

Titre: Mise à l'échelle des analyses de trace dans une architecture
Title: modulaire

Auteur: Ahmad Faour
Author:

Date: 2024

Type: Mémoire ou thèse / Dissertation or Thesis

Référence: Faour, A. (2024). Mise à l'échelle des analyses de trace dans une architecture modulaire [Mémoire de maîtrise, Polytechnique Montréal]. PolyPublie.
Citation: <https://publications.polymtl.ca/58336/>

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

URL de PolyPublie: <https://publications.polymtl.ca/58336/>
PolyPublie URL:

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

Programme: Génie informatique
Program:

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

Mise à l'échelle des analyses de trace dans une architecture modulaire

AHMAD FAOUR

Département de génie informatique et génie logiciel

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

Génie informatique

Avril 2024

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

Ce mémoire intitulé :

Mise à l'échelle des analyses de trace dans une architecture modulaire

présenté par **Ahmad FAOUR**

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

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

Martine BELLAÏCHE, présidente

Michel DAGENAIS, membre et directeur de recherche

Tarek OULD-BACHIR, membre

REMERCIEMENTS

Je tiens à exprimer toute ma gratitude à mon directeur de recherche, Michel Dagenais. Il m'a offert l'opportunité de participer à ce projet de recherche au sein du laboratoire DORSAL. Son soutien et son expérience m'ont guidé au fil des années et m'ont aidé à surmonter ce défi.

Je tiens à remercier tout particulièrement Arnaud Fiorini et Herve Kabamba pour leurs conseils et leur assistance tout au long de mon projet. Leurs connaissances et leur disponibilité pour soutenir les étudiants du laboratoire DORSAL m'ont inspiré tout au long de mon travail.

Je remercie également 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.

Par ailleurs, j'aimerais exprimer ma reconnaissance envers tous mes collègues du laboratoire DORSAL. Sans vous, ces dernières années n'auraient pas été aussi enrichissantes.

Enfin, je tiens à remercier ma famille ainsi que mes amis pour leur soutien constant.

RÉSUMÉ

Les systèmes de calcul de haute performance deviennent de plus en plus nécessaires pour répondre aux besoins auxquels nous sommes confrontés afin de fournir des services dont des millions d'utilisateurs dépendent chaque jour. Il est donc important de disposer d'outils axés sur la compréhension du fonctionnement du système, en collectant les informations nécessaires permettant de dresser des analyses. Dans ce contexte, le traçage, qui est une technique largement connue permettant de collecter les informations sur les fonctionnements internes et les états du système, constitue l'une des meilleures approches. Dans ce cas précis, l'information collectée nécessite le développement de visualisations basées sur des analyses définies en amont, et permettant l'interaction avec l'utilisateur dans son cycle de développement et d'opération. Il existe des solutions pour analyser les traces collectées dans les systèmes de calcul de haute performance, mais aucune d'entre elles ne résout complètement les besoins en évolutivité du traitement des analyses et en flexibilité du développement des analyses. Par conséquent, nous proposons une nouvelle solution qui permet la mise à l'échelle des analyses dans une architecture modulaire.

Notre travail s'étend à l'implémentation d'une architecture distribuée visant à accroître l'évolutivité des outils d'analyse et visant à résoudre les problèmes de flexibilité lors du développement d'analyses. De plus, nous avons étendu le Trace Server Protocol. Cela nous a permis d'implémenter des analyses globales à l'échelle d'une grappe de calcul et faciliter l'analyse des traces connectées en exploitant les événements réseaux. Des bancs de tests ont été menés afin d'évaluer le gain en performance et le surcoût associés à l'architecture distribuée Maître-Ouvrier par rapport à l'approche précédente, l'architecture distribuée client-serveur.

En conclusion, les résultats montrent un gain en performance significatif en fonction du nombre de nœuds mis en place et indiquent un faible surcoût. La majorité du surcoût est causée par la sérialisation du format JSON. Nous avons conclu que la solution pourrait facilement être portée dans les systèmes de calcul de haute performance, mais aussi dans tout système distribué qui nécessite une puissance d'analyse de traces supérieure et une vaste gamme d'options d'analyse. Cependant, dans les cas où le volume des traces est faible, le gain pourrait être nul ou même se transformer en légère perte.

ABSTRACT

High-performance computing systems are becoming increasingly necessary to meet the demands of services used by millions of users every day. Therefore, it is important to have tools focused on understanding system operation by collecting necessary information for analysis. In this context, tracing, a widely known technique for gathering information on internal operations and system states, emerges as one of the best approaches. In this specific case, the collected information requires the development of visualizations based on predefined analyses, enabling interaction with the user in their development.

While there are solutions for analyzing traces collected in high-performance computing systems, none completely address the scalability needs of analysis processing and the flexibility of analysis development. Therefore, we propose a new solution that enables scalable analysis within a modular architecture.

Our work extends to the implementation of a distributed architecture aimed at increasing the scalability of analysis tools. This modular distributed architecture is designed to address flexibility issues encountered during analysis development. It extends the Trace-Server-Protocol to enable global analysis at the scale of a computing cluster and to facilitate the analysis of connected traces by leveraging network events. We conducted a benchmark to evaluate the performance gain and associated overhead of the Master-Worker distributed architecture compared to the previous Client-Server distributed architecture.

In conclusion, the results demonstrate a significant performance gain based on the number of nodes deployed and indicate a low overhead. The majority of the overhead is caused by JSON serialization. We concluded that the solution could easily be implemented in high-performance computing systems as well as any distributed system requiring enhanced trace analysis capabilities and a wide range of analysis options. However, in cases where trace volume is low, no performance or even a degradation may be experienced.

TABLE DES MATIÈRES

REMERCIEMENTS	iii
RÉSUMÉ	iv
ABSTRACT	v
TABLE DES MATIÈRES	vi
LISTE DES TABLEAUX	ix
LISTE DES FIGURES	x
LISTE DES SIGLES ET ABRÉVIATIONS	xi
CHAPITRE 1 INTRODUCTION	1
1.1 MÉTHODOLOGIE	2
1.1.1 Étape de collecte des données	2
1.1.2 Étape d'analyse des données	2
1.1.3 Étape de validation	3
1.2 ÉLÉMENTS DE LA PROBLÉMATIQUE	3
1.3 OBJECTIFS DE LA RECHERCHE	4
1.4 CONTRIBUTIONS ORIGINALES	4
1.5 PLAN DU MEMOIRE	4
CHAPITRE 2 REVUE DE LITTÉRATURE	6
2.1 Architectures distribuées	6
2.1.1 Client-serveur	6
2.1.2 Pair-à-pair	8
2.1.3 Maître-Ouvrier	9
2.1.4 Sérialisation	11
2.1.5 Interopérabilité de systèmes hétérogènes	14
2.1.6 Trace Server Protocol	20
2.2 Traçage	21
2.2.1 Outils de traçage	21
2.2.2 Outils de traçage distribués	25

2.3	Analyse, traitement et visualisation de trace	26
2.3.1	Gestionnaire d'état	26
2.3.2	Outils de visualisation de trace	27
2.3.3	Analyse distribué de trace	31
2.4	Conclusion de la revue littéraire	35

CHAPITRE 3 ARTICLE 1 : SCALING TRACE ANALYSIS IN A MODULAR DISTRIBUTED ARCHITECTURE

		36
3.1	Abstract	36
3.2	Introduction	36
3.3	Background	38
3.4	Related Work	39
3.4.1	Trace Server Protocol	39
3.4.2	Trace Visualization Tools	40
3.4.3	Distributed analysis of trace	41
3.5	Proposed Solution	44
3.5.1	Motivation	44
3.5.2	Overall architecture	46
3.5.3	Distributed Analysis of Traces	47
3.5.4	Deployment	56
3.5.5	Visualization	57
3.5.6	Generating the Configuration file	58
3.6	Results	59
3.6.1	Thread Status Analysis	59
3.6.2	CPU Usage Analysis	59
3.6.3	Query Protocol	60
3.6.4	Execution time overhead	60
3.6.5	Test Environment	60
3.6.6	Trace API	62
3.6.7	Experiment API	63
3.6.8	TimeGraph API	63
3.6.9	XY API	66
3.7	Discussion	68
3.8	Conclusion and Future Work	69
3.9	Acknowledgement	70

CHAPITRE 4 DISCUSSION GÉNÉRALE

4.1	Interopérabilité	71
4.2	Extensibilité	71
4.3	Retour sur les résultats	72
CHAPITRE 5 CONCLUSION		74
5.1	Synthèse des travaux	74
5.2	Limitations	74
5.3	Améliorations futures	75
RÉFÉRENCES		76

LISTE DES TABLEAUX

Tableau 3.1	Analysis Tool comparison	44
Tableau 3.2	Benchmark - Response time	62
Tableau 3.3	TimeGraph API Benchmark - Response time	64
Tableau 3.4	Benchmark XY API - Response time	67

LISTE DES FIGURES

Figure 2.1	Client-Serveur	8
Figure 2.2	Pair-À-Pair	9
Figure 2.3	Architecture Maître-Ouvrier	10
Figure 2.4	Diagramme de séquence du protocole de serveur de langage	18
Figure 2.5	Architecture Monto	19
Figure 2.6	Gestionnaire d'état	27
Figure 2.7	Trace Compass	28
Figure 2.8	Theia - Trace Viewer Extension	29
Figure 2.9	Grafana - Tableau de board	30
Figure 2.10	Gestionnaire d'état avec un évènement perdu	33
Figure 3.1	Gestionnaire d'état	38
Figure 3.2	Trace Compass Server Architecture	39
Figure 3.3	Proposed scalable distributed trace analysis	47
Figure 3.4	Merging JSON	50
Figure 3.5	Critical Path	54
Figure 3.6	Theia - Trace Viewer Architecture	57
Figure 3.7	Call Stack View - Open Trace	63
Figure 3.8	Call Stack View - Create Experiment	63
Figure 3.9	Call Stack View - Get Time Graph Tree	65
Figure 3.10	Call Stack View - Get Time Graph States (4GB)	66
Figure 3.11	Call Stack View - Get Time Graph States (40GB)	66
Figure 3.12	Get Time Graph Arrows	66
Figure 3.13	Call Stack View - Get XY Tree	68
Figure 3.14	Call Stack View - Get XY (4GB)	68
Figure 3.15	Call Stack View - Get XY (40GB)	68

LISTE DES SIGLES ET ABRÉVIATIONS

LTng	Linux Trace Toolkit next generation
JSON	JavaScript Object Notation
REST	Representational State Transfer
HPC	High Performance Computing
API	Application Programming Interface
TSP	Trace Server Protocol
DAP	Debug Adapter Protocol

CHAPITRE 1 INTRODUCTION

Le développement d'applications consiste aujourd'hui, à prendre en compte plusieurs aspects, tels que la performance et la modularité. Tout autant, la phase de conception revêt une importance capitale dans un contexte d'intégration de l'ensemble des aspects essentiels au bon fonctionnement des applications. Il demeure très difficile, voire impossible, de tout prévoir sans rencontrer d'embûches lors du développement.

Dans la grande majorité des cas, la mise en place de cadriciels spécifiques permet d'accélérer le développement et mieux, faciliter la détection d'anomalies en amont, par exemple les cadriciels de développement et les cadriciels de test. La nature des problèmes soulevés par le fonctionnement des applications informatiques appelle à des outils particuliers qui permettent de déboguer, d'identifier et de remonter à la source des erreurs. Le problème devient beaucoup plus complexe dans un contexte de systèmes distribués, dans lequel il s'avère impérieux d'employer des outils qui tiennent compte de la complexité de l'agrégation des informations, de l'impact de leur transfert sur les performances du système, pour n'en citer que ceux-là.

Il est alors important d'avoir des outils axés sur la compréhension du fonctionnement du système, en collectant l'information nécessaire permettant de dresser des analyses, qui passent par la création et la corrélation des métriques importantes renseignant sur la santé de ce dernier. Dans ce contexte, le traçage, qui est une technique largement connue permettant de collecter les informations sur les fonctionnements internes et les états du système, constitue l'une des meilleures approches. Dans ce cas précis, l'information collectée nécessite le développement des visualisations basées sur des analyses définies en amont, et permettant l'interaction avec l'utilisateur dans son pipeline de développement et opération.

Il peut être problématique de gérer une grande quantité de traces lors de l'exécution de l'analyse. En effet, les systèmes nécessitant une puissance de calcul élevée, comme les environnements de calcul haute performance (HPC) ou les systèmes distribués, produisent une quantité considérable de traces nécessitant de très longues périodes pour leur indexation, vu le contexte linéaire de la lecture.

Cette recherche vise à répondre à ce problème en proposant une approche nouvelle basée sur une architecture distribuée modulaire. Tenant compte des contraintes de flexibilité des analyses, cette dernière vise à contribuer à l'état de l'art, en mettant en avant une approche permettant aux outils d'analyse, de s'adapter à des échelles variables tout en préservant une souplesse pour le développement d'analyses.

1.1 MÉTHODOLOGIE

D'une manière générale, l'étude de la performance des applications comprend deux étapes importantes : la collecte des données et leur analyse.

1.1.1 Étape de collecte des données

Dans un contexte distribué, comprendre le fonctionnement d'un système nécessite d'abord de collecter des données sur celui-ci. L'infrastructure de collecte adopte généralement le traçage comme technique, car il permet de suivre les états du système et d'en extraire des informations pertinentes avant de les soumettre à une analyse. Parfois, une phase d'instrumentation est nécessaire, impliquant l'insertion de points de trace à des endroits spécifiques du système ou de son environnement pour générer des événements lorsque le système les rencontre.

Dans le cadre de cette recherche, nous avons opté pour l'utilisation du traceur logiciel LTTng, reconnu pour son faible impact sur le système tracé et son applicabilité dans les environnements de production.

L'instrumentation statique a été utilisée pour insérer les points de trace, en modifiant au besoin le code source des applications. Le traçage a été effectué à deux niveaux : dans l'espace utilisateur et dans l'espace noyau du système d'exploitation. Le traceur choisi offre des fonctionnalités de traçage multi-niveaux, évitant ainsi la gestion de la synchronisation des événements pour les analyses menées dans les deux espaces.

1.1.2 Étape d'analyse des données

L'objectif final de la collecte des données est de comprendre les différents états du système. Ces données, appelées événements, sont écrites dans des fichiers de trace de taille variable. Selon la granularité de l'instrumentation et la durée de la session de traçage, la taille du fichier de trace peut augmenter rapidement pour atteindre des tailles considérables. Les événements peuvent se compter par millions, voire par milliards dans une trace. De plus, lorsque la session de traçage est activée sur chaque nœud d'un système distribué, cela augmente significativement le nombre d'événements à traiter lors de l'analyse et, le cas échéant, lors de l'agrégation des différentes traces.

Il est alors extrêmement difficile, voire impossible, d'extraire manuellement des informations permettant de comprendre le fonctionnement du système. De plus, les différents événements présents dans la trace fournissent une meilleure indication lorsqu'ils sont corrélés à d'autres événements du système pour décrire son état. Il est donc crucial de développer des analyses

ou d'en utiliser certaines pour permettre la flexibilité des visualisations et faciliter l'interprétation.

1.1.3 Étape de validation

Cette recherche vise principalement à proposer une nouvelle approche préservant la flexibilité des analyses et leur mise à l'échelle. Pour valider cette approche, des cas d'utilisation spécifiques basés sur des modèles de visualisation seront employés et confrontés à des tailles différentes de fichiers de traces sur différentes infrastructures.

1.2 ÉLÉMENTS DE LA PROBLÉMATIQUE

Les systèmes de calcul de haute performance deviennent de plus en plus nécessaires pour répondre aux besoins auxquels nous faisons face, pour être en mesure d'offrir des services que des millions d'utilisateur utilisent chaque jour.

De la même manière qu'une application peut avoir besoin du traçage pour identifier les goulots d'étranglement, ceci est encore plus vrai pour les systèmes de calcul de haute performance. L'inconvénient est que les systèmes de calcul de haute performance se retrouvent avec des centaines ou même des milliers de noeuds à être tracés autant au niveau du noyau qu'au niveau de l'espace utilisateur. Cela génère une quantité astronomique de traces sur chaque noeud, ce qui complexifie l'utilisation des outils d'analyse et de visualisation comme ils le sont actuellement.

Pour commencer, transporter une quantité astronomique de données dans le réseau est difficile et peut prendre du temps. De plus, il se peut que le noeud sur lequel nous aimerions analyser ces traces ne possède pas l'espace d'emménagement nécessaire pour recueillir toutes les traces. Ensuite, supposons que nous arrivions à regrouper toutes les traces dans un emplacement. L'outil d'analyse doit être capable de traiter les traces en un temps raisonnable. Si l'analyse prend ne serait-ce qu'une heure à traiter et à visualiser, en excluant le temps de rassembler les fichiers de trace, ce temps d'attente retarderait de une heure le diagnostic de l'analyse pour résoudre le goulot d'étranglement. En espérant que les événements qui exposerait le goulot d'étranglement ait été sauvegardé dans la trace, sinon il faudrait répéter toutes les étapes.

Pour le moment, certaines solutions existent pour analyser des volumes astronomiques de trace, mais elles manquent grandement de flexibilité pour les choix d'analyses disponible ce qui rend le processus de détection de goulot d'étranglement plus difficile. À l'opposé, certaines solutions se concentrent à offrir un vaste choix d'analyse, mais n'arrivent pas à évoluer leur

architecture, assez souvent monolithique, dans une architecture distribuée.

Jusqu'à ce jour, les développeurs réussissent, tant bien que mal, à développer ou améliorer des systèmes de calcul de haute performance et d'y trouver des goulots d'étranglement pour avoir un gain de performance, mais cela n'est pas une tâche facile et c'est un processus qui est très chronophage.

1.3 OBJECTIFS DE LA RECHERCHE

Les problèmes soulevés dans les sections précédentes permettent de mettre en avant une solution prometteuse susceptible d'assurer la mise à l'échelle des outils d'analyse de traces, tout en conservant une modularité sur le développement et le type de ces dernières.

Ce travail poursuit les objectifs spécifiques suivants :

1. Analyser les solutions actuelles qui répondent à une partie de la solution recherchée.
2. Proposer une solution qui respecte les contraintes des systèmes de calcul de haute performance.
3. Implémenter un prototype qui concorde avec la solution proposée.
4. Valider que la solution offre une mise à l'échelle adéquate tout en n'affectant pas la modularité du système pour accepter de nouvelles analyses.

1.4 CONTRIBUTIONS ORIGINALES

Les contributions de cette recherche se présentent comme suit :

1. Une architecture distribuée visant à accroître l'évolutivité des outils d'analyse.
2. Une extension du Trace Server Protocol pour permettre la réalisation d'analyses globales à l'échelle d'une grappe de calcul.
3. Une extension du Trace Server Protocol pour faciliter l'analyse de traces connectées en exploitant les événements réseau.
4. Une architecture distribuée modulaire conçue pour résoudre les problèmes de flexibilité rencontrés lors du développement d'analyses.

1.5 PLAN DU MEMOIRE

Le chapitre 2 présente l'état de l'art des architectures logicielles distribuées, les techniques de traçage, et les différentes solutions qui existent au niveau de la visualisation et l'analyse de

traces. Nous mettons une emphase sur les travaux actuels qui portent sur l'analyse distribuée des traces. Par la suite, nous présentons dans le chapitre 3 l'article principal de ce mémoire sous forme d'article scientifique qui présente en détail la solution et les résultats obtenus. Nous poursuivons avec une discussion générale dans le chapitre 4. Pour finir, nous conclurons ce mémoire au chapitre 5.

CHAPITRE 2 REVUE DE LITTÉRATURE

Ce chapitre présente plusieurs concepts et solutions qui aideront à comprendre la problématique et les objectifs de ce mémoire.

Pour commencer, nous allons présenter les différentes architectures distribuées qui peuvent résoudre de manière assez simple les problèmes d'évolutivité des applications monolithiques. Nous avançons une problématique des systèmes distribués qui est causée par l'interopérabilité de systèmes hétérogènes. Ensuite, nous allons présenter les différents outils de traçage utilisés aujourd'hui, les problèmes qu'ils résolvent, et le cadre où ces outils sont plus efficaces. Pour finir, nous allons discuter des différents outils de visualisation et d'analyse. Il existe plusieurs solutions actuellement disponibles, mais dans le contexte du calcul de haute performance, qui fait partie d'un des objectifs de ce mémoire, il existe des lacunes de mise à l'échelle.

2.1 Architectures distribuées

L'architecture logicielle est une représentation abstraite de la structure et de l'organisation d'un ou plusieurs systèmes informatiques. Un système est formé par un ensemble de composants logiciels, et les interactions entre ces composantes sont désignées sous le nom de "connecteurs". En utilisant des diagrammes et des schémas, l'architecture logicielle décrit comment un système doit être conçu pour répondre à des exigences techniques et fonctionnelles, tout en prenant en compte d'autres caractéristiques logicielles telles que la performance, la réutilisabilité, et la flexibilité [1]. Il devient de plus en plus crucial de prendre en considération des critères à long terme lors du choix d'une architecture, car une mauvaise architecture peut entraîner des coûts considérables et altérer la qualité du service offert. [2]

2.1.1 Client-serveur

Cette architecture est couramment évoquée lorsqu'il est question de systèmes distribués. Elle détient une importance historique considérable et demeure la plus largement adoptée. La figure 2.1 illustre une structure simple dans laquelle les processus endossent les rôles de clients ou de serveurs. Dans ce modèle architectural, deux entités se distinguent par leurs responsabilités : d'une part le client, dont l'objectif est d'accéder à une ressource, et de l'autre le serveur, chargé de gérer ces ressources [3]. Généralement, un serveur désigne une machine ou un processus dotés de puissance de calcul suffisante pour répondre à plusieurs requêtes concurrentes de clients. Les clients n'interagissent pas directement entre eux, mais

passent par l'intermédiaire du serveur. En particulier, les processus clients interagissent avec des processus serveur individuels sur des ordinateurs hôtes potentiellement séparés, ce qui leur permet d'accéder aux ressources partagées qu'ils gèrent. Par ailleurs, les clients ne sont pas nécessairement hébergés sur la même machine physique que le serveur, ce qui nécessite la mise en place d'un protocole de communication entre le client et le serveur. Les serveurs peuvent à leur tour jouer le rôle de clients vis-à-vis d'autres serveurs, comme le montre la figure. Par exemple, un serveur web agit en tant que serveur pour un navigateur web, mais devient lui-même client lorsqu'il interagit avec une base de données [4].

L'architecture client-serveur, également connue sous le nom de modèle client-serveur, est un concept de réseau qui répartit les tâches et les charges de travail entre les clients et les serveurs, qu'ils résident sur le même système ou soient connectés via un réseau informatique. Cette architecture se subdivise en plusieurs sous-catégories, notamment le client lourd et le client léger. Dans le premier cas, le client assume une grande part de la responsabilité en ce qui concerne les calculs et le traitement des données, ce qui réduit la tâche du serveur à la simple transmission des informations demandées. Dans le second cas, c'est l'inverse : le serveur prend en charge les calculs et le traitement des données, tandis que le client se charge principalement de l'affichage. Ces deux variantes servent à répondre au besoin de l'application et chacune mitige certains risques des deux côtés.

La communication dans une architecture client-serveur repose largement sur le modèle de protocole de requête-réponse. En réalité, ce protocole est une structure imposée sur un service sous-jacent de transmission de messages, conçue pour faciliter la communication entre les composants du système client-serveur. Ces protocoles impliquent généralement un échange bidirectionnel de messages, allant du client vers le serveur, puis du serveur vers le client. Le premier message contient à la fois une description de l'opération à exécuter sur le serveur et un tableau d'octets contenant les arguments correspondants. Le second message contient les résultats de l'opération, également codés sous forme de tableau d'octets [4]. Ce paradigme est relativement basique et trouve son utilité principalement dans les systèmes embarqués où les performances sont d'une importance cruciale. De plus, cette approche est également employée dans le protocole HTTP.

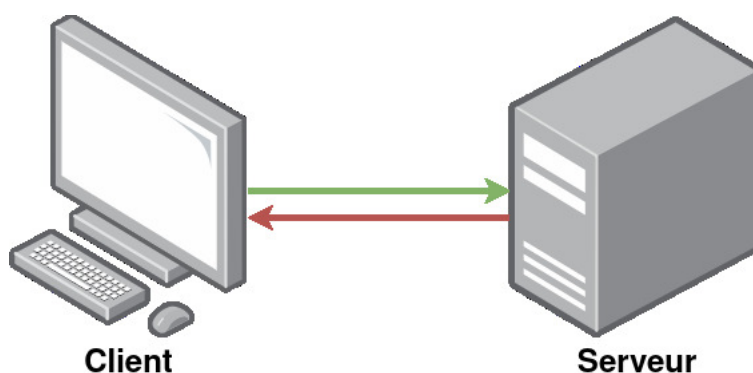


FIGURE 2.1 Client-Serveur

2.1.2 Pair-à-pair

Bien que le modèle client-serveur offre une approche directe et relativement simple du partage de données et d'autres ressources, il est peu évolutif. La centralisation de la fourniture et de la gestion des services qu'implique le placement d'un service à une seule adresse ne va pas bien au-delà de la capacité de l'ordinateur qui héberge le service, et de la bande passante de ses connexions réseau.

