Titre: Title:	Architecture pour les analyses distribuées et parallèles de traces logicielles
Auteur: Author:	Quoc-Hao Tran
Date:	2022
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Tran, Q.-H. (2022). Architecture pour les analyses distribuées et parallèles de traces logicielles [Mémoire de maîtrise, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/10481/

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Architecture pour les analyses distribuées et parallèles de traces logicielles

QUOC-HAO TRAN
Département de génie informatique et génie logiciel

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

Août 2022

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Architecture pour les analyses distribuées et parallèles de traces logicielles

présenté par **Quoc-Hao TRAN**
en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

Martine BELLAICHE, présidente
Michel DAGENAIS, membre et directeur de recherche
François GUIBAULT, membre

DÉDICACE

À tous mes ami(e)s de Montréal et dans d'autre parties du monde.
TT, NN, HT, P, KM, TA, L, et MP,
aucune dédicace n'est suffisante pour votre support...

REMERCIEMENTS

Je tiens avant tout à exprimer toute ma reconnaissance à mon directeur de recherche, Michel Dagenais, pour m'avoir donné cette précieuse opportunité de découvrir le monde de la recherche et du traçage, au sein du laboratoire DORSAL. Son expérience et ses conseils m'ont guidé tout au long des doutes et des difficultés durant la réalisation de ce projet. Son suivi et son encouragement, sans aucun doute, m'ont aidé à compléter la maîtrise.

Je tiens aussi à remercier les partenaires industriels du laboratoire : Ericsson, Ciena, EfficiOS et AMD pour leur soutien financier pendant la durée de ce projet de recherche.

Je voudrais également remercier Abdellah Rahmani pour son aide durant le projet, ses retours sur la rédaction de ce mémoire et pour avoir été toujours disponible en cas de besoin. Un grand merci aussi à Geneviève Bastien, Matthew Khouzam et l'équipe Trace Compass d'Ericsson pour toutes les discussions et le support technique. Merci pour leur patience, leur expertise sur Trace Compass et sur plusieurs sujets différents, c'était une contribution absolument essentielle.

Finalement, j'aimerais exprimer ma gratitude à toutes les personnes exceptionnelles que j'ai rencontrées durant le projet à DORSAL et ailleurs. Merci pour toutes les discussions, les sourires, les repas ensemble, merci à Matthew de m'avoir nourri avec ses BBQs, merci à Arnaud, PF, Bohémond, Erica, Paul et encore d'autre personnes, les temps agréables passés avec eux m'ont fait progresser dans les moments le plus difficiles.

RÉSUMÉ

Avec l'augmentation de la complexité dans le développement de logiciel, les développeurs ont besoin d'outils sophistiqués pour faciliter leur travail. Les outils de traçage aident à comprendre le comportement réel des systèmes complexes grâce au relèvement des évènements bas-niveau, au moment précis des anomalies, avec très peu de surcoût, en production.

Une fois que ces traces sont collectées, des outils d'analyse de trace sont nécessaires pour extraire des informations utiles, car il est presqu'impossible de les analyser manuellement. L'expansion de l'utilisation des systèmes distribués et parallèles crée des défis particuliers pour le domaine du traçage. Les outils traditionnels, créés pour analyser les traces des appareils personnels, ne sont pas adéquats pour analyser les traces générées par ces systèmes qui peuvent être gigantesques.

Il existe une multitude de solutions dédiées au traçage des systèmes distribués. Ces outils supportent l'exécution des opérations pour agréger les analyses des traces de grande taille. Cependant, ces opérations sont souvent très couplées au type d'application ciblé par l'outil. Trace Compass est un outil très flexible. Il supporte plusieurs formats de fichiers de trace et offre des analyses très sophistiquées. Il est aussi facile de rajouter le support pour un nouveau type de trace ou de développer une nouvelle analyse, grâce à son architecture avec une base de données puissante qui permet d'effectuer n'importe quel type d'analyse de manière efficace. Cependant, Trace Compass, dans son état actuel, n'est pas encore utilisable dans un environnement distribué. Il lui manque la capacité de fournir une vue globale sur des traces de grande taille, ou de se déployer dans une organisation avec plusieurs instances parallèles, pour être capable de gérer un plus grand volume de traces.

L'objectif de ce travail est de trouver une solution qui prend les meilleures caractéristiques des solutions existantes : un outil de traçage distribué qui fonctionne bien avec les traces de grand volume et offre des opérateurs puissants qui permettent aux utilisateurs de filtrer les données récupérées, tout en gardant la flexibilité pour faciliter l'adoption de l'outil dans des environnements différents.

Nous avons fait une revue complète de la littérature sur les différentes techniques de traçage, et surtout les solutions existantes pour le traçage distribué en particulier. Nous avons souligné les avantages et les limitations de l'architecture de ces outils et avons fait ressortir les points sur lesquels nous pourrions apporter des contributions. Nous avons remarqué qu'il existe un patron d'architecture commun entre ces outils, et que la capacité d'offrir des opérateurs sur le résultat, à travers des requêtes, est très puissant. Nous avons ensuite cherché à proposer

une architecture distribuée, avec Trace Compass comme le module principal d'analyse, pour offrir les avantages du traçage distribué tout en gardant la flexibilité de Trace Compass.

Pour atteindre cet objectif, nous avons cherché d'abord à comprendre le fonctionnement de Trace Compass, et en particulier les interactions entre ses composants principaux, le serveur Trace Compass et son interface graphique. Cela inclut la compréhension du Trace Server Protocol (TSP) utilisé par Trace Compass. Par la suite, nous avons analysé la mise à l'échelle de Trace Compass dans son état actuel. Nous avons ensuite déterminé les opérations d'agrégation potentiellement intéressantes pour notre cas d'usage, et finalement avons proposé une organisation avec plusieurs instances parallèles de serveur de Trace Compass et avons décrit les interactions basiques dans ce modèle. Nous avons aussi fait une revue des protocoles de communication et avons conclu que le protocole TSP est le meilleur compromis en ce moment. Pour la séparation de requête et l'agrégation des analyses, nous avons défini un modèle d'exécution et de programmation qui facilite le développement de nouveaux types d'agrégation, en bénéficiant des avantages de l'approche diviser-rassembler avec de multiples aller-retour de requêtes.

Avant de procéder à l'implémentation, nous avons fait une analyse théorique sur le temps d'exécution de notre modèle et ses surcoûts. Nous avons aussi établi un budget de temps d'exécution pour les opérations de séparation et d'agrégation, qui aide au choix d'un algorithme avec un impact minimal sur la performance. Nous avons ensuite implémenté un prototype et avons fait en sorte qu'il soit compatible avec l'interface graphique existante.

Finalement, les mesures de temps d'exécution ont été menées pour vérifier l'hypothèse initiale. Nous avons pu confirmer l'efficacité de la solution proposée dans certains cas d'usage. Nous avons discuté aussi des limitations de cette solution. Nous avons conclu que la solution proposée comporte un potentiel intéressant et est une preuve de concept d'un déploiement de plusieurs instances parallèles de Trace Compass. Ce concept peut bien fonctionner dans un environnement distribué, cependant, il sera intéressant d'analyser une plus grande variété de cas d'utilisation pour mieux comprendre les gains de performance associés et, dans certains cas, une légère dégradation de performance.

ABSTRACT

The increasing complexity in software architecture requires sophisticated tools to help engineers in their development process. An efficient tool for this purpose is a tracer, which allows deep understanding of the behaviour of the running program, as well as the complex underlying system, by collecting low level events during execution at the near-exact moment of the anomaly, with minimal overhead, in production.

Once these traces are collected, another set of tools are used to analyse and extract useful information from the raw data. Indeed, it is almost impossible to analyse this data manually in modern days, due to their size. The rapid expansion of distributed and parallel systems requires these tools to be adapted, since these types of systems raise specific challenges due to their nature. Very often, traditional tools created for analysing single machines are not suitable for analysing the traces generated from these large parallel systems.

There are currently several solutions explicitly engineered for distributed tracing. These tools allow users to execute operators on the requested data through their query protocol, to aggregate the analysed data for large size traces. However, these operators are often semantically tightly coupled to the type of application for which the tool was created. Trace Compass is a very flexible tool in this sense. It supports various trace formats and sophisticated analyses. It is also relatively easy to add support for new trace formats, or new types of trace analysis, thanks to its architecture and its powerful database that offer efficient queries for any type of analysis. That being said, Trace Compass in its current form has limitations on scalability. It cannot be deployed over multiple parallel instances, to carry the analysis in parallel, and offer an aggregated view, for very numerous or very big traces.

The goal for this work is to find a solution that brings the best of both world: a tool for tracing distributed and parallel systems with good scalability, offering powerful operators to allow complex combinations of operations on the requested data, while maintaining a certain level of flexibility to ease the adoption of the tool in different types of environments.

We did a thorough literature review on trace analysis tools, with a particular focus on existing solutions for distributed tracing. We underline the advantages and limitations in their architecture, and pointed out the areas where we can bring potential contributions. We noticed some common design patterns among these tools and confirmed that the ability to run operators on the requested data is particularly interesting. From there, we attempt to find a working model for distributed tracing, with Trace Compass as the main analysis module, to bring the advantages of existing solutions to the tool, while maintaining its flexibility.

In order to achieve this goal, we first look into the architecture of Trace Compass to understand the interactions between its main components, and the communication between its backend and frontend. This includes understanding its Trace Server Protocol (TSP). Thereafter, we studied the scalability of Trace Compass in its current state. We then defined some common aggregated operators that are interesting for our use cases and proposed a tracing platform with multiple instances of Trace Compass server. We also laid out the basic details of the interactions between its main components. We also reviewed different communication protocols, used in various distributed tracing tools, but concluded that TSP stays a good compromise at the moment of this work. Regarding the separation of request and aggregation of analysis data, we defined an execution model and a programming paradigm that facilitate the development of new aggregation algorithms, while benefiting from the scatter-gather pattern with a multiple pass approach.

Before proceeding with the implementation, we studied the running time analysis of our model, regarding the overhead of the separation and aggregation workload, and also established a running time budget for them based on the original running time analysis. This budget gives insights into how to choose an aggregation algorithm, without punishing too much the overall performance. We then implemented a prototype and in the process made it compatible with the existing front-ends.

In the end, we ran some benchmarks to measure the running time of our model, in order to verify the initial assumptions. We could confirm the feasibility of our model in some specific use cases. We also had discussions on various limitations of our proposed solution. We concluded that the prototype offers an interesting potential, and is a proof of concept that Trace Compass can be deployed with multiple instances and work well in a distributed environment. Nevertheless, for the prototype to be ready for production, more insights about the model would be useful. We also need a wider variety of realistic use cases to more completely characterise the performance gains, or slight degradation in some specific cases.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

TSP	Trace Server Protocol
PID	Process Identifier
TID	Thread Identifier
CPU	Central Processing Unit
RAM	Random Access Memory
LTTng	Linux Trace Toolkit next generation
CTF	Common Trace Format
API	Application Interface
PromQL	Prometheus Query Language
REST	Representational State Transfer
JSON	JavaScript Object Notation
XML	Extensible Markup Language
HPC	High Performance Computing
TC	Trace Coordinator
TS	Trace Compass Server

LISTE DES ANNEXES

CHAPITRE 1 INTRODUCTION

Avec l'augmentation de la complexité des développements de logiciels, les développeurs ont besoin d'outils sophistiqués pour faciliter leur travail. Les outils de traçage aident à comprendre le comportement réel des systèmes complexes grâce au relèvement des évènements de bas-niveau, au moment précis des anomalies, avec très peu de surcoût en production.

Une fois que les traces sont collectées, des outils d'analyse de trace sont nécessaires pour extraire des informations utiles, car il est presqu'impossible de les analyser manuellement. L'expansion de l'utilisation des systèmes distribués et parallèles crée des défis particuliers pour le domaine du traçage. Les outils traditionnels créés pour analyser les traces des appareils personnels ne sont pas adéquats pour analyser les traces générées par ces systèmes. Notamment, pour comprendre le comportement global de ces systèmes, il faut comprendre l'interaction complexe entre leurs composants. Cette information n'est pas forcément facile à extraire à partir de ces traces. De plus, chaque type de système distribué, comme les systèmes hétérogènes ou les grappes de calcul de haute performance, possède des caractéristiques particulières qui demandent un traitement spécifique de son analyse.

Il existe une multitude de solutions dédiées au traçage des systèmes distribués. Ces outils supportent l'exécution des opérations sur les traces pour obtenir des données agrégées qui fournissent une vue globale pour les traces de grande taille. Cependant, ces opérations sont souvent très couplées au type d'application ciblé par l'outil. L'objectif de ce travail est de proposer une architecture qui offre des capacités similaires et en plus une flexibilité pour adapter l'outil aux différents types de systèmes distribués. À travers ce travail, nous apportons les contributions suivantes :

— Une architecture d'analyse de trace distribuée et parallèle pour les applications de grande échelle.
— Un modèle d'exécution pour l'agrégation d'analyses de traces et un cadriciel pour le développement de nouveaux types d'opérateurs d'agrégation.
— Un prototype de la solution proposée basé sur Trace Compass.
— Une évaluation de sa faisabilité, de son efficacité et de sa performance.

1.1 Définitions et concepts de base

1.1.1 Traçage

Le traçage signifie que les techniques utilisées pour enregistrer les événements produisent généralement un surcoût le plus petit possible pendant le cycle de vie d'un logiciel en production. La collection de ces événements peut se passer en plusieurs niveaux : en espace d'utilisateur ou en espace de noyau. Ces événements représentent souvent des changements d'état dans le système (une ouverture de fichier) et sont sauvegardés dans les fichiers (traces). Il existe plusieurs formats de fichiers de traces différents.

Un traçage peut être dynamique ou statique. Le traçage statique requiert que les développeurs placent des points de traces dans le code source, tandis que le traçage dynamique cherche à dynamiquement placer ces points de trace lors de l'exécution des programmes.

1.1.2 Analyse et visualisation de trace

L'analyse de trace est l'extraction des informations utiles à partir des traces enregistrées. Le travail consiste à lire les fichiers de trace puis reconstruire l'état du système à travers le temps. Il est ensuite possible d'étudier ces états pour déduire les informations comme l'usage des processeurs ou de la mémoire, ainsi les interactions entre les processus du système.

Ces informations peuvent être représentées visuellement (e.g. tableaux, graphiques) pour faciliter la compréhension des traces par les utilisateurs. Un outil de visualisation de trace peut faire partie d'un outil d'analyse de trace ou aussi être connecté avec plusieurs outils d'analyse différents, à travers une interface bien définie.

1.1.3 Systèmes distribués et parallèles

Un système distribué est constitué de plusieurs composantes logicielles qui communiquent entre elles à travers une interface d'application, tout servant un but commun. Ces composantes peuvent être géographiquement éloignés les unes des autres, et communiquent entre elles à travers l'internet. Elles peuvent aussi se situer dans le même réseau local. Un système distribué peut être hautement parallèle, notamment les grappes de calcul, mais il peut aussi être exécuté de manière séquentielle comme le patron micro service.

1.1.4 Traçage distribué

Le traçage distribué est le nom commun pour appeler les techniques de collecte, d'analyse et de visualisation de traces adaptées pour les systèmes distribués. Ces systèmes, en raison de leur nature, demandent des traitements spécifiques lors de l'analyse des données de traces.

1.2 Éléments de la problématique

Les systèmes distribués nécessitent des techniques spécifiques pour analyser leurs traces. Malgré un protocole standard d'échange de données de trace, les solutions existantes dans le domaine du traçage distribué sont souvent très couplées avec leur écosystème, et supportent seulement des formats de traces précis. C'est un choix tout à fait raisonnable, parce que, vue la taille de trace générée dans l'environnement des systèmes distribués, ces outils offrent des opérateurs qui permettent de filtrer les données récupérées. Pour que ces opérateurs aient du sens, il faut qu'ils connaissent bien la structure des données, de manière bien définie. Cela rend difficile l'adaptation des outils d'analyse de trace pour les systèmes distribués, comme les grappes de calcul de haute performance ou les systèmes hétérogènes, où chacun a besoin d'analyses de traces spécifiques.

Trace Compass est un outil très flexible pour ce cas. Il supporte plusieurs formats de fichiers de trace et offre des analyses très sophistiquées. Il est aussi facile de rajouter le support pour un nouveau type de trace, ou de développer une nouvelle analyse, grâce à son architecture avec une base de données puissante qui permet d'effectuer n'importe quel type d'analyse de manière efficace. Son code source, bien structuré de style cadriciel, facilite aussi le travail des développeurs.

Cependant, Trace Compass, dans son état actuel, n'est pas encore utilisable dans un environnement distribué. Il lui manque la capacité de fournir une vue globale sur des traces de très grande taille, ou de se déployer dans une organisation avec plusieurs instances parallèles d'analyse de trace, pour être capable de gérer un plus grand volume de traces.

1.3 Objectifs de recherche

Les problématiques mentionnées dans la section précédente encouragent une recherche pour trouver une solution qui prend les meilleures caractéristiques des solutions existantes : un outil de traçage distribué qui fonctionne bien avec les traces de grand volume et offre des opérateurs puissants qui permettent aux utilisateurs de filtrer le nombre de données récupérées, tout en gardant la flexibilité pour faciliter l'adoption de l'outil dans des environnements différents. La recherche peut être formulée selon plusieurs objectifs spécifiques suivants :

- Analyser le fonctionnement de Trace Compass dans un environnement distribué, avec un gros volume de données de trace, dans son état actuel.
- Proposer une organisation pour déployer Trace Compass dans une infrastructure parallèle pour en augmenter sa capacité de traitement.
- Implémenter un prototype de la solution proposée.

— Évaluer sa performance et analyser son surcoût par rapport aux différentes configurations de déploiement.

