoes several transitions. A timestamp is assigned to each state, which matches a distinct event recorded in the trace. The SHT retains a history of the sequence of system states and transitions. Hence, it is possible to ascertain the state of the system at a specified timestamp by querying the SHT.

Upon reading from the connection, `Redis` acknowledges the client.

The tracepoint write_to_client_start is triggered whenever a message is transmitted from the server to the client. As the connection is not yet withdrawn from the context, the analysis modifies the system state to point out the beginning of the client response. The FD, obtained from the `write_to_client_start` event, is used to identify the SHT branch position that corresponds to the initial client connection.

Figure 6.4 depicts the system move to the *"Write to client"* state. The algorithm uses the `write_to_client_end` event to signify the end of the response transmission to the client. The FD is used to match the two events. Finally, the constructed data structure is used by the state system to provide a snapshot of the system state at a specific timestamp.

The same example can be used to show how the `Bus` branch is populated during the analysis. The bus defines the state of network communication within a `Redis` cluster, and is populated by the `cluster_send` event, that is triggered when Redis broadcasts the publish message to the nodes on cluster. In this case, the status of the `Volume` resource in the SHT takes the value of the calculated volume of data sent metric. The `Type` resource is appended with the value of the communication type, in this case "broadcast".

All the branches of the SHT are populated based on event attributes. The FD is a discriminator for different "Connections" and "Requests" objects. The `Data structure` resources store all the objects for a given branch. For instance, it stores all the connections represented by the FDs, in the case of the `Connections` branch.

TC extensibility enables the development of extension modules that leverage the SHT as the data source for a variety of views. These extensions result in visual and interactive tools that facilitate the debugging of performance issues in `Redis`.

Integration Analysis

The integration process assembles various analyses to provide a comprehensive and detailed view of the system operation, facilitating performance analysis with a high level of granularity. For instance, this integration enables the visualization of interactions between different components of the microservice architecture as spans. However, it is crucial to be able to pinpoint the cause of a bottleneck when one is detected, whether due to a bug, a blocked process, or another issue. Delving deeper into a problem, at a finer granularity, is necessary to trace back to the root cause.

When a bottleneck is identified within Redis, the Redis analysis can be used to accurately determine the cause. Similarly, if the bottleneck is located within a Node.js microservice, the

granular analysis of that environment can be leveraged for precise identification of the cause. The same applies to heterogeneous microservices developed with other technologies.

Integration offers a unified view of the system by observing interactions between microservices, while granular analyses of each environment enable tracing back to the root cause of a bottleneck.

In tracing components of the microservice architecture, events such as `http_client_request`, `http_server_receive`, `http_server_response`, and `http_client_response` are generated with each microservice interaction. These events capture the moments when a microservice sends a request to another, receives a request, sends back a response, and when the response is received by the sender, respectively.

The analysis relies on the attributes of these events to connect components. Attributes include source and destination addresses, source and destination port numbers, invoked services, and file descriptor numbers.

This information allows for the reconstruction and alignment of different interactions, connecting them with analyses developed for various environments, to seamlessly integrate them into a unified analysis tool.

While we used two environments (Node.js and Redis) in our case, this approach can be extended to various environments, such as Java, Golang, and others.

6.5 Experiments

This section introduces the context of our experiments to validate our approach. We present the environment configuration for running the experiments. We then introduce practical and real use cases of performance issues encountered by the community. Then, we measure the overhead incurred by our approach. This measurement is crucial to determine its applicability.

6.5.1 Environment configuration

The experiments were conducted with `Node.js` versions 12.22.3 and 16.1.0 with `Redis` 7.0.11, on an I7 Core running Ubuntu 20.4 with 16 GB RAM.

6.5.2 Use cases

Multi-plateform Analysis Integration

In this use case, to demonstrate the efficiency and advantages of our integrated performance analysis approach, the application *MICROSERVICE-RESTful-Nodejs*[1] was deployed and extended to add additional features, such as message publishing for subscribers.

The microservices application is developed in `Node.js` and uses `Redis` as a cache. It was modified to include message broker functionalities. It consists of five microservices organized as follows :

1. The "user" microservice manages user-related requests within the system.
2. The "Order" microservice handles different orders made by users.
3. The "Gateway" microservice receives user requests and redirects them to the appropriate microservice.
4. The "Redis Gateway" microservice is contacted by other microservices looking to interact with Redis. It forwards the requests to the Redis server and returns responses to the relevant microservices.
5. The "MySQL Gateway" microservice is responsible for managing various requests with the MySQL database.

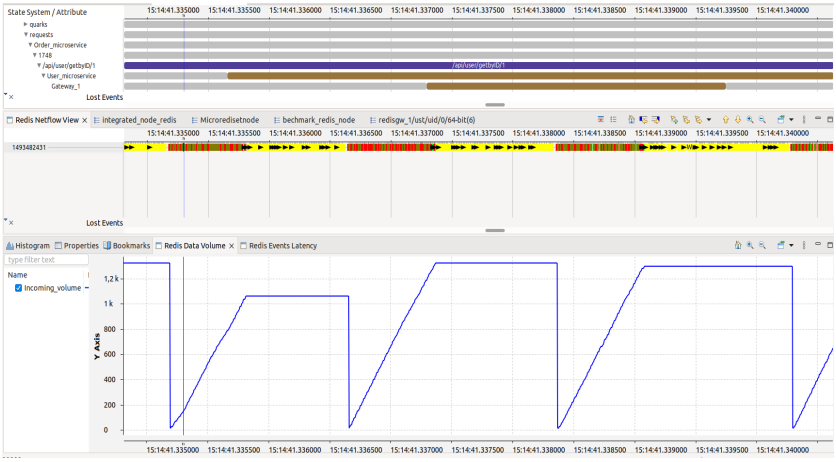


FIGURE 6.5 Constructed blocks from different analysis

To enable the assembly of blocks from the two different environments (Node.js and Redis) for performing granular performance analysis, the original `Redis` and `Node.js` images were replaced with instrumented images.

1. `https://github.com/tudtude/MICROSERVICE-RESTful-Nodejs`

In the case of `Node.js`, Kabamba et al. [130] proposed a low-overhead approach for transparent tracing for Node.js-based microservices. The reader is referred to their work for more details.

Running the application generates local traces in each node of the system. These traces are aggregated and loaded into TC. During the loading process, algorithms performing optimized analyses handle trace events, extract metrics from their attributes, and build a model on disk for each assembled environment.

The execution of the analyses produces two concurrent models under TC, one related to `Node.js` and the other to `Redis`. The extensions of the views generated by the various analyses allow them to be combined into a unified, integrated, and precise tool for performance analysis. The granularity is defined by each of the analyses developed for the different environments.

An overview of the outcomes of the performed analyses is presented in Figure 6.5. In the latter figure, the top perspective pertains to operations executed in `Node.js`. It can be seen how the integration analysis aligns all executions. The lower granular perspectives pertain to operations executed in `Redis`. Notably, these outcomes are acquired in a transparent manner, devoid of any involvement from the developers. The system deployment, tracing, and analysis are all carried out in a seamless fashion. The responsibility of the developer is to accurately determine the nature of the performance issue by using the interactive tools that are generated.

Leveraging Model Abstraction for Redis cluster request life-cycle tracking

The present use case sufficiently demonstrates one of the major features of our tool, which allows tracking the life cycle of a request through a Redis infrastructure. Available monitoring and performance tracking tools primarily rely on established metrics to detect performance issues in Redis. These metrics may include disk write speed, the number of queries, memory-related information, to name a few. However, in a Redis infrastructure, such as a cluster, it may be essential to trace the execution path of a request to precisely identify where and when a request might be blocked, if necessary. This requires correlating information from all nodes in the cluster, synchronizing them using efficient analytical methods, to establish the path a request follows and visualize its journey through various nodes until the end of its life cycle.