Dans l'architecture pair-à-pair, tous les processus impliqués dans une tâche ou une activité jouent des rôles similaires, interagissant de manière coopérative en tant que pairs, sans aucune distinction entre les processus client et serveur, ou les ordinateurs sur lesquels ils s'exécutent. Concrètement, tous les processus participants exécutent le même programme et offrent entre eux le même ensemble d'interfaces. La nécessité de distribuer les ressources partagées beaucoup plus largement, afin de partager les charges informatiques et de communication encourues, permettent de mobiliser un nombre beaucoup plus grand d'ordinateurs et de liens réseau. L'idée clé qui a conduit au développement des systèmes pair-à-pair est que les ressources réseaux et informatiques détenues par les utilisateurs d'un service pourraient également être utilisées pour prendre en charge ce service. Cela a pour conséquence utile que les ressources disponibles pour exécuter le service augmentent avec le nombre d'utilisateurs. La capacité matérielle et les fonctionnalités du système d'exploitation des ordinateurs de bureau d'aujourd'hui dépassent celles des serveurs d'hier, et la majorité d'entre eux sont équipés de connexions réseau haut débit permanentes. Le but de l'architecture pair-à-pair est d'exploiter les ressources (à la fois données et matériel) d'un grand nombre d'ordinateurs participants pour l'accomplissement d'une tâche ou d'une activité donnée. Des applications et des systèmes pair-à-pair ont été construits avec succès pour permettre à des dizaines, voire des centaines de milliers d'ordinateurs, de fournir un accès aux données et à d'autres ressources qu'ils

stockent et gèrent collectivement [4]. L'un des premiers exemples était l'application Napster permettant de partager des fichiers musicaux numériques [5]. Bien que Napster ne soit pas une pure architecture pair-à-pair, sa démonstration de faisabilité a abouti au développement du modèle architectural dans de nombreuses directions précieuses. Les systèmes de calcul à haute performance, les systèmes de partage de fichiers (BitTorrent) [6] et la chaîne de blocs (blockchain) [7] sont des exemples d'applications qui exploitent l'architecture pair-à-pair.

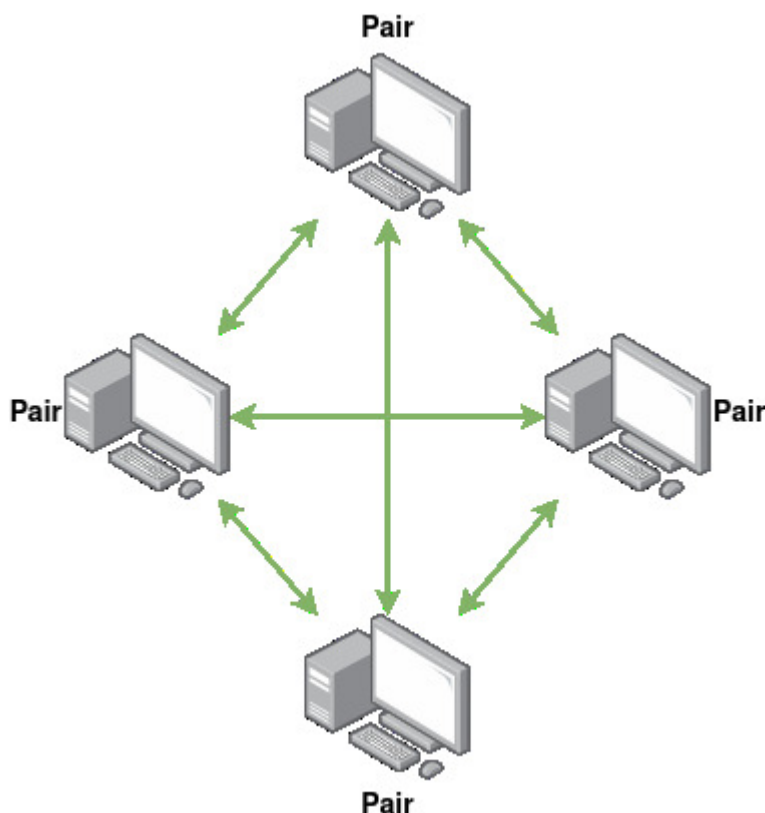


FIGURE 2.2 Pair-À-Pair

2.1.3 Maître-Ouvrier

L'architecture Maître-Ouvrier est une sorte d'architecture de système de calcul parallèle typique, dans laquelle existe un noeud maître. Le noeud maître est responsable de la segmentation de l'ensemble de la tâche, de la distribution des différentes sous-tâches après segmentation à chaque noeud ouvrier pour le calcul, et de la synthèse de tous les résultats de calcul des noeuds ouvrier pour obtenir le résultat final du calcul. Un schéma de l'architecture du système est présenté à la figure 2.3. [8]

Dans une telle architecture, chaque noeud ouvrier doit procéder au travail de calcul intense, tandis que le noeud maître s'occupe de coordonner les résultats finaux. Le noeud maître sera

principalement responsable des données d'entrée avant le calcul, de la sortie des résultats finaux après le calcul, ainsi que de la segmentation et de la distribution des sous-tâches pendant le processus de calcul. Par conséquent, en utilisant une telle architecture, il sera plus facile de gérer et de contrôler les noeuds de calcul, afin notamment d'équilibrer la charge déléguée à chaque noeud ouvrier. Lorsque la disposition des noeuds maître-ouvrier est adoptée, chaque noeud doit effectuer sa tâche et le calcul associé.

Pour le noeud ouvrier, chaque noeud peut être dans un état différent à différents moments du processus de calcul [9], les états suivants sont inclus :

- INCONNU signifie que l'état du noeud est inconnu, généralement avant l'initialisation du système ;
- LIBRE signifie que le noeud est inactif et disponible pour recevoir des tâches de calcul ;
- OCCUPÉ signifie que le noeud a commencé à calculer ;
- OCCUPÉ_À_LIBRE signifie que le noeud a terminé le calcul et attend les données renvoyées. Si les données sont retournées au noeud maître, le statut passera au statut LIBRE ;
- MORT signifie que le noeud n'est pas disponible, probablement en raison d'une panne de réseau entraînant une absence de réponse du noeud.

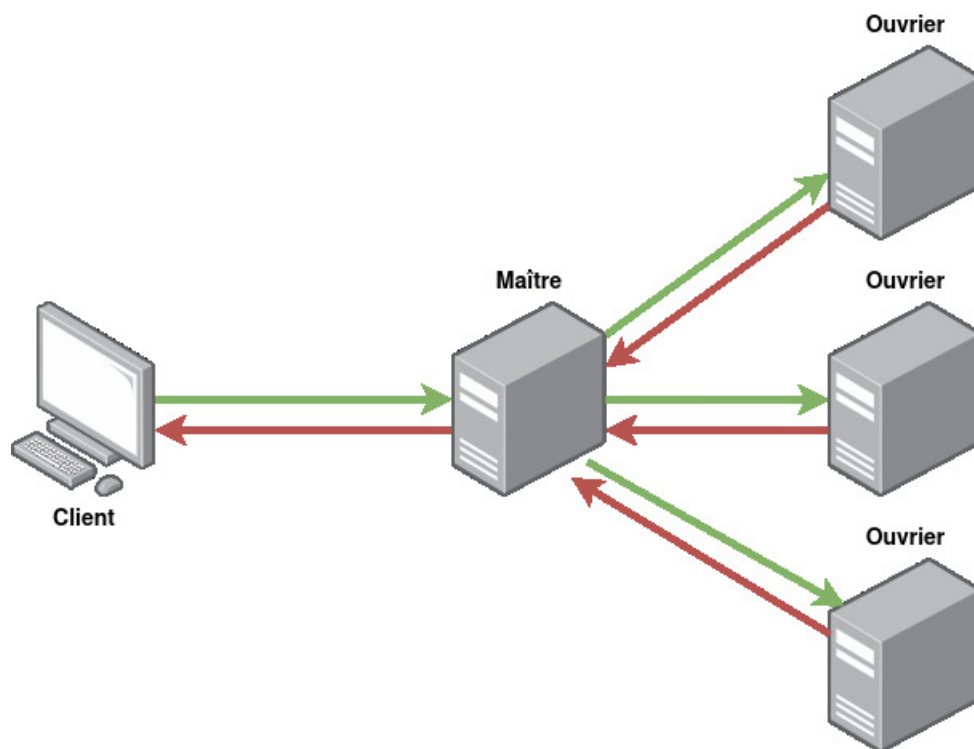


FIGURE 2.3 Architecture Maître-Ouvrier

En résumé, nous avons exploré 3 types d'architecture distribuée, chacun répondant à un besoin bien spécifique. L'architecture Client-Serveur est principalement utilisée pour répartir les tâches et les charges de travail entre les clients et les serveurs. L'architecture Pair-à-Pair est principalement utilisée pour que les ressources réseaux et informatiques détenues par les utilisateurs d'un service puissent également être utilisées pour prendre en charge ce service. L'architecture Maître-Ouvrier est principalement utilisée lorsque nous souhaitons facilement répartir le temps de calcul entre plusieurs machines, pour une mise à l'échelle horizontale selon les besoins. Toutes les architectures distribuées nécessitent la mise en place d'un protocole de communication. À ce moment, il devient important d'explorer les différentes solutions d'interopérabilité entre les systèmes pour les objectifs recherchés.

2.1.4 Sérialisation

La sérialisation des données est définie comme un processus d'encodage ou de traduction de données dans des formats de fichiers neutres, afin qu'elles puissent être transférables ou récupérables par d'autres applications [10]. Inversement, la désérialisation est de convertir un message par transformation, pour en obtenir un objet [11]. Ce flux peut être soit textuel, soit binaire, et il existe plusieurs formats de sérialisation permettant d'organiser et de structurer les données. Nous exposons ici quelques-uns des formats de sérialisation pris en charge par la majorité des langages de programmation couramment utilisés.

Format textuel

Le Extensible Markup Language (XML) utilise des balises pour délimiter les différentes parties des données. Les balises sont des éléments de marquage encadrés par des chevrons. Les données XML sont organisées sous forme d'une structure arborescente, également appelée arbre XML. Chaque balise peut contenir des balises imbriquées, créant ainsi une hiérarchie.

Listing 2.1 XML example

```
1 <person>
2   <name>John Doe</name>
3   <age>30</age>
4 </person>
```

Le JavaScript Object Notation (JSON) est un format de données fortement inspiré de la notation des objets du JavaScript [12]. JSON utilise une syntaxe basée sur des paires clé-valeur. Les données sont représentées sous forme de paires clé-valeur associées, séparées par des virgules et délimitées par des accolades. JSON prend en charge plusieurs types de données,

notamment les chaînes de caractères, les nombres, les objets, les tableaux, les booléens et les valeurs nulles. Les valeurs peuvent être imbriquées pour créer une structure complexe. La syntaxe simple de JSON le rend facile à lire et à écrire pour les humains, tout en restant facilement interprétable par les machines.

Listing 2.2 JSON example

```
1 {  
2     "name": "John Doe",  
3     "age": 30,  
4     "city": "Exampleville"  
5 }
```

YAML Ain't Markup Language (YAML), est un format de sérialisation de données qui met l'accent sur la lisibilité par l'homme. Il est souvent utilisé pour la configuration de logiciels, la représentation de données complexes et d'autres situations où la lisibilité est une priorité. YAML utilise une syntaxe simple basée sur l'indentation et l'utilisation des deux-points pour indiquer une paire clé-valeur. Les listes sont représentées par des tirets. YAML organise les données en utilisant l'indentation pour définir la structure. Les niveaux d'indentation définissent les niveaux hiérarchiques. YAML prend en charge différents types de données, tels que les chaînes de caractères, les nombres, les booléens, les listes et les objets.

Listing 2.3 YAML example

```
1 person:  
2     name: John Doe  
3     age: 30  
4     city: Exampleville
```

Format binaire

Le Binary JSON (BSON) est un format binaire utilisé pour représenter des documents JSON de manière binaire. Contrairement à JSON qui est un format de texte, BSON est conçu pour être plus efficace en termes d'espace de stockage et de rapidité de traitement, ce qui le rend adapté pour la transmission de données entre applications et pour le stockage de données. Pour cela, BSON est utilisée au sein de MongoDB pour le transfert et le stockage des données [13]. BSON prend en charge divers types de données, y compris les types de données de base tels que chaînes de caractères, nombres, booléens, tableaux, objets, ainsi que des types spécifiques comme les dates, les identifiants, les données binaires, etc. Contrairement

à JSON, BSON conserve l'ordre des champs dans un document. Cela peut être important dans certains contextes où l'ordre des champs a une signification.

Les Protocol Buffers (Protobuf), le format de sérialisation de Google, sont un format binaire avec un langage de description d'interface (Interface Definition Language (IDL)) [14]. Protobuf prend en charge une multitude de langages de programmation très populaires et continue de s'étendre sur des langages récemment développés. L'utilisateur spécifie les structures de données à partager ainsi que le type de chaque propriété (entier, chaîne de caractères, etc.) dans un fichier .proto. Ensuite, le compilateur protoc génère le code source des classes ou des composants logiciels responsables de l'envoi et de la réception de ces structures de données. Pour l'encodage des entiers, Protobuf utilise la technique des *varints*, qui implique l'utilisation d'un nombre variable d'octets. Cette méthode permet de représenter un nombre en éliminant les octets les plus significatifs nuls. Un avantage que Protobuf offre est qu'il génère le code client et serveur dans le langage dont vous avez besoin. Cela simplifie énormément la maintenance du code qui sert d'interface entre le client et le serveur.

Apache Thrift est un cadre de communication et de sérialisation de données open source développé par Apache Software Foundation qui offre un format de sérialisation binaire. Il a été initialement créé par Facebook et a été mis en code source libre en 2007 [15]. Thrift offre deux formats de sérialisation : TBinaryProtocol et TCompactProtocol [16]. Thrift est conçu pour faciliter la communication entre différentes applications écrites dans des langages de programmation différents en permettant la définition de structures de données et de services partagés. Dans le même sens que Protobuf, Thrift prend en charge la génération de code pour plusieurs langages de programmation. Thrift utilise un protocole binaire efficace pour sérialiser les données. Cela permet de réduire la taille des messages échangés et d'optimiser les performances de la communication. Les différences de performance entre Thrift et ProtoBuf sont essentiellement négligeables et les deux offrent en grande partie les mêmes avantages [17]

En résumé, la décision entre la sérialisation d'un format textuel ou d'un format binaire devient primordial. Les formats textuels offrent un avantage durant la période de développement, car il est plus simple de confirmer l'exactitude de l'information transmise entre les systèmes. Bien que le format textuel soit intéressant lors de la période de développement, les problèmes de performance des formats textuels sont un inconvénient dans les systèmes en production, et surtout dans les systèmes de calcul de haute performance. À l'opposé, le format binaire offre une performance bien plus fiable pour les systèmes mis en production et les systèmes de calcul de haute performance, mais complique le développement, car il est plus difficile de vérifier l'exactitude du format binaire transmis entre les systèmes. Il est possible d'avoir un cadre qui utilise les avantages de chaque format au moment opportun. Dans cet esprit, Protobuf

spécifie une syntaxe pour la représentation des données protobuf sous forme de texte pour faciliter les tests [18].

2.1.5 Interopérabilité de systèmes hétérogènes

L'interopérabilité se réfère à la capacité d'un système à partager des données avec un autre système, même s'ils n'utilisent pas le même langage de programmation. Pour ce faire, les systèmes peuvent interagir de deux manières : soit de manière indirecte en utilisant un référentiel commun, soit directement via un protocole de communication réseau. Dans le premier cas, les systèmes se coordonnent en utilisant des éléments tels qu'une base de données, un système de fichiers distribués ou une zone de mémoire partagée. Dans le second cas, les systèmes communiquent en utilisant des protocoles tels que TCP, UDP ou HTTP et échangent des données dans un format compréhensible mutuellement. Cette section se concentre exclusivement sur les méthodes d'interaction directe entre les systèmes.

REST

Le standard REST [19] est un ensemble de règles et de conventions permettant de créer et d'interagir avec des services Web ou des applications Web. REST est un style architectural pour la conception d'applications en réseau qui se repose sur le protocole de communication HTTP [20].

Le standard REST est largement utilisé pour créer des serveurs Web en raison de sa simplicité, de son évolutivité et de sa facilité d'intégration. REST est devenu un standard pour les applications Web et est utilisé dans une large gamme de services, des plateformes de médias sociaux aux sites Web de commerce électronique et aux applications mobiles. Voici quelques principes et concepts clés associés au standard REST : ressources, méthodes HTTP, communication sans état, interface uniforme, représentation [20].

Ressources : Les données et les fonctionnalités sont exposées en tant que ressources. Ces ressources sont identifiées par des URL, et chaque ressource possède un URL unique. Par exemple, un API REST pour une librairie peut avoir des ressources telles que `/livres`, `/auteurs` et `/commandes`.

Méthodes HTTP : Les méthodes HTTP permet d'effectuer des opérations sur les ressources. Les méthodes HTTP les plus couramment utilisées dans REST sont :

- GET Utilisé pour récupérer des données d'une ressource ;
- POST Utilisé pour créer une nouvelle ressource
- PUT Utilisé pour mettre à jour une ressource existante ou la créer si elle n'existe pas ;

- DELETE Utilisé pour supprimer une ressource ;
- PATCH Utilisé pour effectuer des mises à jour partielles d'une ressource.

Communication sans état : Une communication sans état signifie que chaque requête doit contenir toutes les informations nécessaires pour comprendre et traiter la requête. Le serveur ne stocke aucune information de session, spécifique au client, entre les requêtes. Cela implique que l'effet d'une requête exécutée doit être idempotent. Cette simplicité rend les API REST évolutives et faciles à mettre en cache.

Interface Uniforme : Une interface uniforme inclut l'utilisation de codes d'état HTTP pour les réponses (par exemple, 200 pour succès, 404 pour non trouvé, 500 pour les erreurs du serveur) et l'appellation des URIs doit être claire et cohérente pour les ressources.

Représentation : Les données sont transférées sous forme de représentations, qui peuvent être dans différents formats, tels que JSON, XML, HTML ou autres. Les clients peuvent demander le format de représentation qu'ils préfèrent en définissant les en-têtes HTTP appropriés.

GraphQL

Fondamentalement, GraphQL [21] offre aux clients la possibilité d'interroger une base de données en utilisant un schéma comme représentation. Cette approche marque une différence majeure par rapport aux standards REST. Dans REST, les serveurs exposent une liste de point d'accès que les clients peuvent appeler, tandis que les serveurs GraphQL mettent à disposition une base de données que les clients peuvent interroger avec un seul point d'accès. Une base de données GraphQL est définie par un schéma, qui peut être considéré comme un réseau de relations [22]. Dans ce réseau, les noeuds représentent des objets qui définissent un type et comprennent une liste de champs. Chaque champ a un nom et un type associé. Des liens se forment lorsque les objets définissent des champs dont les types sont d'autres objets. GraphQL propose un langage de définition de schéma (DSL) simple. Enfin, les schémas incluent généralement un type pré-défini appelé "Query", qui sert de point d'entrée aux API GraphQL. Un objet de type Query expose les types d'objets qui peuvent être interrogés par les clients, ainsi que les arguments nécessaires pour effectuer ces requêtes [23].

Nous pouvons constater que GraphQL résout 2 problèmes qui sont sur-télécharger et sous-télécharger. Sur-télécharger est quand le client fait une requête et reçoit une réponse du serveur contenant des informations superflues, ce qui force le client à implémenter la logique de son côté pour les filtrer. La conséquence est qu'une grosse quantité de données inutiles est transmise, gaspillant la bande passante et potentiellement augmentant le temps de transmission, d'encodage et de décodage. Sous-télécharger est le phénomène inverse, quand le point d'accès au service ne fournit pas suffisamment d'information et donc force le client à faire

plusieurs requêtes pour récolter les ressources recherchées. Le problème de sous-télécharger peut être réglé facilement en suivant plusieurs points d'accès en une seule requête. Régler le problème de sur-télécharger, en revanche, demande des techniques plus sophistiquées, car le besoin de filtrage peut facilement changer d'une application à l'autre [24].

GraphQL est relativement nouveau par rapport à REST. Brito et al [23] ont examiné un article de blog (littérature grise) pour comprendre les avantages et les caractéristiques clés de GraphQL, tels que perçus par les praticiens. Ils concluent que GraphQL est plus efficace en termes de réduction de la taille en octets (99%) des documents JSON. Hartig et Perez [22], [25] fournissent la sémantique d'une requête formelle et analysent ensuite le langage. Leurs résultats montrent que GraphQL est un langage relativement moins complexe. Wittern et coll. [26] a évalué la faisabilité d'encapsuler automatiquement GraphQL pour les API REST existants avec la spécification Open API (OAS). Sur cette base, ils ont étudié les gains obtenus entre GraphQL et REST. En outre, ils discutent des défis liés à la création de tels encapsuleur, tels que l'assainissement des données, l'authentification ou le traitement des requêtes imbriquées.

PromQL

En ce qui concerne les outils de traçage, très peu interagissent avec une base de donnée, mais accèdent plutôt directement des fichiers de trace. Il reste que certains outils emmagasinent les données de trace dans un format spécifique et les déposent ensuite dans une base de donnée, ou les rendent accessibles à travers un langage de requête semblable à ceux utilisés pour les bases de données. Un exemple concret serait PromQL [27]. PromQL est le langage de requête exposé par le serveur Prometheus pour l'accès aux traces sauvegardées. La base de données de Prometheus est organisée sous forme de séries temporelles. Par conséquent, une requête triviale consiste à extraire le point de donnée enregistré pour chaque instant dans le temps. Ensuite, pour filtrer la réponse, il est possible de spécifier les étiquettes (mots-clés) nécessaires. Ainsi, seuls les points de données dotés de ces étiquettes seront renvoyés. De plus, Prometheus prend en charge divers opérateurs d'agrégation. Par exemple, si l'on souhaite calculer la somme des points de données renvoyés, cela peut être réalisé grâce à ces opérateurs d'agrégation.

Remote Procedure Call

Les Remote Procedure Call (RPC) [28] permettent à une machine A d'appeler des procédures sur une machine distante B, comme si elles étaient disponibles dans l'espace d'adressage local [4].

JSON-RPC [29] est un protocole RPC sans état, qui utilise le format JSON comme format d'échange et est conçu pour être simple. Une requête du client contient les champs suivant [30] :

- `jsonrpc` : Permet de spécifier la version du protocole ;
- `method` : Contient le nom de la méthode à exécuter ;
- `params` : Permet de lister les valeurs des paramètres de la méthode ;
- `id` : Permet d'associer une réponse du serveur à la requête du client.

Une requête doit toujours recevoir une réponse. Le protocole supporte également les notifications, un message qui n'a pas besoin d'avoir de réponse. Finalement, JSON-RPC ne spécifie pas la couche de transport par laquelle les messages sont acheminés.

gRPC (Google RPC) [31] est une infrastructure multiplateforme, indépendante du langage et de la plate-forme, à usage général, utilisée par la compagnie Google et rendue publique en 2015. Elle peut générer automatiquement des stubs client et serveur idiomatiques pour un service dans une variété de langues et plates-formes. gRPC utilise *ProtoBuf* un protocole sous-jacent qui constitue un mécanisme flexible, efficace et automatisé pour la sérialisation binaire des données structurées. L'utilisateur définit des services dans un fichier `.proto` et le compilateur protoc génère les API qui seront utilisés par le client et le serveur.

gRPC offre une meilleure performance que le protocole JSON-RPC [32]. Cette différence de performance est causée par le format de transmission. JSON-RPC utilise un format textuel (JSON) pour la transmission et gRPC utilise un format binaire (ProtoBuf). Le format binaire produit un encodage plus petit, et accélère la phase de décodage, puisque la conversion nécessaire est minimale. Le temps de transmission réseau se voit également réduit. Cela donne un avantage à gRPC lorsque la performance est un critère important.