1.4 Plan du mémoire

Le chapitre 2 présente l'état de l'art du domaine du traçage, les particularités du traçage distribué, et quelques patrons de conception dans le développement des logiciels distribués. Il présente également les avantages et inconvénients des outils de tracage distribué ainsi que leur achitecture. Nous allons voir ensuite dans le chapitre 3 les étapes suivies pour réaliser cette recherche. Le chapitre 4 présente l'article principal de ce mémoire, avec les résultats et leur évaluation. Finalement, nous discutons des détails des résultat obtenus dans le chapitre 5 et conclurons ce mémoire au chapitre 6.

CHAPITRE 2 REVUE DE LITTÉRATURE

Nous allons commencer ce chapitre par présenter le domaine du traçage, puisque qu'il est important de savoir les notions fondamentales sur lesquelles notre travail s'appuie (traçage distribué). Ensuite, nous allons voir le traçage distribué et ses défis techniques. Nous allons regarder aussi en détails les plateformes de traçage distribué existantes, leurs avantages et leurs inconvénients, ainsi que quelques protocoles d'interrogation.

2.1 Traçage

Cette partie présente les différents sujets autour du domaine du traçage. Nous allons commencer par une revue des théories et des techniques de traçage, avec quelques exemples sur les outils de traçage populaires dans l'industrie. La collection de trace crée un besoin pour les outils d'analyse et de visualisation de trace, pour lequel les solutions existantes seront exposées. En particulier, nous allons décrire en détails Trace Compass, l'outil sur lequel s'appuie ce travail.

2.1.1 Synthèse et outils de traçage

Le traçage signifie la collection des événements durant la vie d'un système. La différence fondamentale entre un traceur et une librairie de journalisation est l'importance de minimiser son surcoût pour ne pas perturber le système tracé. Les événements dans une trace sont créés à l'aide de l'exécution de points de trace insérés à plusieurs endroits dans un logiciel, qui lèvent seulement un signal d'entrée/sortie ou bien prennent la forme d'une petite fonction qui récupère certains types de données. Le placement de ces points de trace ne se limite pas à la couche d'application (l'espace utilisateur), mais peut aussi se trouver dans les niveaux plus bas, notamment les appels système, les interactions entre le noyau et la mémoire, le disque, et d'autres parties (l'espace noyau) (Desnoyers et Dagenais, 2006) [4]. Sans avoir explicitement des points de trace dans l'espace utilisateur, le traçage noyau souvent révèle quand même une partie importante des informations sur le comportement de l'application utilisateur, car la plupart des applications passent par les appels systèmes (Gregg, 2014) [5].

Le traçage est une version plus inclusive du profilage, dans le sens où il fournit non seulement les mesures, mais aussi le contexte de l'exécution, ce qui est souvent nécessaire pour le débogage. Le traçage diffère du débogage par le fait qu'il ne retient pas l'état exact d'un programme, comme la valeur des variables, mais plus une capture globale des informations

les plus importantes, par exemple son utilisation de mémoire. Ainsi, par sa nature, le traçage peut être utilisé directement en production, grâce à son surcoût souvent très petit (Toupin, 2011) [6]. Il permet la compréhension du comportement non seulement d'un programme, mais aussi de son interaction avec les autres programmes en concurrence, le système d'exploitation ou encore d'autres machines dans un environnement distribué. Ceci diffère du débogage, qui fournit une vue locale, limitée à un environnement théorique qui peut parfois s'éloigner de celui de production. Cela ouvre la porte à plusieurs types d'applications où l'utilité du débogage est très limitée, notamment l'analyse de blocages dans les systèmes distribués (Fournier et Dagenais, 2010) [7], ou de performance du stockage distribué (Daoud et Dagenais, 2020) [8].

Il existe aujourd'hui 2 techniques de traçage : par l'instrumentation statique ou dynamique. La première technique demande aux développeurs de placer les points de trace dans le code source avant la compilation. Évidemment, il a une meilleure efficacité parmi les 2 méthodes, en présentant toutefois un compromis sur la flexibilité, demandant une recompilation du code source à chaque changement de points de trace. Une autre difficulté est qu'il nécessite aussi une bonne connaissance du code source de la part de l'utilisateur. La flexibilité de l'instrumentation dynamique est compromise par un surcoût souvent plus grand par rapport à l'instrumentation statique, à cause d'une variété de raisons. Cependant, les travaux récents réduisent de plus en plus cet écart (Fahem, 2012) [9].

Plusieurs outils existent pour la collection de traces. LTTng est un outil ayant la capacité de tracer le noyau et les systèmes en temps réel. Le développement est initié par Desnoyers en visant à fournir une grande quantité d'information avec un surcoût minimal, voire zéro surcoût quand les traces ne sont pas actives (Desnoyers et Dagenais, 2006) [10]. LTTng sauvegarde les traces dans des fichiers de format binaire CTF supporté par différents outils de visualisation. SystemTap est un autre traceur, développé par Red Hat et inspiré par DTrace (Prasad et al., 2005) [11], qui compile les scripts de l'utilisateur puis les exécute en différents modes, en tant que module noyau, ou comme instrumentation en espace utilisateur. Une autre alternative de LTTng est ftrace, un choix populaire pour tracer les fonctions du noyau (Bird, 2009) [12].

2.1.2 Analyse et visualisation de trace

Trace Compass

Trace Compass est un outil d'analyse et de visualisation de trace développé en code source ouverte sous le chapeau de la fondation Eclipse. Il est basé sur le cadriciel Eclipse Rich Client Platform (RCP) - la fondation de l'EDI Eclipse pour faciliter la structuration vue-modèle de l'application. Trace Compass supporte une variété de formats de trace comme

CTF, GDB...etc et est prêt à utiliser avec le binaire fourni. Pour d'autres formats non listés, il est aussi facile d'implémenter un analyseur syntaxique sur mesure, puis recompiler le projet pour avoir un produit final. Des analyses avancées sont mises en production, notamment une analyse de flux de thread, d'utilisation de CPU, de mémoire, etc., et l'utilisateur peut aussi définir de nouvelles analyses en format XML (Wininger et al, 2017) [13]. L'outil est construit à partir des composants principaux suivants :

Analyseur syntaxique : produire un format lisible par Trace Compass à partir des formats de trace différents. Les traces importées par l'utilisateur dans Trace Compass sont transformées par l'analyseur à ce format. Les contributeurs du projet peuvent facilement ajouter le support pour un nouveau type de trace en ajoutant des règles de transformation pour le type. Lors de l'importation, l'analyseur syntaxique peut aussi fusionner et synchroniser l'horloge des traces de plusieurs machines en mettant en correspondance des événements, notamment les événements d'envoi et de réception de paquets réseau.

Système d'états : Les analyses dans Trace Compass sont basées sur un système d'états - une base de données formatée pour faciliter les requêtes lors des analyses. Trace Compass supporte plusieurs types de stockage pour la sauvegarde de cette base de données, en mémoire ou sur disque. Le but de ce système d'états est de conserver le contexte de l'exécution entre les transitions d'un événement à l'autre. Il atteint cet objectif en enregistrant un arbre d'attributs - simplement dit, toutes les valeurs de contexte qui sont liées à l'événement à chaque changement d'événement. Un état signifie un attribut et sa valeur a un moment donné. Tous les états produits par la trace sont gardés dans un arbre d'histoire (Prieur-Drevon et al, 2018) [14], où l'implémentation exacte dépend du type de stockage choisi.

Module d'analyse : Un module d'analyse fournit d'abord un "State Provider" - un composant qui produit un arbre d'attributs en lisant les événements. Ce composant est ensuite utilisé par le système d'états pour construire son arbre historique. Naturellement, plusieurs types d'analyses peuvent partager un "State Provider" qui possède des attributs en communs (Ericsson) [15]. Le module d'analyse ensuite fait des requêtes sur le système d'états, maintenant construit grâce au "state provider" pour déduire le résultat de l'analyse souhaitée. Ce résultat est ensuite donné à la vue affichée pour l'utilisateur.

Data provider : À partir de la version 3.3, Trace Compass est découplé en 2 parties, le module cœur qui s'occupe des analyses et la vue pour l'affichage, communiquant à travers une interface (le composant "data provider" (Chen Kuang Piao, 2018) [16]). Chen Kuang Piao et al. a proposé une architecture dans laquelle Trace Compass agit comme un serveur, fournit des résultats prêts à servir à un client qui fait l'affichage. Le résultat du module d'analyse, au lieu d'être donné à la vue comme avant, est maintenant pris par la data provider qui

exporte ces informations. Le module ensuite reformate ces données pour construire le modèle d'affichage du client. Le format de requête et réponse du client de Trace Compass s'appelle Trace Server Protocol TCP.

Le travail de Chen Kuang Piao propose une architecture distribuée de Trace Compass, dans laquelle on peut avoir plusieurs serveurs de trace exécutant en même temps, ce qui augmente la capacité d'analyse pour des traces de grande échelle. Grâce à cette architecture, il est maintenant possible d'avoir une vue développée dans les différentes plateformes qui utilisent un même serveur de trace. Un exemple typique de ce cas est le "trace-extension", une vue développée dans l'EDI Theia dans un environnement Web, qui existe aussi en tant qu'une extension de VSCode et donne une expérience presque similaire.

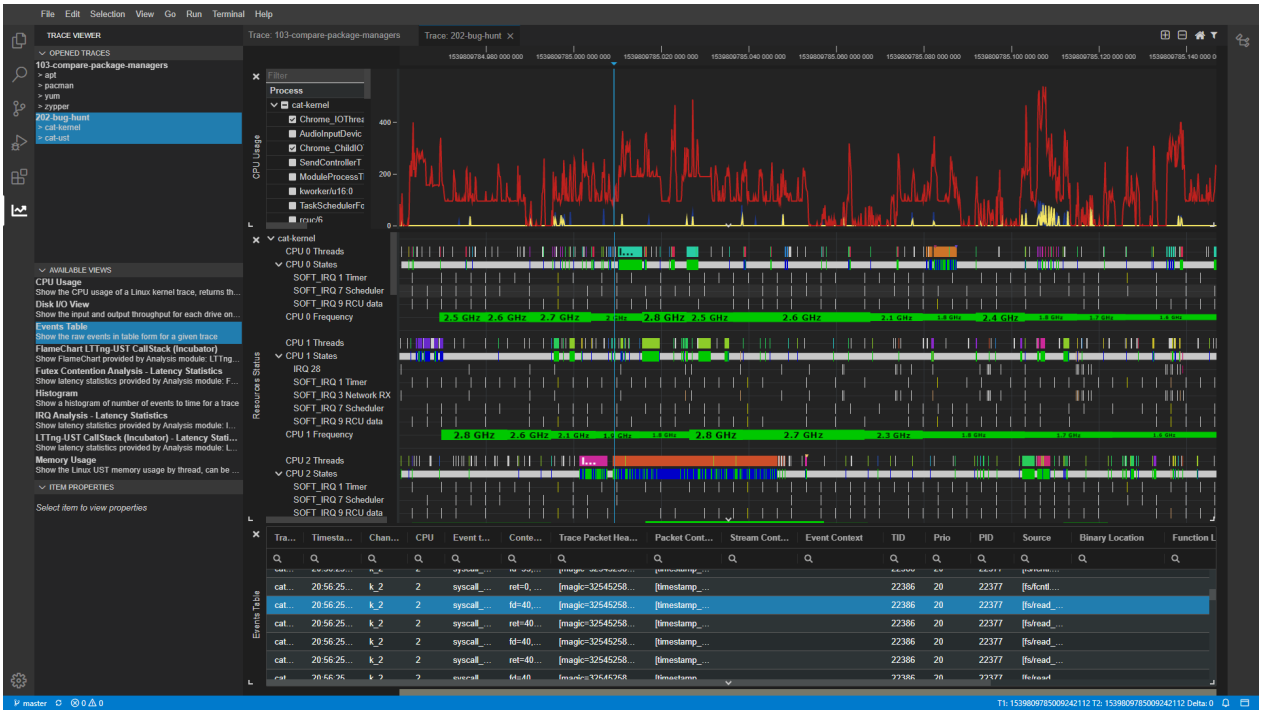

Figure 2.1 Visualisation de trace dans Theia avec l'extension

Il est important de noter qu'il existe plusieurs approches pour paralléliser l'analyse des traces dans Trace Compass. (Reumont-Locke, 2015) [17] et (Martin, 2018) [18] ont travaillé sur la parallélisation du traitement des traces en "multithread", en divisant le flux d'événements puis corrigeant les informations manquantes. Cette parallélisation prend place dans le module d'analyse et aide à accélérer l'analyse existante. Un autre type de parallélisation est quand nous souhaitons agréger les résultats d'analyse, lorsqu'il y a des milliers de nœuds instrumentés dans une grappe, ce qui peut être accompli en utilisant plusieurs serveurs de

Trace Compass.

Trace Compass est très flexible, étant un outil avec des analyses configurables et aussi un cadriciel qui facilite le développement d'une multitude de types d'analyse, grâce à son système d'états et sa base de données de l'historique d'état. Cependant, le calcul de ses analyses et l'écriture de la base de données prend du temps. Une fois la base de données créée, la visualisation de ces états et la navigation dans le temps se font efficacement pour des traces de relativement grande taille.

Perfetto

Perfetto, développé par Google, est une suite d'outils fournissant une solution de traçage complète, du début(instrumentation) à la fin(analyse et visualisation), orientée vers le traçage de noyau de manière traditionnelle. Il consiste en trois parties principales : l'instrumentation de trace, l'analyse de trace et la visualisation de trace.

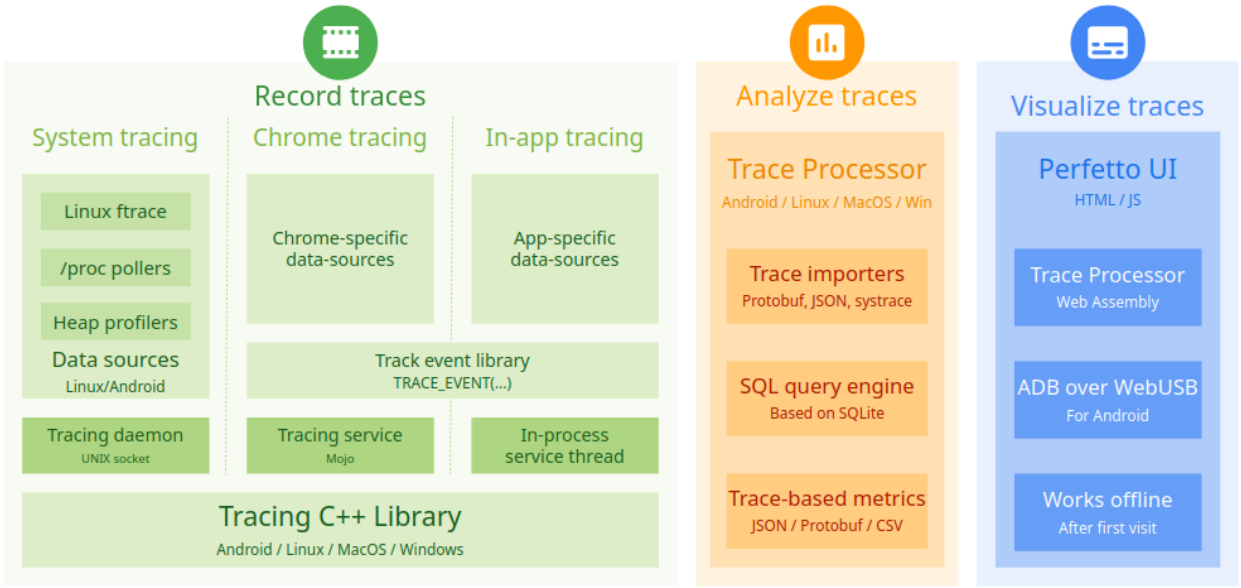

Figure 2.2 Les composants principaux de l'outil Perfetto © Google [1]

Le module d'instrumentation de Perfetto est incorporé à des solutions existantes, lorsque c'est possible. Par exemple, il intègre l'outil ftrace de Linux pour fournir la capacité de tracer le noyau Linux. Ce module est assez performant, il n'introduit pas de surcoût important même dans des applications temps-réel (Drews, 2021) [19]. Pour une comparaison avec Trace Compass, la partie d'analyse de trace de Perfetto prend le rôle du système d'états. Similaire

à Trace Compass, les traces générées sont importées dans ce module, puis transformées et sauvegardées dans une base de données sur mesure, exposée à travers une interface SQL. La visualisation de trace de Perfetto est une interface développée en tant qu'une page web et interagit avec le module d'analyse pour récupérer les données nécessaires. Il prend avantage de la nouvelle technologie "web-worker" et est donc "multithread" pour éviter les blocages d'interaction. L'avantage de Perfetto est que le module analyse n'est pas un composant distinct, mais est intégré dans le navigateur comme un composant WebAssembly. Cela veut dire qu'il est livré directement dans le navigateur lors de connexion à l'interface de visualisation, au contraire de Trace Compass serveur, qui demande un déploiement et des configurations. Cela fait que Perfetto est l'un des outils les plus portables aujourd'hui, tant que l'utilisateur possède un navigateur web moderne sur son poste de travail. Cette architecture permet aussi de visualiser les traces en mode hors-ligne, une fois qu'elles sont importées par l'utilisateur.

La différence fondamentale ici est que le module d'analyse n'est pas distribué comme un fichier binaire, mais plutôt comme une librairie C++. Les différents clients utilisent ensuite cette librairie pour créer leur propre backend. Malgré qu'ils partagent les mêmes APIs fournies par la librairie C++, ces clients ne pourraient pas communiquer directement avec le backend de l'un à l'autre. Étant un module WebAssembly, le module d'analyse est aussi limité par l'interface application du navigateur. Par exemple, ce n'est pas trivial de supporter plusieurs types de service de stockage, comme dans le cas de Trace Compass. Alors, bien qu'il ait conceptuellement une architecture modulaire, Perfetto est toujours couplé et difficile d'être déployé dans un environnement distribué dans son état actuel. Un déploiement avec plusieurs modules d'analyse en parallèle n'est pas envisageable non plus, à cause de cette raison et un manque de protocole de communication entre eux, ce qui limite donc l'aspect de mise à l'échelle. Il demeure un bon outil pour son but original, conçu principalement pour tracer les appareils Android et Linux personnels.