This capability potentially allows for the precise localization of a performance issue occurring

on the critical path of a request. To demonstrate this unique feature of our approach, we modified the Redis-bench client to enable it to send various types of commands, such as "publish" and "subscribe." The benchmark is then executed with 20,000 requests for each command (get, set, subscribe, publish). Figure 6.6 illustrates the flow of some requests processed in

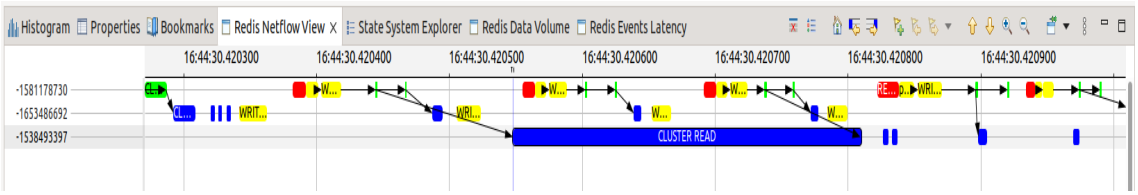


FIGURE 6.6 Redis cluster request flows

the Redis cluster. A performance issue can be observed, occurring during the reading of a message transmitted from one node to another. The read state is represented in the blue segment. When compared to various other requests, reading a message in the cluster takes less time.

By parallelizing the Redis Netflow view, which shows the life cycle of requests, and the Redis Data Volume view, a correlation is observed between the performance issue, and a significant increase in the volume of data leaving through the bus. See Figure 6.7. This directly suggests a relationship between the performance issue and the state of the queue at the network interface.

Such a view also allows visualizing the actual execution time of a command, how long the request takes on each node visited, and how much time it takes within the cluster before reaching the client.

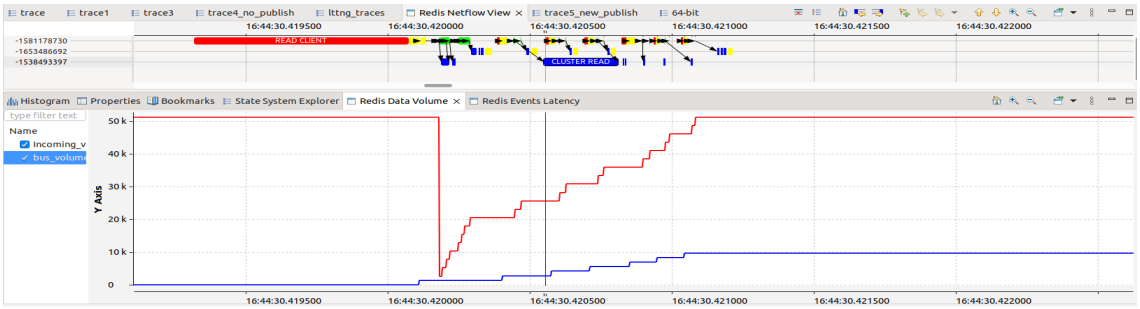


FIGURE 6.7 Request traveling on the cluster bus. Colors represent the operations performed on a timeline

Leveraging Model Abstraction in Redis Cluster Internal Communication

In this use case, we demonstrate the effectiveness of our tool in detecting performance issues related to the communication within a Redis cluster. This specific case identifies a performance problem raised on GitHub by developers. This use case clearly demonstrates how our model level of abstraction, in terms of the communication bus, can be applied to conduct a performance analysis under Redis. The bug described on GitHub, which was still unresolved, pertains to the context of "publish" requests sent to a Redis infrastructure consisting of a cluster with a Primary/Replica or Primary/Primary configuration. Initial experiments indicate, for instance, that when 10KB of data is published in a Redis cluster, subscribers indeed receive 10KB of data, but the data exchanged between the cluster nodes is ten times greater. The use of a network monitoring tool confirms that the volume of data leaving the network interface imposes a very high overhead on the cluster. To reproduce the performance

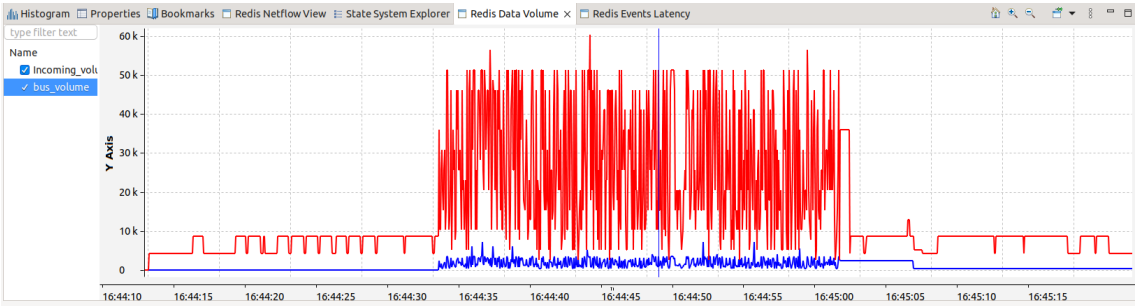


FIGURE 6.8 Redis cluster data volume traveling within. The bus is an abstraction of the communication link to multiple nodes

problem, a Redis cluster with 3 nodes was configured. The microservice application "nextjs-express-redis-microservice-architecture" was used, along with `Jmeter` [93], to generate users load for publishing messages to the cluster, with each client writing to a specific channel. The connection is established on Node 1. On Node 2, a client has subscribed to messages on the same channel, and is supposed to receive all the messages emitted by the publisher.

Figure 6.8 shows the result of the executed benchmark. The blue curve shows the progress of the volume of data entering the cluster over time, via the connection established by the client on Node 1. This same volume of data is received by the client on Node 2. However, there is a drastic increase in the volume of data transmitted within the cluster bus, as indicated by the red curve. Further investigation reveals that the overhead imposed on the data volume increases linearly with the number of nodes connected to the cluster. Two elements significantly impact the volume of data exiting the interface. First, Redis uses a specific protocol called Gossip to encode communications between the cluster nodes. The encapsulation of the data

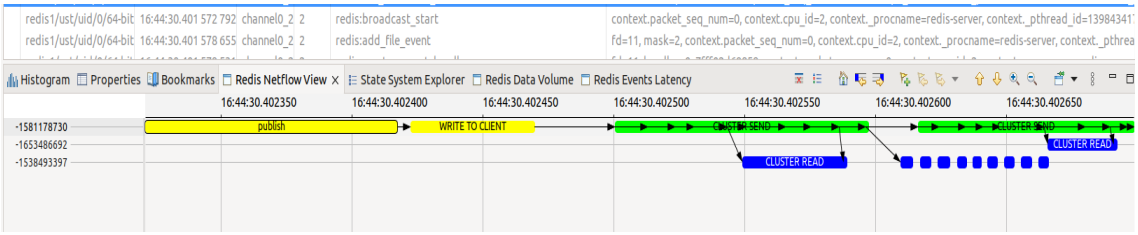


FIGURE 6.9 Redis Node is broadcasting the same message to the two other nodes

received at Node 1 is done using `Gossip`, which inherently introduces a very high overhead for small-sized data. The second element affecting the volume of data exiting the interface is that the communication on the bus is performed for each "publish" request in a broadcast manner. In other words, Node 1 broadcasts the same message to all nodes connected to the cluster, as depicted in Figure 6.9.

This use case reveals a Redis performance problem inherent to the Gossip protocol it employs. Resolving this issue requires a reevaluation of its communication mechanism within the cluster.

Leveraging Model Abstraction to Expose Resource Conflicts

This use case reveals a performance issue that occurs when sending a request containing a large item, in multiple parts, to `Redis`. The problem arises when `Redis` is compiled with `TLS` support and is run with multi-threading enabled for I/O operations.

To reproduce the performance problem, traffic is subsequently generated using `Memtier`[140], which supports TLS, to execute multiple concurrent requests on `Redis`. This results in the activation of threads to handle various I/O operations. During the benchmark execution, sending an isolated request with a large volume of data arriving in parts forces `OpenSSL` to have remaining bytes in the buffer. This automatically creates a performance problem that leads to a `Redis` crash. Our approach accurately exposed the sequence of events that leads

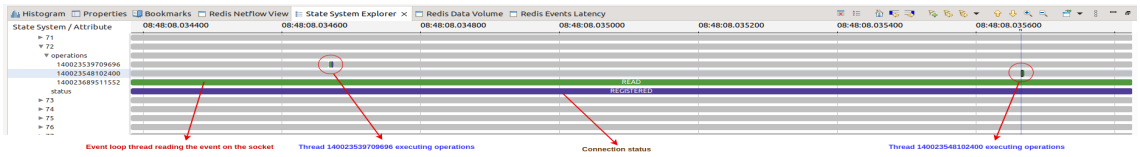


FIGURE 6.10 Persistent Model `Threads` branch view. Numbers on the left represent connections FDs.