Language Server Protocol

Le *Language Server Protocol* (LSP) [33] est un protocole de communication basé sur JSON-RPC, développé par Microsoft, qui permet l'interaction entre un éditeur de programme ou un environnement de développement intégré (EDI) et un serveur de langage de programmation. Un serveur de langage fournit diverses fonctionnalités telles que l'autocomplétion de code, la coloration syntaxique, la navigation vers la définition, et la signalisation des erreurs dans le code. L'objectif principal du LSP est de standardiser la manière dont ces fonctionnalités sont mises à disposition des EDI. Cela permet de rendre l'implémentation d'un EDI indépendante du serveur de langage, favorisant ainsi la réutilisation des serveurs de langage dans différents EDI. Initialement conçu pour VS Code [34], le LSP a évolué grâce à la contribution d'organisations telles que Red Hat et Codeenvy. Plusieurs entreprises, dont la fondation Eclipse,

Github, Sourcegraph et Red Hat, ont adapté leurs outils de développement populaires, tels qu'Eclipse, Atom, IntelliJ, Emacs ou Sublime, pour prendre en charge le protocole [35]. Actuellement, plus de 200 langages de programmation, y compris Java, C/C++, C#, PHP, Python et JavaScript, prennent en charge le LSP [36].

Le LSP définit trois types de messages : les requêtes, les réponses et les notifications. Les requêtes sont initiées par le client, en utilisant des appels de procédures JSON-RPC, et le serveur répond à chaque requête par une réponse. Les notifications peuvent être envoyées par le serveur ou le client, sans le retour d'une réponse. La figure 2.4 ci-dessous illustre un exemple d'interaction et d'échange de messages entre un EDI et un serveur de langage, pendant une session d'édition de code. De plus, le LSP supporte l'annulation de requêtes et les rapports de progression. Grâce à cela, il est possible d'annuler des requêtes en cours et d'obtenir des réponses partielles.

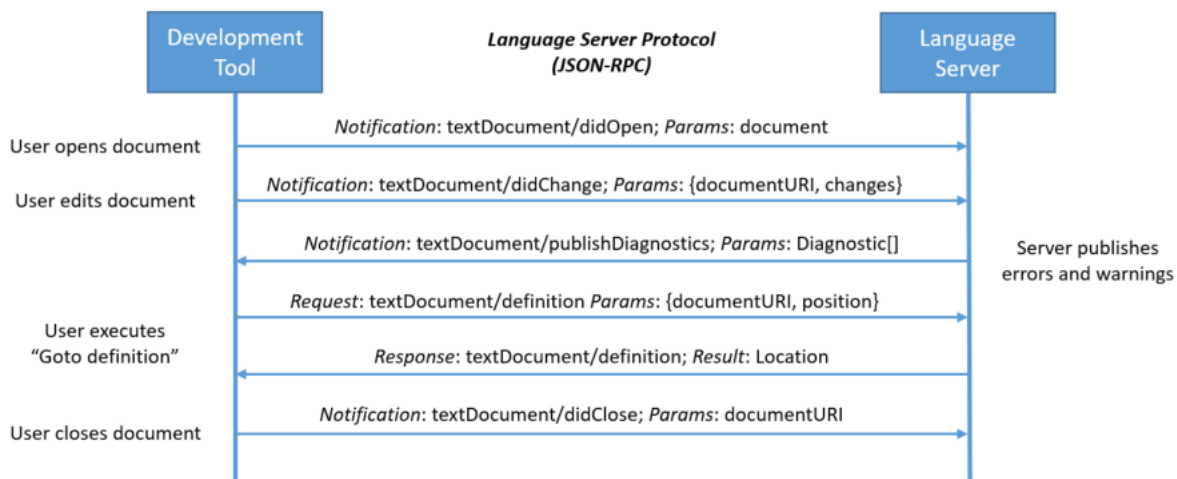


FIGURE 2.4 Diagramme de séquence du protocole de serveur de langage

Source: <https://microsoft.github.io/language-server-protocol/overviews>

Les interactions entre le client et le serveur se basent sur des modèles indépendants du langage de programmation et de l'EDI [37]. Par exemple, lorsqu'un client souhaite accéder à la définition d'une procédure, il échange un Uniform Resource Identifier (URI) avec le serveur pour localiser un document sur disque et une position dans ce document. Le LSP permet également au serveur de signaler quelles fonctionnalités il prend en charge, laissant au client la gestion de la durée de vie du serveur.

Monto [38] est une architecture alternative à LSP qui connecte un EDI à des composants offrant des fonctionnalités de langage, adoptant une approche orientée service. Chaque fonctionnalité de langage, comme l'autocomplétion, est fournie par un service qui peut être décomposé

en sous-services. L'architecture permet à l'EDI de connaître les fonctionnalités disponibles enregistrées auprès d'une composante appelée broker. Un module d'extension dans l'EDI gère la communication avec le broker. La figure 2.5 illustre une vue d'ensemble de l'interaction entre les composants de l'architecture Monto.

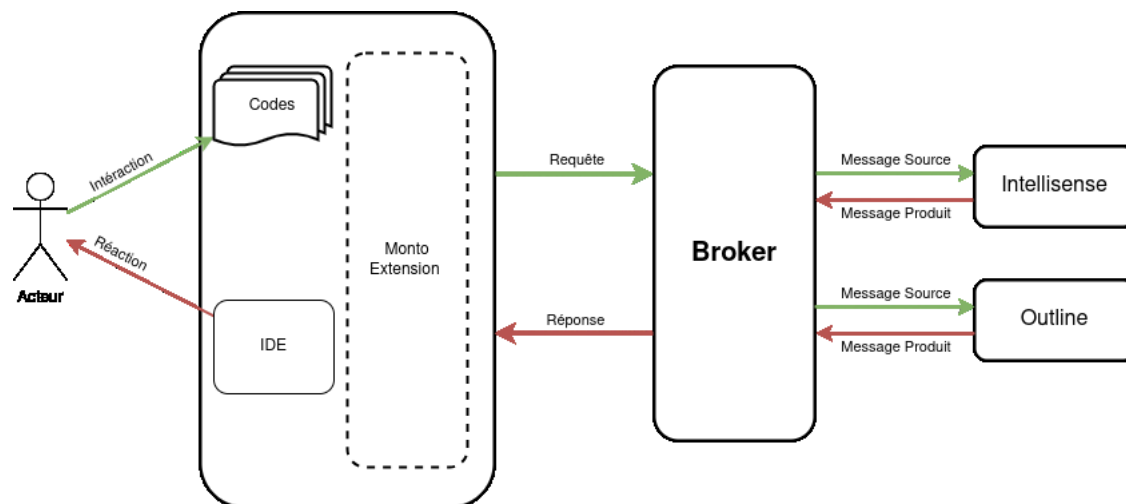


FIGURE 2.5 Architecture Monto

L'architecture Monto définit deux types de messages : les messages source et les messages produit. Les messages source, envoyés du module d'extension de l'EDI vers le broker, contiennent un identifiant unique, le nom du fichier, le langage de programmation et le contenu du fichier. Le broker distribue les messages source aux services appropriés et répond au module d'extension de l'EDI lorsque les opérations sont terminées. Les messages produits encapsulent une représentation intermédiaire générique, indépendante du langage et de l'EDI. Les communications sont effectuées via ZeroMQ [39], une bibliothèque de messagerie asynchrone de haute performance, avec l'encodage des messages au format JSON. Cette approche permet d'avoir un serveur sans état et n'exige pas de conserver une copie des documents ouverts, contrairement au LSP. Cependant, l'envoi du contenu complet d'un fichier au format JSON peut ne pas être optimal en termes de transfert de données sur le réseau.

Language Server Index Format Specification

Language Server Index Format (LSIF) [?] est un protocole et un format de données utilisés dans le cadre du LSP. LSIF a été développé pour améliorer les performances et l'efficacité de la communication entre l'éditeur de code et le serveur de langage. Il s'agit de générer et de stocker un index des informations du projet source, ce qui permet de réduire les délais de réponse lors de l'interaction avec le code source.

1. Génération de l'index : Le serveur de langage génère un index LSIF en analysant le projet source. Cet index contient des informations sur la structure du code, les définitions de fonctions, les variables, les références croisées, etc.
2. Stockage de l'index : Une fois généré, l'index LSIF est stocké dans un fichier ou une base de données. Ce fichier est souvent enregistré à un emplacement spécifique du projet.
3. Utilisation par l'éditeur de code : Lorsque l'utilisateur interagit avec le code source dans l'éditeur, celui-ci peut interroger l'index LSIF pour obtenir des informations pertinentes, telles que des suggestions de complétion, la documentation, ou pour effectuer des actions de réingénierie (*refactoring*). Cela permet à l'éditeur de code de répondre plus rapidement aux demandes de l'utilisateur, car il n'a pas besoin de réanalyser tout le projet à chaque interaction.

LSIF est particulièrement utile pour les projets de grande envergure, où l'analyse statique du code peut être coûteuse en termes de ressources. En utilisant un index pré-généré, les performances de l'éditeur de code s'améliorent significativement.

En résumé, LSIF est un format d'indexation utilisé dans le cadre du protocole LSP pour améliorer les performances de l'interaction entre les éditeurs de code et les serveurs de langage, lors de la programmation. Il permet de stocker des informations structurées sur le code source, ce qui facilite la fourniture de fonctionnalités avancées aux développeurs.

2.1.6 Trace Server Protocol

Le Trace Server Protocol (TSP) [40] a été développée lors de l'introduction de l'architecture client-serveur dans l'outil d'analyse et de visualisation de traces *Trace Compass* par l'équipe de Ericsson en 2018. Ce protocole suit la philosophie des REST API et utilise actuellement JSON comme format d'échange de données. Cependant, le travail de Chen Kuang Piao a suggéré qu'il serait possible d'obtenir des améliorations significatives en termes de taille des messages et d'efficacité de sérialisation/désérialisation en utilisant Protobuf à la place. En effet, la sérialisation/désérialisation devient un facteur très important dans le temps de réponse, lorsque nous essayons de traiter des traces énormes [41]. Il devient alors très important d'évaluer les solutions existantes qui nous permettraient de réduire l'impact de la sérialisation/désérialisation. Cet impact devient encore plus important lorsque nous essayons de transitionner vers une architecture distribuée qui risque de nécessiter plusieurs communications réseau, et donc plusieurs sérialisations/désérialisations pour une seule requête.

La transition vers un autre protocole est envisageable, car REST, en tant qu'approche, n'est pas lié à un format de données spécifique, mais plus aux ressources accessibles par les différents

points d'accès. En plus des points d'accès permettant de récupérer des données de traces ou de sessions d'analyse, les points d'accès pour obtenir le modèle de visualisation des traces sont organisés par type de visualisation. Il existe des points d'accès dédiés aux analyses sous forme de graphes XY, de graphe temporel et de tableau. Ce protocole est encore en évolution alors que le développement de Trace Compass se poursuit et que de nouvelles analyses et visualisations s'ajoutent régulièrement. Il est alors important de considérer que le protocole risque de diverger de ses objectifs primaires en terme de principe, comme le standard RESTful.

Dans le protocole TSP, il est possible d'ajuster la résolution des données, ainsi que les champs de données souhaités, en spécifiant les points de temps ciblés en tant que paramètres [42]. Cette pratique n'est pas courante dans les normes REST qui n'impliquent généralement pas de filtrage des réponses dans le corps de la requête, mais plutôt dans les paramètres de la requête. La principale exception est que nous retrouvons des paramètres de filtrage dans le corps de la requête lorsque nous rencontrons un problème lié à la limite du nombre de caractères possible dans un URL.

Le protocole TSP revêt une importance particulière dans cette étude. En effet, il est utilisé par Trace Compass, l'outil pour lequel nous recherchons une architecture plus efficace afin de mettre à l'échelle l'analyse et la visualisation de traces. Ceci doit permettre de prendre en charge des traces immenses générées par les supercalculateurs exaflopiques, constitués de plusieurs milliers de noeuds.

2.2 Traçage

2.2.1 Outils de traçage

Le traçage implique une utilisation spécialisée de la journalisation qui enregistre les événements d'exécution concernant l'exécution d'un programme. Le traçage est utilisé pour détecter et identifier les problèmes de performance, les problèmes de comportement ou tout autre problème concernant l'utilisation des ressources comme les coeurs de calcul, la mémoire ou les entrées-sorties. Le traçage peut être effectué à plusieurs niveaux, notamment l'espace utilisateur, le noyau, le matériel, l'hyperviseur, le réseau, etc. Le logiciel qui capture ces événements est appelé traceur. Pour capturer ces événements, les points de trace doivent être insérés de manière statique ou dynamique. Dans le cas où le point de trace est inséré de manière statique, le code doit être modifié pour inclure les macros de traçage aux endroits ciblés et doit être recompilé. Il est aussi possible pour le compilateur d'insérer automatiquement des points de trace statiques à des endroits spécifiques, comme les entrées et sorties de fonctions ou les accès en mémoire. Dans le cas où le point de trace est inséré dynamiquement, les points

de trace sont ajoutés dynamiquement au fichier binaire d'un programme compilé, voire même en cours d'exécution [43]. Lors de l'exécution du programme, des événements seront émis et un traceur les capturera. À ce stade, un traceur produira un fichier de trace, organisé dans un format spécifique. Étant donné que le traceur doit s'exécuter en même temps que l'application principale, cela aura un impact sur le temps d'exécution de l'application principale. C'est pourquoi un traceur doit idéalement avoir une surcharge d'exécution minimale. Lorsque le traceur a fini de produire un fichier de trace, nous pouvons appliquer une analyse de trace.

Perf

Perf constitue un ensemble d'outils d'analyse de performance intégré au noyau du système d'exploitation Linux, permettant de tracer les compteurs de performance matériels et logiciels [44]. Cet ensemble d'outils robustes offre une variété d'informations relatives aux performances du système d'exploitation. Leur gamme de fonctionnalités comprend l'échantillonnage basé sur des compteurs de performances, des points d'instrumentation spécifiques, la journalisation pour enregistrer les données collectées et la visualisation à travers des profils textuels. Ces compteurs mesurent divers éléments tels que le nombre d'instructions exécutées, les fautes de cache, les fautes de page et les changements de contexte. Les données collectées peuvent être enregistrées dans un fichier au format texte ou directement affichées sur la console sous forme de statistiques. Cependant, perf utilise un format de fichier monolithique et ne propose pas une méthode évolutive et conviviale pour analyser et visualiser les délais résultants. Souvent, perf est utilisé pour la surveillance des performances, puis la trace résultante est convertie dans un format de fichier différent, afin de pouvoir utiliser des outils de visualisation de chronologie [45].

Strace

Strace est un utilitaire de traçage, de diagnostic et de débogage en espace utilisateur avec une interface de ligne de commande traditionnelle pour Linux [46]. Il est utilisé pour surveiller et altérer les interactions entre les processus et le noyau Linux, qui incluent les appels système, la transmission de signaux et les changements d'état du processus. Le fonctionnement de strace est rendu possible par la fonctionnalité du noyau connue sous le nom de *ptrace* [47]. Son utilisation ne nécessite ni recompilation ni reconfiguration du code source de l'application. Les traces produites suivent un format textuel. Ce format textuel facilite la lisibilité des traces pour les utilisateurs. Cependant, sa souplesse est restreinte, car la gamme d'événements traçables reste fixe. De plus, le coût additionnel du traçage est notable (e.g., plusieurs fois plus lent), impliquant de multiples changements de contexte, entre l'application et strace, à

chaque appel système [48].

Ftrace est un traceur interne conçu pour aider les développeurs et les concepteurs de systèmes à découvrir ce qui se passe à l'intérieur du noyau Linux. Il peut être utilisé pour déboguer ou analyser les latences et les problèmes de performance qui surviennent en dehors de l'espace utilisateur [49]. FTrace est un outil de noyau pour tracer l'exécution de fonctions dans le noyau Linux. FTrace ajoute la possibilité de tracer l'entrée et la sortie de chaque fonction. Cela permet de mesurer la durée de la fonction, ce qui ajoute un outil particulièrement puissant pour trouver les zones chronophages d'exécution du noyau [50]. Bien que ftrace soit généralement considéré comme un traceur de fonctions, il s'agit en réalité d'un cadre composé de plusieurs utilitaires de traçage variés. Il existe un suivi de latence pour examiner ce qui se passe entre les interruptions désactivées et activées, ainsi que pour la préemption, et à partir du moment où une tâche est réveillée jusqu'à ce qu'elle soit réellement exécutée. L'une des utilisations les plus courantes de ftrace est le traçage d'événements. Tout au long du noyau, des centaines de points d'événements statiques peuvent être activés via le système de fichiers tracefs, pour voir ce qui se passe dans certaines parties du noyau. Ftrace offre la possibilité de collecter les informations sur les interruptions et les événements d'ordonnancement. Avec ces informations, il devient possible de détecter des problèmes de latence [51]. Les fichiers de configuration et les traces produites par ftrace se trouvent sur le système de fichier debugfs [52]. Le format du contenu des traces est textuel et lisible par l'utilisateur.

Linux Trace Toolkit next generation

Le Linux Trace Toolkit next generation (LTTng) est un traceur à code source ouvert développé initialement à Polytechnique Montréal en 2006 [53]. Il permet non seulement de tracer le noyau, mais également les programmes en espace utilisateur avec le module LTTng-UST. LTTng utilise des tampons circulaires dans lesquels les événements sont d'abord insérés avant d'être consommés et écrits sur disque. Pour les tampons circulaires, LTTng n'utilise pas les pages. Au lieu de cela, il utilise des sous-tampons par CPU. L'avantage de cette approche est que la taille d'un événement peut être supérieure à la taille d'une page, et que l'impact sur les performances des barrières de mémoire est mieux amorti (les barrières ne sont requises qu'aux limites du sous-tampon plutôt qu'aux limites des pages). LTTng a été conçu et implémenté dans l'objectif d'avoir une surcharge minimale sur les machines, tout en évitant les interruptions. En effet, lorsqu'un sous-tampon est plein, il est vidé par un processus séparé, sans avoir besoin de bloquer l'application tracée. En outre, l'ajout des données de trace dans les tampons se fait avec des opérations atomiques, sans verrous, et de ce fait les points de trace peuvent être placés même dans les contextes d'interruptions comme les signaux ou

les NMI (*Non Maskable Interrupt*). Comme objectif supplémentaire, LTTng vise à simplifier l’extension de son instrumentation. LTTng peut être utilisé dans les domaines embarqués pour réduire les problèmes de contraintes dans le noyau et également dans les application en temps réel [54]. LTTng permet de recueillir des données similaires à celles de ftrace et Perf, présentant un surcoût légèrement réduit [48]. De plus, il offre des fonctionnalités supplémentaires telles que la gestion de sessions de traçage concurrentes, le traçage en mode utilisateur, et la rotation des fichiers de trace.

HPCToolkit

HPCToolkit [55] est une suite d’outils d’analyse des performances qui se concentre sur l’échantillonnage d’applications parallèles, prenant en charge MPI et OpenMP. Il utilise la synthèse pour fournir des profils de performances. De plus, une vue chronologique basée sur la journalisation est disponible. HPCToolkit comprend des informations provenant d’échantillons de pile d’appels, qui sont traités dans l’espace utilisateur avec des techniques de déroulement sophistiquées, et des compteurs de performances de l’interface de programmation d’applications.

MPI Parallel Environment

MPI Parallel Environment (MPE) est un ensemble de logiciels pour MPI (Message Passing Interface). Dans cet ensemble de logiciels, nous retrouvons des bibliothèques de profilage pour collecter des données sur le comportement de MPI. Lors de l’utilisation de ces librairies, elles s’occupent de générer les journaux d’activités pour ensuite les visualiser durant l’exécution du programme utilisateur. Chaque noeud de calcul se lie avec les librairies de traçage MPE et génère une fichier de trace. Il en résulte plusieurs fichiers de trace contenant des informations complémentaires avec des horloges désynchronisées. Cela affecte l’ordre dans lequel les événements seront affichés, si aucun traitement n’est effectué [56].

Score-P

Score-P est une suite d’outils hautement évolutive, utilisée pour le profilage et le suivi des événements des applications de calcul à haute performance (High Performance Computing, ou HPC). Score-P est en mesure d’instrumenter les paradigmes procéduraux, OpenMP et MPI. Pour être en mesure d’instrumenter une application, l’utilisateur doit recompiler l’application en utilisant l’interface de commande de Score-P. Score-P détecte le paradigme et insère les différents points de trace selon ce paradigme [57]. Score-P permet de produire les traces au

format Cube4 et OTF2 (Open Trace Format) qui est un format binaire [58].

Open SpeedShop

Open SpeedShop est un ensemble d'outils multiplateforme dédié à l'analyse de performance des applications HPC fonctionnant sur les systèmes utilisant des processeurs Intel, AMD, ARM, ou PowerPC [59]. Cet outil rassemble des informations sur les compteurs de performance, les opérations d'entrée-sortie et l'utilisation de la mémoire. De plus, il trace et profile les appels de fonctions MPI, OpenMP et CUDA. Les traces générées sont au format OTF. Il n'est pas nécessaire de recompiler l'application de l'utilisateur pour récolter des informations au niveau des fonctions ou des bibliothèques. Malheureusement, le projet ne semble plus avoir de contribution sur Github depuis 2 ans [60].

2.2.2 Outils de traçage distribués

Dapper est l'infrastructure de traçage de systèmes distribués de Google [61]. Dapper est le premier à avoir formalisé et popularisé le traçage distribué, comme Docker l'a fait pour l'utilisation des conteneurs. Sa conception est similaire à celle d'autres traceurs comme Magpie et X-Trace. Dapper est capable de limiter son instrumentation à un petit nombre de bibliothèques, puisque l'environnement de chacune des applications chez Google est très similaire. Dapper trace récursivement les requêtes utilisant des Remote Procedure Calls (RPC) et leur assigne un numéro d'identifiant unique. Chaque événement est alors associé à un numéro d'identifiant unique, ce qui permet de regrouper par la suite les événements par requête. Cela permet à Dapper d'être en mesure de recréer le chemin de contrôle d'une requête avec très peu de changements dans l'application. Même si Dapper se concentre sur l'environnement de Google, qui utilise principalement le protocole RPC, il n'y est pas contraint. Il est possible de générer des traces dans des applications utilisant des sessions SMTP, des requêtes HTTP et des requêtes vers des serveurs SQL.

OpenTracing, inspiré de Dapper, est un projet de la Cloud Native Computing Foundation dont l'objectif est de standardiser l'instrumentation des systèmes distribués pour le traçage. Dans OpenTracing, le concept de trace permet de suivre l'état d'une transaction ou une requête à travers le système. Pour ce faire, ce dernier fournit un API et décrit une trace comme étant un graphe orienté acyclique d'intervalles appelés *span*. Un *span* est un intervalle de temps qui représente le temps écoulé pour l'aboutissement d'une action. Finalement, OpenTracing étant une spécification, les développeurs peuvent ajouter ou changer très facilement d'implémentation.

OpenTelemetry, la fusion d'OpenTracing et OpenCensus [62], est un projet du Cloud Native Computing Foundation dont l'objectif est de facilement instrumenter vos applications ou systèmes, quels que soient leur langue, leur infrastructure ou leur environnement d'exécution [63]. OpenTelemetry, focalise sur la génération, la collection, la gestion et l'exportation de données de télémétrie. Dans OpenTelemetry, divers concepts ont été définis : le concept de *span* contient la description d'une mesure de latence à un moment précis. Il est livré avec une variété d'attributs visant à fournir une description exhaustive de l'action mesurée. La notion de ressource est également définie [64]. Dans le même sens qu'OpenTracing, OpenTelemetry utilise le concept de trace pour suivre l'état d'une transaction ou d'une requête à travers le système. De plus, il offre la possibilité de récolter des métriques du système et de les rassembler, pour ensuite les exporter, le tout étant configuré dans le code.

L'échantillonnage par requête qu'offrent les outils de traçage distribués comme Dapper et OpenTelemetry amènent des avantages très clairs lorsque nous essayons de comprendre le comportement de notre système distribué. Cela permet de suivre la requête et de récolter certaines métriques sur les machines hôte, sans être forcé d'introduire du code dans notre système à l'aide d'agents. Tout cela est possible en utilisant des bibliothèques instrumentées de RPC et en organisant l'information par requête au lieu de l'organiser par noeud. Cependant, contrairement aux outils de traçage traditionnels, il est plus difficile d'avoir de l'information sur le noyau du système, comme par exemple le nombre de fois qu'un fil d'exécution a été activé ou mis en pause.

2.3 Analyse, traitement et visualisation de trace

2.3.1 Gestionnaire d'état

L'objectif d'un gestionnaire d'état est de construire une machine à états à partir des événements d'une trace [65]. Avant de construire la machine à états, la trace doit être lue en ordre chronologique, partant du début jusqu'à la fin. Lors de l'initialisation du gestionnaire d'état, des états vides y sont attribués. Il y a donc une période durant laquelle la machine à états n'est pas initialisée et comprend des informations incomplètes. La figure 2.6, inspirée des travaux de Wininger, illustre l'évolution des états d'un fil d'exécution à travers le temps, depuis son initialisation, en fonction des événements de la trace. Avant le premier `sched_switch`, l'attribut n'existe pas. Par la suite, il change d'état en fonction des événements qui lui sont associés.