2.2 Traçage distribué

2.2.1 Systèmes distribués et parallèles

Les systèmes parallèles sont utilisés partout aujourd'hui, dans les systèmes personnels ainsi que ceux industriels. Cependant, il existe des limites sur le nombre de cœurs qu'un processeur est capable de tenir avant qu'il ne devienne trop cher, ou des contraintes physiques qui empêchent d'avoir toute la puissance de calcul dans un même endroit. C'est pourquoi les systèmes distribués jouent toujours un rôle indispensable. Avec l'augmentation fulgurante du nombre de compagnies qui fournissent des services sur l'Internet, les systèmes distribués de-

viennent encore plus pertinents. Par rapport à un système monolithique, ils sont plus adaptés pour ces compagnies sur Internet qui fournissent les services globaux 24/7 en développant rapidement de nouvelles fonctionnalités. Les systèmes distribués en général ont plus de tolérance de panne, ou dans le pire des cas peuvent fonctionner en mode dégradé, grâce à la séparation et l'isolation des parties différentes du programme dans des machines différentes, ce qui limite dans une grande mesure la propagation d'une panne quelconque. Une panne dans un système monolithique affecte tout ce qui existe dedans. Un système distribué réduit aussi la friction et augmente la flexibilité pour l'équipement de développement. Puisque les composants communiquent maintenant à travers des APIs, chaque équipe peut développer sa part basée complètement sur l'API stable, sans connaître les détails derrière. Ceci nécessite alors moins de planification et de synchronisation. Puisque ces parties sont découplées, elles peuvent évoluer horizontalement ou verticalement, dépendant du cas d'utilisation. Elles peuvent être déployées de manière différente, sur des matériaux différents, selon le besoin, tant que l'API définie est bien respectée. Il existe de nombreux services qui exécutent souvent les tâches similaires sur des jeux de données différents et distincts. Il va sans dire que la réplication des machines standard est souvent aussi efficace que chercher plus de cœurs dans une seule machine, pour ce cas d'utilisation, car le facteur de parallélisation est presque 100%, et tout ça avec un coût moindre. Cela donne naissance aux grappes de calcul avec des centaines, voire des milliers de machines.

Le développement de logiciel pour ces types d'infrastructure peut être simple ou complexe, en fonction de l'application souhaitée, mais le paradigme est plutôt bien défini grâce aux nombreuses librairies et cadriciels qui facilitent la communication entre ces machines. Le traçage dans ces systèmes est aussi important, puisque le nombre de points de défaillance est en proportion avec la taille de grappe. De plus, une anomalie dans des grappes ramène potentiellement des impacts plus graves que sur un système non distribué (Matloff) [20]. Les outils de traçage pour ces infrastructures sont un composant important pour assurer l'évolutivité du développement.

2.2.2 Traçage distribué

Un grand défi du système distribué est l'investigation en cas de panne. Vu que le programme se distribue partout dans le réseau, possiblement dans un environnement différent sur chaque noeud, développé dans un langage différent, exécuté sur un matériel différent, il est presque impossible de mettre ce programme dans un débogueur et retracer étape par étape son exécution pour comprendre son comportement. Il faut donc avoir des outils de caractérisation spécifiquement conçus pour ces systèmes, d'où le besoin pour le traçage distribué. Le tra-

çage distribué partage des bases communes avec les techniques de traçage ordinaires, mais aussi introduit des défis en plus de ceux existants. Les outils traditionnels sont souvent très efficaces pour observer le comportement d'une application, ou l'interaction inter-processus d'une machine locale. Cependant, dans un environnement distribué, notamment un réseau de plusieurs machines connectées qui chacune contribue une partie de la fonctionnalité globale (patron micro-service), ils ont de la difficulté à offrir une bonne visibilité tout au long de l'exécution à travers plusieurs machines (Uber, 2017) [2]. Ce défaut est dû au fait que ces outils n'ont pas de notion de contexte de l'exécution lors de l'instrumentation, et les données tracées manquent des détails nécessaires pour pouvoir déduire le contexte plus tard.

La solution la plus courante pour contourner ce problème est la propagation de contexte distribué, ou la propagation de "metadata", comparé aux autres techniques comme "Black-box inference" ou "Schema-based" (Shkuro, 2019) [21]. L'idée fondamentale est d'inclure un identifiant unique lorsqu'une exécution commence dans un service quelconque, qui est ensuite propagé à travers les machines comme un paramètre dans les sous-requêtes qui en découlent. Un désavantage de cette méthode est la nécessité d'avoir explicitement une librairie client dans chaque service, cette librairie est responsable de créer le tracé avec l'identifiant de contexte unique, et de synchroniser ce contexte lorsque c'est nécessaire. Ceci dit, son résultat est beaucoup plus fiable que les autres approches, et c'est pourquoi les projets comme Open Tracing sont conçus en essayant d'optimiser et standardiser la manière dont des fournisseurs différents peuvent développer leur version de librairie client.

2.2.3 Patrons de systèmes distribués

Les outils de traçage distribués sont eux-mêmes des systèmes distribués. C'est pourquoi, avant d'en parler en détail, nous allons examiner ici des patrons typiques pour la conception des systèmes distribués qui peuvent être utilisés pour ces systèmes.

Un outil de traçage distribué est souvent un système distribué lui-même. Concevoir un système distribué est une procédure propice à faire des erreurs à cause des interactions complexes et de l'augmentation du nombre de points de défaillances possibles. Suivre les patrons nous permet d'appliquer les meilleures pratiques et de bénéficier de la connaissance apprise sur les systèmes qui utilisent un patron similaire, ce qui facilite le développement et le débogage (Burns, 2018) [22].

Le patron le plus simple consiste en un nœud contrôleur et plusieurs nœuds travailleurs sans notion d'état. Le contrôleur surveille les nœuds travailleurs en continu pour savoir leur état de travail. Lorsqu'il reçoit une requête, le contrôleur choisit un nœud travailleur libre et lui assigne la requête. Ce patron est utilisé pour fournir une redondance par rapport à

un système central. En cas de panne d'un serveur, le système peut continuer à fonctionner. Ce modèle peut ensuite inclure une notion de session. Ce concept est utile lorsqu'on veut avoir une persistance de données à travers plusieurs requêtes. Par exemple, le résultat d'une requête de type A, une fois calculé, peut être mis en cache sur un serveur quelconque. Quand le contrôleur reçoit le même type de requête une deuxième fois, il doit savoir à quelle session de quel nœud travailleur appartient cette requête, pour avoir rapidement le cache (Burns, 2018) [23].

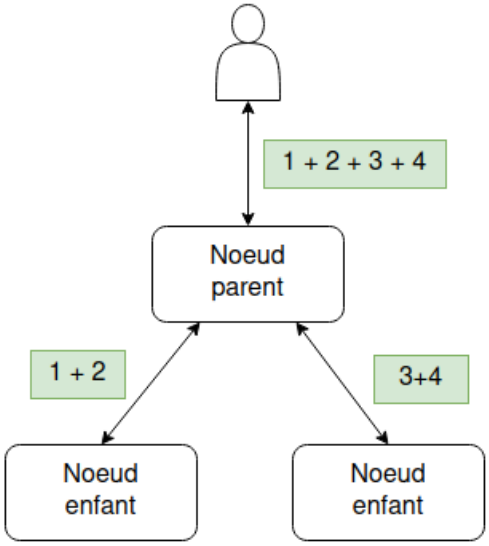

Figure 2.3 Le patron de conception diviser-rassembler

Ces deux patrons fournissent une redondance de travailleur, mais à chaque fois la requête est traitée par un seul travailleur. À la place, on peut diviser la requête pour que chaque nœud travailleur en traite une partie. Ils retournent ensuite les réponses partielles qui sont fusionnées par le contrôleur. Ceci est connu comme le patron diviser-rassembler et est la base du fameux MapReduce. Ces patrons ne sont pas mutuellement exclusifs, par exemple avec trois nœuds travailleurs, le contrôleur peut parfaitement utiliser le diviser-rassembler avec deux travailleurs et envoie une deuxième requête complète au troisième travailleur. Évidemment, le contrôleur doit être assez sophistiqué pour dynamiquement changer la stratégie en fonction de charge de travail (Burns, 2018) [24].

Toujours avec le même patron, on peut ensuite avoir un contrôleur choisi dynamiquement ou étant un travailleur pour les requêtes plus légères, afin de sauvegarder le temps de réseau introduit lorsqu'il faut communiquer avec un autre travailleur (Burns, 2018) [25].

2.2.4 Outils de traçage distribué

Nous allons maintenant consulter quelques outils de traçage distribués les plus populaires. Ces outils ont adopté des formats de trace ou métrique légèrement différents, avec lesquels des fonctionnalités spécifiques sont développées. Ce qui nous intéresse vraiment, c'est de voir comment le flux de données est organisé, à partir de machine instrumentée, de la phase d'analyse, jusqu'à la visualisation. Nous faisons attention à ses avantages et ses inconvénients, les points communs et sa possibilité de mise à l'échelle sur le long terme.

Jaeger

Jaeger est un outil de collection et de visualisation de trace développé initialement par Uber. Il y a plusieurs années, Uber possédait une architecture monolithique écrite en Python. A ce stade, les outils traditionnels comme Merckx étaient suffisants pour tracer tout le système. Avec l'augmentation rapide de nombre de services fournis, son équipe d'ingénieurs a graduellement adopté l'architecture micro-service pour mieux découpler les fonctionnalités. Avec le changement, les outils traditionnels n'étaient plus appropriés pour le travail. Jaeger a été conçu pour fournir la visibilité sur l'ensemble des micro-services et est prévu pour suivre la mise à l'échelle à de nombreux futurs services.

Le premier prototype de Jaeger est seulement la librairie client qui aide à créer les traces avec le contexte d'exécution pris en compte. Ces traces sont en format de Zipkin, car le prototype utilisait Zipkin comme son serveur central de trace pour l'analyse. L'interface graphique de Zipkin était utilisée pour la visualisation de traces. Ces composants Zipkin ont ensuite été remplacés progressivement.

L'architecture de Jaeger aujourd'hui consiste en des librairies du côté client, implémentées en suivant l'API de OpenTracing en plusieurs langages de programmation différents. Ils sont intégrés avec les cadriciels pour créer des traces à chaque requête reçue de manière transparente. Au lieu d'envoyer ces traces à un serveur central directement, cette librairie les envoie au jaeger-agent, un processus qui tourne localement et est souvent déployé en tant qu'un composant dans l'infrastructure. Le jaeger-agent ensuite envoie ces traces en lot à une grappe de collecteurs (serveur central de traçage) pour les sauvegarder sous la forme d'une base de donnée pré-traitée et prête à être analysée. Cet agent a pour but d'effectuer la découverte du collecteur et d'effectuer l'acheminement à travers le réseau des données produites par l'instrumentation. Cette approche permet aussi de centraliser la stratégie d'instrumentation au niveau du serveur de traçage central et de l'ajuster en fonction du volume de trafic apporté par les services participants. La stratégie mise à jour est récupérée localement par les services

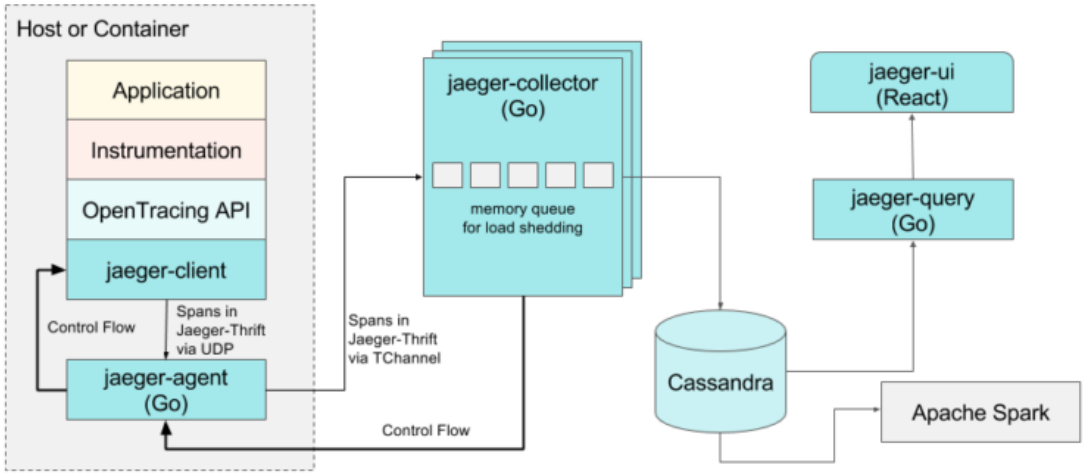

Figure 2.4 L'architecture de l'outil Jaeger © Uber Engineering Blog [2]

à chaque intervalle fixe. En effet, Jaeger n'est pas seulement un outil pour analyser les traces, mais une suite d'outils qui couvrent du début (l'instrumentation) jusqu'à la fin (l'analyse et la visualisation de trace) du travail de traçage.

La contribution importante de Jaeger n'est pas dans la phase d'analyse ou la phase d'instrumentation, mais dans la mécanique qu'il utilise pour suivre une trace à travers plusieurs machines dans le réseau au moment de l'instrumentation. Avec la publication du projet OpenTelemetry, on obtient non seulement une couche compatible vers l'arrière avec l'API de OpenTracing, mais aussi une implémentation par défaut officielle. La partie d'instrumentation de Jaeger est graduellement remplacée par OpenTelemetry (Shkuro, 2019) [26].

Nagios

Nagios est l'un des outils le plus utilisé dans l'industrie pour la surveillance des systèmes infonuagiques (Tamburri et al, 2020) [27] avec une architecture distribuée dans laquelle plusieurs agents envoient les données à un serveur Nagios central. Contrairement à Jaeger qui remonte les traces très spécifiques à chaque application, Nagios est un outil de surveillance de type boîte noire, c'est -à -dire qu'il se concentre sur les métriques générales du système comme le disque, le CPU, etc. L'avantage de cette approche est qu'il n'a pas besoin d'une librairie du côté client pour fonctionner. Cependant, c'est aussi le point faible de l'outil, puisqu'il n'y a pas assez de détails, il ne sert seulement qu'à montrer l'état actuel, plutôt

que de permettre d'expliquer la transformation du système à un état quelconque. Il est aussi possible de développer des "plug-ins" pour ajouter des fonctionnalités à Nagios, par exemple un système d'alerte en cas d'anomalie (Renita et Elizabeth, 2017) [28] pour envoyer des notifications sur les plateformes externes. Originalement, cette architecture limite l'évolutivité du système, parce que même si l'instrumentation est distribuée, le serveur central est limité à un seul nœud et donc n'évolue pas bien avec l'augmentation de nombre de nœuds instrumentés. Certains outils ont été développés, comme une extension basée sur Nagios, pour régler ce problème (Ciuffoletti, 2016) [29].

Dans les dernières versions, il est possible de déployer Nagios avec une organisation fédérée de monitoring qui permet d'avoir plusieurs serveurs Nagios locaux, qui ensuite reportent les informations à un serveur Nagios central pour un rapport centralisé du système (Nagios) [30]. Il existe maintenant aussi le module "Nagios Fusion" qui fonctionne comme le monitoring fédéré à grande échelle. Il offre en plus la capacité d'identifier et de configurer automatiquement les serveurs Nagios locaux à partir du nœud "Nagios Fusion" central.

Il est intéressant de voir la solution apportée par l'équipe de Nagios pour contourner le problème de mise à l'échelle. En ajoutant la capacité de communication entre les serveurs centraux, il est possible d'augmenter l'échelle de fonctionnement sans modifier les parties principales de l'outil.

Prometheus & Grafana

Prometheus est le nouveau standard de technologie pour le traçage distribué publié en 2016 par SoundCloud. D'une vue globale, l'organisation des composants principaux de Prometheus ressemble partiellement à Nagios, mais avec beaucoup plus de fonctionnalités à l'état de l'art. Le cœur de l'écosystème est défini dans le module serveur qui détermine la spécification du protocole de communication ainsi que le format de trace supporté. Le module d'instrumentation de Prometheus est une collection de librairies clientes implémentées d'après la spécification ; la plupart d'entre elles sont officiellement maintenues par la communauté. Similaire à Jaeger, Prometheus permet d'avoir des traces sur mesure, tant qu'elles suivent bien l'un des quatre types définis. Le serveur central récupère les traces et les sauvegarde comme une table de série d'événements dans le temps. Il est donc possible de comprendre le transfert d'état du système à travers le temps en inspectant son histoire d'état. Ceci diffère de Nagios qui montre l'état actuel du système, mais manque la capacité de déduire la cause d'une panne, à cause de sa nature de monitoring boîte noire, surtout dans un environnement dynamique.