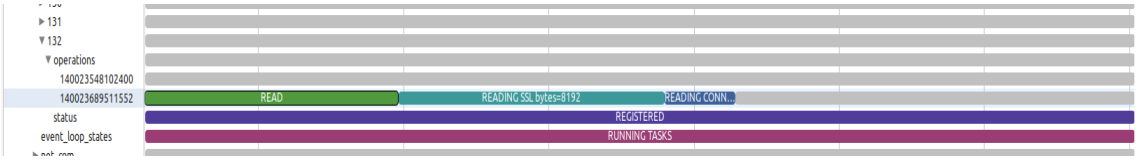


FIGURE 6.11 First reading loop of SSL connection

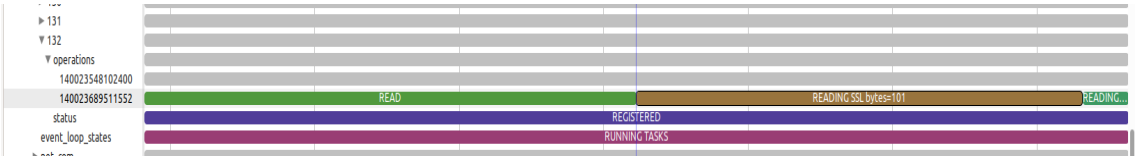


FIGURE 6.12 Second reading loop of SSL connection

to this performance problem. As can be observed in Figure 6.10, which shows a portion of the Threads branch of the model, the view allows the visualization of the history of different established connections stored in the SHT. The numbers on the right represent file descriptors. In `Redis`, events arrive under different connections where they are read and processed. At the end of their processing, the events become unreadable only by being removed from the context of the connection to which they belonged. This operation is instrumented by the tracepoint `delete_file_event`.

However, the connection remains open and can be reused for reading various other events, unless it is explicitly closed or due to abnormal behavior in `Redis` execution. By carefully observing Figure 6.10, it can be seen that, for the connection identified by number 72, the `Operations` attribute constitutes the logical resource associated, attached as per the model description. In other words, the resource is not physical but rather logical in nature in this instance. In this case, the various operations executed on this connection will be associated with the thread that performs them.

A particular thread that draws attention is the one responsible for running the event loop. As described in Section 6.3, the EL reads incoming events on the connection, decodes the command to be executed, and queues it for processing in its next phase. The *"Read"* state indicates the event reading by the thread, while the status described as *"Registered"* indicates the state of the connection handler.

Based on the foundations laid out above, a description of the unique features of our tool, in accurately pinpointing the root cause of the bug, is given. This particular use case illustrates a `Redis` bug concerning the `TLS` support with `OpenSSL`. When a big item is transmitted to `Redis` in numerous elements, `Redis` optimizes its operations by exclusively requesting the

FIGURE 6.13 Third reading loop of SSL connection

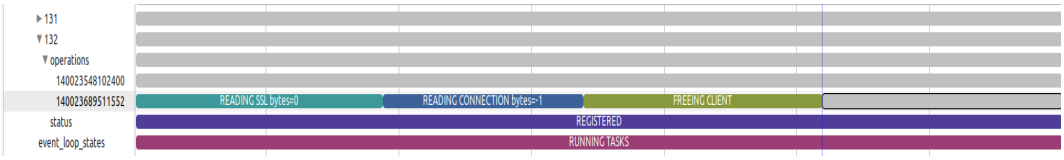


FIGURE 6.14 Fourth reading loop of SSL connection. No more data in the buffer, returns −1. Frees the connection.

FIGURE 6.15 Fifth reading loop of SSL connection by another thread. No pending data in buffer, double-frees the connection

data required to finish the reading. Consequently, it prevents the creation of extra copies. On occasion, this optimization strategy may results in OpenSSL buffers retaining bytes.

`Redis` features mechanisms for periodically determining whether pending data is present in the buffer to mark the connection as such. When the multi-threading reading option is activated, threads from the pool will execute the connection reading. Given that the connection is identified as having pending data in the buffer, the next phase of the EL will entail a thread reading the connection to retrieve the pending data. Once no further data is available for reading, the connection will no longer be identified as such. It can thus be reused in response to subsequent incoming events.

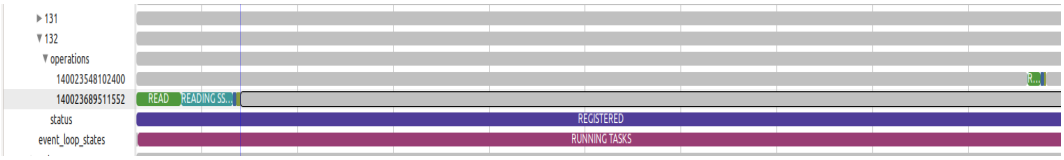


FIGURE 6.16 A zoom out on how the freeing operations are performed by the threads.

In the specific case of our experiment, the forced closure of the connection on FD 132 triggers

a new read event on the connection. Figure 6.11 depicts the reading of the client request by the EL. Since TLS support is enabled, the SSL handler calls the internal SSL reading function within `Redis`. A detailed look on Figure 6.11 captures the state of the thread transitioning to *"Reading SSL bytes=8192"*. A straightforward deduction provides more precision regarding the amount of data that `Redis` requested from `OpenSSL` during the connection reading. In this case, 8192 bytes are read. After the reading, the EL checks whether additional data is still pending in the buffer. If so, the connection is added to a list for processing in the *"RUNNING TASK"* phase. In Figure 6.11, the thread reading the connection is the main `Redis` process that manages the EL.

As soon as the EL enters the next *"RUNNING TASK"* state, the tracepoint handleClientWithPendingReadsUsingThreads activated. This indicates the involvement of the I/O threads due to the activation of this support during `Redis` launch. Figure 6.12 depicts the second iteration of reading the connection. In this specific phase, the concerned thread traverses the list and generates read events for each of the connections with pending data in the buffer. The same connection FD 132 is read again, as indicated by the thread status in Figure 6.12. This time, the EL reads 101 bytes in this second iteration.

The check for pending data in the buffer is performed once more. In this case, the connection is again added to the list of connections with pending data for processing in the buffer. In the subsequent phase of the loop, represented by the *"RUNNING TASK"* state, as shown in Figure 6.13, the thread once again goes through the list and generates read events on each connection present. This time, 18 bytes are read. In the fourth iteration, Figure 6.14, since the quantity of data is not greater than zero, indicating that there is no more data pending in the buffer, the resources associated with the client represented by the connection are released. This can be observed in Figure 6.14 under the *"FREEING CLIENT"* state.

The connection is then removed from the list of connections with pending data in the buffer. However, it is not removed from the list of connections requiring the activation of threads for I/O processing. With the resources of this connection now released, the FD appears as readable. In the next phase of the EL (RUNNING TASK), given that there is no more pending data in the buffer for connection FD 132, the activation of the I/O context, allowing for thread-based processing delegation, is not triggered, as there is no more data waiting to be read.

In the next cycle of the EL, since the connection is still readable, a read event is generated. This can be observed in Figure 6.15. This time, a different thread comes into play and performs the reading on the connection, adding it to the list of connections requiring the activation of the thread context for I/O processing. With careful attention, this operation

adds the same connection a second time to the thread I/O processing list.

The consequence is that when there is no more data pending in the buffer, the thread frees the client connection for a second time, leading to the crash of `Redis`. A focused view of those operations is depicted by Figure 6.16. In response to this unexpected behavior, the connection is automatically closed, as depicted by Figure 6.17, and the sequence of events results in a system crash.

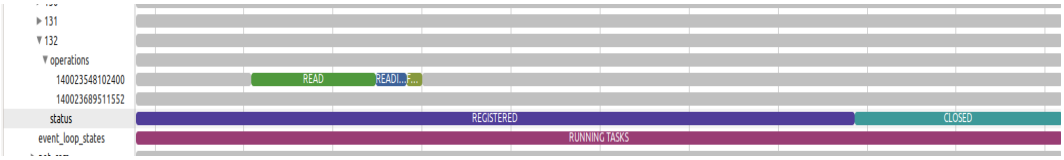


FIGURE 6.17 Closing the connection after encountering unexpected behaviour.