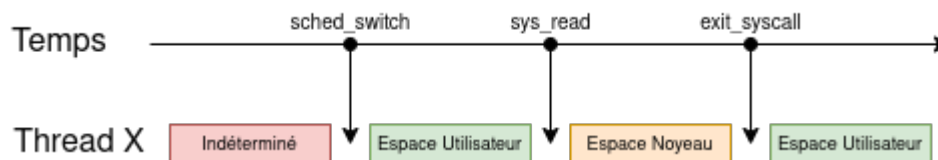


FIGURE 2.6 Gestionnaire d'état

Plus une trace possède d'évènements, plus le coût impliqué dans la construction d'une machine à états est importante. Pour connaître l'état d'un fil d'exécution à l'instant t , cela implique de devoir relire la trace depuis le début, ce qui s'avère inefficace. Pour pallier ce problème, Montplaisir et al. propose de sauvegarder les intervalles d'état, sur toute la période couverte par la trace, dans une arborescence appelée State History Tree [66] sur le disque. Pour cela, ils posent comme hypothèse que les intervalles de statut arrivent séquentiellement. Les critères de recherche basés sur le temps sont garantis et l'arbre est construit de sorte qu'il n'a pas besoin de se rééquilibrer pendant la construction. Cette solution prend en charge des traces d'une taille de plusieurs dizaines de gigaoctets.

Une autre méthode proposée par Chan et al. consiste à sauvegarder les états dans la trace elle-même [67]. La trace contient donc à la fois des événements et des états ponctuels. Cependant, leur approche n'est pas très flexible, car les informations sur l'état du moniteur sont étroitement liées à l'affichage.

2.3.2 Outils de visualisation de trace

Il existe une multitude d'outils pour visualiser les traces. Certains se spécialisent dans un certain format de traces ou un certain système d'exploitation, tandis que d'autres offrent plus de flexibilité et d'extensibilité. Cette section présente quelques outils notables.

Trace Compass

Trace Compass est un outil multiplateforme de visualisation de traces basé sur Eclipse [68]. L'outil permet d'analyser les traces au format GNU Debugger (GDB), Best Trace Format (BTF), CTF et Packet CAPture (PCAP). Plusieurs analyses, telles que l'état des fils d'exécution, les utilisations des ressources CPU, l'utilisation de la mémoire, l'activité des disques, les compteurs de performance, ou encore le chemin critique, sont offertes. Les travaux de Kouamé ont permis aux utilisateurs de définir leurs propres analyses par le biais d'un fichier XML qui doit suivre une certaine structure [69]. L'outil supporte de grandes traces, mais son architecture est monolithique et plusieurs composantes de l'interface utilisateur sont fortement

couplées aux composants responsables d'analyser les traces.

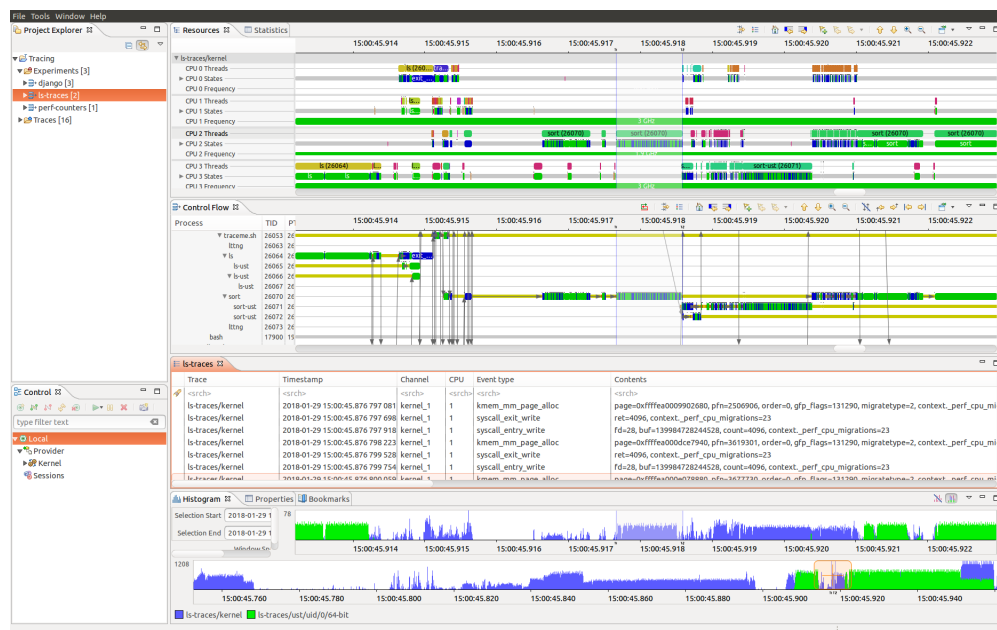


FIGURE 2.7 Trace Compass

Source: <https://eclipse.dev/tracecompass/>

Theia - Trace Viewer Extension

Efftinge et Kosyakov ont présenté Theia [70], un nouveau logiciel libre cadriciel pour les EDI. Theia a été mis de l'avant pour montrer la prochaine génération d'EDI après Eclipse. L'architecture de Theia a été pensée pour être extensible, supporter plusieurs langages, supporter le lancement et le débogage d'application. L'avantage principal de Theia est qu'il peut être exécuté dans les navigateurs web, ainsi qu'en local comme une application de bureau directement sur le système d'exploitation. Theia est un cadriciel d'EDI, ce qui signifie qu'il est utilisé pour créer d'autres EDI qui auront tous les avantages que Theia offre, mais qui seront personnalisés pour les besoins de l'utilisateur. Le projet partage de nombreuses similitudes avec l'éditeur de code Microsoft Visual Studio Code, au niveau des technologies, de l'interface graphique et les personnalisations de l'EDI par plugins. Theia utilise Node.js et Electron comme technologies, et implémente les protocoles LSP et DAP. Theia permet également d'installer des extensions lors de la compilation, pour qu'elles fassent partie intégrante de l'EDI lors de la distribution. Les extensions sont compatibles entre Theia et VsCode.

L'extension Trace Viewer est la nouvelle application open source client-serveur développée

par Ericsson et la Fondation Eclipse, avec d'autres contributeurs. Elle est disponible actuellement dans l'EDI Theia. Son objectif est de remplacer Trace Compass en tant qu'outil de traçage à usage général d'Eclipse. Tel que décrit précédemment, Trace Viewer utilise une architecture client-serveur, dans laquelle le client communique avec le serveur via un protocole appelé Trace Server Protocol (TSP) [42]. L'extension s'intègre dans l'EDI permettant de programmer, de tracer et de visualiser les traces avec le même cadriciel. Ceci permet d'itérer plus rapidement dans les différentes phases de développement. L'extension permet de visualiser les traces sous plusieurs vues, selon les analyses disponibles pour la trace dans le serveur.

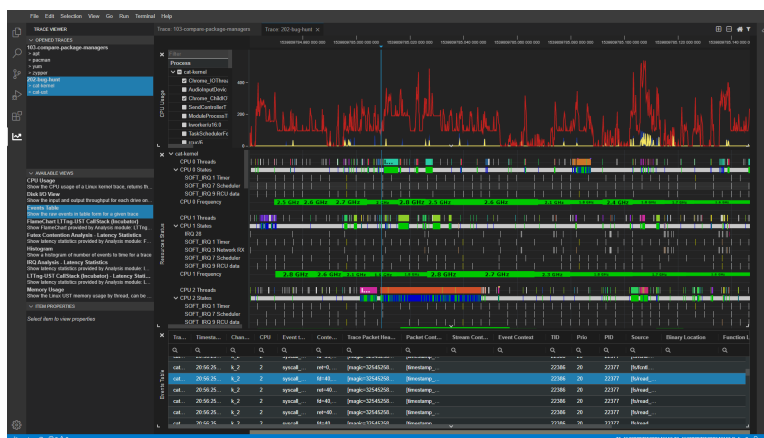


FIGURE 2.8 Theia - Trace Viewer Extension

Source: <https://github.com/eclipse-cdt-cloud/theia-trace-extension>

L'extension Trace Viewer utilise le protocole Trace-Server-Protocol pour interroger les modèles de données et visualiser différentes analyses de trace. Une analyse de trace lit une trace de manière séquentielle et stocke ses résultats (appelés modèles d'état) dans une structure de données pouvant être écrite sur disque. À partir des événements de trace, l'analyse extrairait les intervalles des valeurs d'état pour différentes ressources système (CPU, processus, disques, etc.). [71] Une analyse de trace transforme les événements de trace en un ensemble de données pour effectuer plus d'investigation et les réorganise en structure de données pour un accès plus rapide [68].

Grafana

Grafana est une plateforme logicielle libre qui fournit des fonctionnalités pour interroger, visualiser et comprendre les métriques. Grafana a répondu aux questions sur la façon de visualiser les données, ce qui a mis au défi de nombreux développeurs ces dernières années. Grafana apporte toutes les données ensemble et les partage entre les équipes. Grâce à divers

plugins, les utilisateurs peuvent obtenir ce qui leur convient le plus, ou même créer des plugins adaptés à leurs besoins. Grafana offre plusieurs fonctionnalités pour permettre à l'utilisateur de visualiser les données, d'organiser les données ou même de notifier sur différente plateforme tel que slack ou PagerDuty. Grafana peut également se connecter à différentes sources de donnée externes, telles que Excel ou une base de donnée SQL, et immédiatement permettre de visualiser ses données. Cela est très pratique puisque Grafana interprète directement les données, au lieu de le faire manuellement, ce qui nous sauve une étape qui peut parfois prendre du temps. Grafana permet également d'afficher plusieurs sources de données différentes dans la même vue. Grafana est assez simple lorsqu'il est temps d'afficher les données. Après s'être connecté aux différentes sources de données, l'utilisateur doit construire une requête pour extraire l'information pertinente. Ensuite, selon le cas d'utilisation l'usager choisit l'outil de visualisation. [72] Grafana ne s'occupe pas d'instrumenter les applications, mais laisse plutôt le travail à d'autres outils plus spécialisés pour ce traitement, comme Prometheus. À prendre en compte que Grafana n'offre pas d'analyses sur un ensemble de donnée, comme le fait Trace Compass, mais permet d'afficher différentes données selon la requête construite.



FIGURE 2.9 Grafana - Tableau de bord

Source: <https://grafana.com>

Vampir

Vampir fournit un cadre facile à utiliser qui permet aux développeurs d'afficher et d'analyser rapidement le comportement arbitraire d'un programme. La suite d'outils implémente des algorithmes d'analyse d'événements optimisés et des affichages personnalisables [73]. L'optimisation des performances avec Vampir implique plusieurs étapes pour améliorer les performances des applications informatiques parallèles. Vampir est un outil d'analyse de performance souvent utilisé en conjonction avec l'instrumentation Score-P pour collecter des

données sur les performances des applications [74]. Le processus commence généralement par l'instrumentation de votre code et son exécution, en collectant des données. Vampir est ensuite utilisé pour visualiser et analyser ces données. L'analyse se concentre sur l'identification des goulots d'étranglement, tels que les algorithmes inefficaces, les modèles d'accès aux données problématiques, les charges de travail déséquilibrées et les zones à forte utilisation du processeur. Les modèles de communication et de synchronisation sont essentiels, et Vampir aide à évaluer comment les processus communiquent et où se produit la surcharge de synchronisation. Les modèles d'utilisation de la mémoire sont également examinés, car une consommation élevée de mémoire et un accès inefficace à la mémoire peuvent affecter les performances. Les stratégies de parallélisation sont évaluées et des modifications sont apportées en fonction des informations tirées de l'analyse Vampir. Ces changements peuvent impliquer des modifications d'algorithmes, une optimisation des structures de données, une réduction des transferts de données et un meilleur équilibrage de charge. Une analyse comparative continue est cruciale pour mesurer l'impact des efforts d'optimisation. Pour les systèmes distribués, Vampir peut être utilisé pour le profilage parallèle, vous permettant d'analyser les performances sur plusieurs noeuds. La collaboration et la documentation des résultats sont essentielles, car des perspectives multiples et des efforts collectifs conduisent souvent à des solutions d'optimisation plus efficaces. En résumé, Vampir est un outil précieux dans le processus itératif d'optimisation des performances, qui implique un profilage, une analyse minutieuse et des modifications du code.

2.3.3 Analyse distribué de trace

Les efforts récents pour paralléliser l'analyse des traces, comme en témoignent les travaux de Reumont-Locke [75] et Martin [76], ont visé à améliorer l'efficacité de ce processus sur un seul noeud de calcul. Reumont-Locke propose une méthode efficace de parallélisation de l'analyse de traces en tenant en compte plusieurs facteurs : partitionnement des données, équilibrage de la charge, verrouillage et synchronisation, et dépendances des données. La résolution de dépendance explique bien les problèmes lorsque nous avons plusieurs traces que nous essayons de raccommoder ensemble, lorsque nous essayons de séparer la charge de travail entre plusieurs coeurs ou même entre plusieurs noeuds de calcul. Martin discute de comment déduire les états manquants lors de l'analyse d'une trace. Ceci peut se produire lorsque des événements ont été perdus car les tampons du traceur étaient surchargés.

La même difficulté se produit lorsqu'on veut analyser en parallèle plutôt que de manière séquentielle différentes portions d'une trace. Par exemple, divisons l'analyse de trace en deux morceaux (deux moitiés de l'intervalle de temps couvert par la trace) pour en faire l'analyse

en parallèle. Le fil d'exécution qui tente de calculer l'état des ressources pendant le second intervalle, à partir de la seconde moitié de la trace, commence avec des états inconnus pour toutes les ressources. Au fur et à mesure que les événements de cette seconde moitié de trace sont traités, l'état de certaines ressources peut être obtenu. Une fois l'analyse terminée, l'analyse de la première moitié de trace, qui a été faite en parallèle, peut être utilisée pour combler les états inconnus qui demeurent au début de l'intervalle associé à la deuxième moitié de trace.

Marie Martin [76] a proposé un cadriciel qui permet souvent de calculer le nouvel état d'une ressource, suite à un événement, même si son état initial était inconnu. Ceci peut être utilisé pour faire le calcul en parallèle de plusieurs morceaux d'une même trace, qui correspondent à des intervalles de temps consécutifs. Son travail va plus loin et permet de détecter des incohérences dans les changements d'état et ensuite d'appliquer une correction pour éviter l'incohérence. Ceci est utile pour les cas où la collecte d'événements n'est pas fiable et certains événements sont manquants. Dans un tel contexte, cette capacité est précieuse, car la perte d'un événement peut causer une mauvaise interprétation de l'analyse, comme le montre la figure 2.10. Il est également important de comprendre que lors d'une analyse de plusieurs traces, la perte d'un événement pourrait causer l'échec d'une analyse lors de la résolution de dépendances en vue de calculer le chemin critique.

Néanmoins, l'optimisation de l'analyse de traces par plusieurs fils d'exécution en parallèle, sur un seul noeud, ne peut à elle seule répondre pleinement aux défis d'évolutivité inhérents aux environnements distribués. Même si un outil d'analyse de trace fonctionne avec une efficacité maximale, il reste limité par les capacités de calcul du matériel sous-jacent, généralement équivalentes au système tracé dans la plupart des cas. Lorsque nous augmentons le nombre de noeuds tracés jusqu'à des centaines ou des milliers, l'outil d'analyse devient rapidement submergé. Ce goulot d'étranglement devient particulièrement problématique dans les applications nécessitant une disponibilité continue, car les anomalies peuvent rapidement s'aggraver au sein des clusters [77], et les retards dans l'analyse des données pendant les enquêtes ne font qu'aggraver la situation.

Il est intéressant de noter que les multiples traces, prises sur différents noeuds de calcul distribué, sont beaucoup moins couplées que les différents morceaux d'une même trace. Ainsi, alors qu'il est plutôt difficile de paralléliser l'analyse d'une même trace sur plusieurs coeurs de CPU, ou même sur plusieurs noeuds, il est plus facilement envisageable de paralléliser l'analyse de multiples traces, prises sur différents noeuds de calcul, sur un nombre égal ou inférieur de noeuds d'analyse de trace. C'est le contexte qui nous intéresse dans ce mémoire, alors que l'objectif est de mettre à l'échelle l'analyse de trace pour des applications de calcul

de haute performance, sur des grappes de calcul contenant des centaines ou des milliers de noeuds.



FIGURE 2.10 Gestionnaire d'état avec un évènement perdu

Nagios

Nagios est une application open-source de surveillance, offrant une extensibilité facile. Elle permet de vérifier l'état des dispositifs réseau et de leurs services, tout en fournissant un système d'alertes. Notamment, Nagios a adopté une approche similaire à Prometheus pour résoudre ses propres défis de mise à l'échelle [78]. Les versions récentes permettent de déployer Nagios avec une structure de surveillance fédérée, autorisant plusieurs serveurs Nagios locaux qui remontent ensuite les informations vers un serveur Nagios central, pour un monitoring systématique (Nagios). Un module nommé Nagios Fusion permet de surveiller et de configurer automatiquement les serveurs Nagios locaux.

Prometheus

Prometheus [79] est un système de surveillance et d'alerte open-source publié en 2016 par SoundCloud. Prometheus répond aux problèmes d'évolutivité en permettant l'organisation des instances Prometheus dans une fédération, comme l'a démontré Reback en 2021 [80]. Cette approche facilite l'établissement d'une hiérarchie à plusieurs niveaux de serveurs Prometheus, chaque niveau regroupant et transmettant les données au niveau supérieur. Bien que la mise à l'échelle horizontale des serveurs d'analyse soit une stratégie efficace, il est essentiel de reconnaître que les analyses Prometheus consistent principalement en des métriques simples, ce qui rend l'agrégation possible. Cependant, appliquer le même modèle à des processus d'analyse plus complexes, tels que ceux proposés par Trace Compass, présente des défis considérables. La nature de leurs agrégations varie considérablement, selon les différents types d'analyse, ce qui rend difficile l'application directe de l'approche Prometheus dans de tels cas. De plus, Prometheus possède la capacité d'automatiser l'alerte d'anomalie.

L'analyse distribuée du chemin critique dans Trace Compass

Le contexte particulier du calcul de haute performance guide nos efforts et nos choix récents pour paralléliser l'analyse du chemin critique, comme en témoignent les travaux de Denys et Dagenais. Ils ont visé à améliorer l'efficacité de ce processus. L'un des avantages de l'architecture distribuée mise en place est que cela ne nécessite aucun transfert de fichiers entre le client et les noeuds d'analyse. Cela indique en contrepartie que l'algorithme doit permettre la distribution du calcul d'analyse du chemin critique sur chacun des noeuds. Pour cela, ils explorent un algorithme pour résoudre les dépendances distantes [77]. Les événements de noyau nécessaires à l'exécution de l'analyse sont identifiés : `sched_switch`, `sched_ttwu`, `interrupt_entry`, `interrupt_exit`, `inet_sock_local_in`, and `inet_sock_local_out`. Denys et Dagenais mettent de l'avant deux étapes pour le calcul du chemin critique : la construction du graphe d'exécution et la résolution du chemin critique. Le graphe d'exécution peut facilement être calculé sur différents noeuds, puisqu'il ne nécessite aucune dépendance envers les autres traces. Les complications surviennent lors de la parallélisation de la seconde étape, le calcul du chemin critique, puisque celui-ci nécessite les graphes d'exécution de chacune des traces sur le même noeud.

La solution proposée par Denys et al est de transmettre des informations très spécifiques aux noeuds capables de résoudre l'état bloquant [81]. L'architecture utilisée est une architecture pair-à-pair pour établir la communication entre les différents noeuds de calcul [82]. Le chemin critique se résout en cascade entre les différents noeuds d'analyse mis à disposition, sans revenir au noeud d'analyse qui a commencé la requête, tant que le chemin critique n'est pas complété. Pour minimiser les interactions entre les noeuds d'analyse, un index des traces sur chaque noeud et des interactions entre les graphes d'exécution de chaque noeud est construit. L'avantage d'avoir un index est de limiter les appels réseaux, ce qui pourrait causer une latence à la réponse au fur et à mesure que le chemin critique est calculé. Un inconvénient de l'architecture et de l'index combiné est que tous les noeuds doivent avoir les index des autres noeuds d'analyse, pour être capables de demander la suite du chemin critique, sans retourner au noeud d'analyse principal. Cela rajoute un défi au niveau de l'infrastructure, de s'assurer que chaque noeud possède les index de chaque autre noeud. Un autre inconvénient serait que la communication entre les noeuds d'analyse est uniquement conçu dans le cadre du chemin critique, et spécifiquement pour Trace Compass. Cela limite les utilisateurs à devoir utiliser Trace Compass, ou à reproduire l'entièreté de cette solution dans leur outil de visualisation de trace.

Pour conclure, les recherches progressent sur la mise à l'échelle des analyses comme le chemin critique, ou l'aggrégation de métriques simples comme celles supportées par Prometheus.

Cependant, plusieurs se demandent comment nous pouvons rendre tout cela disponible de manière modulaire, à travers un standard comme le protocole TSP, tout en conservant une flexibilité sur les analyses possibles, mais également en permettant une mise à l'échelle horizontale, par exemple dans un contexte de calcul de haute performance.

2.4 Conclusion de la revue littéraire

À travers cette revue de littérature, nous avons vu qu'il a eu beaucoup de travail effectué pour pallier au problème de récolter, d'analyser et de visualiser des traces. Toutes ces applications proposent des modèles bien précis à leur besoin avec la mise à l'échelle en tête. Cela permet d'identifier ce qui semble marcher ou ce qu'il faudrait améliorer. Une lacune évidente des outils de traçage est dû au manque de standardisation ce qui rend le flux de travail instrumentation-collection-analyse-visualisation difficile. Par chance, le protocole TSP offre une fondation pour pouvoir offrir un standard robuste pour favoriser la réutilisation des serveurs d'analyse pour les différents outils de visualisation. Cependant, le protocole n'avait pas la mise à l'échelle en tête.

En conséquence, nous souhaitons d'abord d'évaluer les limites du protocole TSP sur des grappes de calcul. Nous souhaitons ensuite proposer une extension aux protocoles pour permettre aux serveurs d'implémenter des analyses qui nécessitent les dépendances entre les traces comme le chemin critique.

CHAPITRE 3 ARTICLE 1 : SCALING TRACE ANALYSIS IN A MODULAR DISTRIBUTED ARCHITECTURE

Soumis aux Transactions on Parallel Computing

Date de soumission : Le 5 Avril 2024

Autheurs : Ahmad Faour, Hervé Mbikayi Kabamba, Michel Dagenais

3.1 Abstract

Trace analysis plays a pivotal role in optimizing the performance and efficiency of computing systems, particularly in high-performance and distributed environments. Recent efforts have focused on parallelizing trace analysis on single compute nodes, resulting in significant advancements in efficiency. However, challenges persist in extending this scalability to distributed environments, where traces are fragmented across multiple remote machines. Existing approaches lack flexibility or scalability, necessitating alternative solutions for efficient trace analysis. In this paper, we propose a novel solution that enhances the scalability of trace analysis for high-performance computing applications across distributed clusters. Our solution speeds up by a factor of 8 to 16 the trace analysis, when 20 parallel are used.

We investigate various aspects including data storage, transfer, distributed computation, and visualization, aiming to provide insights into the performance of our proposed solution for distributed systems. This paper contributes to the ongoing discourse on scalable trace analysis methodologies and offers valuable insights for researchers and practitioners in the field of high-performance computing.

3.2 Introduction

In recent years, the advancement of trace analysis methodologies has been pivotal in enhancing the efficiency of computing systems. Parallelization of trace analysis, as demonstrated by [75], and [76], has emerged as a prominent strategy aimed at optimizing the computational processes on individual nodes. Reumont-Locke [75], focused on pivotal aspects such as data partitioning, load balancing, locking and synchronization, and data dependencies, contributing to significant strides in parallelizing trace analysis. Similarly, [76], delves into the challenge of inferring missing states during trace analysis, particularly when events are lost due to buffer overload.

Despite the progress made in parallelizing trace analysis on single nodes, significant challenges persist, particularly in distributed environments. The fragmentation of traces across multiple remote machines poses a considerable obstacle, as existing tools such as Theia [70], lack the inherent capability to seamlessly explore files from diverse nodes simultaneously. This limitation necessitates alternative solutions, to facilitate efficient trace analysis across distributed systems. Martin [76], provided a notable contribution in this regard, enabling parallel computation of multiple trace segments while detecting and rectifying inconsistencies in state changes.