En plus d'un outil d'analyse de trace traditionnel, Prometheus possède la capacité d'automa-

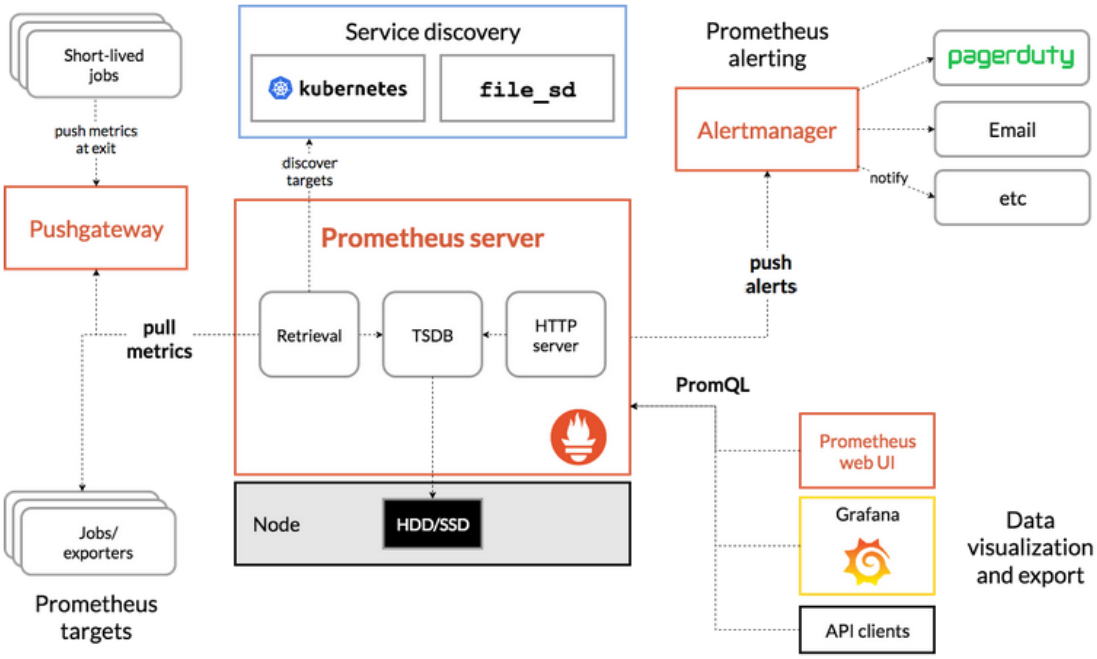

Figure 2.5 L'architecture de l'outil Prometheus © Prometheus, 2014 [3]

tiser l'alerte d'anomalie. L'utilisateur peut définir les règles sur lesquelles Prometheus base ses décisions d'envois d'alertes. En effet, grâce à sa flexibilité, il est capable de connecter les modules principaux de Prometheus aux autres services souhaités pour encore augmenter la fonctionnalité de base. Par exemple, on peut connecter le module d'alerte à un service externe pour automatiser les notifications par mail ou sur Slack. On peut aussi adapter le service de stockage utilisé par Prometheus serveur, notamment avoir une instance ElasticSearch pour la sauvegarde historique, et donc améliorer la performance d'itération dessus (Sukhija et al., 2020) [31].

Avec une seule instance, Prometheus rencontre le problème de mise à l'échelle comme Axios, car le serveur central constitue une limite de puissance de calcul. En revanche, le projet a été conçu avec la mise à l'échelle en tête et offre différentes options pour amener Prometheus à une échelle plus grande. En résumé, les serveurs Prometheus sont capables d'échanger leur base de données de l'un à l'autre. La technique pour orchestrer plusieurs serveurs Prometheus s'appelle "fédération" (Reback, 2021) [32]. Récemment, le projet Thanos a été créé dans le but de gérer les instances de Prometheus et fournir une interface agrégée.

Grafana est un outil de visualisation qui est souvent recommandé pour utiliser avec Prometheus. Grafana permet de personnaliser les tables et schémas à afficher à partir des points de données obtenus de Prometheus. Cette possibilité est très puissante, car différents utilisateurs

peuvent adapter le format d'affichage de Grafana aux différents besoins, sans être contraint par celui de défaut (Na et al., 2020) [33].

Les autres systèmes de surveillance distribuée

Il existe certains autres travaux qui partagent des techniques communes avec le traçage distribué. Ces projets ont tous pour but de relever, récupérer et analyser les données de manière distribuée et fonctionnent à grande échelle. En regardant ces approches dans des domaines similaires, nous pourrons potentiellement nous inspirer des solutions encore inexistantes en ce moment dans notre domaine.

Les recherches autour du problème d'analyse de journaux d'exécution de manière distribuée ont gagné en popularité. Les compagnies sur Internet qui vendent aux utilisateurs finaux produisent une grosse quantité de journaux tous les jours. Les informations contenues dans ces journaux aident à comprendre le comportement des utilisateurs et donc à personnaliser les produits de manière plus pertinente. L'analyse de ces journaux doit être en continu et rapide, en raison de constants changements de choix par les utilisateurs tous les jours. Il est crucial alors d'avoir un mécanisme pour les analyser efficacement, étant donné le grand volume de contenu créé dans ces journaux. Ce travail est aussi problématique, parce que les interactions des utilisateurs sont avec des requêtes différentes, qui sont fournies par des services différents, et qui ensuite produisent des journaux différents. Réunir et relier ces informations différentes pour obtenir une vue globale ressemble beaucoup à notre travail en traçage distribué. Les modèles comme MapReduce sont typiquement appropriés pour ce type de partition et d'analyse de données (Dewangan al., 2016) [34]. Le même modèle peut être ensuite intégré dans les cadriciels comme Hadoop (Bhosale et al, 2018) [35] puis Spark (Li et al., 2018) [36] pour bénéficier des avantages comme la tolérance aux pannes, ou son excellente performance (Blanas et al., 2010) [37]. Log Lens est un outil d'analyse de journaux dynamique qui prend une approche similaire (Debnath et al., 2018) [38]. Le projet doit utiliser une version modifiée de Spark, pour supporter sa requête de dynamiquement changer le format des journaux, sans redémarrer tout le service.

Les nouveaux outils proposés pour l'analyse de trafic réseau en temps réel possèdent une architecture qui ressemble aussi à celle du traçage distribué, dans laquelle plusieurs nœuds qui capturent l'information de réseau envoient leurs données à un serveur central, qui lui est connecté à un service approprié pour le traitement (Turcato et al., 2020) [39]. Les systèmes de caméras de surveillance sont un autre type d'application qui partage des caractéristiques communes avec notre intérêt. L'architecture du projet INVISUM comporte des composants principaux occupant des rôles très similaires à ceux de Jaeger ou Prometheus : Les données

brutes, dans ce cas des images vidéo, sont envoyées à un serveur central pour extraire de l'information. Un autre module, qui permet de configurer les règles, analyse ces informations pour déterminer s'il faut envoyer une alerte aux services correspondants (Diego et al, 2018) [40].

2.3 Protocole d'interrogation

Les protocoles d'interrogation influencent fortement le développement d'un système de traçage distribué. Un protocole approprié doit, en raison des diverses combinaisons de types d'analyse et de visualisation de traçage possibles, pour les cas d'usage commun, fournir des options assez spécifiques pour les requêtes des utilisateurs, afin qu'il puisse ajuster le niveau de détails dans la réponse souhaitée. Ceci évite d'avoir à implémenter de la logique complexe dans les différents types de clients qui sont supposés être légers. En même temps, le protocole doit être assez flexible pour s'adapter facilement aux nouveaux besoins, sans introduire d'incompatibilité face à la nature dynamique du domaine du traçage. Puisque nous cherchons à transformer Trace Compass vers une nouvelle architecture et supporter de nouvelles fonctionnalités, il est nécessaire de revoir l'utilité du protocole actuel afin d'affirmer sa capacité d'adaptation aux nouveaux besoins et d'évoluer facilement avec les potentiels futurs changements. Dans ce contexte, nous allons regarder d'abord le Trace Server Protocol (TSP), le protocole utilisé actuellement par Trace Compass, puis, regarder aussi les autres protocoles populaires utilisés dans les outils de traçage distribués.

TSP est l'API développé lors de l'introduction de l'architecture client-serveur dans Trace Compass (Chen Kuang Piao, 2018) [16]. TSP est développé en suivant la philosophie REST API et en ce moment utilise JSON comme le format d'échange de données, mais le travail de Chen Kuang Piao a suggéré qu'on peut facilement faire des gains en termes de taille de message et d'efficacité de sérialisation/dé-sérialisation en utilisant Protobuf à la place (Chen Kuang Piao, 2018) [16]. Ceci est tout à fait faisable, puisque REST, en tant que guide, n'a aucun couplement à un format de données spécifique. À part les points d'accès pour récupérer simplement les données appartenant à une trace ou à une session d'analyse, les points d'accès pour obtenir le modèle de visualisation de trace sont organisés par type de visualisation. On en retrouve notamment un pour les analyses qui sont présentées sous forme de graphes XY, et un pour le graphe temporel [41]. Il est possible d'ajuster la résolution de graphe en passant en paramètre les points de temps souhaités, malgré que ce ne soit pas une pratique très commune du standard REST d'effectuer un filtrage sur la réponse en fonction de paramètres de la requête. Un problème avec TSP en ce moment vient de son format de message JSON et de la manière qu'il fonctionne dans les clients de web développés en JavaScript (JS). Même si

le format de temps est bien défini dans la spécification de TSP comme int64 (Ericsson) [42], le parseur JSON par défaut du JS ne peut traiter les int64 et donc introduit une perte de précision de 512 ns pour chaque sélection sur le graphe (Khouzam, 2021) [43]. Pire, le format JSON n'est pas fortement typé et donc ne supporte pas un type comme int64. En outre, puisque JS traite int16 (nombre) et int64(BigInt) comme deux types différents, qui ne sont pas convertibles, même avec les analyseurs syntaxiques qui supportent BigInt, il faut compter sur des solutions spécifiques (Maréchal, 2021) [44] (Tran, 2021) [45] pour traiter correctement ces cas en fonction de la longueur des valeurs de temps reçues. Cela compromet le but initial de fournir une interface homogène pour tous types de client, sans dépendance sur des librairies externes.

Avant de commencer à parler de GraphQL, on doit définir d'abord 2 notions importantes : le problème de sur-télécharger et sous-télécharger dans la conception d'un API. 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. 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 accomplir son objectif. 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.

Les critiques fréquentes sur les API REST sont que REST souffre du problème de sur-télécharger et sous-télécharger (Vadlamani et al., 2021) [46]. De son côté, GraphQL, un protocole et un langage de requêtes publié par Facebook en 2016 a été créé comme solution pour fournir exactement les informations demandées, rien de moins et rien de plus (Facebook) [47]. La différence fondamentale est que GraphQL expose une base de données aux clients sous un seul point d'entrée, tandis que REST expose plusieurs points d'entrée prédéfinis. Techniquement parlant, il est possible de régler le sur-téléchargement dans la conception de REST. Une première option est d'ajouter des points d'accès supplémentaires au service pour fournir des réponses partielles. La deuxième option est de retourner une réponse dynamique en fonction de paramètres passés en requête, ce qui est une approche similaire à celle implémentée par le protocole TSP. GraphQL est une solution qui intègre ces 2 options d'une manière plus élégante et uniforme, en fournissant une spécification et un cadriciel qui automatisent la création des combinaisons de point d'entrée de données, au lieu de manuellement les développer, pour simplifier l'implémentation. Le résultat du principe de GraphQL est que l'utilisateur définit explicitement le graphe de données. Ensuite, les requêtes explicitent les

données requises dans la réponse, ce qui évite aussi les changements de contenu associés aux points d'accès lors de changements incompatibles (Brito et al., 2018) [48] (Xavier et al., 2017) [49], et évite l'enfer de version de REST (Amundsen, 2021) [50]. Un autre avantage souvent associé à GraphQL est la réduction du nombre de requêtes grâce au fait qu'on peut effectuer une seule requête complexe pour récupérer toutes les informations nécessaires (Vadlamani, 2021) [46]. En réalité, malgré que ce soit une possibilité, cet objectif représente un défi technique assez complexe, dû au fait qu'on organise souvent les fonctionnalités autour des fonctions distinctes qui chacune demande une partie de l'information (Brito et al., 2019) [51]. GraphQL souffre souvent du problème "1+N" (le vrai nom du problème "N+1" (Cronin, 2019) [52]), mais ce n'est pas un problème pour le cas d'usage de Trace Compass au moment d'écrire ce mémoire. En effet, aucune réponse ne prend la forme d'une liste complexe de structures. Il y a eu plusieurs travaux pour comparer la performance de GraphQL aux autres protocoles comme REST, avec des résultats qui sont parfois en conflit (Sayago et al., 2020) [53] (Hartina et al., 2018) [54] (Lee et al., 2020) [55]. Mesurer le temps de réponse total ne nous donne pas nécessairement l'information pertinente, parce que les implémentations de ces protocoles varient d'un langage à l'autre (REST n'a même pas de spécification technique). Une mesure de performance pour GraphQL, s'il y a lieu, doit mesurer spécifiquement le surcoût du temps d'exécution de traiter la requête GraphQL. Sans regarder la performance pure, un autre gain potentiel se trouve dans le temps de transmission de données, grâce au réducteur de taille de message (Guo et al., 2018) [56] (Brito et al., 2019) [51]. Au moment d'écrire ces lignes, TSP, quoique basé sur REST, ne souffre pas énormément du problème de sur-télécharger, et n'économiserait donc pas beaucoup sur la taille des messages en cas de changement de protocole.

PromQL est le langage de requête exposé par le serveur Prometheus pour l'accès aux traces sauvegardées. Ce n'est pas un protocole standard ou une librairie en soi, dans le sens où nous pouvons l'entendre pour REST ou GraphQL, car il est fortement couplé avec les concepts de Prometheus. Nous regardons quand même ce langage pour s'inspirer de ses concepts de requête d'une base de données de trace, qu'un protocole pourrait offrir. En effet, le langage de requête LogQL de Grafana Loki est conçu d'après PromQL. La base de données de Prometheus est une série temporelle, ainsi la requête la plus simple est de récupérer le point de donnée sauvegardé pour chaque valeur de temps. On peut ensuite filtrer la réponse souhaitée en spécifiant par exemple les étiquettes (mots-clés) requises, et donc seulement les points de données qui ont cette étiquette seraient retournés. Plusieurs opérateurs d'agrégation sont aussi supportés, par exemple si on veut calculer la somme des points retournés. Cela permet d'avoir la logique de ces calculs qui réside du côté serveur, avec plus de puissance de calcul, et aussi d'éviter de dupliquer la logique commune à travers plusieurs clients différents. Cela per-

met aussi de sauvegarder la bande passante, si une agrégation réduit de manière importante la quantité d'information à retourner dans la réponse. En cas de besoin spécifique, on peut toujours implémenter un de ces opérateurs dans le client en inspectant la réponse. Il existe par défaut aussi les fonctions qui transforment la réponse. Ces opérations peuvent souvent être exécutées beaucoup plus efficacement du côté du serveur, car on peut facilement avoir un point de donnée en plusieurs formats et une transformation devient simplement un accès de données, au lieu d'une procédure complète si implémenté dans le client.

2.4 Conclusion de la revue de littérature

A travers cette revue de littérature, nous avons vu que la modularisation de Trace Compass a créé la fondation pour pouvoir le déployer dans un environnement distribué et qu'il existe plusieurs travaux autour du sujet du traçage distribué qui se concentrent sur différentes parties. Nous avons aussi vu que le flux de travail instrumentation-collection-analyse-visualisation/alerte est aussi commun en dehors du domaine du traçage. Toutes ces applications proposent aussi des modèles robustes avec la mise à l'échelle en tête.

Pourtant, les opérateurs d'agrégation dans ces outils, s'il y a lieu, sont souvent prédéfinis et fortement couplés avec le format de trace et l'écosystème, malgré qu'ils possèdent un langage de requête puissant. Cela gêne l'adaptation lors des cas d'usage spécifiques et force les utilisateurs à réinventer la logique.

En conséquence, nous souhaitons d'abord proposer une organisation qui permet un fonctionnement efficace sur des grappes de calcul pour Trace Compass, en s'inspirant de l'architecture des outils de trace distribués à l'état de l'art. Avec cette nouvelle architecture, nous proposons ensuite une extension du protocole actuel permettant de construire un modèle de programmation qui facilite le développement de nouveaux types d'agrégation dans la visualisation de trace.

CHAPITRE 3 MÉTHODOLOGIE

Nous décrivons dans ce chapitre les étapes suivies pour atteindre les objectifs de recherche. Il aide à mieux comprendre le déroulement de la présentation de notre travail dans les chapitres suivants.

3.1 Proposition de solution

Grâce à la revue de littérature, nous avons vu que le domaine du traçage distribué demande des techniques particulières et plusieurs outils ont été créés pour répondre à ces besoins.

Suivant nos objectifs de recherche, la première étape est d'analyser la mise à l'échelle de Trace Compass dans son état actuel en utilisant simplement l'outil sans aucune modification avec des collections de traces de gros volume représentant les traces d'une grappe de calcul. Pour créer ces données, nous avons dupliqué plusieurs fois les traces d'une seule machine pour simuler une grande quantité de traces. La duplication est faite en utilisant un script qui copie et aussi modifie les "meta" données pour s'assurer que Trace Compass les reconnait en tant que des traces individuelles. Le but de cet exercice est d'étudier le temps d'exécution des portions différentes de Trace Compass pour identifier les morceaux de travail parallélisables et aussi de trouver la limite dans sa capacité de traitement. Par la suite, la nouvelle architecture proposée est supposée de résoudre cette limitation.

Afin d'arriver à une proposition, nous avons d'abord identifié le patron d'architecture commun des outils d'analyse de traces dans notre revue de littérature. Nous avons aussi constaté que la capacité d'offrir des opérateurs pour agréger le résultat à travers les requêtes est très puissant et est une nécessité vue la surcharge de l'information de traces à afficher pour les traces d'une grappe de calcul. Nous avons donc choisi une architecture de type "Map Reduce" pour paralléliser l'exécution de Trace Compass, avec une phase "Reduce" qui inclut une capacité d'agréger les résultats de l'analyse de traces de la phase "Map".

Pour pouvoir appliquer cette architecture à notre problème, nous avons d'abord cherché à comprendre le fonctionnement de Trace Compass, en particulier l'interaction entre le serveur Trace Compass et l'interface graphique. Cela inclut la compréhension du Trace Server Protocol (TSP), utilisé par Trace Compass. Nous avons utilisé l'outil d'inspection du navigateur de web pour observer les échanges de requêtes et réponses entre le serveur Trace Compass et l'interface graphique web lors d'une utilisation typique d'analyse de traces. Nous cherchons à comprendre spécifiquement les étapes d'exécution dans le serveur Trace Compass lorsqu'il

reçoit une requête d'analyse. Pour cela, nous devons consulter le code source et effectuer des simulations de requêtes pour observer son comportement lors de l'exécution. Nous avons consulté aussi l'équipe de développement de Trace Compass pour plusieurs aspects techniques sur cet outil, durant ces étapes. Cela permet de bien observer le flux de données traversant l'application et déterminer les endroits dans le flux où une agrégation serait applicable. Nous avons aussi inspecté manuellement en détail les réponses TSP retournées par le serveur Trace Compass pour déterminer les opérations d'agrégation à faire pour notre cas d'usage. Nous avons étudié les types d'analyse différents pour choisir ceux qui sont plus facilement généralisables pour attaquer le problème.