This problem exposes a synchronization flaw in `Redis` 7. Furthermore, we extended the experiment by introducing delays in the input/output thread reading operation, using a timer to simulate contention during the execution of the thread read callback. In some cases, this led to a race condition, detected by our tool, clearly showing that, in the same context, two different threads accessed the same connection, resulting in abnormal system behavior.

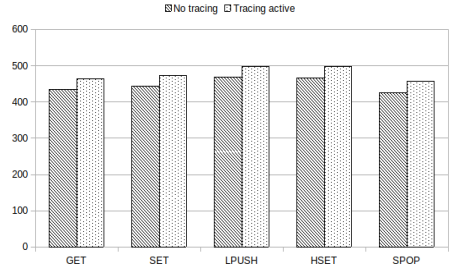
6.5.3 Evaluation

Latency overhead

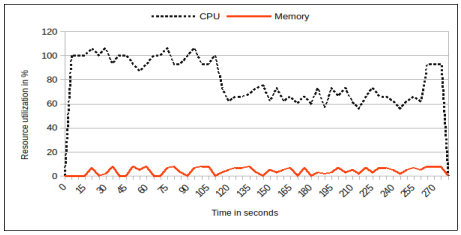
The average execution time of various commands issued by the client is illustrated in Figure 6.18(a). Redis-benchmark was utilized to conduct the investigation. To assess the effectiveness of our solution in tracing, one million queries were sent to the server for each command on a local computer. A comparison between the execution of commands with and without tracing enabled reveals a disparity in the average execution time (see Figure 6.18(a)). The overhead incurred by tracing is approximately 7%.

The fixed minimum guaranteed duration for processing the tracepoint by `LTTng` is the cause of this. However, the investigation is conducted directly on the internal processing of `Redis` and not in response to a request from a network service in a microservices context. `LTTng` is presently considered to be the world fastest tracer and imposes a minimal amount of system overhead.

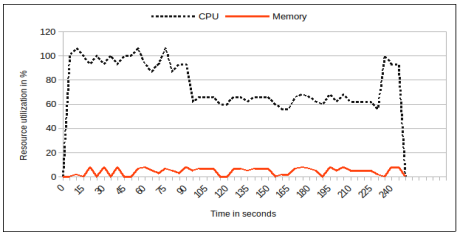
`Redis` interacts with modules or external applications in the network, as part of its normal execution, to provide a highly available infrastructure. The fixed overhead introduced by tracing with our approach, which is approximately 70 microseconds in a configuration of



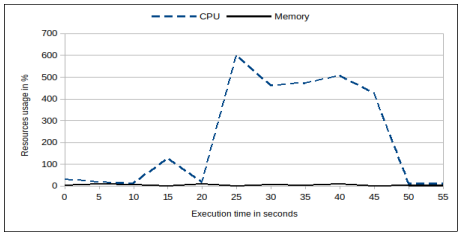
(a) Latency in microseconds with tracing activated and with no tracing.



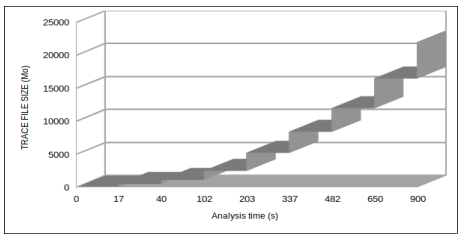
(b) Resources utilisation with tracing activated



(c) Resources utilisation with no tracing.



(d) Analysis resources utilisation



(e) Analysis running time in second vs trace file sizes

FIGURE 6.18 Analysis resources utilisation

concurrent command processing issued on 100 open connections, is absorbed by the communication time of microservice infrastructures.

Resources Consumption

The resource consumption of the executed benchmark is illustrated in Figure 6.18(b) and Figure 6.18(c), corresponding to its usage with and without trace activation. It is clear that, on average, the curve does not impose a substantial burden on the system. This is due to the optimization of the instrumentation of the application, and the type of tracer employed. Comparatively to the non-traced application, the CPU consumption curve for the traced application exhibits a comparable pattern. The memory consumption curves exhibit a nearly identical profile in both scenarios, indicating that our solution has minimal effect on resource utilization.

Trace Analysis

Trace analysis is a critical component of our methodology, as it enables applying algorithms that have been designed to discover correlations among metrics derived from event attributes. It facilitates the creation of a data structure that is stored on disk and that contains the built model. This data structure is then used by the visualization tools needed for studying system performance.

In Figure 6.18(d), the resources consumption incurred by running the analysis is depicted. To speed up the process, TC leverages a multi-threading approach, which explains the peaks in CPU consumption. However, memory is not much affected, since the designed data structure, storing the model data, is written to the disk. Figure 6.18(e) presents the time it takes to perform the analysis based on the trace file size. The analysis time is the duration necessary to parse a provided trace and construct the data structure that stores the model on disk. This data structure can then be queried by implemented views, in order to display correlated information, facilitating an interactive comprehension of the operation of the system.

6.6 Conclusion

The evolving landscape of applications, and the continuous growth of digital resources, have led to a paradigm shift in how content and information are accessed. Real-time interactions and minimal latency have become critical requirements for modern applications, leading to the adoption of cloud-based architectures and in-memory data storage solutions. `Redis`, as an

open-source, in-memory data storage system, has emerged as a key player in this ecosystem, offering exceptional performance, versatility, and low-latency data access.

Debugging performance issues, in distributed systems, being a complex endeavor, specialized tools, that can correlate events and metrics to debug performance issues, are required.

This paper introduced a novel methodology for the integrated debugging of `Redis`-based microservices performance issues. It leverages user-level tracing and a two-level abstraction analysis model, based on the identification of actors directly impacting `Redis` performance. The methodology employs two advanced tools for achieving its results. It leverages the state system technology and an SHT to store historical states of the system. Thus, stored metrics and information could be correlated, to provide a comprehensive solution for performance debugging.

Our presented technique offers a valuable resource for diagnosing and addressing performance issues in `Redis`, ensuring that applications can maintain optimal performance and deliver an excellent user experience in the current demanding digital landscape.

An interesting avenue for future work could be to extend the model, to consider different types of resources and actors for performance debugging of other system architectures.

CHAPITRE 7 DISCUSSION GENERALE

Dans ce chapitre, nous présentons les limites de notre solution avant de conclure la thèse avec nos recommandations.

L'analyse des performances des systèmes distribués passe par plusieurs phases, notamment celle de la collecte, suivie d'une étape d'analyse des données collectées, d'où le besoin d'instrumentation de l'application. Dans ce contexte, la vision globale de la solution est de fournir des outils d'analyse de performance qui limitent les efforts des développeurs dans la collecte des données et leurs analyses.

Cependant, vu du côté du fournisseur de la solution, les changements constants des versions des systèmes disponibles exigent des efforts pour les instrumenter de nouveau, afin de garantir l'utilisation permanente des outils par la communauté. Cela nécessite donc des efforts de maintenance continus. Des démarches pour l'intégration, par exemple des points de trace directement dans les versions livrées de Node.js, sont en cours. Cependant, les outils de traçage employés ont parfois des versions nouvelles qui tendent à briser les dépendances avec les points de trace disponibles dans les applications.

Il devient alors nécessaire de les instrumenter avec les nouvelles versions de librairies, même lorsque les points de trace font déjà partie des versions des systèmes disponibles.

En second lieu, le véritable défi est de trouver des compromis entre la granularité de l'analyse et sa généralisation. En effet, plus les analyses vont être granulaires, plus elles vont limiter leur capacité à détecter des problèmes divers. Le défi est au niveau du degré d'abstraction de l'analyse, car plus il est élevé, plus l'analyse masquera certains détails.

L'exemple typique est celui de l'emploi des traceurs distribués. Le niveau d'abstraction est très élevé du fait qu'ils sont mieux adaptés au traçage de haut niveau, tel que les interactions entre composants dans un système distribué. De ce fait, ils peuvent renseigner sur la latence des requêtes exécutées et aider l'usager à identifier des goulots d'étranglement. Cependant, ils ne peuvent pas aider dans la détection des problèmes tels que bogues, problèmes de logique de code, ou autres problèmes de bas niveau.

Remonter à la source du problème nécessite d'employer des analyses beaucoup plus granulaires. Dans ce cas, un compromis est nécessaire entre le degré d'abstraction et le besoin d'être en mesure d'identifier au moins un groupe de problèmes de performance particuliers.