However, extending the scalability of trace analysis beyond parallel execution on singular nodes remains a formidable task. Existing approaches, such as those proposed by Kabamba et al. [83], have yet to adequately address the scalability concerns inherent in distributed environments. As computing clusters grow in size and complexity, the scalability of trace analysis becomes increasingly critical. Challenges such as data transmission across networks and optimization of computational tasks across multiple nodes pose significant hurdles in achieving scalable trace analysis.

Moreover, the intricacies of analyzing distributed traces extend beyond mere parallelization on multiple nodes. The coupling of traces collected from disparate computing nodes presents unique challenges, demanding innovative approaches to ensure comprehensive analysis while maintaining scalability. This article aims to address these challenges by proposing a novel solution that enhances the scalability of trace analysis for high-performance computing applications across distributed clusters, using a modular approach such as the Trace Server Protocol (TSP).

We delve into the intricacies of distributed trace analysis, exploring various aspects including data storage, transfer, distributed computation, and visualization. We investigate the performance of our proposed solution in distributed systems, aiming to strike a balance between scalability and flexibility. By leveraging our solution, we seek to empower users to conduct comprehensive trace analysis across distributed computing clusters comprising thousands of nodes.

This paper presents a novel method for distributed trace analysis. It suggests the implementation of a trace coordinator adhering to the Master-Worker distributed architecture. Communication between the master node and the worker nodes relies on the TSP, for which we propose an extension to address issues arising from the inability to analyze all traces on a single worker node. Aggregation algorithms have been implemented to cater to diverse analysis requirements. Additionally, resource and execution graph APIs have been integrated into the protocol to enhance the flexibility of distributed trace analysis.

The remainder of this paper is organized as follows : Section 3.3 establishes the background of the research. Section 3.4 provides an overview of existing visualisation tools such as Nagios and Prometheus, highlighting their scalability strategies. Section 3.5 introduces our proposed solution, outlining its motivation and key objectives. Section 3.6 presents the obtained results. In section 3.7 we provide a discussion on the obtained results while in Section 3.8, we conclude the work and highlight some future work avenues.

3.3 Background

Trace Compass (TC) is a cross-platform trace visualization tool based on Eclipse [68]. It facilitates the analysis of traces across various formats including GNU Debugger (GDB), Best Trace Format (BTF), Common Trace Format (CTF), and Packet Capture (PCAP) [84]. Several analyses, such as the state of execution threads, CPU resource usage, memory usage, disk activity, performance counters, and even the critical path, are offered. Previous work on TC permitted users to define analyses utilizing XML files [69]. Although this capability remains supported, its limitations have led to a proliferation of scripted and hard-coded. These alternatives afford greater control over the development of new features, thus rendering them more appealing for users seeking enhanced flexibility and customization. The tool supports large traces, but its architecture is monolithic and several components of the user interface are strongly coupled to the components responsible for analyzing the traces.

Trace Compass implements a state manager that constructs a state machine from the events of a trace [65]. Before building the state machine, the trace must be read in chronological order, starting from the beginning to the end. When the state manager is initialized, empty states are assigned to it. There is therefore a period during which the state machine is not initialized and includes incomplete information. Figure 3.1, inspired by the work of Wininger, illustrates the evolution of the states of an execution thread over time, since its initialization, depending on the events in the trace. Before the first `sched_switch`, the attribute does not exist. Subsequently, it changes state depending on the events associated with it.

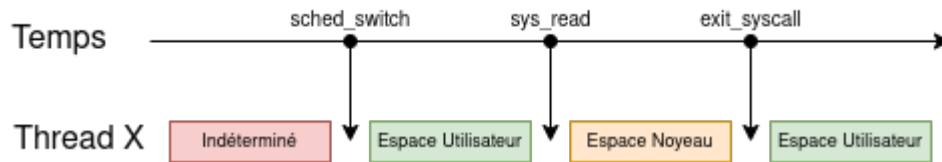


FIGURE 3.1 Gestionnaire d'état

The State History Tree (SHT), used to store the state system on disk, is a scalable and

efficient tree-based state history data structure, and a number of algorithms are available to store, manage and retrieve the different state values for an arbitrary trace size, for any number of system resources [66]. The SHT can take a lot of time to build, because the state tree is built sequentially when reading the trace. That step happens when we first request the analysis *tree* (e.g. Time Graph Tree, XY Tree). In Trace Compass Server, the SHT of some analysis can be generated by default when the experiment is created, for instance as a background pre-processing. This will extend the time of creating the experiment, but when we will request the analysis that uses this analysis module, it will already have been created, and thus the response will be much faster. For benchmarking purposes, during our experiments, this feature was disabled to measure precisely each processing step.

A Trace Compass Server was build using the main components of Trace Compass. The difference between Trace Compass and the Trace Compass Server is the addition of an abstraction layer to loose the coupling between the analyzes and views offered by Trace Compass. This Data Provider layer, presented in Figure 3.2, is used through the REST protocol as an interface to build and fetch the generic models available by URI.

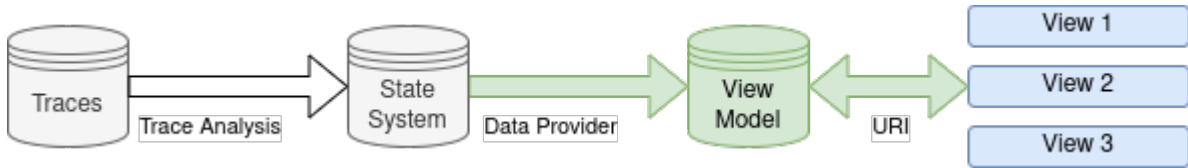


FIGURE 3.2 Trace Compass Server Architecture

3.4 Related Work

This section is structured into three distinct subsections. The first segment delves into recent advancements concerning the decentralization of visualization and analytical processes, leveraging established protocols. In the subsequent section, recent developments in trace visualization tools are explored. Lastly, the third section investigates various methodologies employed to achieve distributed analysis on trace data.

3.4.1 Trace Server Protocol

The Trace Server Protocol (TSP) [40], was developed during the introduction of the client-server architecture in **Trace Compass** (TC), a powerful framework for trace visualization and analysis, by the development team at Ericsson in 2018. This protocol follows the REST API philosophy and currently uses JSON as the data exchange format. In this context,

access points for accessing the trace visualization model are arranged based on the type of visualization. There are dedicated access points for analyses in the form of XY graphs, time graphs and tables. This protocol is still evolving as development of Trace Compass(TC) continues and new analyses and visualizations are added regularly. In the TSP protocol, it is possible to adjust the data resolution, as well as the desired data fields, by specifying the targeted time points as parameters [42].

In the realm of distributed architectures, a more pronounced impact is noted, primarily because of the engagement of numerous network communications. This results in a heightened frequency of serializations and deserializations operations for a single request. As a result, it is important to evaluate the existing solutions to reduce the impact of the serialization processes. Work by Chen Kuang Piao [41], suggested that it would be possible to achieve significant improvements in message size and serialization efficiency by using Protobuf [14] instead. Indeed, serialization becomes a very important factor in response time, when processing huge traces.

Given that TC employs the TSP protocol, the need for an efficient architecture to scale trace analysis and visualization is paramount. This architecture needs to be able to scale based on the number of nodes, to handle a large number of traces generated by recent supercomputers, made up of several thousands of nodes.

3.4.2 Trace Visualization Tools

Within the domain of trace visualization, a plethora of tools are available, each designed to fulfill unique needs and preferences. While certain tools are optimized for particular trace formats or operating systems, others prioritize adaptability and extensibility. This section serves to introduce several noteworthy tools within this landscape.

The **Theia Trace Viewer** extension and the trace server are part of the new open source client-server application developed by Ericsson and the Eclipse Foundation, along with other contributors. It is currently available in the Theia IDE [70]. Its goal is to replace Trace Compass an Eclipse general-purpose tracing tool. As described previously, **Theia Trace Viewer** uses a client-server architecture, in which the client communicates with the server via a protocol called the Trace Server Protocol (TSP) [42]. The extension seamlessly integrates into the IDE, enabling users to program, trace, and view traces using the same framework. This unified approach facilitates a rapid iteration across various development phases. Additionally, the extension provides the capability to visualize traces through multiple views, contingent upon the analyses accessible for the trace within the server.

The **Theia Trace Viewer** extension uses the Trace-Server-Protocol to query data models and visualize different trace analyses. A trace analysis reads a trace sequentially and stores its results (called state models) in a data structure that can be written to disk. From the trace events, the analysis would extract the intervals of state values for different system resources (CPUs, processes, disks, etc.) [71]. A trace analysis reads the trace events and transforms them into a dataset for further investigation, while storing them in a flexible and extensible disk-based data structure, accessible with a logarithmic complexity [68].

Grafana is an open source software platform that provides functionality for querying, visualizing and understanding metrics. **Grafana** propose a way to visualize data, which has challenged many developers in recent years. **Grafana** brings all the data together and shares it between teams. Through various plugins, users can get what suits them most, or even create plugins tailored to their needs. **Grafana** offers several features allowing user viewing and organizing data, or even notification on different platforms such as **Slack** or **PagerDuty**. **Grafana** can also connect to various external data sources, such as Excel or an SQL database, and immediately visualize its data. This functionality is highly practical as Grafana directly interprets the data, eliminating the need for manual interpretation. This streamlined process saves valuable time bypassing a step that can occasionally be time-consuming. **Grafana** also allows displaying multiple different data sources in the same view. **Grafana** is quite simple when it comes to displaying data. After connecting to the different data sources, the user must build a query to extract the relevant information. Then, depending on the use case, the user chooses the visualization tool [72]. **Grafana** does not instrument applications, but rather leaves the work to more specialized tools such as **Prometheus**. It is important to acknowledge that **Grafana** does not provide analyses on a dataset in the manner that TC does. Rather, it enables the visualization of various datasets contingent upon the constructed query.

3.4.3 Distributed analysis of trace

Recent efforts to parallelize trace analysis, as evidenced by the work of Reumont-Locke [75], and Martin [76], have aimed to improve the efficiency of this process on a single compute node. Reumont-Locke provides an efficient method for parallelizing trace analysis by considering several factors : data partitioning, load balancing, locking and synchronization, and data dependencies. Dependency resolution effectively illustrates the challenges encountered when attempting to integrate multiple traces together, while separating the workload between multiple cores or multiple compute nodes. Martin [76], discusses how to infer missing states when analyzing a trace. This can occur when events were lost because the tracer buffers were overloaded. The same difficulty occurs when analyzing different portions of a trace in parallel

rather than sequentially.

Prometheus [79], is an open-source monitoring and alerting system released in 2016 by SoundCloud. Prometheus addresses scalability issues by allowing **Prometheus** instances to be organized into a federation, as demonstrated by Reback in 2021 [80]. This approach makes it easy to establish a tiered hierarchy of **Prometheus** servers, with each tier aggregating and passing data to the next level up. While horizontally scaling analytics servers is an effective strategy, it is critical to recognize that **Prometheus** analytics consist primarily of single metrics, which makes aggregation possible. However, applying the same model to more complex analysis processes, such as those offered by TC, presents considerable challenges. The nature of their aggregations varies considerably, depending on different types of analysis, making it difficult to directly apply the **Prometheus** approach in such cases.

Previous research conducted by Tran et al. [41] proposed a distributed tracing architecture that offers the capability to extend the aggregation operations. The distributed architecture (Master-Worker) of the solution offers a good improvement in the response time. The disadvantage of their approach is the limitation of aggregations to only apply to TSP models. Certain aggregations and analyses require that everyone be aware of the cluster information, before starting the analysis.

Previous research conducted by Kabamba et al. focused on distributed trace analysis on a singular node [83], yet failed to adequately address scalability concerns. This approach neglected to consider the challenges associated with transferring substantial volumes of trace data across the network, along with the optimization of analysis duration through the distribution of computational tasks across multiple nodes.

Even if a trace analysis tool operates at maximum efficiency, it is still limited by the computational capabilities of the underlying hardware, which is generally equivalent to the traced system in most cases.

As the number of plotted nodes escalates to thousands, the analysis tool promptly encounters saturation. This bottleneck assumes heightened significance within contexts necessitating uninterrupted availability, as anomalies possess the propensity to swiftly proliferate within clusters [77], exacerbating the situation further by augmenting delays in data analysis throughout the investigative processes.

It is interesting to note that the multiple traces, taken on different distributed computing nodes, are much less coupled than the different pieces of the same trace. Thus, while it is rather difficult to parallelize the analysis of the same trace on several CPU cores, or even on several nodes, it is however possible to parallelize the analysis of multiple traces, taken on

different computing nodes, on an equal or less number of trace analysis nodes.

This particular context forms the focus of this article, which aims to enhance the scalability of trace analysis, for high-performance computing applications, on computing clusters comprising thousands of nodes.

Distributed analysis of the critical path in Trace Compass

The particular context of high performance computing guides our efforts and our recent choices to parallelize the analysis of the critical path, as evidenced by the work of Denys and al. [81]. They aimed to improve the efficiency of this process. An advantage of the implemented distributed architecture is its independence from any necessity for file transfers between the client and the analysis node. This indicates in return that the algorithm must allow the distribution of the critical path analysis calculation on each of the nodes. To do this, they explore an algorithm to resolve remote dependencies [77]. The authors put forward two steps for the calculation of the critical path : the construction of the execution graph and the resolution of the critical path.

The solution proposed in [81], is to transmit very specific information to the nodes capable of resolving the blocking state. The architecture used is a peer-to-peer architecture to establish communication between the different computing nodes [82]. The critical path is resolved in cascade between the different analysis nodes made available, without returning to the analysis node which started the request, as long as the critical path is not completed. To minimize the interactions between the analysis nodes, an index of the traces on each node and of the interactions between the execution graphs of each node is constructed. The advantage of having an index is to limit network calls, which could cause response latency as the critical path is calculated.

To conclude, research is progressing on the scaling of analyses like the critical path, or the aggregation of simple metrics like those supported by Prometheus. However, many wonder how we can make all of this available in a modular way, through a standard like the TSP protocol, while maintaining flexibility on possible analyses, but also allowing horizontal scaling, for example in a high performance computing context. It is important to mention that the volume of the trace inside a cluster makes it difficult to move the traces from the traced node to the analysis node. This is why our solution focused on an architecture that can process them in parallel without transmitting too many traces over the network.

Table 3.1 provides a summary of the highlights of the tools discussed in Section 3.1. It is therefore obvious that our solution attempts to fill the lack of flexible tool in distributed

analysis.

	Flexibility	Scalability	Standardization	Open Source
Trace Compass	✓			✓
Theia - Trace Viewer	✓		✓	✓
Prometheus		✓	✓	✓
Our Solution	✓	✓	✓	✓

TABLEAU 3.1 Analysis Tool comparison

Each tool and approach offers unique strengths and limitations in scalability. While some, like TC, provide comprehensive trace analysis capabilities but face challenges in scaling due to architectural constraints, others, like **Theia** and **Grafana**, offer modern frameworks and visualization capabilities but may be limited by dependencies or lack of in-depth analysis features. Distributed analysis methods aim to improve efficiency but encounter challenges in managing distributed environments. Addressing scalability and flexibility concerns requires comprehensive strategies tailored to the specific requirements of trace analysis in distributed environments, including efficient data transfer, parallel processing, and optimized communication protocols.

3.5 Proposed Solution

This article presents a new solution for distributed analysis that increases the scalability in terms of system size and load, without compromising the flexibility of the analysis. To understand well the solution we will go through every aspect of the distributed analysis : data storage, data transfer, distributed computation and visualization. In particular, this section investigates the ability of the methods to perform well on distributed systems.

3.5.1 Motivation

Conventional methodologies advocate employing distributed tracers to monitor high-level operations like request tracking within distributed environments. These tracers facilitate a deeper understanding of the lifecycle of specific system requests as they traverse different nodes in the distributed systems. Nevertheless, challenges emerge when conducting analyses based on data collected from both user space, with detailed insights, or/and kernel space events. In such instances, the volume of events generated across nodes cannot be effectively managed by distributed tracers alone. Addressing such scenarios presents a significant challenge within distributed systems environments. If the tracker does not store traces on a machine similar to Jaeger [85], users must handle this task themselves.

Consider a high-performance cluster generating traces on every node, each trace being several gigabytes in size. Manually consolidating all these traces onto a single machine presents formidable challenges. Firstly, this is due to the finite storage capacity of individual computers. Secondly, transferring traces over the network is constrained by limited bandwidth, potentially requiring hours to move large volumes of data from one computer to another.

While it is not realistic to collect all traces of a high performance cluster on a single node for analysis, it is not always possible to use the entire cluster with each node storing its trace and participating in its analysis. A common case is to have an analysis cluster of smaller size than the calculation cluster. Each node in the analysis cluster collects traces from numerous nodes in the computing cluster (e.g. 10 or 100) and takes care of their analysis.

Another challenge arises when attempting to analyze a system, such as a High-Performance Computing (HPC) cluster. The sheer volume of traces becomes a significant obstacle, even if all the traces are aggregated onto a single computer and adequately stored. Computing an analysis on these traces, which can collectively contain billions of events, requires an enormous amount of time. It is not uncommon for a single analysis to necessitate up to three hours of computation.

TC excels in its ability to support various trace formats, making it a standout framework in this regard. This platform offers a seamless way to incorporate custom front-end parser and even introduce novel types of analyses. Notably, the separation between the TC client and server components positions it as a promising option for distributed deployment. While the current Trace Compass server can communicate with a remote client through the TSP, it has limitations. It can only run as a single instance, making it inadequate for handling substantial volumes of trace data in environments like HPC clusters. This limitation becomes apparent when considering the demands on CPU, storage, and RAM memory resources. Additionally, the Trace Compass server presently requires all trace data to be stored locally for analysis, which may not align with certain use cases.

Certainly, running multiple TC servers, each dedicated to assessing the traces of individual machines within a cluster, is a feasible approach. However, it is crucial to emphasize the importance of having an overarching view of the entire cluster. The parallelization of the analyses of multiple traces, collected from various nodes, can be achieved readily. However, the challenge lies in consolidating them for simultaneous display and obtaining aggregated metrics, particularly when dealing with intricate analyses such as the critical path. To address this, we propose a novel architecture aimed at facilitating both the parallel analysis and visualization of traces originating from high-performance clusters.

To make TC more adept at tracing extensive and diverse systems, a fundamental re-architecture

with scalability in mind is imperative. Our aim is to equip TC with robust query capabilities similar to those found in established platforms, all while preserving its inherent flexibility for the development of unique analysis types and trace formats. Achieving this objective necessitates the implementation of an abstract layer, allowing for the creation of new query operators with minimal effort, while maintaining the overall behavior of the platform. This concept aligns with the notion of the trace parser or the DataProvider layers, which provide an abstraction of the view, irrespective of the specific type of analysis within TC.

3.5.2 Overall architecture

The architecture platform consists of four main parts, the traced nodes, the analysis nodes, the coordinator node and the user interface (UI). An analysis node serves as the designated location for processing trace data that originates from traced nodes. This configuration facilitates the deployment of the analysis node directly where the traces are located, which is more convenient than relocating the traces to the same node. In the proposed configuration, each analysis node refers to a TC Compass Server. The coordinator node plays a pivotal role when a client or view initiates a query. It, in turn, dispatches queries to the relevant analysis nodes and subsequently combines their responses to furnish an integrated analysis outcome. When a client transmits a request to a server seeking trace analysis data, the predominant portion of the response typically comprises data points, frequently presented in the form of a time series array.

In our context, there is no necessity for a protocol enabling the construction of complex operations. This is because the flexibility to create analyses within the architecture already encompasses the capability for intricate operations. Therefore, we opted to use the concepts put forward by the TSP as the communication protocol for trace data and analysis requests. Several factors guided this decision. Firstly, the protocol is designed to break away from existing analytics implementations and focuses on displaying data.

This grants the ability of maintaining the inherent flexibility found in many trace analysis tools for the addition or modification of analyses. Furthermore, the protocol must possess the functionality to filter data points based on timestamps and should be readily expandable to incorporate additional information based on the current view, such as zooming in or out. Simultaneously, it constrains the computational resources required on the coordinator, as its primary task involves aggregating data. Our objective is to concentrate compute-intensive tasks at the parallel analysis node. Thirdly, the protocol must possess the capability to select a set of traces for inclusion in the analyses. Additionally, this feature assists in the maintenance of the coordinator by ensuring that aggregation relies on the views provided by the protocol

rather than the implementation of the analysis itself.

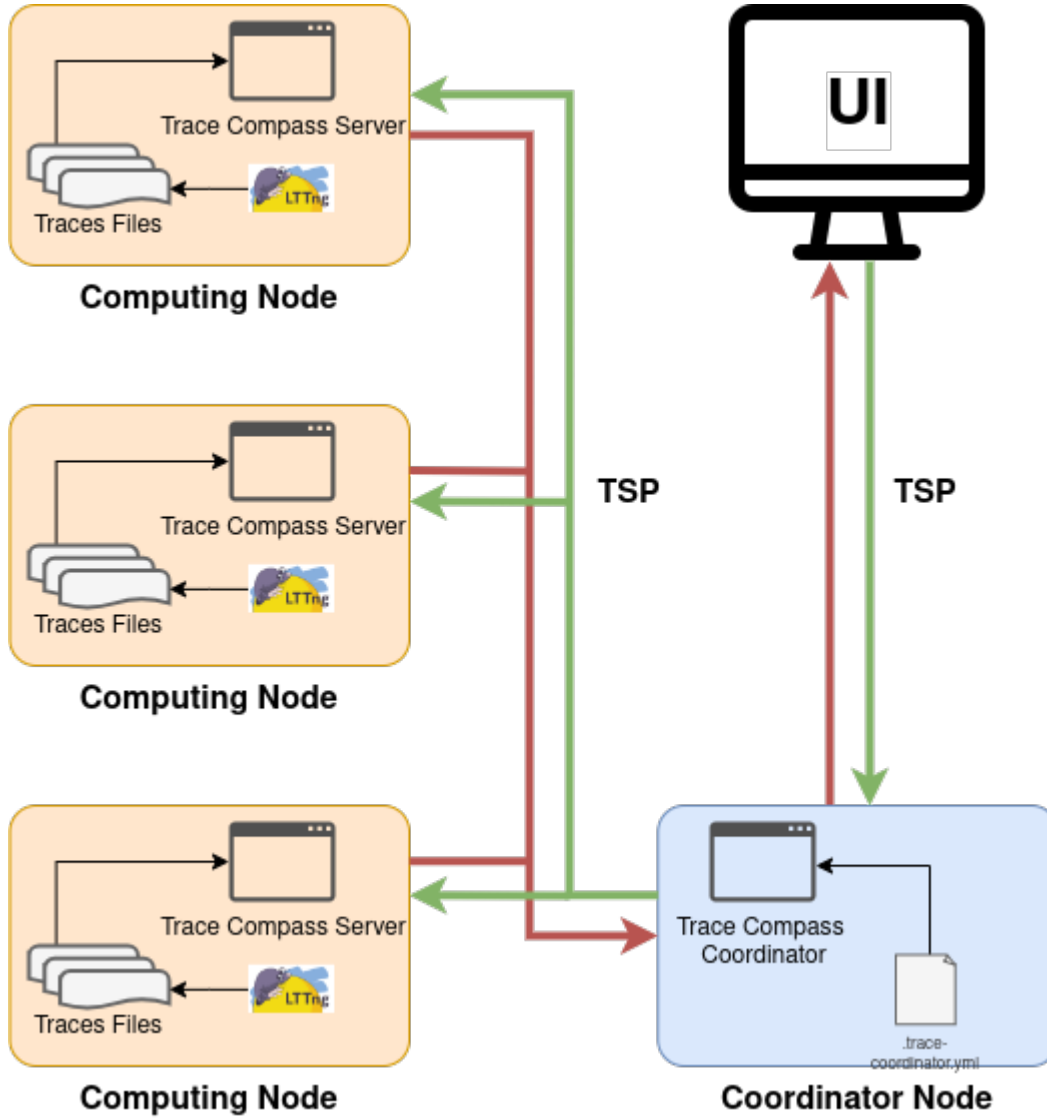


FIGURE 3.3 Proposed scalable distributed trace analysis

3.5.3 Distributed Analysis of Traces

As an integral aspect of our solution, the protocol should facilitate the broadcasting of the analysis. To achieve this, it is imperative to identify the traces that need to be considered in the analysis. Therefore, we use the concept provided by the TSP called *experiments*. An *experiment* is essentially a compilation of all the desired traces required for a particular analysis. In our case, the coordinator will ask each analysis node to create an *experiment* which will comprise the traces necessary for the analysis. When the user will ask to get the

information about the *experiment*, the coordinator will broadcast the request and aggregate the response from each analysis node.