Finalement, nous proposons une organisation avec plusieurs instances de serveur de Trace Compass et décrivons les interactions basiques dans cette organisation. Nous décrivons en détail la séparation des requêtes et l'agrégation des réponses effectuées par le composant coordonnateur. Pour cela, nous faisons aussi une révision sur les protocoles de communication, pour déterminer si le protocole actuel est approprié à notre besoin.

3.2 Implémentation de la solution

Puisque le serveur Trace Compass et le TSP est un mode d'exécution récent de Trace Compass (d'après l'architecture proposée par Yonni Chen Kuang Piao, 2018) [16], il existe certaines fonctionnalités de Trace Compass qui n'étaient pas encore implémentées pour ce mode, ou des analyses qui ne sont pas encore exposées à travers TSP. Pour commencer la phase d'implémentation, nous avons implémenté l'analyse de statistiques de type d'événements pour le serveur Trace Compass, qui est un candidat pour le cas d'usage de Trace Compass en mode distribué. Nous l'avons développé en tant qu'un "plugin" Eclipse dans l'incubateur Trace Compass, ce qui nécessite une préparation de l'environnement de travail avec les étapes suivantes : installer la version de Eclipse pour les "committers" ; créer et configurer un projet avec le code source de Trace Compass et l'incubateur Trace Compass ; créer un nouveau plugin en utilisant le script fourni par l'incubateur.

Ensuite, nous avons déterminé l'architecture du composant coordonnateur. Nous avons choisi l'environnement NodeJS parce qu'il permet de développer rapidement des prototypes, avec plusieurs librairies largement supportées en code source ouvert. Afin de s'assurer la compatibilité avec l'interface graphique existante de Trace Compass, nous avons consulté aussi son code source pour vérifier les requêtes basiques auxquelles le coordonnateur doit être capable de répondre. Essentiellement, le coordonnateur est un serveur qui est capable de traiter les requêtes TSP. Nous avons utilisé alors la librairie "Fastify" pour développer le module de serveur HTTP (TSP est un protocole de HTTP). Cette librairie facilite l'implémentation et la

configuration des fonctions de traitement des requêtes, ainsi que la journalisation du serveur. L'envoi des requêtes aux serveurs Trace Compass par le coordonnateur est développé avec la librairie "tsp-typescript-client", ce qu'utilise l'interface graphique officielle de Trace Compass. Cette librairie facilite la mise en forme des données et paramètres pour les requêtes TSP.

À la fin de cette étape, nous obtenons un prototype simple d'un coordonnateur qui est capable d'échanger les données avec plusieurs instances de serveur de Trace Compass. Le coordonnateur à cette étape passe simplement les requêtes du client à plusieurs serveurs de Trace Compass, puis retourne leurs réponses. Nous écrivons ensuite des tests qui envoient des requêtes au coordonnateur et s'assurent qu'il retourne les réponses dans le format souhaité. Une fois que le coordonnateur fonctionne dans ce mode simple, nous cherchons à implémenter les opérations d'agrégation sur les résultats reçus des serveurs de Trace Compass.

3.3 Évaluation de la solution

Nous évaluons l'implémentation de notre solution en vérifiant d'abord le bon format et contenu de réponse retourné par le composant coordonnateur. Nous avons aussi utilisé le coordonnateur directement comme un serveur pour l'interface graphique de Trace Compass, pour vérifier sa compatibilité.

En un second temps, nous effectuons une mesure de performance de notre implémentation. Pour cela, nous définissons les scénarios d'analyse avec une collection de traces avec pour but de tester le tout. Initialement, les mesures sont prises en utilisant la même collection de trace dupliquée que nous avons utilisée pour l'analyse de la mise à l'échelle de Trace Compass. Cependant, pour que les scénarios soient plus proches des usages réels, nous avons régénéré les traces en traçant 4 ordinateurs différents, chacun exécutant un serveur de Trace Compass avec une charge de travail similaire. Leurs traces sont collectées pendant un même intervalle de temps, et en fonction leur spécification, la taille des traces varient légèrement d'une machine à l'autre. Nous avons ensuite mesuré le temps d'exécution dans ces scénarios de notre prototype. Nous avons mesuré aussi celui d'une instance de serveur de Trace Compass pour le comparer à notre solution et avoir des notions sur le facteur de gain associé à la parallélisation. Pour effectuer les tests, nous déployons les instances de serveur de Trace Compass sur des ordinateurs dans une même salle de travail à Polytechnique, pour minimiser le coût de latence du réseau. Nous avons effectué les tests la nuit, lorsqu'il n'y a aucun autre utilisateur dans la salle, pour éviter toute interférence sur le fonctionnement de ces ordinateurs. Les ordinateurs utilisés sont tous sur le système d'exploitation Fedora 34 et possèdent un processeur Intel(R) Core(TM) i5-9400F CPU @ 2.90GHz avec 16GO de mémoire principale.

CHAPITRE 4 ARTICLE 1 : SCALABLE DISTRIBUTED ARCHITECTURE FOR TRACE ANALYSIS

Dans ce chapitre, nous présentons le cœur de ce mémoire : l'article scientifique qui décrit notre proposition en détail, ainsi que son évaluation. L'article présenté suivant est entièrement écrit en anglais.

Authors

Quoc-Hao Tran <quoc-hao.tran@polymtl.ca>

Michel R. Dagenais <michel.dagenais@polymtl.ca>

Abdellah Rahmani <abdellah.rahmani@polymtl.ca>

Department of Computer and Software Engineering, École Polytechnique de Montréal

Submitted to

Concurrency and Computation : Practice and Experience, July 21th 2022

Keywords

architecture, analyse, analysis, tracing, trace, parallel, distributed, aggregation

Abstract

Trace analysis is a powerful tool for troubleshooting with information from the production environment, especially with the rising complexity of software applications. Recent trends in favor of microservices and High-Performance Computing cluster architectures bring challenges in distributed tracing, both in terms of scalability and usability. Various tracing platforms like Jaeger and Prometheus were created to trace distributed systems, each of them focusing on solving a specific problem. Modern projects like Prometheus, come with a set of end-to-end tools for tracing, and offer powerful query capabilities into its trace database. However, the query & aggregation operators offered by these frameworks are tightly coupled to its ecosystem. Meanwhile, supporting external services, and developing custom operators, results in a trade-off that implies losing a large number of supported features. Our work

offers a distributed tracing framework, based on existing solutions, with processing time as well as storage and memory scalability in mind. Furthermore, it provides a standard paradigm for developing new query operators with ease, without sacrificing useful features. The evaluation demonstrates that the proposed approach is feasible, flexible enough to support different trace types, and introduces a minimal overhead. This is important towards making distributed tracing more easily adopted for heterogeneous systems.

4.1 Introduction

The rising complexity in software development requires ever more sophisticated development tools. Beyond fixing targeted logic bugs, non-functional problems such as performance bottlenecks are common in modern software applications, and more often than not are considered just as critical as downtime problems, especially in High-Performance Computing (HPC) applications. Profilers or loggers are often used to retrieve global system measurements for anomaly detection purposes. This practice, however, compels switching back to a debugger for further investigation of the detected problem, because the profiler alone lacks contextual knowledge to be able to infer causality of the events leading to the problematic issue. Even then, traditional debuggers have become incrementally less adequate for developers to inspect complex behaviors of software applications, as the typical interaction involves stopping the execution and examining the source code around the current instruction pointer location. Meanwhile, the applications are increasingly multiprocess, multi-thread, and interact in a complex manner. Hence, understanding these applications means understanding multiple contexts that are not usually available at a glance, at once. Furthermore, stopping online applications, that interact through protocols with timeouts, is not suitable for debuggers that are invasive and not designed to have a minimal impact. Thus, even running a debugger in production would not guarantee reproducing the problematic behavior under investigation, since the impact of attaching the debugger would change the environment significantly (Toupin, 2011) [6].

This is where tracing comes into play. Trace collecting tools are often capable of running alongside the main application, in production, without changing its behavior, and yet can provide enough details about the underlying runtime, with data acquired at the near-exact moment of anomaly. However, it is impractical and most of the time impossible to analyse trace files manually. Therefore, other tools are required to analyse and visualize trace data, such as matching the trace data recorded across processes and threads together, to quickly give an overview of the overall picture to the developers. Given the goal of understanding the global performance of a distributed system, and also the details of specific complex behaviors,

trace analysis tools are bound to expand with new capabilities, to deal with rapidly changing architectures and practices in software development. Creating tracing tools is a constantly evolving domain of expertise. As parallel systems gained in popularity, various works in tracing applications opened the doors to tools that go beyond what debuggers and profilers can achieve (Fournier and Dagenais, 2010) [7]. With parallel systems come parallel traces. Previous research examined how to parallelize the process of trace analysis, to better handle the increasing volume of trace data (Reumont-Locke, 2015) [17], (Martin, 2018) [18].

The continuous rapid expansion of the distributed infrastructure made this even more necessary, as newer distributed systems bring a new set of challenges to the existing tools. As software applications were moved to the Cloud and modularized, so were tracing tools. Recent work in developing tracing applications emphasize an organization with distinct modules that all work together to provide the functionality, instead of the traditional monolithic design. Examples include Perfetto, developed by Google [1], or the introduction of the Data Provider in Trace Compass, to separate its processing and presentation layers (Chen Kuang Piao, 2018) [16], all favoring a lightweight web-like frontend approach. Nonetheless, tracing distributed systems demands some additional processing to connect nested requests. Several large and sophisticated projects are dedicated to distributed systems tracing and monitoring, like Nagios [30] or Jaeger. Distributed tracers such as Jaeger introduced concepts like context propagation, and patterns that set the standard on how distributed tracing tools should behave (Yuri Shkuro, 2019) [21]. The OpenTracing API [57] was co-developed by large Cloud operators, so different tools can develop their own set of features, and yet still talk to each other through a well-defined communication protocol. These projects typically employ a central server responsible for aggregating the whole system trace data. This data is then queried by a display client. Even when providing a powerful query language, as in the case of Prometheus [58], the type of available aggregation algorithms is either basic or tightly coupled to the tracing tool itself. This is a drawback, as the trace data now comes from increasingly diverse sources, especially in heterogeneous systems. Therefore, more flexibility is required for defining newer aggregation types.

To overcome this particularity, we propose an architecture that, inspired by existing solutions following the scatter/gather paradigm, separates analysis requests into parallel jobs executed on different nodes and then aggregates the results at the end. This yields another major advantage that consists in exploiting the resources of the analysis nodes, thus offering a greater capacity of available storage and RAM memory, which allows handling larger amounts of data. We also provide a flexible component that allows to easily develop new aggregation algorithms for different types of analysis requests. Our main contributions are as follows :

— Proposing a scalable distributed trace analysis architecture.
— Proposing an execution model for trace analysis aggregation and a programming paradigm / framework for developing new types of aggregation operators.
— Implement, based on Trace Compass, a prototype of the proposed framework.
— Evaluate the feasibility, efficiency and performance of the proposed implementation.

This paper is organized in four sections. We first review the related work in section II. The proposed architecture and its motivation are discussed in detail in section III, followed by the description of our prototype implementation in section IV. We then evaluate the prototype in section V and finally conclude in section VI.

4.2 Related Work

This section is divided into three parts. We will first look at different works published recently regarding tracing applications, we will then provide an overview of the techniques and tools dedicated to distributed tracing. Finally, we will zoom on the query and communication protocols used by these tools.

4.2.1 Recent work on tracing tools

Trace Compass is an open source project supported by Ericsson for trace analysis and visualization. It was originally developed using the Eclipse Rich Client Platform (RCP) to take advantage of the view-model skeleton of the platform. The heart of the tool is a component called the state system. The trace data is read through a parser which converts it to an internal format, so that only a new parser is needed to support a new trace type, for which a time series of states is generated. The trace data may contain very detailed events about the execution. However, some of the more important events (e.g. scheduling change) are recognized by the state provider components to model the state of the traced system (e.g. currently executing thread on each processor core) (Ericsson) [15]. An analysis module then queries this state system and performs various computations on it. The result is data points used for creating the model from which different graphs and charts are drawn.

The problem with this monolithic architecture is that the whole application is tightly coupled. The DataProvider layer (Chen Kuang Piao, 2018) [16] was then introduced, sitting in front of the analysis module and providing the ready-to-use view model through an API. This allows breaking up the link and split Trace Compass into two parts : the server which is everything before the DataProvider, and the view that queries the data providers. Hence, the Trace Compass viewer component becomes very lightweight, since it does not contain the

logic for creating the system model anymore, and is separate from the rest of the application. This allows the views to be developed independently of the Trace Compass server backend, with modern user-interface technologies such as React. More importantly, nothing stops the view from sitting on a distant platform, communicating through the Internet. For example, taking advantage of browser technology, this sets the foundation for using Trace Compass in a distributed manner.

Another modern work that uses a similar approach for trace analysis is Perfetto, developed by Google. The core module, Trace Processor, contains all the common logic for computing the ready-to-use view model and exposing it as SQL tables. SQL queries are then used by the views in both Android Studio and the Perfetto UI [59]. This avoids reinventing the wheel to a great extent.

The notable difference between Perfetto and Trace Compass is that the Perfetto Trace Processor is not distributed as a binary but rather a C++ library. The clients then wrap that library to create their own backend. Despite sharing the same API and core functionality provided by the C++ library, different views of Perfetto cannot access each others backend directly. This includes the backend developed as a WebAssembly module integrated in the web browser for the Perfetto UI, giving it the capability to run offline [59]. Being a WebAssembly module also means that the Trace Processor is limited by the browser API. For example, it is not straightforward to offer multiple storage backends, as can offer Trace Compass. Thus, even though having different core components modularized in concept, Perfetto is still coupled and difficult to deploy in a distributed environment as is. Having multiple Trace Processor modules running together is also not viable, due in part to the lack of a communication protocol, hence affecting its scalability prospects. It is still a great tool for the original purpose, mainly designed for tracing personal Android and Linux devices.

4.2.2 Distributed tracing

Distributed tracing does not necessarily require high scalability. Some distributed systems are compact, due to physical constraints, and stay relatively small. It is probably why Nagios, one of the most popular surveillance platforms for distributed applications (Tamburri et al, 2020) [27] suffered from scalability issues. The tool laid out a pattern for collecting and analysing distributed trace data, as well making itself extendable with a plug-in system, allowing connections to third party services (Renita et Elizabeth, 2017) [28]. Nonetheless, the fact that there is only one central processing instance holds it from going any further. There were numerous works aiming at solving this issue (Ciuffoletti, 2016) [29] by embedding multiple Nagios instances, thus extending its capabilities.

The recent work for parallelizing trace analysis (Reumont-Locke, 2015) [17], (Martin, 2018) [18] shows attempts at improving the efficiency of the process. This alone, however, is not sufficient to address the scalability issue in distributed environments. As efficient as it can ever be, a single instance of a trace analysis tool is bound by the computational power of the underlying machine, which in extreme cases is at best as powerful as the traced one. Multiplying the traced machine by hundreds or thousands of nodes, we quickly overwhelm the analysis tool. This contention point would be problematic for applications where 24/7 uptime is wanted. An anomaly can escalate quickly in clusters (Matloff) [20] and a delay in analysing its data during investigation does not help. A similar pattern found among distributed tracing platforms consists of multiple tracers, each running on a traced machine, all sending its trace data to a central collector for analysis work. In some cases, the tracer takes the form of a client library to create custom events with specific information related to the application, in contrast to the black-box tracing nature of Nagios. Prometheus achieves scalability by making it possible to organize Prometheus instances into a federation (Reback, 2021) [32]. Indeed, it is possible to set up a multi-level hierarchy of Prometheus servers, each level reporting aggregated data to the upper level. This approach was in fact also used by Nagios, later on, to overcome its own scalability issue (Nagios) [30]. Scaling analysis servers horizontally is a good approach. However, Prometheus analyses are simple, with mostly metrics, and thus the aggregation is viable. Attempting to apply the same model for more sophisticated analysis processes, such as the ones that Trace Compass offers, would be difficult. Their aggregations are highly different, from one type to another.

Scalability is not the only challenge facing distributed tracing. Merging trace data from different machines is not a trivial task either. Trace Compass allows developers to define how to select events to correlate with causality links, in order to time align and merge different sets of trace data together, and follow dependency links for the critical path computation. Linking together traces at different levels like this is somewhat context dependent. For instance, it has been implemented in Trace Compass to link traces of processes on different nodes exchanging TCP packets, to link traces from KVM virtual machines interacting with their Linux host trace, and also to link traces from different processors on heterogeneous systems. However, a specific implementation is required for each type of application (Ericsson) [15]. Traditional tracers are not aware of execution context, and inferring the context during the analysis phase is also difficult, due to the lack of necessary details in the trace data (Uber, 2017) [2]. The most adopted solution for this problem is called context propagation, which is the technique employed by Jaeger. The idea is to include a unique request identifier in every event generated, and to link together requests and their nested requests. This approach is more invasive, because of the need to generate and propagate requests identifiers in the

middleware libraries, but is more reliable and lower overhead as compared to techniques like "Black-box inference" or "Schema-based" (Shkuro, 2019) [21].