Dans tous les cas, les analyses peuvent être modifiées au besoin, bien que cela nécessite des efforts. Ainsi, la collecte de plusieurs cas d'utilisation différents peut aider à améliorer les

modèles d'abstraction des analyses, de manière à ce que le niveau de généralisation puisse être beaucoup plus élevé.

Au-delà de cela, notre solution démontre avec suffisance comment des outils peuvent être combinés pour effectuer des analyses efficaces des performances des systèmes distribués asynchrones. La combinaison des outils adaptés aux environnements monolithiques, tels que les traceurs LTTng, avec des outils et méthodes employés dans les contextes distribués, permet d'atteindre un niveau de granularité élevé.

Au niveau de la transparence dans la collecte des données d'exécution du système sous Node.js, l'approche que nous proposons constitue une avancée significative dans la facilitation et la réduction des coûts liés aux efforts du déploiement des infrastructures de collecte. Elle constitue une première étape vers sa généralisation sur d'autres environnements et appelle à un changement de paradigme. En effet, les avantages qu'offre notre approche, non pas seulement dans la collecte des données d'analyse, mais en étendant le niveau de transparence à leur analyse, sont inégalés. L'approche peut être étendue sur d'autres systèmes asynchrones, connus pour leur complexité dans la gestion du contexte d'exécution, en vue de migrer vers un nouveau paradigme de traçage et d'analyse de performance en général.

Les systèmes asynchrones étant connus pour leur complexité, les analyses de performance dans des contextes distribués, qui intègrent l'hétérogénéité de ces derniers, permettent d'avoir une vue globale et détaillée sur le fonctionnement du système.

De ce fait, la démarche d'intégration des différentes analyses proposées dans ce travail, ainsi que des analyses futures des systèmes différents, permet de définir le niveau de granularité dans l'analyse de performance.

Effectivement, en présentant une vue unifiée du système distribué global, des goulots d'étranglement peuvent être identifiés à certains endroits du système, alors que la localisation des causes de ces problèmes fait recours aux analyses granulaires préalablement définies pour chaque environnement intégré. Dans ce travail, une preuve de concept a été présentée, par le biais de l'intégration de deux analyses différentes : une développée avec Node.js et la seconde développée pour Redis. L'intégration transparente de ces deux analyses permet d'aligner les interactions dans un outil unifié permettant d'observer les performances du système et de faire appel à ces dernières pour remonter à la cause des problèmes.

L'adoption de ces approches d'intégration, permettant de considérer les analyses hétérogènes comme des fournisseurs de métriques, permet de les assembler comme des blocs, pour construire des systèmes unifiés d'analyse de performance, changeant ainsi les paradigmes employés aujourd'hui.

CHAPITRE 8 CONCLUSION

L'objectif de cette recherche était de proposer de nouvelles méthodes visant le développement d'outils d'analyse de la performance des systèmes distribués asynchrones. La granularité de l'analyse a été un élément clé qui a orienté nos recherches vers l'introduction de nouvelles approches adaptées à ces environnements.

En effet, les systèmes asynchrones se démarquent des autres architectures en raison de leur complexité de fonctionnement. Parmi eux, les systèmes multicouches tels que Node.js présentent une architecture interne complexe, intégrant initialement un asynchronisme inhérent à leur environnement d'exécution, ce qui nécessite des techniques spécifiques pour leur analyse.

Ces systèmes exposent une architecture interne composée de plusieurs acteurs interagissant en symbiose pour exécuter les tâches soumises par les couches supérieures. La coordination des exécutions s'effectue verticalement en interne, allant de la soumission du code JavaScript à son orchestration par une couche particulière appelée Libuv, puis à son exécution grâce à l'intervention de la machine virtuelle interne. Cette complexité soulève un véritable problème de suivi du contexte d'exécution, comme démontré dans le Chapitre 4 de cette thèse.

Nous avons proposé des approches d'instrumentation, et des modèles réutilisables d'analyse pour leur développement et leur application dans différents systèmes asynchrones. Ces approches permettent d'effectuer des analyses granulaires des systèmes asynchrones, ce qui permet d'aller plus en profondeur en vue d'identifier les causes des problèmes de performance.

Node.js a été choisi pour cette recherche en raison de sa complexité inhérente à l'asynchronisme de son fonctionnement et de son statut d'environnement mature, largement adopté par la communauté de développement.

Nous avons proposé un modèle de représentation des exécutions des requêtes prenant en compte le contexte particulier de cet environnement. Ce modèle, appelé VSM, permet de représenter une requête ou une tâche selon deux dimensions : une dimension horizontale pour visualiser la durée de l'exécution de la requête, et une dimension verticale pour visualiser le flux vertical des différentes exécutions atomiques de la requête, à mesure qu'elle traverse les différentes couches internes de Node.js. Ce modèle d'identification nous a permis de mettre en évidence des problèmes de performance et des bogues inhérents à des bibliothèques qui ne pouvaient pas être identifiés par les outils actuels. Ainsi, ce modèle offre une granularité très élevée dans l'analyse des performances des applications développées sous Node.js.

Dans le Chapitre 5, nous avons exploré la conception de méthodes de traçage transparent dans les environnements asynchrones pour réduire les efforts liés au déploiement de l'infrastructure de collecte, qui peut potentiellement altérer le comportement du système lorsque l'instrumentation est mal faite. Nous avons utilisé Node.js à cette fin, en raison de sa complexité pour élever le degré de généralisation de l'approche à d'autres systèmes asynchrones. Nous avons proposé une technique basée sur la reconstruction interne du contexte d'exécution, pour atteindre nos objectifs de transparence.

Cette approche représente un changement de paradigme dans la conduite de l'analyse de performance des systèmes distribués, car elle ne nécessite aucune intervention de l'utilisateur pour déployer l'infrastructure de collecte. L'utilisateur se contente d'utiliser des images dans des conteneurs et de déployer ses applications, sans modifier les procédures du pipeline de développement. L'application en cours d'exécution est tracée de manière transparente, et les analyses sont effectuées sur la trace pour reconstruire automatiquement les contextes des interactions entre les composants du système distribué. En d'autres termes, cette nouvelle approche permet de comprendre de manière transparente l'architecture de fonctionnement d'un système distribué dont on n'a aucune connaissance.

Cette méthode s'appuie sur l'identification des séquences internes des interactions des composants réseau sous Node.js, pour reconstruire les contextes d'exécution des requêtes dans un environnement distribué sous Node.js. Cependant, cette méthode peut être étendue à d'autres environnements offrant de l'asynchronisme, tels que Java et Golang, afin de permettre le traçage transparent des composants.

Enfin, une fois les deux premiers objectifs de la recherche atteints, le Chapitre 6 s'est penché sur le troisième objectif, visant à intégrer ces approches pour offrir transparence et granularité dans l'analyse de performance des systèmes distribués asynchrones. Cette intégration permet de tirer parti de la granularité atteinte par différentes analyses, et de la transparence dans la collecte de la trace d'exécution, unifiant ainsi la démarche de débogage des problèmes de performance dans un seul outil. Deux analyses granulaires de différents environnements ont été intégrées : Node.js et Redis.

L'outil résultant de cette analyse permet de visualiser les performances du système dans son ensemble, afin d'identifier les goulots d'étranglement. Dans ce contexte, l'utilisation des analyses intégrées dans l'outil permet de remonter à la cause du problème de performance observé.

Les analyses sont présentées avec des niveaux d'abstraction qui les rendent compréhensibles à des niveaux élevés et applicables à d'autres systèmes.

Bien que cette recherche ait été menée conformément aux objectifs initiaux, elle ouvre la voie à plusieurs axes d'amélioration et d'extension.

En ce qui concerne la collecte transparente des traces, l'identification des séquences d'exécution s'est basée sur des protocoles de communication synchrones. Le protocole HTTP a été utilisé avec le modèle RestFul, qui est un mode d'interaction entre composants dans les systèmes distribués. La recherche pourrait être étendue à d'autres protocoles tels que le WebSocket et d'autres protocoles légers utilisés dans l'écosystème, élargissant ainsi l'application de notre approche.