When the *experiment* is created, the different available analyses are pulled and sent back to the user. Every analysis available has to be associated with one or more views. For each association, a data provider has to be implemented to transform the analysis into the view model. Presently, TSP provides four distinct view models for displaying the results of an analysis, depending on the data provider available in the back-end : Table (e.g. Events Table), XY Graph (e.g. CPU Usage), Time Graph (e.g. Thread Status), and Data Tree (e.g. IRQ Analysis - Latency Statistics). In our implementation, we focused on two of those views : XY Graph and Time Graph. These views models allow the visualization tool to directly display data to the user, without prior processing. This facilitates the avoidance of resource constraints that may exist in specific workstations or applications, allowing for the optimal utilization of the remote resources that are available. In these views models there are two key axes : the vertical and horizontal axes. The vertical axis represents all the processes within the trace and is visually represented in rows (e.g. state, cpu usage), while the horizontal axis signifies time and is visually displayed in columns.

When an experiment is created and traces are added to it, if the traces are independent, they merely contribute additional rows to the basic analysis, thereby extending the data along the vertical axis. It could be that certain analysis (number of events, CPU usage) aggregates the traces in the experiment for some views models. However, if the traces originate from the same tracing session and this session is segmented into multiple trace parts, as exemplified in "Lttng Log Rotation" [86], the data can expand both along the vertical and horizontal axes.

In certain scenarios, specific analyses may necessitate access to more data for completing the analysis, such as the critical path analysis. Unfortunately, the aim of allowing the analysis node to conduct the analysis and only return transformed data does not facilitate the straightforward execution of complex analyses that involve interrelations between traces. TSP does not currently provide any URI for directly retrieving this unprocessed data. Consequently, we propose a solution to extend the architecture, enabling users to access more data for the various views and at the same time allow more features that could be useful for users in the future.

Aggregation of independent trace (vertically)

Our architecture, conveniently exposes the analysis results in the form of a view model. As previously discussed, this approach offers the advantage of enabling the client to directly

display the data without the need for additional processing. Consequently, our proposed solution focuses its aggregation efforts on this transformed result, as opposed to the raw data retrieved by the analysis node.

Most of the analyses are currently represented as a series of data points over a timeline. Therefore, when requesting an analysis result via the protocol, it primarily necessitates providing the timestamps for the analyzed timeline and the requested items for the analyzed process needing to be displayed. Listing 3.1 depicts the body parameters that are sent through the HTTP request. Upon receiving requests from the client or the view, the coordinator node can efficiently retain the request in its original form, duplicate it, and broadcast it to the other nodes. Subsequently, it aggregates the returned responses in a cohesive manner.

Listing 3.1 TSP - Query Example

```

1 "parameters": {
2   "requested_timerange": {
3     "start": 1539975458466483700,
4     "end": 1539975465845162800,
5     "nbTimes": 43455
6   },
7   "requested_items": [
8     0,
9     1
10  ]
11 }
```

This approach works well because the traces are analyzed from different machines, generated independently, but during the same time frame. As a result, the querying time interval remains consistent. In essence, we can liken this process to a Map/Reduce paradigm, with the distributed data processing involving the analysis nodes acting as the workers during the Map phases, and the coordinator node overseeing the Reduce phases. Upon receiving a request, the coordinator queries the analysis nodes and then merges all their responses, to produce the final response for the client. Most of the analyses are trivial, we do not recalculate anything and the coordinator has for only job to join the responses as depicted in Figure 3.4.

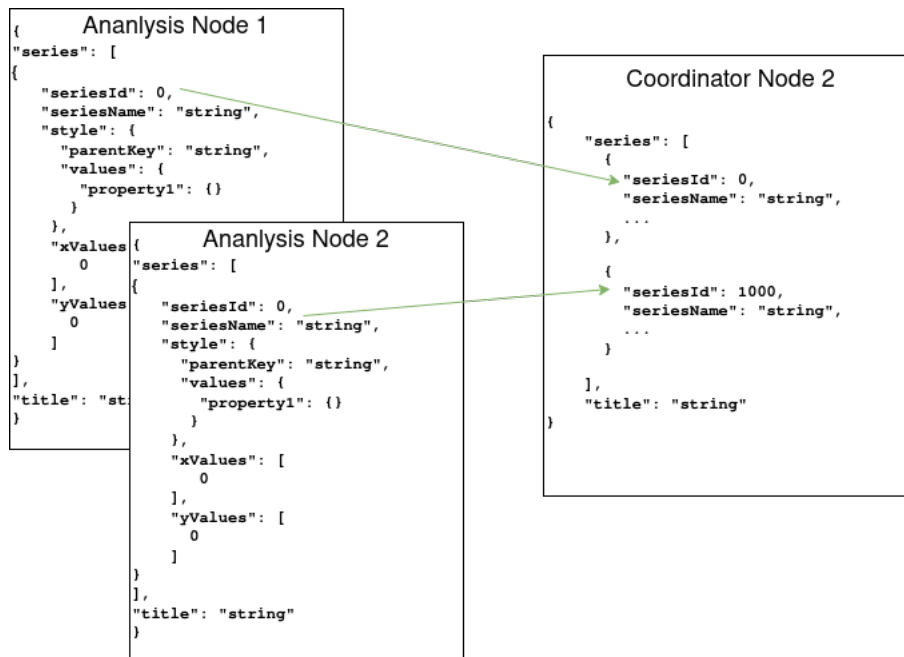


FIGURE 3.4 Merging JSON

Consider a use case wherein data aggregation from various sources involves computing the ratio of each value to the total sum of data (e.g., the percentage of CPU usage). Given that the traces are independent and the percentage is calculated based on the resources of the machine where the trace was collected, there is no necessity to recalculate the ratio of each value.

The issue observed in this particular scenario pertains to the absence of information regarding the potential outputs of other analysis nodes. Each row presented within the view necessitates a distinct identifier. The generation of such identifiers is intrinsic to an analysis node, operating without awareness of the outcomes derived from similar processes in other nodes. Consequently, predicting the potential overlap of identifier numbers proves challenging, as the analyses engage with an abstract layer unbeknownst to them, believing they are directly interacting with views. When generating the data, each analysis node will increment a counter for each row that will be created, starting from 0. Therefore, when the coordinator will aggregate the response from each analysis node, it will send back to the client/views many series of data points with the same identifier.

The first option is to add an URI to the protocol to communicate with the analysis node and constrain the range of identifiers available before requesting an analysis. This solution could work and be efficient, but will require configuring the analysis node. The alternative approach available resembles the initial option. However, rather than applying the reserved value range

to generate identifiers within the analysis node, we opt for its direct implementation within the coordinator node.

A significant advantage of this approach is that there is no need to add an URI. Since assigning the parser node `id` is an arithmetic operation or a simple bitwise one, the latter is not really expensive and can be performed in parallel on the coordinator node. Both options allow the filtering of the analysis node that have to contribute to the analysis, if the item ids are requested in the query. The proposed solution implements the second approach, because the operation is centralized in the coordinator node, and experiments revealed a negligible cost of this operation, even with large amounts of data, on the processing speed. The Algorithm 1 presents a pseudo code that chooses the different steps for a trivial aggregation.

Algorithm 1: Simple Aggregation

Input: computing nodes $CN[]$, body of the request $Body$
Output: response with aggregate data $Response$

```

1 function aggregate( $CN[], Body$ )
2   for  $CN$  in  $CN[]$  do
3      $RequestedItem[] \leftarrow Body.requestedItems$ 
4      $Body.requestedItems \leftarrow []$ 
5     for  $RequestedItem$  in  $RequestedItem[]$  do
6       if  $isValidEncodeEntryId(RequestedItem)$  then
7          $entryId \leftarrow decodeEntryId(CN.id, requestedItem)$ 
8          $Body.requestedItems.add(entryId)$ 
9      $Data[] \leftarrow CN.sendRequest(Body)$ 
10  forall  $Data[]$  do
11     $Data.entryId \leftarrow encodeEntryId(CN.id, Data.entryId)$ 
12   $Response \leftarrow Data[]$ 
13  return  $Response$ 
14 function encodeEntryId( $CNId, EntryId$ )
15    $Step \leftarrow Long.MAX\_VALUE/N$ 
16   return  $EntryId + Step * CNId$ 
17 function decodeEntryId( $CNId, EncodeEntryId$ )
18    $Step \leftarrow Long.MAX\_VALUE/N$ 
19   return  $EntryId - Step * CNId$ 
20 function isValidEncodeEntryId( $CNId, EncodeEntryId$ )
21    $Step \leftarrow Long.MAX\_VALUE/N$ 
22   return
      $EncodeEntryId \geq Step * CNId \ \&\& \ EncodeEntryId < step * (CNId + 1)$ 

```

Cluster Analysis

Gaining scalability involves ensuring that distributed nodes handle local analyses. The analysis of collected traces can be independently visualized on each analysis node of the cluster, without altering the basic architecture. The coordinator merely consolidates the information received from the workers and does not process the trace data. Since the volume of presented information remains constant (e.g., on a 4K screen), the coordinator should not encounter scalability issues in this scenario.

In the second scenario, an analysis may be local on the worker, but the number of workers exceeds the number of pixel rows on the display. Consequently, the coordinator aggregates the responses from multiple workers to display each row of pixels. It is essential to note that, in this scenario, the coordinator experiences higher demand and risks becoming a bottleneck in the infrastructure. However, we do not anticipate encountering this situation in the near future (i.e., having more than 1000 workers).

In cases where there are more workers than pixel rows on the screen, our proposed solution involves indicating the requested items (e.g., processes) using the identifiers generated by the analysis node that the user wants to display. This allows filtering the output for each analysis node and ensures that the sum of each pixel row equals the desired number of pixel rows displayed by the user.

Furthermore, some analyses can be summarized as computing metrics dependent on tracing data. In such cases, we expect the analysis node to perform the bulk of the work upfront to calculate intermediate metrics. These intermediate metrics can then be easily aggregated by the coordinator. The coordinator may retrieve the necessary data to calculate these global metrics (e.g., total number of CPU cores) and request each worker to compute the process utilization rate based on this global value. This results in predominantly local analysis, with a globally calculated factor considered. Generally, calculating these metrics requires accessing specific resource information (e.g., number of CPUs) behind the scenes. Hence, it is beneficial to expose these resources used by the analysis through a URI.

Moreover, we consider that users might want to adjust these resources for the analysis. This approach aids in tasking each analysis node to perform the complete analysis. However, if the coordinator aims to enhance the analysis at its level, it will alter the resource information used by each analysis node for the analysis and initiate it accordingly.

Algorithm 2: Cluster Analysis base on global resource

Input: computing nodes $CN[]$
Output: response with new metrics data $Response$

```

1 function clusterAnalysis( $CN[]$ )
2   for  $CN$  in  $CN[]$  do
3      $Resource \leftarrow CN.fetchResource()$ 
4      $GlobalResource[] \leftarrow calculGlobalResource(Resource)$ 
5     forall  $GlobalResource[]$  do
6        $CN.updateResourceForExperiment(GlobalResource)$ 
7     for  $CN$  in  $CN[]$  do
8        $Response \leftarrow CN.analysis()$ 
9   return  $Response$ 

```

Critical Path Analysis

We can consider more comprehensive analyses that require the coordinator to calculate the final value instead of simple aggregation. One such scenario is when the analysis necessitates access to the entire trace, as seen in the critical path analysis.

The critical path analysis aims to identify the longest path of a process to understand what is hindering its progress. Various factors, such as disk access, network issues, or locks, can cause a process to become blocked. While most of these issues can be resolved easily with trace data available on the analysis node, network-related blockages require information from other traces to accurately determine the blocking state. When the analysis encounters a network blocking state, it attempts to find events related to network communication [87]. Typically, the necessary information to match events is found within the network event payload. By identifying two events with the same network identification using an algorithm, a correlation can be established, potentially providing information to resolve the blocking state.

At first glance, there are two approaches to solving the critical path : front to back or back to front. Giraldeau et al. [88] demonstrated that progressing from front to back can encounter challenges. As depicted in Figure 3.5, the source may emit numerous arrows, and traversing all the paths can be computationally expensive. Conversely, moving from back to front simplifies the process of determining the cause of the delay. As shown in Figure 3.5, only one event can awaken the blocking state, facilitating the efficient identification of the path causing the process to stall, and enabling further analysis.

Denys et al. [81], proposed a way to handle the calculation of the critical path in a distributed environment. However, the proposed solution does not take into account the flexibility

constraint which is a crucial factor for the standardization of the approach. The algorithm reliance on peer-to-peer communications, without using any standardized protocol, limits its portability to other systems. Nonetheless, having indexes to target the servers is crucial for resolving the critical path without broadcasting to every analysis node.

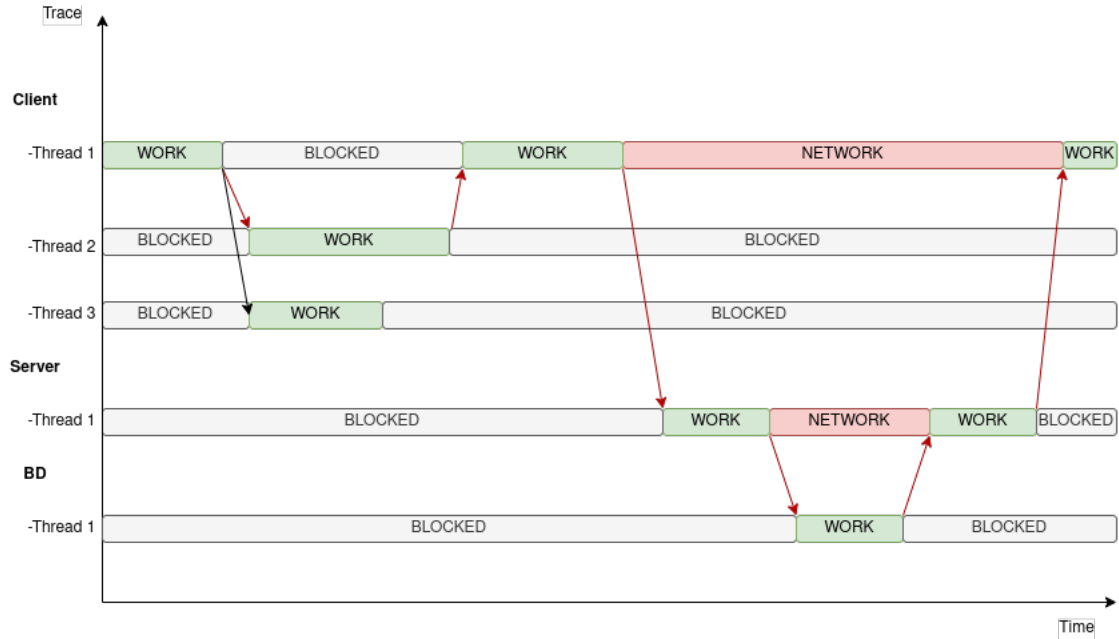


FIGURE 3.5 Critical Path

To solve this challenge, our approach integrates a high level of flexibility and abstraction. In this particular scenario, the type of views for displaying the critical path is determined first. While reusing the Time Graph view model offered by TSP may seem relevant since the critical path display primarily consists of states and arrows. However, it is not sufficient for enabling a distributed analysis. The data required to complete the critical path computation cannot be obtained solely from the time graph responses. Therefore, our solution allows introducing another generalized view that can benefit multiple analyses. To achieve this, we propose implementing a view based on the elements of an execution graph, to allow the system to collect the necessary details for distributed analysis, and expose the following components : vertices and edges.

Listing 3.2 Graph View Model - Vertex

```

1 {
2     timestamp: 0,
3     workerId: 0,
4 }
```

Listing 3.3 Graph View Model - Edge

```
1 {  
2     sourceVertex: {  
3         timestamp: 0,  
4         workerId: 0,  
5     },  
6     targetVertex: {  
7         timestamp: 0,  
8         workerId: 0,  
9     },  
10    context: ""  
11    qualifier: ""  
12 }
```

The only missing information needed to compute a resulting path or graph is contextual data that can match the vertices of the multiple graphs provided by the analysis nodes. The identification of these vertices must be universal and determined by specific factors during implementation. To decouple the implementation from behavior, the flexibility of our solution allows using a generic UUID (Universally Unique Identifier) that is generated according to the implementation. This solution offers the advantage of allowing the analysis to determine the properties for the connection points between the local execution graphs. For instance, in the case of the critical path analysis, properties related to network communication are used to generate the UUID. Each UUID indicates the vertices of each trace that had network communication. Therefore, the indexes have the following model 3.4

Listing 3.4 Graph View Model - Index

```
1 {  
2     vertex: {  
3         timestamp: 0,  
4         workerId: 0,  
5     },  
6     index: "123e4567-e89b-12d3-a456-426614174000"  
7 }
```

Although this solution aims to standardize and make the critical path analysis portable, it is

essential to acknowledge the absence of experimental context. The current implementation of Trace Compass lacks support for this functionality, and integrating such a solution would necessitate alterations to the Trace Compass implementation. However, addressing this issue is beyond the scope of this research.

Algorithm 3: Critical Path Analysis base on Execution Graph API

Input: node that own the process NP
Input: computing nodes $CN[]$
Input: process P
Output: response with critical path *Response*

```

1 function coordinatorCriticalPathAnalysis( $NP, CN[], P$ )
2   for  $CN$  in  $CN[]$  do
3      $Indexes \leftarrow CN.fetchIndexes()$ 
4      $CP \leftarrow NP.criticalPathAnalysis(P)$ 
5      $NetworkEdgeStack \leftarrow findNetworkBlockingState(CP)$ 
6     while  $!NetworkEdgeStack.isEmpty()$  do
7        $NetworkEdge \leftarrow NetworkEdgeStack.pop()$ 
8        $Index \leftarrow findIndex(NetworkEdge.targetVertex)$ 
9        $Vertex \leftarrow findMatchingVertex(index.UUID)$ 
10       $ComputingNode \leftarrow findComputingNode(CN[], index)$ 
11       $Edges \leftarrow ComputingNode.resolveBlockingState(Vertex)$ 
12       $NetworkEdgeStack \leftarrow findNetworkBlockingState(Edges)$ 
13       $CP \leftarrow Edges$ 
14     $Response \leftarrow CP$ 
15  return  $Response$ 

```

3.5.4 Deployment

To simplify the deployment of the cluster, we use a tool to manage the containers and a configuration management tool.

We built Docker images for the coordinator¹ and the Trace Compass Server². Each Trace Compass Server runs inside of a container based on Alpine Linux tagged 3.16.0 and runs on OpenJDK version 17. The coordinator runs inside of a container based on eclipse-temurin tagged 17-alpine and runs on OpenJDK version 17. On each physical computer, Apptainer [89] version 1.1.7-1.fc38 is installed instead of Docker. Apptainer was used because it does not require root access.

1. <https://hub.docker.com/r/spiritus2424/trace-coordinator>

2. <https://hub.docker.com/r/spiritus2424/trace-server>

For the configuration management tool, we use Ansible [90] version 2.10.8 to automate the configuration and the deployment of the cluster.

3.5.5 Visualization

The visualization tool chosen is **Theia** with the **Theia Trace Viewer** extension. The advantage of this solution is that it implements the TSP protocol. This protocol already defines several URIs based on the different views to which the analyses provide results to display them. Since the TSP protocol was implemented in the coordinator, the visualisation tool can be directly used without major changes. The **Theia Trace Viewer** extension architecture 3.6 includes some third-party libraries, a timeline-chart libraries and the `tsp-typescript-client` library³. The **Theia Trace Viewer** Extension needs to access the file system to select the traces for the analysis.

In order for this approach to work, the coordinator needs to interact with the **Theia** Front-end and Back-end. The assumption is made that the **Theia** Back-end instance is running on the same host as the Trace Compass Server. Upon selecting the traces no more interactions happen between the extension and **Theia**. However, the interactions are done only with the Trace Compass Server using TSP.

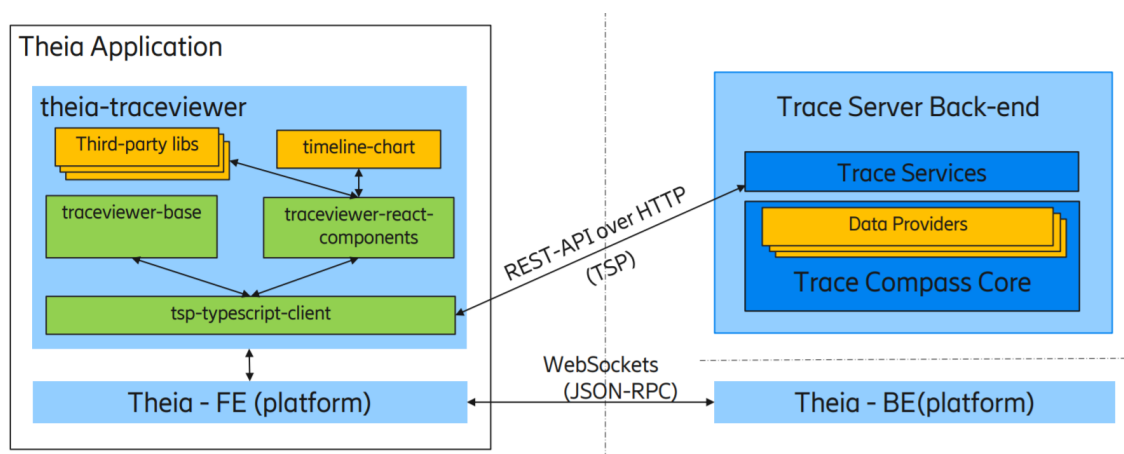


FIGURE 3.6 Theia - Trace Viewer Architecture

Source: <https://github.com/eclipse-cdt-cloud/theia-trace-extension>

Since the TSP protocol does not allow exploring the traces available on the analysis nodes. It was decided that the role falls to the visualization tool to enable finding traces on remote

3. <https://github.com/eclipse-cdt-cloud/tsp-typescript-client>

machines. The **Theia Trace Viewer** tool has a file explorer allowing navigating folders and files available on the machine wherever the application is running.

In the context of the cloud-based version of Theia, the file explorer functionality enables the exploration of files and folders on the remote machine where the Theia server is initiated. However, this setup introduces a challenge : traces are distributed across various remote machines. Since Theia lacks the capability to function as a distributed application, capable of simultaneously exploring files from multiple machines, we have made the decision for this validation prototype to provide an alternative solution that operates independently of Theia. This alternative allows for the opening of desired traces by different analysis nodes. Consequently, other visualization tools that solely implement the TSP protocol, without incorporating a file explorer, will be able to utilize the coordinator. It is conceivable that tracers may contribute features to aid in localizing trace files and simplifying this process, although this falls outside the scope of the present paper.

The user will be able to generate a *.trace-coordinator.yml* configuration file. This file is separated into two sections. The first is the definition of the analysis nodes available in the cluster and the traces to open when the coordinator node is ready. The second part consists of indicating the desired experiments when starting the coordinator. When the file is generated, then the coordinator can be started, the traces are opened and experiments are created. The **Theia Trace Viewer** tool can be configured to directly query the coordinator node instead of an analysis node.

3.5.6 Generating the Configuration file

As discussed in the previous subsection, a configuration file is needed to specify where to find the trace files and the trace servers. To simplify this process, a command line interface in Python, available in the referenced repository⁴, is used.

The command line interface allows generating a *.trace-coordinator.yml* configuration file. The user defines the IP addresses and ports for each Trace Compass servers to list all the IP addresses and the ports in the *.trace-coordinator.yml* file. The trace path on each analysis node can be added as well. Following the generation of the configuration file, the command line tool can be used to add experiments.

```
1 trace-servers:
2   - host: "http://{{server-url}}"
3   port: 5000
```

4. <https://github.com/Spiritus2424/trace-coordinator-cli>

```

4     traces-path: [ "/home/tracing" ]
5 experiments:
6   - name: "Experiment 1"
7     traces-path: [ "/home/tracing/client", "/home/tracing/
      server" ]

```

3.6 Results

The detailed results obtained in this paper and the source code of the experiments are available in this git repository.⁵

In this section, we focus on evaluating the proposed solution. To this end, we implemented the Master-Worker architecture. While the proposed architecture solves the initial problem, we want to ensure, for a wide number of analysis nodes and range of trace sizes, that the overhead in the master is low and the performance improvement scales better than the standalone trace analysis tool. For our experiments we focus on two popular analysis : Thread Status Analysis and CPU Usage Analysis.

3.6.1 Thread Status Analysis

The Thread Status Analysis is an analysis that tracks the state of each thread over the trace duration. In TSP, this analysis can be requested using the *Time Graph API*. The response returned from the analysis server is formatted as states and arrows for a Time Graph. The X axis being the timeline and the Y axis being the thread states. To retrieve the Time Graph models, we have to retrieve the Time Graph tree, which corresponds to the different processes as well as their identification numbers and their hierarchy as they will be displayed in the Time Graph.