4.2.3 Query protocol

Trace Server Protocol (TSP) is the API of the DataProvider layer in Trace Compass. It follows the RESTful philosophy and uses JSON as the data exchange format. However, the work of Chen Kuang Piao showed that the efficiency may be improved by using Protobuf instead (Chen Kuang Piao, 2018) [16]. The TSP endpoints are organized by visualization type, for instance time graphs or XY charts. Users can adjust the resolution of graphs by providing a series of timestamps as parameters upon calling the endpoints [41]. Although it is not a common practice in REST API design, filtering responses based on the provided parameters is possible. In this case, it does not raise any problem, since the responses remain uniform.

Criticisms of REST APIs often relate to the problem of under or over fetching (Vadlamani et al., 2021) [46]. GraphQL is a query protocol and language developed by Facebook with the aim to deliver only the exact amount of data needed. The first and foremost advantage of this is to reduce the amount of data transmitted and thus save bandwidth (Guo et al., 2018) [56] (Brito et al., 2019) [51]. At first glance, it appears that GraphQL can help save bandwidth even further by combining multiple requests in one. However, in practice, this has not proven practical, as we often organize code around small functions (Brito et al., 2019) [51]. Multiple attempts were made to benchmark GraphQL against REST (Sayago et al., 2020) [53] (Hartina et al., 2018) [54] (Lee et al., 2020) [55] with mixed results. The measurement of response time is not meaningful in itself, because REST does not even have a technical specification to begin with, and so the implementation of both protocols is different from one programming language to another.

PromQL is the language used to query the Prometheus server. It is not a standard protocol in the sense that there is no readily distributed library for general purpose usage, outside of Prometheus. However, multiple vendors implemented their PromQL version, based on the open source engine, despite not having the full specification. This brings some interesting ideas in terms of what a query protocol can or not achieve. Aside from simply querying the trace database, PromQL has the concept of “operators” and “functions”. In a few words, operators are for computing operations on the queried trace data, like addition or subtraction, including aggregation operators, and functions are used to derive information from the queried data.

This is what makes PromQL powerful. The operators and functions are passed in the query itself as arguments, allowing filtering and transforming the response in a complex manner,

directly on the server side. This saves bandwidth by filtering unnecessary data, and avoids reimplementing some of this logic on the client side.

4.3 Proposed solution

4.3.1 Motivation & Background

The distributed tracing analysis platforms discussed above, even when following the OpenTracing API, are tightly coupled to their own ecosystem, most of the time, even the ones with strong support for external services, like Prometheus that only supports its in-house trace data format. Understandably, it is easier to provide powerful query operators when the trace format is fixed. As a matter of fact, it is possible to partly adopt such a framework and use it with custom trace formats and storage backends, with the trade-off of losing some PromQL powerful features and relying on other query services (Sukhija et al., 2020) [31]. This approach however is cumbersome and not standardized.

Supporting different types of trace format is where Trace Compass shines. This framework easily allows to implement custom front-end parsers or even new types of analysis. Along with the separation of the Trace Compass client and server sides, it is now a potential candidate for distributed deployment. Trace Compass server as-is can communicate with a remote client, thanks to the Trace Server Protocol, but with the server being able to run only as a single instance, the volume of trace data in an HPC cluster, for example, quickly overwhelms its capability in terms of CPU, storage, and RAM memory. Furthermore, the Trace Compass server currently requires all trace data to be stored locally to be analysed.

It is of course possible to run multiple Trace Compass servers, each used to evaluate the traces of several machines in the cluster independently. Nonetheless, a view on the cluster as a whole is highly desirable. In order to make Trace Compass more suitable for tracing large heterogeneous systems, we need a re-architecture with scalability in mind. We would also want powerful query capabilities, similar to existing platforms, without sacrificing its flexibility regarding the development of new types of analysis and trace format. This requires an abstract layer, such that new query operators can be developed with minimal effort, without changing how the platform behaves overall, similar to the idea of the trace parser or the DataProvider layers, providing an abstraction of the view regardless of the type of analysis in Trace Compass.

4.3.2 Overall architecture

Our proposed architecture follows an approach similar to existing distributed tracing platforms (e.g. Jaeger, Prometheus). Figure 4.1 shows a high level representation of the tracing backend architecture, for which each of its components are about to be discussed in details. The platform consists of three main parts, the traced nodes, the analysis nodes and the coordinator node. An analysis node is where the analysis of trace data, produced by traced nodes, takes place. Each of the analysis nodes is responsible for a different group of traced nodes. The coordinator node, when queried by the client/view, will in turn query the analysis nodes and merge the responses together, returning an aggregated analysis result. When a client sends a request to a server for trace analysis data, the biggest part of the response is the data points, often formatted as a time series array. These data points cannot be filtered by its keys, and thus does not fit into the schema mechanism of GraphQL. Other metadata, like trace details, are very small in size compared to the analysis data points, thus filtering any unnecessary part is not worth the overhead and complexity trade-off of using GraphQL. PromQL, with its operators, brings some interesting insights, especially for complex combinations. However, in our model, operators are quite different. It is not clear if chaining these complex operators would be interesting, or even possible in general. For these reasons, we used the Trace Server Protocol (TSP) as the communication protocol for trace data and analysis requests. First, it already offers the capability of filtering data points through the timestamp parameter, and operators can be integrated easily by slightly extending the protocol. Second, it is already implemented by Trace Compass server, a component we choose for our prototype implementation. More complex query languages can be envisioned in the future if needed.

The grey blocks represent traced nodes. Each of these machines runs an agent process in the background, ideally deployed as an infrastructure module. Upon being configured, the coordinator broadcasts a message containing the hierarchy details to the agents. Knowing the assigned analysis node, the agent then collects and sends over the trace data produced by the traced machine. Upon receiving the data, the analysis node starts the indexing process and the coordinator node can poll the analysis node to know when it is ready for query and display.

This sidecar agent pattern allows the traced machine to actively control its trace data transfer process, and choose the best moment for the job. The agent can also be used by the coordinator to control the tracer indirectly, leaving the choice of tracer flexible. Moreover, the original TSP protocol is designed for trace data to be imported by users to the analysis server, thus having the agents pushing trace data instead of being pulled for it. The agents act as clients

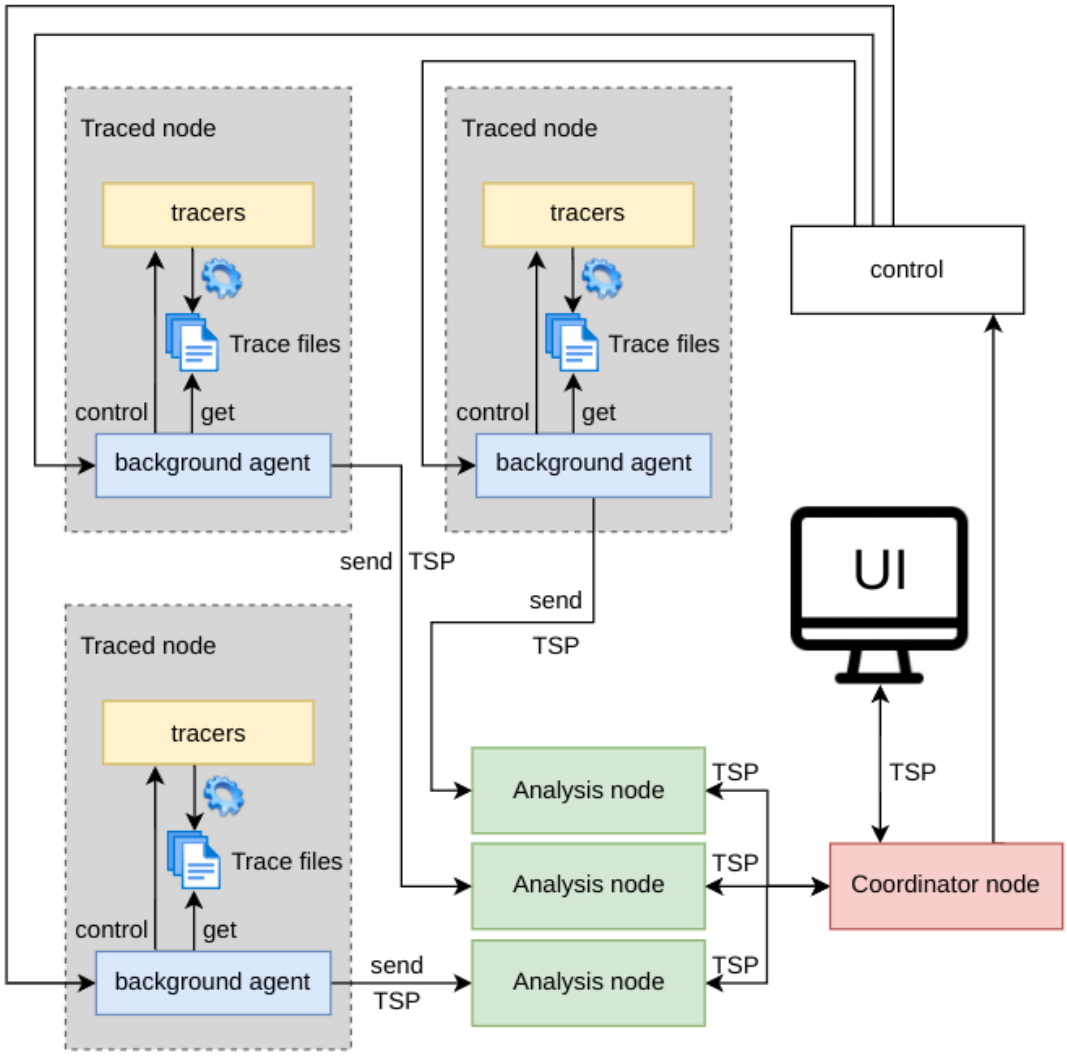

Figure 4.1 Proposed scalable distributed trace analysis platform

and comply with the current protocol. Ideally, we would want to reuse an analysis node easily for resource reasons, meaning it should be possible to analyse traces of different clusters in different moments. The necessary element to make this work is a unique ID associated with a set of trace data (e.g. ID1 for traces from cluster1, ID2 for traces from cluster2), so that the analysis server knows which *trace set* a request is referring to. Indeed, TSP has a notion called "experiment", which is a collection of traces, to group multiple different traces together once imported. Each experiment is identified by a unique ExperimentID. Currently, however, the ExperimentID is supposed to be created by the analysis server and thus is only unique locally. In order for this to work with our architecture, we need an ExperimentID that is unique across the analysis servers. Though, since the goal is to analyse traces from different

machines in a cluster as a single system, the ExperimentID should be the same for traces from machines of the same cluster. It is logical to use the coordinator as the central entity to create this ExperimentID, since only it knows the whole hierarchy details of the tracing platform. The ExperimentID would be included in the broadcasting message to the agents.

Apart from the control broadcasting message system, TSP is used for communications regarding trace data transfers and analysis, between different parties in the platform. A slight change will allow specifying an ExperimentID when importing traces. For the purpose of this paper, only one coordinator node is presented in the figure, but it shall be possible to deploy a multi-level hierarchy, with multiple coordinator nodes, where each level responds with aggregated data to the next higher level.

4.3.3 Aggregation of trace analysis

TSP exposes the analysis results formatted as a view model data. As discussed previously, this has the advantage that the client can just display it directly, without any further processing. As such, the aggregation in our proposed solution is executed on this formatted result, but not the "raw" one as returned from querying the state system in the case of Trace Compass. Currently, most of the analyses are presented as a data points series throughout a timeline. Thus, requesting an analysis result through TSP only essentially requires the timestamps of the analysed timeline to be provided (Listing 1). When receiving requests from the client/view, the coordinator node can simply keep the request as is, duplicate and propagate it down the hierarchy to the other nodes, then aggregate the returned responses all at once. This works because we are analysing the traces from different machines, produced during the same period independently. Hence, the querying time interval is the same. In a way, we are using Map/Reduce except that the processing data is already distributed, with the analysis nodes being the workers for Map phases, and the coordinator node being responsible for Reduce phases.

```
1  {
2    "parameters": {
3      "requested_times": [111200000, 111300000],
4    }
5  }
```

Listing 1 TSP analysis request parameters

Upon receiving a request, the coordinator queries the analysis nodes and then merges all their responses, to produce the final response for the client. This behavior is typically one of the single-pass Map/Reduce. Let us take a use case that, when aggregating data from multiple sources, computes the ratio of each value to the "sum" of data (e.g. percentage of each source). Currently, this computation is carried out only by the coordinator, since only it knows the "sum" after acquiring the results from the analysis nodes. This should not be an issue, regarding storage limitations for example, since the size of the responses together, even for big traces, is very small compared to the size of the trace itself. Thus, it can be handled by a single coordinator machine, with multi-thread if necessary. What can possibly be better is if the coordinator can query only the "sum" of each analysis node first, then computes the total "sum", then sends this value to the analysis nodes for computing the ratios themselves. This becomes a multi-pass Map/Reduce. In the first pass, the Map phase produces the local "sum", followed by a Reduce phase that produces the total "sum", this information is then included in the requests for the second pass Map phase that computes the relative data. The final Reduce phase just collects these values, without further processing, and returns these ready-to-use data to the client. This is especially efficient when the computations in the first and second phases are mostly independent, presenting very little overlap, and the communication latency of multiple phases is not too significant. In order to take advantage of multi-pass Map/Reduce where possible, we need to first rethink how each request is handled by an analysis node.

Analysis automata

To be suitable for multi-pass Map/Reduce execution, the handling of each analysis request needs to be separated into different parts, where each corresponds to a Map phase. Clear boundaries between the parts should be established since the model expects these parts to run independently, one after another. Instead of a long procedure, which is difficult to subdivide, we can define a trace analysis as a series of smaller analyses, each one being a step closer to produce the final result of the "whole" analysis. The next step is executed each time the analysis node receives an analysis request, providing that the required information is included. We can then think of each analysis as a simple automaton A where :

— 0...s represents the steps of an analysis.
For each step,
— $R_i(0 \leq i \leq s)$ is the transition rule required to enter it, for instance it checks if the information necessary for the execution of F_i is provided.
— $F_i(0 \leq i \leq s)$ is the function to be executed.

— $I_i (0 \leq i \leq s)$ is the input from the coordinator.
— $O_i (0 \leq i \leq s)$ is the output.

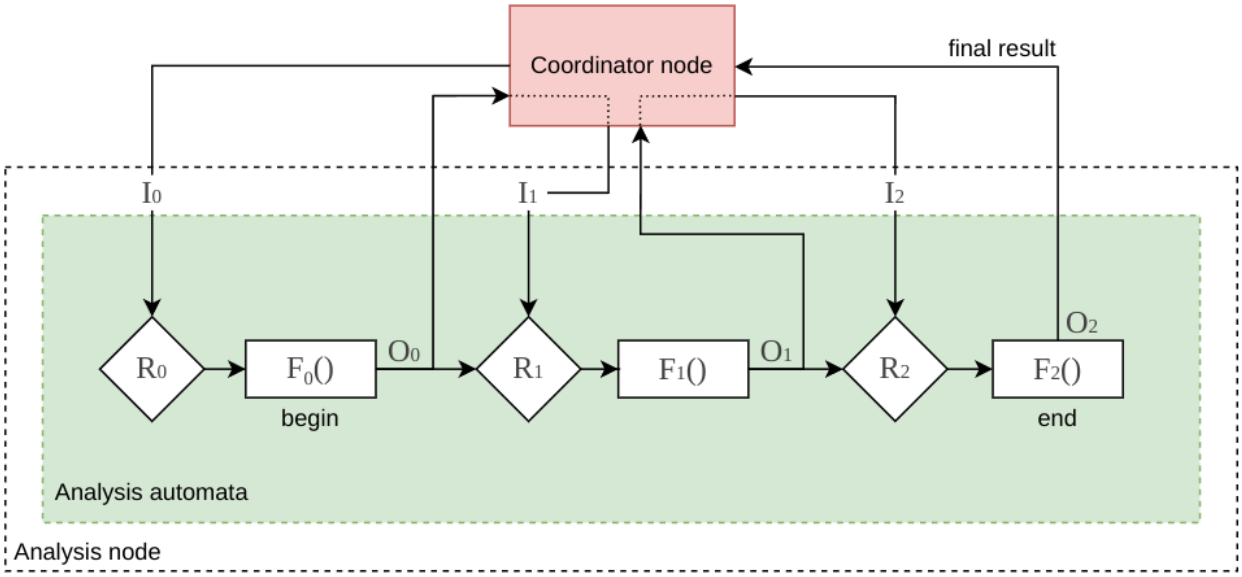

Figure 4.2 Analysis automata

The automaton is an abstract concept that holds any information related to an analysis, each identified with a unique ID. It allows the coordinator to keep track of the progress of each analysis in case the analysis is interrupted unexpectedly and allows the analysis node to alternate between different multi-pass analysis, requested by different coordinators, without having to finish the whole analysis. Theoretically, it also makes caching the analysis intermediate and final results easier.

Each Map phase in the multi-pass Map/Reduce corresponds to an automaton's step, executed upon receiving a request from the coordinator. Figure 4.2 depicts an example of the execution flow of an analysis automaton. The coordinator initiates the execution by sending a request containing parameters I_0 to the analysis node. If I_0 complies with R_0, F_0 executes. Though omitted, R_0 could also require other internal values. The output O_0 of F_0 alone does not fulfill the R_1 requirement, thus the automaton is halted and O_0 is sent back to the coordinator. The coordinator proceeds with a Reduce phase to derive some aggregated data from O_0 of all analysis nodes, which is then included as I_1 in the second request to the analysis nodes. F_1 then executes if R_1 condition is met (which requires the output of the previous step together with I_1), and so on until the result of the final step is returned, which is the final result of the whole analysis.

In the case where "reducing" data in the middle of the analysis is not desired, for instance when there is only one analysis node running, or it is simply not possible to do so with a certain type of analysis, we can change the transition rules so that no external "reduced" data is ever needed to achieve a pass-through of all steps, thus making the procedure single-pass again. Take the execution flow above, this means the I_i coming in R_i in each step would be removed from the equation, making R_i solely relying on the output of the previous step. Thus, the execution should fall through, without being interrupted until completed. Switching an analysis process between single-pass and multi-pass modes is therefore feasible and is a matter of configuration. Hence, the introduction of multi-pass does not affect the current behavior, or demand two distinct implementations.