En ce qui concerne l'approche de traçage transparente introduite, l'analyse permet la collecte transparente des données et la reconstruction du contexte interne des interactions réseau des composants. Le résultat de l'analyse est une vue permettant de visualiser les requêtes des composants du système distribué, similaire à ce que proposent les traceurs distribués. Cependant, à partir du moment où la reconstruction des interactions nécessite de retrouver le contexte de la requête, l'analyse pourrait être améliorée en tenant compte de cette information pour connecter les fonctions appelées dans chaque couche du système. Cela permettrait de lier chaque requête aux fonctions qu'elle exécute à travers les différentes couches de manière transparente. En d'autres termes, la granularité obtenue par les analyses proposées au Chapitre 4, qui s'est basée sur l'instrumentation des applications Node.js pour propager le contexte de la trace, pourrait être obtenue sans instrumentation de l'application de cette manière, ce qui constituerait une amélioration significative de notre solution.

Enfin, nous tenons à souligner que le code relatif aux analyses développées, ainsi que les bancs d'essai utilisés, sont tous disponibles en téléchargement sur des référentiels publics.

RÉFÉRENCES

[1] T. L. Project. (2021) The lttng documentation. [En ligne]. Disponible : https://lttng.org/docs/v2.9/

[2] N. A. Lynch, *Distributed Algorithms*. Morgan Kaufmann, 1996.

[3] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems (TOCS)*, vol. 16, n°. 2, p. 133–169, 1998.

[4] Redis Labs, "Redis," https://redis.io/, 2023, accessed : date-of-access.

[5] Node.js Foundation, "Node.js," https://nodejs.org/, 2023, accessed : date-of-access.

[6] Mozilla Foundation, "Mozilla firefox," https://www.mozilla.org/en-US/firefox/new/, 2023, accessed : date-of-access.

[7] C. E. Perkins et P. Bhagwat, "Highly dynamic destination-sequenced distance-vector routing (dsdv) for mobile computers," *ACM SIGCOMM Computer Communication Review*, vol. 24, n°. 4, p. 234–244, 1994.

[8] E. Bainomugisha *et al.*, "A survey on reactive programming," *ACM Computing Surveys (CSUR)*, vol. 45, n°. 4, p. 1–34, 2013.

[9] E. Meijer, "Observing the unobservable : Programming with events and services," dans *International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*. Springer, 2010, p. 689–703.

[10] ReactiveX Contributors, "Rxjava," https://github.com/ReactiveX/RxJava, 2023, accessed : date-of-access.

[11] Pivotal Software, "Project reactor," https://projectreactor.io/, 2023, accessed : date-of-access.

[12] P. Vasconcelos et F. Täïani, "Reactive programming : A walk on the wild side with monads," *The Knowledge Engineering Review*, vol. 32, 2017.

[13] J. B. V. Kuhn et V. Klang, "Akka : Simplicity meets power," *Lightbend, Inc*, 2010.

[14] G. Salvaneschi, M. Mezini et P. Eugster, "Towards reactive programming for object-oriented applications," *Transactions on Modelling and Computer Simulation (TOMACS)*, vol. 24, n°. 3, p. 1–39, 2014.

[15] A. Adya *et al.*, "Cooperative task management without manual stack management," dans *USENIX Annual Technical Conference*, 2002.

[16] A. Ghosh, *Understanding Event-Driven Programming*. McGraw-Hill Education, 2010.

[17] R. Reese, *Understanding Asynchronous Programming in Node.js*. O'Reilly Media, 2018.

[18] S. Tilkov et S. Vinoski, “Node.js : Using javascript to build high-performance network programs,” *IEEE Internet Computing*, vol. 14, n°. 6, p. 80–83, 2010.

[19] RabbitMQ Team, “Rabbitmq,” https://www.rabbitmq.com/, 2023, accessed : date-of-access.

[20] B. Liskov, “Promises : Abstractions for asynchronous programming,” *Communications of the ACM*, vol. 55, n°. 6, p. 89–97, 2012.

[21] K. Roy et G. Leuck, *Understanding Event-Driven Programming.* Manning Publications, 2019.

[22] A. S. Tanenbaum et M. Van Steen, *Distributed Systems : Principles and Paradigms.* Prentice Hall, 2007.

[23] G. Coulouris, J. Dollimore et T. Kindberg, *Distributed Systems : Concepts and Design.* Pearson Education, 2011.

[24] S. J. Mullender, édit., *Distributed Systems.* ACM Press/Addison-Wesley Publishing Co., 1993.

[25] J. Bacon et T. Harris, “Operating systems : Concurrent and distributed software design,” 2003.

[26] S. Newman, *Building Microservices.* O’Reilly Media Inc., 2015.

[27] M. Fowler et J. Lewis, “Microservices,” *a new way of building applications*, 2014.

[28] N. Dragoni *et al.*, “Microservices : Yesterday, today, and tomorrow,” *Present and Ulterior Software Engineering*, p. 195–216, 2017.

[29] A. Balalaie, A. Heydarnoori et P. Jamshidi, “Microservices architecture enables devops : Migration to a cloud-native architecture,” *IEEE Software*, vol. 33, n°. 3, p. 42–52, 2016.

[30] C. Richardson, *Microservices Patterns : With examples in Java.* Manning Publications, 2018.

[31] D. Kang, F. Wu et C. Kil, “Container and microservice driven design for cloud infrastructure devops,” dans *2016 IEEE International Conference on Cloud Engineering Workshop (IC2EW).* IEEE, 2016, p. 202–203.

[32] C. N. C. Foundation, “Kubernetes,” 2014. [En ligne]. Disponible : https://kubernetes.io/fr

[33] Docker, Inc., “Docker,” https://www.docker.com/, 2023, consulté le [07 Octobre 2023].

[34] J. Hamilton *et al.*, “Scada statistics monitoring using the elastic stack (elasticsearch, logstash, kibana),” 2018.

[35] A. Lahmadi et F. Beck, “Powering monitoring analytics with elk stack,” dans *9th international conference on autonomous infrastructure, management and security (aims 2015)*, 2015.

[36] G. Katsaros, R. Kübert et G. Gallizo, “Building a service-oriented monitoring framework with rest and nagios,” dans *2011 IEEE International Conference on Services Computing*. IEEE, 2011, p. 426–431.

[37] N. Sukhija *et al.*, “Event management and monitoring framework for hpc environments using servicenow and prometheus,” dans *Proceedings of the 12th International Conference on Management of Digital EcoSystems*, 2020, p. 149–156.

[38] M. Holopainen, “Monitoring container environment with prometheus and grafana,” 2021.

[39] B. H. Sigelman *et al.*, “Dapper, a large-scale distributed systems tracing infrastructure,” 2010.

[40] Wikipedia. Instrumentation (computer programming). [En ligne]. Disponible : https://en.wikipedia.org/wiki/Instrumentation_(computer_programming)

[41] C.-K. Luk *et al.*, “Pin : building customized program analysis tools with dynamic instrumentation,” dans *ACM SIGPLAN Notices*, vol. 40. ACM, 2005, p. 190–200.

[42] M. Bunnell, “Dynamic instrumentation event trace system and methods,” mai 16 2006, uS Patent 7,047,521.

[43] P. Caserta, “Analyse statique et dynamique de code et visualisation des logiciels via la métaphore de la ville : contribution à l’aide à la compréhension des programmes.” Thèse de doctorat, Université de Lorraine, 2012.

[44] T. Ball, “The concept of dynamic analysis,” dans *Software Engineering—ESEC/FSE’99*. Springer, 1999, p. 216–234.

[45] B. Cornelissen *et al.*, “A systematic survey of program comprehension through dynamic analysis,” *IEEE Transactions on Software Engineering*, vol. 35, n°. 5, p. 684–702, 2009.

[46] (2018) Qu’est-ce que le débogage. [En ligne]. Disponible : https://docs.microsoft.com/fr-fr/visualstudio/debugger/what-is-debugging?view=vs-2019

[47] R. Lalement. (1996) Qu’est-ce qu’un débogueur ? [En ligne]. Disponible : http://cermics.enpc.fr/polys/info-96/node134.html

[48] P. Barham *et al.*, “Using magpie for request extraction and workload modelling.” dans *OSDI*, vol. 4, 2004, p. 18–18.

[49] B. Paul *et al.*, “Magpie : Online modelling and performance-aware systems.” dans *HotOS*, 2003, p. 85–90.

[50] Y.-Y. M. Chen *et al.*, *Path-based failure and evolution management*. University of California, Berkeley, 2004.