3.6.2 CPU Usage Analysis

CPU Usage Analysis is an analysis that shows the CPU usage of different processes over the trace duration. In TSP, this analysis can be requested using the *XY API*. The response returned from the analysis server is formatted as a model for an XY chart. The X axis being the timeline and the Y axis being the CPU usage. To retrieve the XY model, we have to retrieve the XY tree, which corresponds to the different processes as well as their identification numbers and their hierarchy.

5. <https://github.com/Spiritus2424/trace-coordinator>

3.6.3 Query Protocol

To ensure a reasonable response time, the protocol frames the response with a generic response model, to allow the server to return a partial response in cases where the request completion time would be too long. This requires the client to poll the server for the request as the request completes. In the context of the results, we wanted to evaluate the speedup in relation to the time needed to be able to obtain a complete analysis. This is why we will scrutinize the analysis until we obtain the complete analysis. To better understand the extent of the additional cost of our solution, the additional cost will be based on the partial response.

3.6.4 Execution time overhead

The Master-Worker architecture introduces an execution time overhead. This overhead is mainly composed of aggregating time and serializing/deserializing the data. The network latency is removed from the equation since the cluster is under the same network. The execution time overhead is given by :

$$Overhead(ms) = T_{Aggregation} + T_{Serialization} \quad (3.1)$$

To get the view models, the client makes an HTTP request to the master, which replicates the request for each worker in the cluster, querying the Data Providers. Thus, the time for completing a request is the sum of the longest time that the master had to wait after a worker and the master execution time overhead.

$$T_{Request} = T_{LongestQuery} + Overhead(ms) \quad (3.2)$$

3.6.5 Test Environment

To evaluate our solution, the execution time of different analyses was measured on a single Trace Compass server, and was compared to that of our model which includes a coordinator and different sets of Trace Compass Servers, with the same collection of traces. Trace Compass Server instances and the coordinator were deployed on servers located on the same local network. The cluster used in the experiments was composed of 22 physical computers, 20 configured as analysis nodes, 1 configured as a coordinator node and another as a viewer. Each machine in the cluster contained an Intel Core i5 4570s 2.9 GHz, 16 GB of RAM and 1000 GB of disk and ran Fedora version 38. All traces were generated by tracing the essential kernel events of the machine with LTTng 2.13.9 [86]. For consistency, we had a set of trace

collections used for testing. We made sure the requests were using the same trace collections with our solution and with Trace Compass Server alone. If analyzed by one single instance of the Trace Compass Server, the traces are grouped on the same machine. In the model with the coordinator, the traces are distributed equally among each analysis server. In other words, if we have 20 traces and 5 analysis nodes, each analysis node will alone have 4 traces to process. No traces will be placed on the machine where the coordinator is located since the coordinator does not have the ability to directly read the traces.

During the tests performed to assess data transfer and execution time overhead, the client initiated requests to the server, seeking analysis results covering the entire trace duration with realistic resolution settings. Furthermore, the client issued requests for various analyses while ensuring uniform parameters across each request. Three primary factors were identified as influencing data transfer and execution time overhead : the desired model resolution, the number of distinct processes, and the trace size.

To conduct the analysis, we evaluated the processing speed using two sets of 20 traces. The initial set comprised a total of 40MB, while the second set totaled 40GB. For both sets, analyses were conducted using 1, 5, 10, and 20 analysis nodes. Notably, when employing a single analysis node, the coordinator was omitted as all necessary information resided within the analysis node itself. Consequently, there was no requirement for an intermediary step to aggregate responses before they were returned to the user. It is essential to recognize that our solution introduces a noticeable overhead : serialization/deserialization.

We wanted to evaluate the real improvements that users will experience with our solution, making sure to reproduce queries identical to those of users. To do this, we adjusted our queries to take into account the screen resolution used. Ultimately we want to display the information in a window width of a certain number of pixels width. Information is needed for each pixel and the number of pixel may be much less than the number of states. The resolutions taken into account are 1080 pixels, 2000 pixels and 4000 pixels. The resolution solely impacts the following API URIs : "Get Time Graph States" and "Get XY Models" by necessitating the loading of additional models.

It is essential to recognize that ensuring load distribution, when retrieving visualization models for an analysis, presents challenges. This complexity arises because each node possesses specific traces, and no traces are duplicated within the system. In our approach, when requesting analysis models, it is imperative to specify the visualization entry identifiers. As outlined in section 3.5, our solution filters entries based on user-requested information. To achieve this, we reconstruct the query for each analysis node, incorporating entry identifiers that match the analysis node identifier range, before dispatching a customized query. When

there is no identifier that matches the analysis node identifier range, no query will be sent to the analysis node. An issue arises when all identifiers correspond to a single analysis node, as this implies that all computation will be concentrated on this node. This problem can occur only for the API URIs : *Get Time Graph States* and *Get XY Models*. The same problem can happen when an experiment is created using the *create experiment* URI. Creating an experiment requires specifying the UUIDs of the trace that should be included in the experiment. If the traces of the experiment are focused on a few analysis nodes, the analysis will not be able to benefit from our solution.

In general, it is evident that the solution has enhanced the response time. However, it cannot be conclusively stated that the response meets the standard of 200ms [91]. Each benchmark in Table 3.2 begins with the total size of trace files for that benchmark, followed by the number of workers utilized for each row. The columns "Open Trace" and "Create Experiment" denote the computation time required to fulfill the respective requests.

Trace File Size	Number of Workers	Open Trace (s)	Create Experiment (s)
4GB	1	2.83	277.15
	5	1.17	67.81
	10	0.95	35.39
	20	0.89	17.60
40GB	1	1.78	2850.97
	5	1.40	845.37
	10	1.06	444.04
	20	1.03	152.89

TABLEAU 3.2 Benchmark - Response time

3.6.6 Trace API

The benchmark was conducted on the "open trace" URI to assess whether it could potentially act as a bottleneck in the process of obtaining a complete analysis. As depicted in Table 3.2, there is an improvement in the performance of the trace API. Examining the trace presented in Figure 3.7, it is clear that the majority of the time is spent waiting after the analysis nodes. Each request is asynchronously distributed across all analysis nodes. From this, we can infer that the overhead generated by the coordinator node is less than 4.3ms for a response duration of 1559ms.

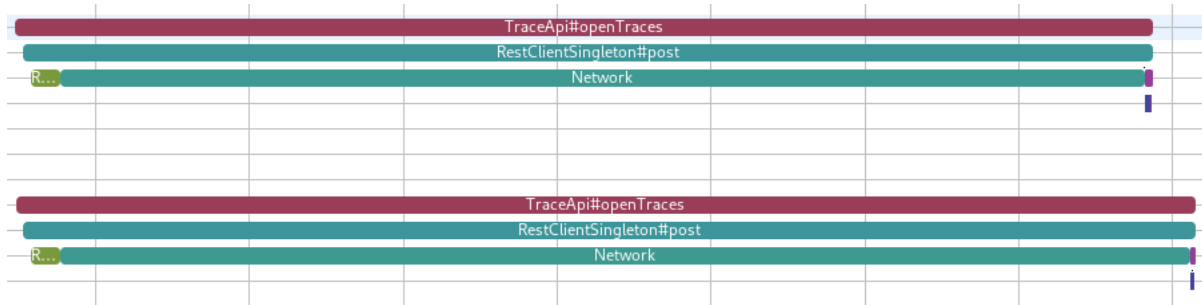


FIGURE 3.7 Call Stack View - Open Trace

3.6.7 Experiment API

The benchmark was conducted on the TSP URIs : "create experiment" and "get outputs descriptors". Figure 3.8 illustrates that most of the time, the coordinator is awaiting a response from the analysis nodes. Fortunately, the solution shows an improvement over using a single analysis node. Creating the experiment represents the lengthiest step in obtaining the results of a trace analysis, as it involves the initial creation of status files, as seen in the tests conducted here. This process iterates through each event from the traces and builds the state system synchronously, one event at a time. Subsequent openings of the same experiment will be faster, as the state files are already created. In conclusion, the overhead generated by the coordinator node is less than 231ms for a partial response that lasts 532.4ms.

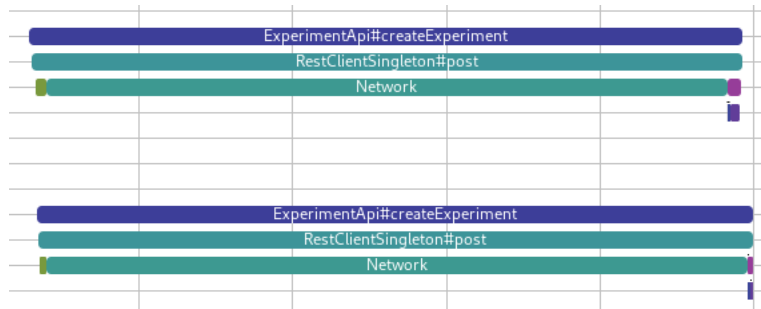


FIGURE 3.8 Call Stack View - Create Experiment

3.6.8 TimeGraph API

Overall, it is evident that the solution has enhanced the response time for the Time Graph API. In each benchmark outlined in Table 3.3, the first column indicates the screen resolution considered. This is followed by the total size of trace files for the benchmark and the number of workers utilized for each row. The columns "Get TimeGraph Tree," "Get TimeGraph States,"

and "Get TimeGraph Arrows" indicate the computation time required to fulfill the request.

Screen Resolution	Trace File Size	Number of Workers	Get TimeGraph		
			Tree (s)	States (s)	Arrows (s)
1080p	4GB	1	362.62	4.46	7.30
		5	43.56	2.42	1.76
		10	31.05	1.97	0.91
		20	24.65	1.77	0.53
	Size of the Response		3.1MB	21.3MB	523.2KB
	40GB	1	3430.79	28.40	36.16
		5	447.29	18.77	8.78
		10	256.73	2.51	8.12
		20	219.49	1.90	5.46
	Size of the Response		139.5MB	24.0MB	32.8KB
2K	4GB	1	315.31	4.50	5.47
		5	43.72	3.14	1.30
		10	30.98	2.79	0.93
		20	23.31	2.54	0.58
	Size of the Response		3.1MB	35.4MB	989.3KB
	40GB	1	3532.46	51.66	86.04
		5	445.16	20.14	11.98
		10	261.71	9.46	8.68
		20	220.66	5.79	2.47
	Size of the Response		139.5MB	39.7MB	250.6KB
4K	4GB	1	380.32	7.28	7.66
		5	43.16	5.54	1.80
		10	31.22	4.39	1.22
		20	24.75	4.36	0.98
	Size of the Response		3.1MB	63.1MB	2.1MB
	40GB	1	2928.45	44.55	65.38
		5	448.91	21.77	13.34
		10	269.43	10.85	8.56
		20	214.35	6.91	4.56
	Size of the Response		139.5MB	71.2MB	1.0MB

TABLEAU 3.3 TimeGraph API Benchmark - Response time

We conducted benchmark tests on the TSP URIs : *Get Time Graph Tree*, *Get Time Graph States*, and *Get Time Graph Arrows*. In an interactive setting, preprocessing the analysis could potentially be performed concurrently with creating the experiment. This could have reduced the time required to receive a response from the *Time Graph Tree* URI. However, in our experiments, we waited until the Trace Compass server was completely terminated before invoking the *Time Graph Tree* URI. Nevertheless, the solution we propose significantly

improves the total response time for obtaining a complete analysis, by a factor of 13 to 15 with 20 analysis nodes, compared to a single node. We utilized a widely used analysis called *Thread Status* in Trace Compass Server. After querying the *Time Graph Tree*, subsequent queries were made for *Time Graph States* and *Time Graph Arrows*.

Additionally, we observed that with smaller traces, the improvement is less pronounced, suggesting that the data size may become too small to efficiently parallelize the analysis. Conversely, with 40 GB of traces, we observed a much greater improvement.

Another noteworthy observation is the correlation between response size and response time. Depending on the user's tracing analysis and screen resolution, this API can generate a significant amount of data. As the resolution increases, so does the time it takes to serialize the data, subsequently increasing the response size. Unfortunately, the coordinator introduces an additional serialization step to aggregate the *Time Graph States* and *Time Graph Arrows*, which impacts response time more than usual due to the serialization process. This added overhead sometimes offsets the benefits, leading to a scenario where employing just one analysis node without a coordinator might be more efficient. Hence, it is essential to minimize the unnecessary overhead in the response. The time spent on serialization can be observed in the call stack figures, where the segment immediately following the network segment represents the serialization segment (`RestClientSingleton#createTspClientResponse`). As depicted in Figure 3.10, the serialization time, indicated in purple, exceeds the computation time of the analysis nodes. Replacing JSON with a binary format, such as protobuf, would reduce the overhead and further enhance the proposed approach efficiency.

In summary, the overhead produced by the coordinator node is minimal. The overhead of the *Get Time Graph Tree* URI is less than 129ms for a partial response that lasts 2223ms. The overhead of the *Get Time Graph States* URI is 293ms for a request that lasts 5068ms. The overhead of the *Get Time Graph Arrows* URI is less than 12ms for a partial response that lasts 4506ms. Increasing the number of workers has a positive impact on response time, and the coordinator overhead on response time is negligible.

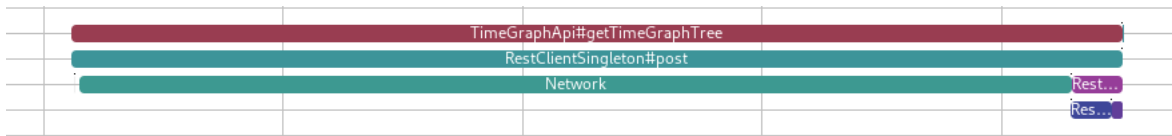


FIGURE 3.9 Call Stack View - Get Time Graph Tree

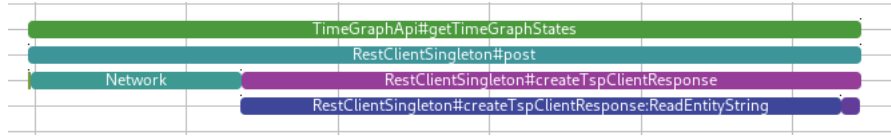


FIGURE 3.10 Call Stack View - Get Time Graph States (4GB)

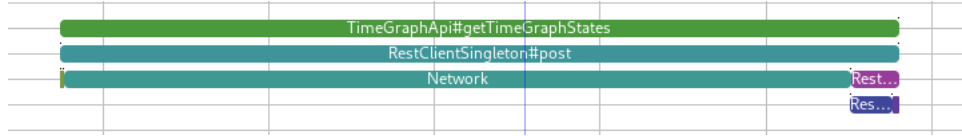


FIGURE 3.11 Call Stack View - Get Time Graph States (40GB)

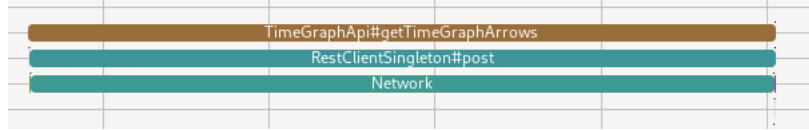


FIGURE 3.12 Get Time Graph Arrows

3.6.9 XY API

In general, it is evident that the solution has enhanced the response time for the Time Graph API. In each benchmark, as indicated in Table 3.4, the first column denotes the screen resolution considered. This is followed by the total size of trace files for the benchmark and the number of workers utilized for each row. The columns labeled *Get XY Tree* and *Get XY Models* denote the time taken to compute the request. The *XY Tree* exposes the entries of the XY graph, which can be utilized to request the *XY Model* via the TSP. An *XY Model* comprises the series of data points required for display in the XY graph.

We conducted the benchmarks on the TSP URIs : *Get XY Tree* and *Get XY Models*. The solution we propose improves the total response time to obtain a complete analysis by a factor of 5 to 12 for 20 analysis nodes, compared with a single node. In our case, we utilized a popular analysis named *CPU Usage* in Trace Compass Server. After querying the *XY Tree*, we proceeded to query the series by employing the *Get XY Models* URI. We also observed that for smaller traces, such as the set of 40MB traces, the improvement might be constrained by the serialization phase, if the amount of data transferred increases. However, for larger traces, there is a much more significant improvement, and increasing the number of workers

Screen Resolution	Trace result File Size	Number of Workers	Get XY	
			Tree (s)	Models (s)
1080p	4GB	1	145.57	10.90
		5	36.99	3.47
		10	33.71	2.12
		20	30.24	1.41
		Size of the Response	1.5MB	7.1MB
	40GB	1	3282.95	704.70
		5	492.16	133.28
		10	339.27	66.00
		20	309.84	37.05
		Size of the Response	130.8MB	7.2MB
2K	4GB	1	141.95	13.75
		5	35.56	4.31
		10	32.18	2.68
		20	30.17	1.93
		Size of the Response	1.5MB	11.8MB
	40GB	1	4992.27	893.76
		5	501.92	180.36
		10	340.31	93.08
		20	303.97	50.41
		Size of the Response	130.8MB	12.0MB
4K	4GB	1	142.48	21.30
		5	37.61	6.06
		10	32.40	3.76
		20	30.09	2.71
		Size of the Response	1.5MB	21.1MB
	40GB	1	3304.87	1435.14
		5	502.20	267.29
		10	349.94	137.52
		20	299.12	77.09
		Size of the Response	130.8MB	21.6MB

TABLEAU 3.4 Benchmark XY API - Response time

also positively impacts the response time. The coordinator’s computation has a minor impact on the response time.

Another aspect we noticed pertains to the response size. Depending on the trace analysis and screen resolution, this API can transmit a substantial amount of data, compared to what we typically observe with the *Time Graph Model* URI. As the resolution increases, so does the time required for serialization and deserialization, consequently resulting in larger response sizes. Unfortunately, the coordinator adds an extra serialization phase to aggregate

the *XY Model*, significantly affecting the response time, due to the additional serialization step. This additional step sometimes outweighs the benefits, potentially making it preferable to use a single analysis node without a coordinator. Thus, it is crucial to avoid overloading the response with unnecessary data. The time spent on serialization can be observed in the call stack figures, with the segment immediately following the network segment being the serialization segment (`RestClientSingleton#createTspClientResponse`). As depicted in Figure 3.14, the serialization segment, shown in purple, contributes to the bottleneck.

In summary, we find that the overhead introduced by the coordinator node is minimal. The overhead of the *Get XY Tree* URI is less than 12ms for a partial response lasting 5463ms, and the overhead of the *Get XY Models* URI is 5ms for a request lasting 1m16s. Increasing the number of workers positively impacts the response time, with the coordinator contributing only a slight overhead to the response time.

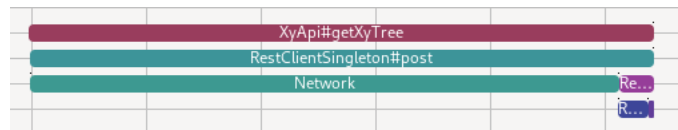


FIGURE 3.13 Call Stack View - Get XY Tree

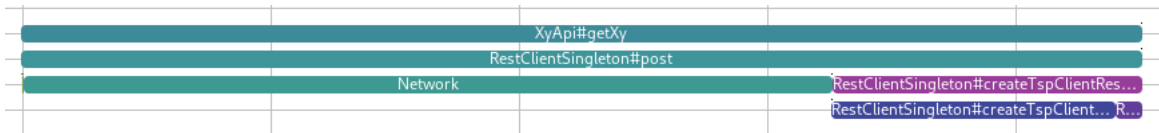


FIGURE 3.14 Call Stack View - Get XY (4GB)

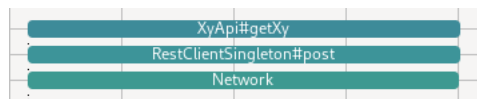


FIGURE 3.15 Call Stack View - Get XY (40GB)

3.7 Discussion

In real-world use cases, such as MPI programs on HPC clusters, programs are run by a large number of nodes with the same architecture and resources. The challenge is to efficiently distribute the computations. Bottlenecks can occur when the load is not perfectly balanced among all the nodes. Other inefficiencies can also happen in the communication between

the different nodes. In our case, because the user screen size limits the number of rows that the user can see, then the request will only be focused on a couple of analysis nodes. The MPI benchmarks presented in this article are representative of real use cases and give an overview of the ability of the new modular distributed architecture to handle large traces, while computing the analysis in a reasonable amount of time.

There is no significant overhead on the coordinator node, except the serialization and the aggregation phase that can be problematic if the amount of data requested is not limited by other factors (e.g. screen resolution). The serialization overhead can be reduced by using a different protocol that uses binary data instead of JSON. Indirectly the overhead of serialization will be less important if the trace files are bigger. The performance gain of our solution is due to the assumption that the traces are independent, but this is not always the case. When the traces are not independent, it means that the traces help to display pixels on the same row on the Y axis. for such case, a new mechanism would be needed.

3.8 Conclusion and Future Work

In this paper, we proposed a solution to improve the scalability of trace analysis for high-performance computing applications on distributed clusters, while maintaining flexibility in developing new analysis. By separating the traces in a cluster of analysis nodes, we were able to drastically reduced the response time. The data provider layer allows us to continue adding analysis based on view models, with very little constraint.

However, our approach enables users to utilize existing analyses, albeit requiring developers to integrate new analyses into the codebase. In some scenarios, we may only need to combine or aggregate a few metrics together. This capability is evident in tools like Prometheus and its PromQL query language, where obtaining the desired information becomes more straightforward, without the need to develop new analyses for each scenario. This querying mechanism facilitates the efficient creation of pseudo-analyses, without the necessity of adding code lines repeatedly.

The current polling protocol generates a partial response and does not provide any indication of when the response will be ready. Consequently, this necessitates frequent polling to check for response readiness, potentially overloading the server when multiple clients attempt to connect and receive updates on their requests. Exploring alternative mechanisms to alleviate unnecessary requests is warranted. Streaming, for instance, allows for opening a connection and transmitting information without the need to close it immediately. Once the server exhausts its data stream, it can then close the stream and the connection simultaneously.

3.9 Acknowledgement

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

CHAPITRE 4 DISCUSSION GÉNÉRALE

4.1 Interopérabilité

Avoir un protocole de communication pour le coordinateur identique à celui des nœuds d'analyse procure l'avantage au client de ne pas avoir besoin d'implémenter un autre protocole pour le bon fonctionnement du système. En d'autres termes, le client n'a pas besoin de savoir si la communication se fait avec un nœud d'analyse ou un nœud coordinateur. Cela permet de facilement appliquer le coordinateur lorsque nécessaire ou de s'en débarrasser sans avoir à le gérer manuellement au niveau de l'outil de visualisation.

Il est intéressant de constater que la responsabilité du coordinateur n'est en aucun cas de lire et d'analyser les traces directement. Nous avons essayé de séparer les responsabilités pour éviter un couplage trop fort entre l'analyse et l'agrégation. Cependant, il est possible que pour des cas très particuliers, l'implémentation d'une analyse ait un impact direct sur l'implémentation de l'agrégation sur le nœud coordinateur. Un exemple assez évident est les analyses qui nécessitent la totalité des traces, comme le chemin critique, pour pouvoir établir les liaisons entre elles.

4.2 Extensibilité

L'extensibilité de la solution est cruciale dans notre contexte. Il peut être très désagréable de devoir naviguer entre plusieurs outils de visualisation qui traitent chacun un format de trace différent et disposent d'une quantité limitée d'analyses. Pour cela, nous avons orienté l'architecture du coordinateur pour qu'elle respecte les mêmes contraintes que celle du nœud d'analyse. Le coordinateur doit être apte à étendre ses agrégations et ses opérateurs sans compromettre les termes du contrat établi par le protocole. C'est pourquoi le coordinateur a la possibilité d'étendre ses agrégations et ses opérateurs en offrant lui-même des analyses ou même une extension d'analyse.

Par exemple, si nous avons une analyse qui cherche à calculer le taux d'utilisation de chaque CPU, mais que nous savons que cette trace a été prise à l'intérieur d'une grappe de calcul et que nous aimerions connaître son taux d'utilisation par rapport au nombre de CPU disponibles dans la grappe de calcul. Le coordinateur pourrait mettre à disposition une analyse spécifique. En plus de l'analyse de base nommée "Taux d'utilisation CPU" que les nœuds d'analyse offrent, on pourrait avoir "Grappe de calcul - Taux d'utilisation CPU" qui viendrait s'ajouter à la liste des analyses disponibles pour le client.

Augmenter le nombre d'analyses disponibles est crucial, tout comme la capacité d'offrir cette même liberté dans les vues disponibles. Notre recherche s'est arrêtée aux modèles de vues de type "XY", "Time Graph" et "Execution Graph". Cela n'empêche pas que l'architecture a la capacité d'implémenter un plus grand nombre de vues modèle (par exemple, des diagrammes circulaires) sans contraindre le reste du système.