As in the case of the ExperimentID, we shall introduce an AutomataID that is unique to each life cycle of an automaton. When the automaton is halted, it should not wait in a blocking state, for it cannot be guaranteed that the next request always takes minimal time to arrive, or if it will arrive at all, due to external factors like network conditions. Therefore, this ID allows the analysis node to completely store the automaton somewhere and be in a "blank" state, even in between steps to handle others things. It can restore the automaton state later to continue if required. Furthermore, it is possible for the coordinator to make multiple requests for different analyses while waiting for the results that might not come in the same order. Thus, this AutomataID needs to be managed by the coordinator, to insure its uniqueness across all analysis nodes, and can be used by the coordinator to link unordered responses.

Use cases examples

We will now look at concrete examples of how the coordinator aggregates the information received from the analysis nodes. These examples are data taken more or less directly from the analyses executed during the benchmarks in subsection 4.4.2.

As mentioned in 4.3.2, traces distributed across analysis servers, that are supposed to be viewed together, are packed into collections called "experiments" on each analysis server and linked by a globally unique ExperimentID created by the coordinator. Experiments are the entity that analyses are run upon. The coordinator might know the semantic of the ExperimentID (e.g. ID1 is for traces from cluster1), but the TSP protocol requires other information regarding the experiment, such as the required time range for the analysis. The coordinator has no knowledge of this information initially (before the aggregation of the experiments), since they are tied to the traces residing only on the analysis nodes. Therefore, before requesting any analysis, the client/view needs to make an initial request to the coordinator in

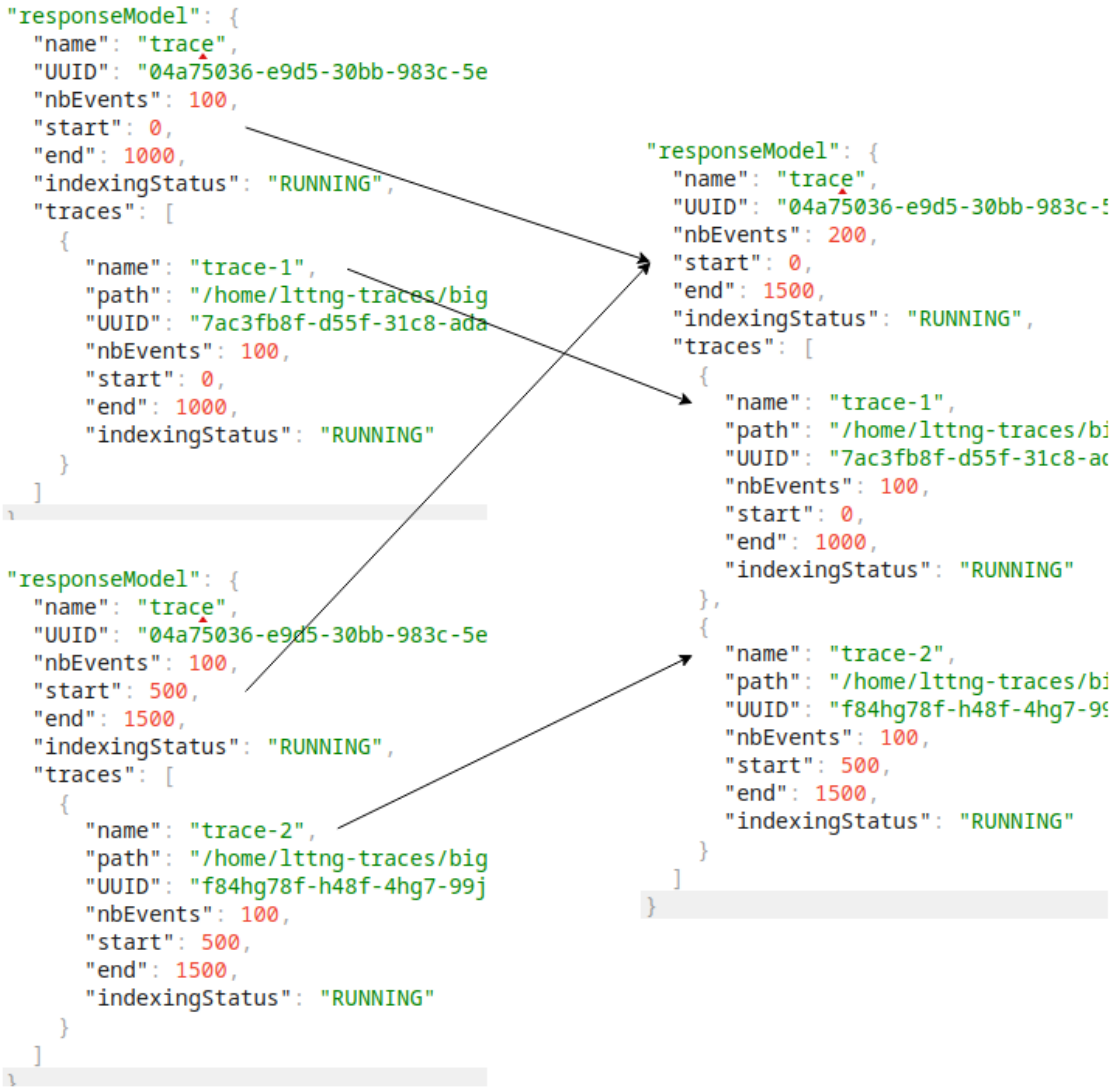

Figure 4.3 Aggregating experiment information

order to retrieve the basic information about the targeted experiment. After receiving the client request, the coordinator issues the same request to all the concerned analysis nodes. These latter will return their responses to the coordinator, that will use them in order to compute the aggregated information. It sums up the number of events "nbEvents", to have the total number of events from all analysis nodes, and keeps only the minimum "start" time and maximum "end" time to reflect the time range that covers all the experiment. This data is then returned to the client, with the updated fields and a concatenated list of traces from all analysis nodes. The client will see the result as one experiment on which it can start issuing analysis requests. In figure 4.3, we show an example of two analysis nodes, each having one experiment (containing one trace) that will be aggregated by the coordinator into one global

experiment, containing the two traces, by aggregating the different fields (number of events, start time, etc.).

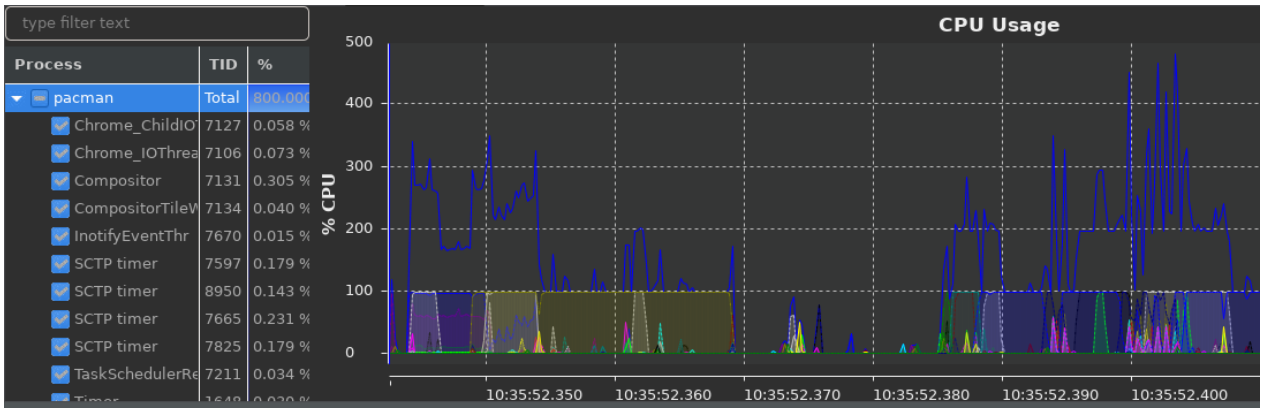

Figure 4.4 CPU Usage analysis

CPU Usage Analysis is a common analysis that can be requested through TSP (figure 4.4), showing the CPU usage of different processes over a timespan. In TSP lingo, this analysis is of type XY, since the responses returned from the analysis servers are formatted as a model for an XY chart (the X axis being the timeline and the Y axis being the CPU usage). For all XY type analysis, the client typically makes two distinct requests to get the full result. The first one is to retrieve the XY tree, a list of elements containing the data that is going to be displayed in the XY chart. In the case of the CPU Usage Analysis, these elements are the processes and threads. The client is required to provide a time range (the request start and end times) so that only the processes that fall in between this time range are included in the tree. In listing 2, because we are querying the full analysis for testing purposes, the time range interval corresponds to the start and end times of the experiment itself.

As the coordinator receives this request from the client, it can send the exact same request to all analysis nodes, without changing the time range parameter. First, the analysis servers already filter out the time range that is outside the valid range, hence a query with a time range larger than the actual existing one should not be a problem. If the client is querying an analysis for the full time range, the start and end times of the experiment, returned by the coordinator to the client, already take into account the earliest start time and the latest end time in all nodes. Hence, it is certain that the full time range covers the biggest possible range, and therefore the coordinator can use the same time range for all analysis nodes. Traces are expected to have the same timescale here, otherwise the coordinator should not

```
{
  "parameters": {
    "requested_times": [
      0,
      1500,
    ],
  }
}
```

Listing 2 CPU usage process tree request parameter

be interfering with the time range regardless, but a separate resynchronization operation should be carried out.

Figure 4.5 shows the aggregation of the responses for this request. Basically, the XY tree is a table with lines and columns. The "entries" field contains lines. Within each line, the columns are represented by the "labels" field. The data in columns are arranged in the order corresponding to their semantic in the "headers" field. In the CPU Usage analysis, this tree acts as a process explorer that allows users to select the element being displayed on the chart, since showing everything can be an expensive task for a big trace. Each element in the CPU Usage analysis tree has a process name, along with their TID, its CPU time during the time range, and its CPU usage percentage.

There are several ways to aggregate the information in the tree, depending on the use case. In this example, assuming that we are analysing the traces for the same programs executing across different machines in a cluster (same parallel computation on multiple nodes for different data points), we will try to merge those similar processes in the trees together, and show its average usage across all traces. This corresponds to updating the third and fourth values of the "labels" fields with their average values across multiple nodes. The problem of finding the similar processes in the trees is out of the scope of this project. The process name may not be a sufficient indication and the process id is basically assigned at random. Nonetheless, finding the corresponding processes on different nodes has a big influence on the feasibility of this type of tree aggregation, because it directly affects the running time of the Reduce phase. The best possible case is that the coordinator can seek the similar process in $\mathcal{O}(1)$, and we simulate this behavior by just finding the entry with the same "id" field, which is conveniently the same as the index of the entry in the "entries" list. The client receives the final result from the coordinator, with only these updated mean values.

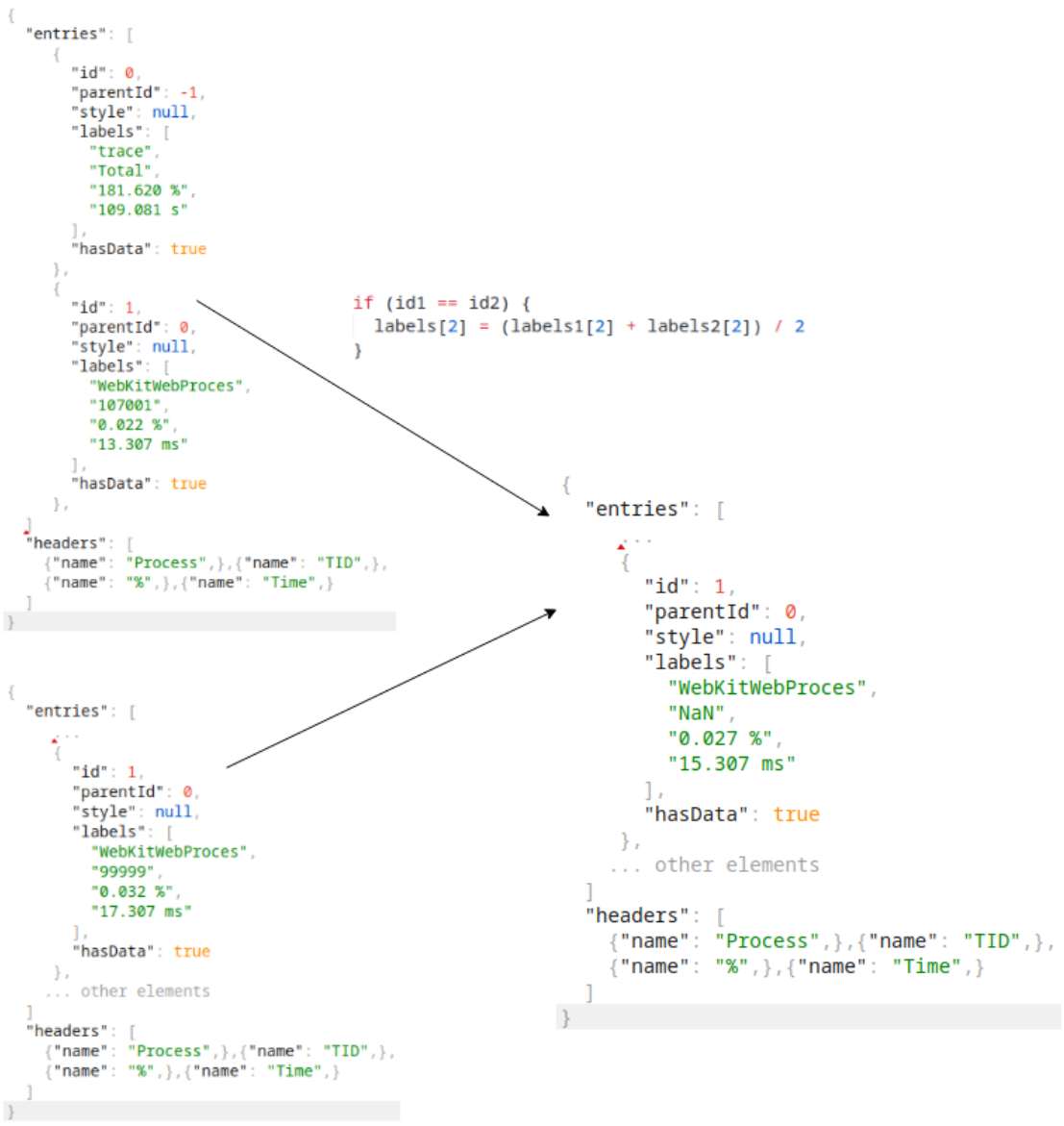

Figure 4.5 Aggregating CPU Usage Analysis's XY tree

Once the client "selects" the process it wants to see on the XY chart, a second request takes place to retrieve the actual data points. As mentioned above, the chart shows a series of data points through a period of time. The time interval between events in trace data is often at the microseconds or nanoseconds level. Showing every single one of them for traces lasting many hours is expensive and unnecessary, as the screen is limited by its resolution anyhow. Therefore, the second request of XY type analysis requires the list of timestamps, which allows controlling how much data the client needs, alongside a list of selected processes. As in listing 3, the "selected items" contains a list of "id" fields of the processes found in the

response of the first request. One problem arises here, the "id" field is created by the analysis servers for its computation, and therefore might be different for the same process from one analysis node to another. However, since the client only sees the merged information, it sees only one process with one identifier. The identifier must somehow be translated from the client side to the server side. The ideal case is if we can seek the similar process in $\mathcal{O}(1)$ while aggregating the tree. This implies that a unique identifier can be computed for its characteristics, and hence be unique for similar processes across all analysis nodes. Otherwise, maintaining a mapping of client side and server side ids in the coordinator would be expensive, since every subsequent request have to go through the mapping, which would affect the performance. A better option is to have the coordinator do the mapping at the aggregation phase. After sending the aggregated response to the client, the coordinator would make another request to the analysis servers for them to update their process ids with the coordinator assigned one. From here, subsequent requests can go through with the unique id directly. This supplementary request does not affect the clients response time, since it is accomplished in "background" while the client is viewing the analysis result. Moreover, this happens only once, since subsequent interactions with the chart (e.g. zooming) make use of the same processes, only the requested data points change. This was not implemented in the prototype, as a trivial mapping algorithm was sufficient fot the use cases in the benchmark.

Listing 4 shows the structure of the XY chart model returned by the analysis servers for this request, with the "yValues" fields containing the list of data points for the Y axis. Since the analysis servers return their responses using the unique id ("seriesId" field) for each of the processes, the coordinator simply iterates through the processes that have the same id and calculates the average values of each data point for the client, similar to how it calculated the

```
{
  "parameters": {
    "requested_times": [
      0, 5, 10, ...
    ],
    "requested_items": [
      0, 1, 2, ...
    ],
  }
}
```

Listing 3 CPU usage XY data request parameters

```
"model": {
  "series": [
    {
      "seriesId": 0,
      "yValues": [
        34204000,
        81346600,
        ...
      ],
      "xValues": [
        0, 5, 10,...
      ]
    },
    ...
  ],
  "title": "string"
}
```

Listing 4 CPU usage XY data

average values in the CPU process tree. In both the aggregation phases of the CPU process tree and the XY chart data, if seeking to the targeted process takes $\mathcal{O}(1)$, the coordinator iterates over the m "yValues" and the n analysis servers, thus it takes $\mathcal{O}(mxn)$ to complete the aggregation.