[51] T. Gschwind *et al.*, “Webmon : A performance profiler for web transactions,” dans *Proceedings Fourth IEEE International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems (WECWIS 2002)*. IEEE, 2002, p. 171–176.

[52] R. Fonseca *et al.*, “X-trace : A pervasive network tracing framework,” dans *4th {USENIX} Symposium on Networked Systems Design & Implementation ({NSDI} 07)*, 2007.

[53] K. Chaturvedi. Modern distributed tracing systems architecture. [En ligne]. Disponible : https://medium.com/modern-distributed-tracing-systems-architecture/modern-distributed-tracing-systems-architecture-698691b4f998

[54] What is opentracing ? everything you need to get started. [En ligne]. Disponible : https://www.sentinelone.com/blog/what-is-opentracing/

[55] M. S D et D. M., “Distributed request tracing using zipkin and spring boot sleuth,” *International Journal of Computer Applications*, vol. 175, p. 35–37, 08 2020.

[56] A. A. Mesh et A. B. Overview, “Understanding service mesh,” *Journal of Cloud Computing*, 2023.

[57] “Prometheus : Monitoring system & time series database,” 2023, consulté le [04 octobre 2]. [En ligne]. Disponible : https://prometheus.io/

[58] “Jaeger : open source, end-to-end distributed tracing,” 2023, consulté le [07 Octobre 2023]. [En ligne]. Disponible : https://www.jaegertracing.io/

[59] “Istio : Connect, secure, control, and observe services,” 2023, consulté le [07 Octobre 2023]. [En ligne]. Disponible : https://istio.io/

[60] “Linkerd : Ultralight service mesh for kubernetes,” 2023, consulté le [07 Octobre 2023]. [En ligne]. Disponible : https://linkerd.io/

[61] Y. J. Bationo, N. Ezzati-Jivan et M. R. Dagenais, “Efficient cloud tracing : From very high level to very low level,” dans *2018 IEEE International Conference on Consumer Electronics (ICCE)*. IEEE, 2018, p. 1–6.

[62] Y. Gan *et al.*, “Seer : Leveraging big data to navigate the complexity of performance debugging in cloud microservices,” dans *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems*, 2019, p. 19–33.

[63] Y. Geng *et al.*, “Exploiting a natural network effect for scalable, fine-grained clock synchronization,” dans *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, 2018, p. 81–94.

[64] D. Gmach *et al.*, “Workload analysis and demand prediction of enterprise data center applications,” dans *2007 IEEE 10th International Symposium on Workload Characterization*. IEEE, 2007, p. 171–180.

[65] Z. Gong, X. Gu et J. Wilkes, “Press : Predictive elastic resource scaling for cloud systems,” dans *2010 International Conference on Network and Service Management*. Ieee, 2010, p. 9–16.

[66] J. F. Shortle *et al.*, *Fundamentals of queueing theory*. John Wiley & Sons, 2018, vol. 399.

[67] D. Ardelean, A. Diwan et C. Erdman, “Performance analysis of cloud applications,” dans *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, 2018, p. 405–417.

[68] 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.

[69] S. Rostedt. ftrace - function tracer. [En ligne]. Disponible : https://01.org/linuxgraphics/gfx-docs/drm/trace/ftrace.html

[70] S. Rostedts. Using the trace_event() macro (part 1). [En ligne]. Disponible : https://lwn.net/Articles/379903/

[71] A. Aranya, C. P. Wright et E. Zadok, “Tracefs : A file system to trace them all.” dans *FAST*, 2004, p. 129–145.

[72] M. Fleming. (2017) A thorough introduction to ebpf. [En ligne]. Disponible : https://lwn.net/Articles/740157/

[73] M. Gebai et M. R. Dagenais, “Survey and analysis of kernel and userspace tracers on linux : Design, implementation, and overhead,” *ACM Computing Surveys (CSUR)*, vol. 51, nº. 2, p. 1–33, 2018.

[74] A. Ghods, “A study of linux perf and slab allocation sub-systems. (2016).” 2016.

[75] A. R. Ghods, “A study of linux perf and slab allocation sub-systems,” Mémoire de maîtrise, University of Waterloo, 2016.

[76] D. Julien et D. Mathieu, “Lttng-ust vs systemtap userspace tracing benchmarks,” 2016.

[77] R. Fontana *et al.*, “Babeltrace 2 : Tracing and monitoring framework,” dans *2019 Linux Plumbers Conference*, 2019.

[78] A. Montplaisir *et al.*, “State history tree : An incremental disk-based data structure for very large interval data. in 2013 ase,” dans *IEEE international conference on big data*, 2013.

[79] M. Michels. (2020) The art of debugging distributed systems. [En ligne]. Disponible : https://maximilianmichels.com/2020/debugging-distributed-systems/

[80] C. N. C. Foundation. (2022) Opentelemetry client architecture. [En ligne]. Disponible : https://github.com/open-telemetry/opentelemetry-specification/blob/main/specification/overview.md

[81] A. Rashevskaya. (2021) Node.js architecture from a to z. [En ligne]. Disponible : https://litslink.com/blog/node-js-architecture-from-a-to-z

[82] AppDynamics. (2015) Top five node.js performance metrics, tips and tricks. [En ligne]. Disponible : https://kapost-files-prod.s3.amazonaws.com/published/55526fea6c1e4cdd7c0000c2/ebook-top-5-node-dot-js-metrics-tips-and-tricks.pdf

[83] D. Ancona *et al.*, "Towards runtime monitoring of node.js and its application to the internet of things," dans *Proceedings First Workshop on Architectures, Languages and Paradigms for IoT, ALP4IoT@iFM 2017, Turin, Italy, September 18, 2017*, ser. EPTCS, D. Pianini et G. Salvaneschi, édit., vol. 264, 2017, p. 27–42. [En ligne]. Disponible : https://doi.org/10.4204/EPTCS.264.4

[84] X. Chang *et al.*, "Detecting atomicity violations for event-driven node.js applications," dans *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, p. 631–642.

[85] S. H. Jensen, M. Madsen et A. Møller, "Modeling the html dom and browser api in static analysis of javascript web applications," dans *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA : Association for Computing Machinery, 2011, p. 59–69. [En ligne]. Disponible : https://doi.org/10.1145/2025113.2025125

[86] S. Guarnieri et B. Livshits, "Gatekeeper : Mostly static enforcement of security and reliability policies for javascript code." dans *18th USENIX Security Symposium (USENIX Security 09)*. Montreal, Quebec : USENIX Association, 01 2009, p. 151–168. [En ligne]. Disponible : https://www.usenix.org/conference/usenixsecurity09/technical-sessions/presentation/gatekeeper-mostly-static-enforcement

[87] S. Hong, Y. Park et M. Kim, "Detecting concurrency errors in client-side java script web applications," dans *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, 2014, p. 61–70.

[88] J. Davis, A. Thekumparampil et D. Lee, "Node.fz : Fuzzing the server-side event-driven architecture," dans *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17. New York, NY, USA : Association for Computing Machinery, 2017, p. 145–160. [En ligne]. Disponible : https://doi.org/10.1145/3064176.3064188

[89] A. T. Endo et A. Møller, “Noderacer : Event race detection for node.js applications,” dans *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 2020, p. 120–130.

[90] C. Q. Adamsen *et al.*, “Repairing event race errors by controlling nondeterminism,” dans *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, p. 289–299.

[91] ——, “Practical ajax race detection for javascript web applications,” dans *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA : Association for Computing Machinery, 2018, p. 38–48.

[92] L. Nkenyereye et J. Jang, “Performance evaluation of server-side javascript for healthcare hub server in remote healthcare monitoring system,” dans *The 7th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN 2016)/The 6th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH-2016)/Affiliated Workshops, September 19-22, 2016, London, United Kingdom*, ser. Procedia Computer Science, E. M. Shakshuki, édit., vol. 98. Elsevier, 2016, p. 382–387. [En ligne]. Disponible : https://doi.org/10.1016/j.procs.2016.09.058

[93] A. Fondation., “Apache jmeter.” [En ligne]. Disponible : https://github.com/apache/jmeter

[94] M. Madsen, F. Tip et O. Lhoták, “Static analysis of event-driven node.js javascript applications,” *SIGPLAN Not.*, vol. 50, n°. 10, p. 505–519, oct 2015. [En ligne]. Disponible : https://doi.org/10.1145/2858965.2814272

[95] V. Jitani. (2021) Monads explained. [En ligne]. Disponible : https://towardsdatascience.com/monads-from-the-lens-of-imperative-programmer-af1ab8c8790c

[96] D. Janin, “An equational modeling of asynchronous concurrent programming,” dans *Trends in Functional Programming*, A. Byrski et J. Hughes, édit. Cham : Springer International Publishing, 2020, p. 180–203.