La complexité d'offrir aux serveurs une plus grande part de responsabilité pour la définition des entrées et des sorties limite les actions que certaines interfaces utilisateur souhaiteraient mettre en place. C'est pourquoi notre protocole expose la possibilité de définir au niveau du serveur différentes actions selon les vues ou même les éléments qui composent ce modèle de vue. Cela permettra au serveur de fournir les informations d'entrée et de sortie tout en permettant aux interfaces utilisateurs d'avoir des interactions plus personnalisées selon les besoins de chaque implémentation.

4.3 Retour sur les résultats

Les résultats présentés au chapitre 3 reposent sur des valeurs de test qui représentent des cas réels d'application. Pour nous en assurer, nous avons basé nos expériences sur des résolutions d'écran (1080p, 2K et 4K) afin d'extraire une quantité de données qui se rapprochent le plus possible de l'expérience utilisateur. Chaque requête est exécutée une seule fois pour déterminer le temps nécessaire pour afficher une analyse lors du démarrage du système.

Nous avons voulu explorer si le volume des traces impacte le gain en accélération en fonction du volume total des traces à analyser. Pour cela, nous avons récolté deux ensembles de 20 traces faisant au total 4 Go et 40 Go. Nous avons également voulu étudier l'impact de l'accélération en fonction du nombre de nœuds d'analyse disponibles. Pour chaque résolution d'écran et ensemble de traces, nous avons séparé nos résultats selon le nombre de nœuds (1, 5, 10, 20) utilisés au cours de l'expérience.

En ce qui concerne la mise à l'échelle de notre solution, la distribution de calcul sur chaque nœud nous offre une accélération non négligeable sur le temps de réponse. Plus spécifiquement, nous observons un gain en accélération similaire entre les deux ensembles de traces. Cependant, lorsqu'il est question de la génération des modèles de vues, nous remarquons que les gains sont moins importants avec des volumes de traces plus petits, bien qu'ils restent présents. Il est normal de l'observer, car ce qui est chronophage est la taille de la réponse plutôt que le traitement du nœud d'analyse lui-même. Nous pouvons imaginer que les gains seront présents jusqu'à un certain seuil où les traces seront trop petites pour observer une accélération significative.

Répartir les traces sur un plus grand nombre de nœuds présente des avantages, mais il est difficile d'en profiter au maximum. Dans nos expériences, nous avons remarqué qu'assez souvent, les requêtes se concentrent uniquement sur un sous-ensemble de nœuds d'analyse. Cela explique pourquoi, dans certains cas, nous observons une plus grande accélération avec des résolutions plus élevées par rapport à des résolutions moins élevées.

La taille de la réponse peut devenir un goulot d'étranglement au niveau du coordinateur. Il est facile de constater que la taille de la réponse augmente lorsque la résolution augmente. À ce moment, il est pertinent de connaître le pourcentage du temps total que le coordinateur passe à traiter la réponse par rapport au temps consacré à attendre la réponse. Nous avons découvert que le coordinateur passe la majorité de son temps à attendre après les réponses de chaque nœud d'analyse. Dans certains cas, nous passons même la majorité de notre temps à attendre la désérialisation de la réponse avant de pouvoir joindre les réponses des différents nœuds, car la taille de la réponse est plus grande que le temps de traitement du nœud d'analyse. Nous remarquons ces cas spécifiques surtout dans les requêtes qui demandent les modèles de vue pour l'affichage. Le temps consacré à joindre les différentes réponses et à calculer les nouveaux identifiants, selon une plage de valeurs, prend moins de 1 ms, puisque le tout est majoritairement parallélisé au niveau du coordinateur.

CHAPITRE 5 CONCLUSION

Le présent Chapitre présente une synthèse de notre recherche ainsi que ses différentes limitations, et présente les pistes pour des améliorations futures.

5.1 Synthèse des travaux

Nos travaux ont consisté à développer un prototype basé sur une architecture distribuée, visant à résoudre les problèmes de modularité et de mise à l'échelle rencontrés par les outils de visualisation et d'analyse de traces dans le contexte des systèmes de calcul haute performance.

Pour ce faire, nous avons dû explorer les solutions existantes répondant en partie aux besoins identifiés, ainsi que les différents aspects d'un système distribué, afin de concevoir une solution capable de relever les défis de la mise à l'échelle et de la modularité rencontrés par les outils de visualisation et d'analyse de traces. Trace Compass est un outil polyvalent offrant une large gamme d'analyses et prenant en charge plusieurs formats de traces, mais il peine à traiter efficacement les traces volumineuses. En revanche, la combinaison de Grafana et Prometheus offre une solution prometteuse en termes de mise à l'échelle, mais est limitée dans les options d'analyse disponibles, se concentrant principalement sur l'agrégation de métriques selon le langage PromQL. Le protocole TSP constitue une base solide pour la communication avec un nœud d'analyse et pour l'implémentation de nouvelles analyses en exposant des modèles de visualisation via une couche d'abstraction (*Data Provider*).

Par la suite, nous avons implémenté un coordinateur suivant l'architecture Maître-Ouvrier propre aux systèmes distribués. La communication entre le coordinateur et les nœuds d'analyse repose sur le TSP, auquel nous avons apporté des extensions pour résoudre les problèmes liés à la dispersion des traces sur différents nœuds. Nous avons développé des algorithmes d'agrégation pour répondre aux besoins des différentes analyses, et introduit les APIs des ressources et des graphes d'exécution dans le protocole pour accroître la flexibilité des analyses distribuées.

5.2 Limitations

L'une des limitations du prototype développé réside dans la difficulté à localiser automatiquement les traces sur les nœuds. L'utilisateur de notre solution doit préalablement connaître l'emplacement des traces sur chaque nœud pour pouvoir les traiter.

Lors de notre expérience, il s'est avéré difficile de contrôler la répartition de la charge de travail entre les nœuds, certains endossant une charge de calcul plus importante que d'autres. Une autre limitation, liée à la charge de travail, mais également à notre choix de conception, est que chaque nœud d'analyse est laissé à sa propre implémentation, ce qui limite notre contrôle sur l'utilisation des ressources disponibles. Il sera intéressant de regarder comment choisir quelle trace devrait être envoyée à quel nœud d'analyse pour plus d'efficacité.

La plus grande limitation de notre implémentation survient lorsque nous souhaitons exécuter une analyse nécessitant l'ensemble des traces. L'implémentation doit être réalisée à la fois sur le nœud d'analyse et sur le coordinateur pour agréger les différentes parties de l'analyse.

Une limitation rencontrée lors de la collecte des résultats est liée au protocole d'interrogation, qui nécessite l'envoi de plusieurs requêtes au serveur pour connaître le temps total nécessaire à la complétion de la requête. Cela peut surcharger le serveur si plusieurs clients tentent de se connecter simultanément pour obtenir des informations sur l'état de leurs requêtes.

5.3 Améliorations futures

Notre approche permet d'utiliser les analyses disponibles, mais nécessite que les développeurs ajoutent des analyses dans le code. Dans certains cas, il serait plus pratique de simplement agréger quelques métriques ensemble. Nous pourrions explorer des mécanismes similaires à PromQL pour faciliter l'accès à l'information sans nécessiter le développement d'une nouvelle analyse à chaque fois.

Le protocole d'interrogation actuel génère une réponse partielle sans indiquer quand la réponse complète est prête. Il serait intéressant d'explorer d'autres mécanismes pour éviter la nécessité d'une surveillance constante. Des mécanismes tels que les flux ou l'utilisation de jetons pourraient être envisagés pour améliorer l'efficacité et la performance du système.

RÉFÉRENCES

- [1] A. Martens *et al.*, “Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms,” dans *Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering*, ser. WOSP/SIPEW '10. New York, NY, USA : Association for Computing Machinery, 2010, p. 105–116. [En ligne]. Disponible : <https://doi.org/10.1145/1712605.1712624>
- [2] M. Kolny, “Scaling up the prime video audio/video monitoring service and reducing costs by 90%,” Mar 2023. [En ligne]. Disponible : <https://www.primevideotech.com/video-streaming/scaling-up-the-prime-video-audio-video-monitoring-service-and-reducing-costs-by-90>
- [3] J. Terra, “What is client-server architecture? everything you should know : Simplilearn,” Aug 2023. [En ligne]. Disponible : <https://www.simplilearn.com/what-is-client-server-architecture-article#:~:text=The%20client%2Dserver%20architecture%20refers,model%20or%20client%20server%20network.>
- [4] G. Coulouris *et al.*, *Distributed systems : Concepts and design*. Addison-Wesley, 2012.
- [5] Q. H. Vu, M. Lupu et B. C. Ooi, *Architecture of Peer-to-Peer Systems*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2010, p. 11–37. [En ligne]. Disponible : https://doi.org/10.1007/978-3-642-03514-2_2
- [6] D. Qiu et R. Srikant, “Modeling and performance analysis of bittorrent-like peer-to-peer networks,” *ACM SIGCOMM computer communication review*, vol. 34, n^o. 4, p. 367–378, 2004.
- [7] J. Abdella *et al.*, “An architecture and performance evaluation of blockchain-based peer-to-peer energy trading,” *IEEE Transactions on Smart Grid*, vol. 12, n^o. 4, p. 3364–3378, 2021.
- [8] H.-C. Lu, F. Hwang et Y.-H. Huang, “Parallel and distributed architecture of genetic algorithm on apache hadoop and spark,” *Applied Soft Computing*, vol. 95, p. 106497, 2020. [En ligne]. Disponible : <https://www.sciencedirect.com/science/article/pii/S1568494620304361>
- [9] J. Liu, N. Ding et J. Xu, “The research of the double master/slave system architecture in the mpi parallel environment,” dans *2010 International Conference on Electrical and Control Engineering*, 2010, p. 3098–3101.
- [10] A. Adibfar et A. M. Costin, “Review of data serialization challenges and validation methods for improving interoperability,” *Computing in Civil Engineering 2021*, p. 522–

- 529, 2021.
- [11] D. Allam, H. Grall et J.-C. Royer, “From object-oriented programming to service-oriented computing : How to improve interoperability by preserving subtyping,” dans *WEBIST 2013-9th international conference on web information systems and technologies*. SciTePress Digital Library, 2013, p. 169–173.
 - [12] E. International, “ECMA-404 - Ecma International — ecma-international.org,” <https://ecma-international.org/publications-and-standards/standards/ecma-404/>, [Accessed 21-02-2024].
 - [13] “Architecture Guide — mongodb.com,” <https://www.mongodb.com/collateral/mongodb-architecture-guide>, [Accessed 21-02-2024].
 - [14] Google, “Protocol buffers are a language-neutral, platform-neutral extensible mechanism for serializing structured data,” <https://developers.google.com/protocol-buffers/>, [Accessed 21-02-2024].
 - [15] M. Slee, A. Agarwal et M. Kwiatkowski, “Thrift : Scalable cross-language services implementation,” *Facebook white paper*, vol. 5, n°. 8, p. 127, 2007.
 - [16] A. Prunicki, “Apache Thrift — objectcomputing.com,” <https://objectcomputing.com/resources/publications/sett/june-2009-apache-thrift>, 2009, [Accessed 21-02-2024].
 - [17] A. Sumaray et S. K. Makki, “A comparison of data serialization formats for optimal efficiency on a mobile platform,” dans *Proceedings of the 6th international conference on ubiquitous information management and communication*, 2012, p. 1–6.
 - [18] “Text Format Language Specification — protobuf.dev,” <https://protobuf.dev/reference/protobuf/textformat-spec/>, [Accessed 29-02-2024].
 - [19] G. Brito et M. T. Valente, “Rest vs graphql : A controlled experiment,” dans *2020 IEEE international conference on software architecture (ICSA)*. IEEE, 2020, p. 81–91.
 - [20] [En ligne]. Disponible : https://ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
 - [21] A. Quiña-Mera *et al.*, “GraphQL : A systematic mapping study,” *ACM Computing Surveys*, vol. 55, n°. 10, p. 1–35, 2023.
 - [22] O. Hartig et J. Pérez, “An initial analysis of facebook’s graphql language,” *11th Alberto Mendelzon International Workshop on Foundations of Data Management (AMW)*, 2017.
 - [23] G. Brito, T. Mombach et M. T. Valente, “Migrating to graphql : A practical assessment,” dans *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, p. 140–150.

- [24] S. L. Vadlamani *et al.*, “Can graphql replace rest ? a study of their efficiency and viability,” dans *2021 IEEE/ACM 8th International Workshop on Software Engineering Research and Industrial Practice (SER&IP)*, 2021, p. 10–17.
- [25] O. Hartig et J. Pérez, “Semantics and complexity of graphql,” dans *Proceedings of the 2018 World Wide Web Conference*, 2018, p. 1155–1164.
- [26] E. Wittern, A. Cha et J. A. Laredo, “Generating graphql-wrappers for rest (-like) apis,” dans *International Conference on Web Engineering*. Springer, 2018, p. 65–83.
- [27] M. Holopainen, “Monitoring container environment with prometheus and grafana,” *Metropolia University of Applied Sciences*, 2021.
- [28] B. Zhu *et al.*, “Octopus+ : An rdma-enabled distributed persistent memory file system,” *ACM Transactions on Storage (TOS)*, vol. 17, n^o. 3, p. 1–25, 2021.
- [29] [En ligne]. Disponible : <https://www.jsonrpc.org/specification>
- [30] C. Samsel *et al.*, “Web service to json-rpc transformation,” dans *International Conference on Software Engineering and Applications*, vol. 2. SCITEPRESS, 2013, p. 214–219.
- [31] K. Seemakhupt *et al.*, “A cloud-scale characterization of remote procedure calls,” dans *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, p. 498–514.
- [32] S. Kiraly et S. Szekely, “Analysing rpc and testing the performance of solutions,” *Informatica*, vol. 42, n^o. 4, 2018.
- [33] D. Barros *et al.*, “Editing support for software languages : implementation practices in language server protocols,” dans *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*, 2022, p. 232–243.
- [34] O. Turan et Ö. Ö. TANRIÖVER, “An experimental evaluation of the effect of solid principles to microsoft vs code metrics,” *AJIT-e : Academic Journal of Information Technology*, vol. 9, n^o. 34, p. 7–24, 2018.
- [35] I. Gornev et T. Liakh, “Ride : Theia-based web ide for the reflex language,” dans *2021 IEEE 22nd International Conference of Young Professionals in Electron Devices and Materials (EDM)*. IEEE, 2021, p. 503–506.
- [36] “Language Servers — microsoft.github.io,” <https://microsoft.github.io/language-server-protocol/implementors/servers/>, [Accessed 29-02-2024].
- [37] R. Rodriguez-Echeverria *et al.*, “Towards a language server protocol infrastructure for graphical modeling,” dans *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, 2018, p. 370–380.

- [38] S. Keidel, W. Pfeiffer et S. Erdweg, “The ide portability problem and its solution in monto,” dans *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, 2016, p. 152–162.
- [39] J. Lauener, W. Sliwinski et G. CERN, “How to design & implement a modern communication middleware based on zeromq,” dans *Proc of ICALEPCS*, vol. 17, 2017, p. 45–51.
- [40] Ericsson, “Trace server protocol,” <https://raw.githubusercontent.com/theia-ide/trace-server-protocol/master/API.yaml>.
- [41] Q.-H. Tran, “Architecture pour les analyses distribuées et parallèles de traces logicielles,” Mémoire de maîtrise, Polytechnique Montréal, August 2022. [En ligne]. Disponible : <https://publications.polymtl.ca/10481/>
- [42] Ericsson, *Trace server protocol openapi specification*. [En ligne]. Disponible : <https://raw.githubusercontent.com/theia-ide/trace-server-protocol/master/API.yaml>
- [43] B. Gregg et J. Mauro, *DTrace : Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*. Prentice Hall Professional, 2011.
- [44] J. Edge, “Perfcounters added to the mainline,” *LWN, July*, vol. 10, p. 50, 2009.
- [45] R. Schöne *et al.*, “Scalable tools for non-intrusive performance debugging of parallel linux workloads.” Ottawa Linux Symposium Comittee, 2014.
- [46] “strace — strace.io,” <https://strace.io/>, [Accessed 26-02-2024].
- [47] B. Gregg, “strace Wow Much Syscall — brendangregg.com,” <https://www.brendangregg.com/blog/2014-05-11/strace-wow-much-syscall.html>, [Accessed 21-02-2024].
- [48] 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, n°. 2, p. 1–33, 2018.
- [49] “ftrace - Function Tracer 2014 ; The Linux Kernel documentation — kernel.org,” <https://www.kernel.org/doc/html/v5.0/trace/ftrace.html>, [Accessed 26-02-2024].
- [50] T. Bird, “Measuring function duration with ftrace,” dans *Proceedings of the Linux Symposium*, vol. 1. Citeseer, 2009.
- [51] S. Rostedt, “Finding origins of latencies using ftrace,” *Proc. RT Linux WS*, 2009.
- [52] Y. C. K. Piao, “Nouvelle architecture pour les environnements de développement intégré et traçage de logiciel,” Mémoire de maîtrise, École Polytechnique de Montréal, August 2018. [En ligne]. Disponible : <https://publications.polymtl.ca/3282/>

- [53] 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.
- [54] M. Desnoyers et M. Dagenais, “Low disturbance embedded system tracing with linux trace toolkit next generation,” dans *ELC (Embedded Linux Conference)*, vol. 2006, 2006.
- [55] L. Adhianto *et al.*, “Hpctoolkit : Tools for performance analysis of optimized parallel programs,” *Concurrency and Computation : Practice and Experience*, vol. 22, n^o. 6, p. 685–701, 2010.
- [56] C. Wu *et al.*, “From trace generation to visualization : A performance framework for distributed parallel systems,” dans *SC '00 : Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, 2000, p. 50–50.
- [57] D. A. Mey *et al.*, “Score-p : A unified performance measurement system for petascale applications,” dans *Competence in High Performance Computing 2010 : Proceedings of an International Conference on Competence in High Performance Computing, June 2010, Schloss Schwetzingen, Germany*. Springer, 2012, p. 85–97.
- [58] C. Roessel, “VI-HPS : : Projects : : Score-P — vi-hps.org,” <https://www.vi-hps.org/projects/score-p/>, [Accessed 27-02-2024].
- [59] “Open|SpeedShop : An Open Source Infrastructure for Parallel Performance Analysis — doi.org,” <https://doi.org/10.3233/SPR-2008-0256>, [Accessed 27-02-2024].
- [60] “GitHub - OpenSpeedShop/openspeedshop : Open|SpeedShop is a community effort by The Krell Institute with current direct funding from DOE’s NNSA and Office of Science. It is building on top of a broad list of community infrastructures, most notably Dyninst and MRNet from UW, libmonitor from Rice, and PAPI from UTK. Open|SpeedShop is an open source multi platform Linux performance tool which is targeted to support performance analysis of applications running on both single node and large scale Intel, AMD, ARM, Intel Phi, PPC, GPU processor based systems and on Blue Gene and Cray platforms. — github.com,” <https://github.com/OpenSpeedShop/openspeedshop>, [Accessed 02-03-2024].
- [61] B. H. Sigelman *et al.*, “Dapper, a large-scale distributed systems tracing infrastructure,” Google, Inc., Rapport technique, 2010. [En ligne]. Disponible : <https://research.google.com/archive/papers/dapper-2010-1.pdf>
- [62] D. G. Reichelt, S. Kühne et W. Hasselbring, “Overhead comparison of opentelemetry, inspectit and kieker,” 2021.
- [63] “What is OpenTelemetry? — opentelemetry.io,” <https://opentelemetry.io/docs/what-is-opentelemetry/>, [Accessed 27-02-2024].

- [64] C. Cassé *et al.*, “Using distributed tracing to identify inefficient resources composition in cloud applications,” dans *2021 IEEE 10th International Conference on Cloud Networking (CloudNet)*. IEEE, 2021, p. 40–47.
- [65] F. Wininger, “Conception flexible d’analyses issues d’une trace système,” Mémoire de maîtrise, École Polytechnique de Montréal, April 2014. [En ligne]. Disponible : <https://publications.polymtl.ca/1359/>
- [66] A. Montplaisir *et al.*, “Efficient model to query and visualize the system states extracted from trace data,” dans *Runtime Verification : 4th International Conference, RV 2013, Rennes, France, September 24-27, 2013. Proceedings 4*. Springer, 2013, p. 219–234.
- [67] A. Chan, W. Gropp et E. Lusk, “An efficient format for nearly constant-time access to arbitrary time intervals in large trace files,” *Scientific Programming*, vol. 16, n°. 2-3, p. 155–165, 2008.
- [68] L. Prieur-Drevon, R. Beamonte et M. R. Dagenais, “R-sht : A state history tree with r-tree properties for analysis and visualization of highly parallel system traces,” *Journal of Systems and Software*, vol. 135, p. 55–68, 2018.
- [69] K. G. Kouamé, “Langage dédié et analyse automatisée pour la détection de patrons au sein de traces d’exécution,” Mémoire de maîtrise, École Polytechnique de Montréal, August 2015. [En ligne]. Disponible : <https://publications.polymtl.ca/1900/>
- [70] A. K. Sven Efftinge, “Theia - one ide framework for desktop & cloud,” dans *EclipseCon France 2017*, Toulouse, France, June 2017.
- [71] Y. Chen Kuang Piao, N. Ezzati-Jivan et M. R. Dagenais, “Distributed architecture for an integrated development environment, large trace analysis, and visualization,” *Sensors*, vol. 21, n°. 16, p. 5560, 2021.
- [72] B. K. Do, “Data visualization with integration of grafana and quanta platform,” Thèse de doctorat, UNIVERSITY OF APPLIED SCIENCES, 2021.
- [73] G. GmbH, “Vampir - performance optimization.” [En ligne]. Disponible : <https://vampir.eu/>
- [74] H. Mix, C. Herold et M. Weber, “Visualization of multi-layer i/o performance in vampir,” dans *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2018, p. 387–394.
- [75] F. Reumont-Locke, “Méthodes efficaces de parallélisation de l’analyse de traces noyau,” Thèse de doctorat, École Polytechnique de Montréal, 2015.
- [76] M. Martin, “Analyse détaillée de trace en dépit d’événements manquants,” Mémoire de maîtrise, École Polytechnique de Montréal, August 2018. [En ligne]. Disponible : <https://publications.polymtl.ca/3248/>

- [77] N. Matloff, “Programming on parallel machines,” *University of California, Davis*, vol. 39319, 2011.
- [78] 2019. [En ligne]. Disponible : https://assets.nagios.com/downloads/general/docs/Monitoring_Architecture_Solutions_For_Large_Organizations.pdf
- [79] R. Boncea et I. Bacivarov, “A system architecture for monitoring the reliability of iot,” dans *Proceedings of the 15th International Conference on Quality and Dependability*, 2016, p. 143–150.
- [80] Oct 2023. [En ligne]. Disponible : <https://logz.io/blog/prometheus-architecture-at-scale/#~:text=The%20most%20straightforward%20way%20to,%3A%20hierarchical%20and%20cross%2Dservice>.
- [81] P.-F. Denys, Q. Fournier et M. R. Dagenais, “Distributed computation of the critical path from execution traces,” *Software : Practice and Experience*, 2023.
- [82] P.-F. Denys et M. R. Dagenais, “Scalable distributed computation of the critical path from execution traces,” *Software : Practice and Experience*, 2023.
- [83] H. M. Kabamba, M. Khouzam et M. R. Dagenais, “Vnode : Low-overhead transparent tracing of node.js-based microservice architectures,” *Future Internet*, vol. 16, n°. 1, p. 13, 2023.
- [84] “Trace Compass — eclipse.dev,” <https://eclipse.dev/tracecompass/>, [Accessed 03-04-2024].
- [85] J. Höglund, “An analysis of a distributed tracing systems effect on performance jaeger and opentracing api,” 2020.
- [86] I. Kohyarnejadfarid *et al.*, “A framework for detecting system performance anomalies using tracing data analysis,” *Entropy*, vol. 23, n°. 8, p. 1011, 2021.
- [87] H. Nemati *et al.*, “Critical path analysis through hierarchical distributed virtualized environments using host kernel tracing,” *IEEE Transactions on Cloud Computing*, vol. 10, n°. 2, p. 774–791, 2019.
- [88] F. Giraldeau et M. Dagenais, “Wait analysis of distributed systems using kernel tracing,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, n°. 8, p. 2450–2461, 2015.
- [89] “Home — apptainer.org,” <https://apptainer.org/>, [Accessed 16-10-2023].
- [90] R. H. Ansible, “Ansible is Simple IT Automation — ansible.com,” <https://www.ansible.com/>, [Accessed 16-10-2023].
- [91] “What Is Response Time & How to Reduce It - Sematext — sematext.com,” <https://sematext.com/glossary/response-time/#~:text=Standards%3A%20What%>

20Is%20a%20Good%20Response%20Time&text=A%20web%20response%20time%
20ranging,and%20needs%20to%20be%20fixed., [Accessed 02-04-2024].