[97] E. Moggi, “Notions of computation and monads,” *Information and Computation*, vol. 93, n°. 1, p. 55–92, 1991, selections from 1989 IEEE Symposium on Logic in Computer Science. [En ligne]. Disponible : https://www.sciencedirect.com/science/article/pii/0890540191900524

[98] E. Org., “Trace compass.” [En ligne]. Disponible : https://projects.eclipse.org/proposals/trace-compass

[99] M. Gebai et M. R. Dagenais, “Survey and analysis of kernel and userspace tracers on linux : Design, implementation, and overhead,” *ACM Comput. Surv.*, vol. 51, n°. 2, mar 2018. [En ligne]. Disponible : https://doi.org/10.1145/3158644

[100] M. Hauswirth *et al.*, “Vertical profiling : understanding the behavior of object-priented applications,” dans *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2004, p. 251–269.

[101] ——, “Automating vertical profiling,” dans *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2005, p. 281–296.

[102] G. Ausiello *et al.*, “k-calling context profiling,” dans *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, 2012, p. 867–878.

[103] P. Moret, W. Binder et A. Villazón, “Cccp : Complete calling context profiling in virtual execution environments,” dans *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, 2009, p. 151–160.

[104] M. Gokhman. (2019) Node.js event-loop : How even quick node.js async functions can block the event-loop, starve i/o. [En ligne]. Disponible : https://snyk.io/blog/nodejs-how-even-quick-async-functions-can-block-the-event-loop-starve-io/

[105] V. Vallet. (2020) Node.js, lots of ways to block your event-loop (and how to avoid it). [En ligne]. Disponible : https://medium.com/voodoo-engineering/node-js-lots-of-ways-to-block-your-event-loop-and-how-to-avoid-it-b41f41deecf5

[106] C.-A. Staicu et M. Pradel, “Freezing the web : A study of {ReDoS} vulnerabilities in {JavaScript-based} web servers,” dans *27th USENIX Security Symposium (USENIX Security 18)*, 2018, p. 361–376.

[107] TypeFox et Ericsson, “Eclipse theia,” 2019. [En ligne]. Disponible : https://www.theia-ide.org/

[108] H. Shah et T. Soomro, “Node.js challenges in implementation,” *Global Journal of Computer Science and Technology*, vol. 17, p. 72–83, 05 2017.

[109] T. Point. (2016) Node.js tutorial. [En ligne]. Disponible : https://www.tutorialspoint.com/nodejs/index.htm

[110] J. Zhu, “A scalability-oriented benchmark suite for node.js in the cloud,” Thèse de doctorat, University of New Brunswick., 2018.

[111] J. Lewis et M. Fowler, “Microservices : a definition of this new architectural term,” *MartinFowler.com*, vol. 25, 2014.

[112] D. Sharma *et al.*, “Hansel : diagnosing faults in openstack,” dans *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*. ACM, 2015, p. 23.

[113] R. R. Sambasivan *et al.*, “Diagnosing performance changes by comparing request flows,” dans *NSDI*, vol. 5, 2011, p. 1–1.

[114] B.-C. Tak *et al.*, “vpath : Precise discovery of request processing paths from black-box observations of thread and network activities,” dans *USENIX Annual technical conference*, 2009.

[115] A. R. Sampaio *et al.*, “Supporting microservice evolution,” dans *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE, 2017, p. 539–543.

[116] J. Kaldor *et al.*, “Canopy : An end-to-end performance tracing and analysis system,” dans *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, p. 34–50.

[117] J. Mace, R. Roelke et R. Fonseca, “Pivot tracing : Dynamic causal monitoring for distributed systems,” dans *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 2015, p. 378–393.

[118] M. Fani Sani, S. J. van Zelst et W. M. van der Aalst, “The impact of biased sampling of event logs on the performance of process discovery,” *Computing*, vol. 103, p. 1085–1104, 2021.

[119] C. Liu *et al.*, “Sampling business process event logs using graph-based ranking model,” *Concurrency and Computation : Practice and Experience*, vol. 33, n°. 5, p. e5974, 2021.

[120] Z. Wang, A. Sanchez et A. Herkersdorf, “Scisim : a software performance estimation framework using source code instrumentation,” dans *Proceedings of the 7th international workshop on Software and performance*. ACM, 2008, p. 33–42.

[121] M. Santana *et al.*, “Transparent tracing of microservice-based applications,” dans *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, 2019, p. 1252–1259.

[122] B. Wassermann et W. Emmerich, “Monere : Monitoring of service compositions for failure diagnosis,” dans *Service-Oriented Computing : 9th International Conference, ICSOC 2011, Paphos, Cyprus, December 5-8, 2011 Proceedings 9*. Springer, 2011, p. 344–358.

[123] M. Y. Chen *et al.*, “Pinpoint : Problem determination in large, dynamic internet services,” dans *Proceedings International Conference on Dependable Systems and Networks*. IEEE, 2002, p. 595–604.

[124] S. Kitajima et N. Matsuoka, “Inferring calling relationship based on external observation for microservice architecture,” dans *Service-Oriented Computing : 15th International Conference, ICSOC 2017, Malaga, Spain, November 13–16, 2017, Proceedings*. Springer, 2017, p. 229–237.

[125] M. K. Aguilera *et al.*, “Performance debugging for distributed systems of black boxes,” *ACM SIGOPS Operating Systems Review*, vol. 37, n°. 5, p. 74–89, 2003.

[126] D. A. Menasce, “Qos issues in web services,” *IEEE internet computing*, vol. 6, n°. 6, p. 72–75, 2002.

[127] “Opentelemetry,” https://opentelemetry.io, accessed on January 2, 2024.

[128] H. K. Cho *et al.*, “Instant profiling : Instrumentation sampling for profiling datacenter applications,” dans *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2013, p. 1–10.

[129] N. B. Seghier et O. Kazar, “Performance benchmarking and comparison of nosql databases : Redis vs mongodb vs cassandra using ycsb tool,” dans *2021 International Conference on Recent Advances in Mathematics and Informatics (ICRAMI)*. IEEE, 2021, p. 1–6.

[130] H. M. Kabamba, D. Michel et K. Matthew, “Vnode : Low-overhead transparent tracing of node.js-based microservices architectures,” *arXiv preprint arXiv :1304.0969*, 2023.

[131] C. Complete, “Steve mcconnel,” 2004.

[132] X. Merino et C. E. Otero, “Microservice debugging with checkpoint-restart,” dans *2023 IEEE Cloud Summit*. IEEE, 2023, p. 58–63.

[133] A. Quinn *et al.*, “Debugging the {OmniTable} way,” dans *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, p. 357–373.

[134] M. Whittaker *et al.*, “Debugging distributed systems with why-across-time provenance,” dans *Proceedings of the ACM symposium on cloud computing*, 2018, p. 333–346.

[135] P. Arafa *et al.*, “Qdime : Qos-aware dynamic binary instrumentation,” dans *2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2017, p. 132–142.

[136] N. Jia *et al.*, “Spire : improving dynamic binary translation through spc-indexed indirect branch redirecting,” *ACM SIGPLAN Notices*, vol. 48, n°. 7, p. 1–12, 2013.

[137] M. Serrano et X. Zhuang, “Building approximate calling context from partial call traces,” dans *2009 International Symposium on Code Generation and Optimization*. IEEE, 2009, p. 221–230.

[138] D. Gross *et al.*, *Fundamentals of Queueing Theory*, ser. Wiley Series in Probability and Statistics. Wiley, 2011, n°. 627.

[139] H. Kasture et D. Sanchez, “Ubik : Efficient cache sharing with strict qos for latency-critical workloads,” *ACM SIGPLAN Notices*, vol. 49, n°. 4, p. 729–742, 2014.

[140] “Memtier : A Redis benchmark tool,” https ://github.com/RedisLabs/memtier_benchmark, 2012, GitHub repository.