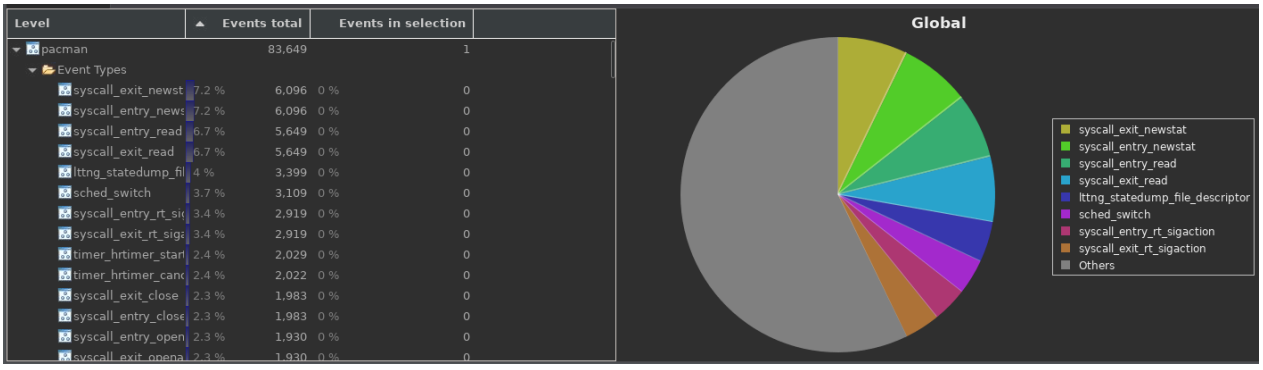

Figure 4.6 Statistics analysis

Statistics is another analysis considered here. This view shows the statistics for each type of event in the analysed trace (Figure 4.6). Only one request is needed to get the statistics data. Similar to the request used to get the process tree in the CPU usage analysis, the client needs to provide the time range in the request parameter. The coordinator also treats it in the same manner.

```
{
  "entries": [
      {
          "id": 0,
          "parentid": -1,
          "style": null,
          "labels": [
              "trace",
              "42372757", // total number of events
              "100"
          ],
          "hasdata": true
      },
      {
          "id": 1,
          "parentid": 0,
          "style": null,
          "labels": [
              "syscall_entry_geteuid",
              "3835",
              "0.009050626561778834"
          ],
          "hasdata": true
      },
      ...
  ],
  "headers": [
      {"name": "level"},{"name": "events total"},
      {"name": "percentage"}
  ]
}
```

Listing 5 Statistics data

Listing 5 shows the response to the statistics analysis request from one analysis server. The

result may be seen as a statistics table, with lines and columns similar to the CPU process tree. In this case, the statistics has three columns for each element, its name, its number of events and its percentage on the total number of events of the trace handled by the analysis server. The total number of events is represented by the first element. From the client side, it would make more sense to see the percentage per event type based of the total number of events of all nodes. For the purpose of this example, we simply do not handle the case of similar entries, like we could have for the CPU process tree. To achieve this outcome, the coordinator would have to iterate through the list in the "entries" field, the whole execution therefore would run in $\mathcal{O}(n)$. This is where multi-pass execution would be beneficial, the coordinator can make one first request to retrieve the total number of events from all nodes (retrieving only the first element of the list), sum it up and gives it back to the analysis nodes. The analysis nodes then use this updated information to calculate the percentage directly, instead of their local total. The aggregation workload in this scenario consists of simply calculating a sum of integer numbers, and therefore runs in $\mathcal{O}(1)$ (for a constant number of data points to be shown). This is a significant improvement over single-pass execution. As discussed at the beginning of this section, this is beneficial with the condition that handling the first request runs much faster than the second request in each analysis server. It is indeed the case in the Statistics analysis of Trace Compass, since the total number of events is not retrieved by summing the number of events of each type, but maintained as a state in the state system itself, thus making the cost of querying this information $\mathcal{O}(1)$ for each data point, regardless of the number of event types.

Aggregation of other types of TSP analysis : During our research, we also investigated aggregations on analysis views other than those mentioned previously, in particular the Resources view and the Control flow view.

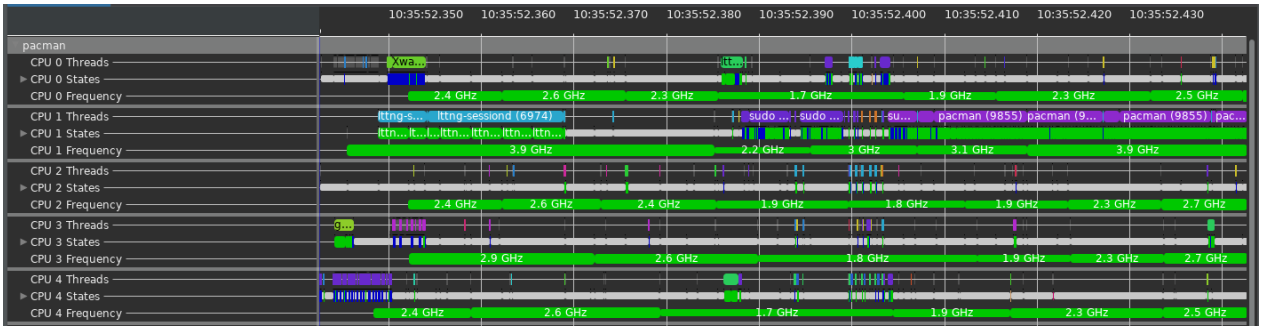

Figure 4.7 Ressources view analysis of Trace Compass

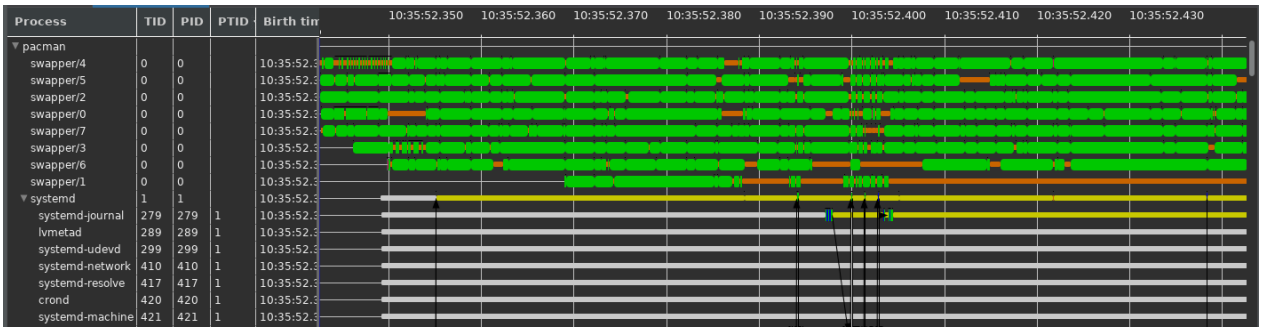

Figure 4.8 Control flow view of Trace Compass

The Resources analysis view (figure 4.7) allows to analyze the state of each CPU core during the execution of a program. Therefore, there is no real benefit in aggregating them with the information of another CPU belonging to another machine. The Control Flow view (figure 4.8) allows the user to analyze the execution flow of a program in details, with CPU wait times as well as CPU information on which the program is running. In the same way, there is no real benefit in aggregating the execution flows of two distinct programs. The potential use case might be in comparing the execution flows of the same process on two different machines, but that is not part of the aggregation operation. If these processes are part of an execution chain, like nested distributed requests, this use case may be better suited for specialized tools like Jaeger or OpenTracing. That said, if necessary, it is relatively easy to implement new aggregation operations for the coordinator, thanks to its modular architecture.

Threads status aggregation into one pixel line : In this paragraph, we are proposing and describing another aggregation approach that might be used with some types of analyses, like the Resources analysis, in some specific use cases.

In (figure 4.7) we can see the Resources view analysis of Trace Compass. Compared to the CPU Usage view, represented as an XY chart with two axis, the Resources view is represented as a series of states throughout the timeline axis (a time graph). Each row corresponds to one resource (CPU) of the running machine. In the case of an HPC with hundreds or thousands of nodes, having multiple rows per machine, each representing one detail, quickly overwhelms the user with information. It then becomes very difficult to properly evaluate the situation. We can instead display for each node (or multiple ones) a single row that contains the most significant piece of information. It serves as a visual alarm of an anomaly. In the case of Resources analysis for example, it may consist in representing the status of all the threads of a specific analyzed node with only one line of pixels, using some simple operation. This

aggregation method is useful when the user might want to have a global view of the status of a sub-cluster, or of all the machines composing the computing cluster itself. Viewing the nodes global status with one line will help to quickly draw the user attention to a specific machine that might be problematic, thereby isolating the debugging from thousands of machines to just 1 or a few machines. This can make anomaly detection easier and faster.

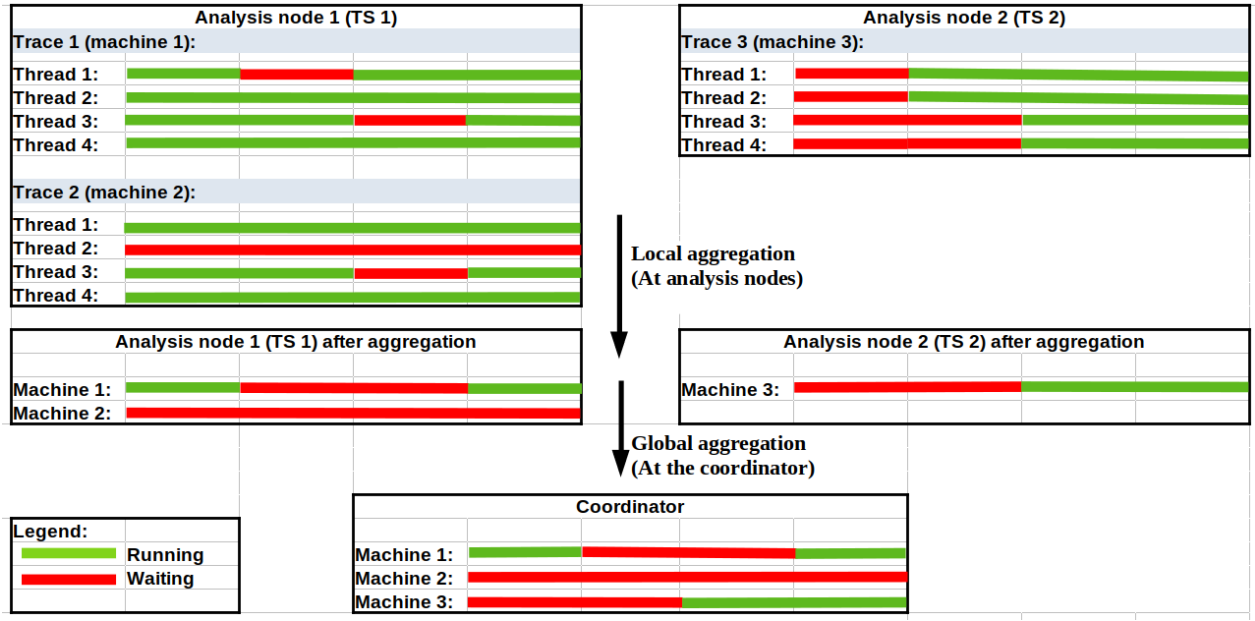

Figure 4.9 An example of one line pixel aggregation applied on threads status

The aggregation should be done on the trace server nodes, so that the job can be done in parallel. This means that it would be necessary to add an additional processing step at the trace server level, before returning its query response. During this step, the aggregation will be performed locally, thus returning the global status of the node, instead of the states of its different threads. This could be done, for example, by adding an optional parameter in the TSP query parameters that specifies the type of the aggregation we wish to use. It is possible that the analysis node processes many traces coming from different analyzed nodes. In that case, the aggregation should be done by regrouping the threads status of each node together. This can be done using the host ID (UUID) from the trace.

An example of the algorithms that may be used to choose the piece of information to be displayed on the one line pixel is min/max/mean, which can be easily applied on numeric values such as the cpu frequency. The exact operation can be choosen by users through a parameter in the TSP query. A slightly more sophisticated algorithm is to choose the "significant" state representing all CPUs in the node. The exact detail of "significant" depends

on the analysis type. In the Resources analysis, for example, we can choose to display an idle state in the node time graph (one or more CPU cores are in the idle state), which allows users to quickly see if some CPU cores become idle for long intervals during their execution. Users can then request more details for that specific node and time range, to "zoom in" the suspicious anomaly. Instead of displaying the wait status, we can also display the predominant state of the node, that is the state which is in majority among its CPU cores. Similar to min/max, users can specify the wanted operation along with its details in the TSP request body. In Figure 4.9, a diagram explains this aggregation method on the threads of 3 traced machines, being analysed by two instances of trace servers. We use an aggregation algorithm based on displaying the "waiting" status as the most significant state in this example.

In this paper, we do not benchmark this aggregation approach, since our prototype has covered more complex approaches. Nonetheless, it may be implemented in the future.

Running time analysis

This proposed architecture is essentially the divide-and-conquer approach, with slight differences. Suppose that we divide-and-conquer-ly analyse a trace with n events, we divide it into α equal sub-traces and analyse each of them, then merge the results at the end. The running time $T(n)$ of the analysis would be.

$$T(n) = \alpha T(n/\alpha) + D(n) + C(n) \quad (1)$$

That is, α times the analysis running time $T(n/\alpha)$ of each sub-trace of n/α events, plus the time $D(n)$ it takes to divide the trace and the time $C(n)$ required to merge (conquer) the results at the end.

In our model, the sub-traces are analysed in parallel by multiple analysis nodes at the same time, thus we do not need to multiply its running time by α number of nodes. Let k be the number of steps we create for such an analysis, $k(D(n) + C(n))$ would be the total time the coordinator takes to separate the requests and aggregate the results of each step. As we saw in the example 3 of the CPU Usage analysis, $D(n)$ most of the time is $\mathcal{O}(1)$ since the timestamp parameter is the same for all analysis nodes, therefore, we can safely replace $kD(n)$ with a constant c. The number of steps k does not affect $T(n/\alpha)$ either, since each step is a linear part of the original analysis running on such a trace, the whole execution of all steps would still take the same time. This would give us.

$$T(n) = T(n/\alpha) + kC(n) + c \text{ with } c > 0 \quad (2)$$

The goal here is to keep the overhead low enough so that our approach performs equally or better than $T_{singlenode}(n)$. The reasoning is that even in the worst case, where there is no performance gain, we already save the time of grouping together traces from different nodes, and also benefit from the enlarged storage. Thus, we can establish the bounds on $C(n)$ using the condition $T(n) \leq T_{singlenode}(n)$

$$T(n/\alpha) + kC(n) + c \leq T_{singlenode}(n) \quad (3)$$

$$C(n) \leq \frac{T_{singlenode}(n) - T(n/\alpha) - c}{k} \quad (4)$$

substitute with $T_{singlenode}(n) = \alpha T(n/\alpha)$

$$C(n) \leq \frac{\alpha - 1}{k} T(n/\alpha) - c \quad (5)$$

There are a few minor simplifications in this analysis. First of all, theoretically, the bound of $C(N)$ should include the overhead from the network latency, since its value also depends on k and n. However, this overhead can be dismissed if we consider it already compensated by the time we saved from transmitting trace data from multiple nodes, in the case of using only one analysis server. The variable n might be different, depending on the trace analysis algorithm. For example, in Trace Compass, n as the number of events is irrelevant because, once the state system is built, subsequent analyses only query the state system and do not depend on the trace size anymore, but only the running time of the queries. As mentioned above, we aggregate the formatted results instead of the “raw” results. Thus, $C(n)$ depends on the TSP result size. In fact, the analysis server also takes time to convert the queried results into this format, thus we can loosely consider that $T(n)$, the running time, "depends" at least on the size of its result and so let n be the size of the result instead of the number of events. Omitting the running time, for building the state system in the analysis, is also justifiable because it does not affect the upper bound. We do not want the analysis to perform worse, even when the state system is already built. Secondly, $T(n/\alpha)$ as the running time of each analysis server is not exactly true in our model, since we do not divide the traces equally but record them independently. The coordinator, however, can only aggregate the results once all of them are received. Thus, n/α can easily be substituted by the maximal value. Most importantly, $T_{singlenode}(n) = \alpha T(n/\alpha)$ strictly assumes that $T_{singlenode}(n)$ is linear with n, which might not always be the case. Even so, the condition would still hold, since the linear case gives us the minimal upper bound. These simplifications are acceptable because we are not analysing rigorously the exact algorithm ($T_{singlenode}(n)$ and $C(n)$ are not known) but merely defining a "budget" for $C(n)$ relative to $T(n/\alpha)$ by a factor $\frac{\alpha-1}{k}$ to help us choose the appropriate aggregation algorithm and easily verify it experimentally.

4.4 Evaluation

4.4.1 Implementation

We implemented a prototype of the proposed solution using Trace Compass as the analysis server. The coordinator is implemented in NodeJS v14.18.12. The side-car agent is omitted, since it does not have any effect on the measure of the analyses and its aggregation. All traces are considered to be already imported in each analysis node.

The two types of views, supported by the Trace Compass server, and chosen for the prototype are the Statistics view (an example of the multi-pass case) and the CPU Usage (an example of the single-pass case).

The analysis phase in our model also includes the DataProvider workload in the case of Trace Compass. Thus, we define the abstract class *AnalysisAutomata* that sits above the DataProvider layer.

```
1  // T return type
2  public abstract class AnalysisAutomata<T> {
3      private int step = 0;
4      private int final_step = 0;
5      private boolean done = false;
6      public void advance(){
7          this.step++;
8          done = step == final_step;
9      }
10     public boolean isDone(){return this.done;}
11     public abstract T execute();
12 }
13
14 public final class AutomataManager {
15     public static HashMap<String, AnalysisAutomata>
16         instance = new HashMap();
17     private AnalysisAutomataManager() {}
18 }
```

Listing 6 Analysis Automata Class definition

Each type of analysis extends the abstract class and defines how *execute* should behave. When implementing the method, developers should define its behavior, based on the current *step* of the analysis, and call *advance*() at the end of each step. There is also an *AutomataManager*

that holds a table of automata linked by their identifier, so that the analysis server can retrieve (or create if it doesn't exist) the correct automata for the received request. The implementation class UML is provided in figure 4.10.

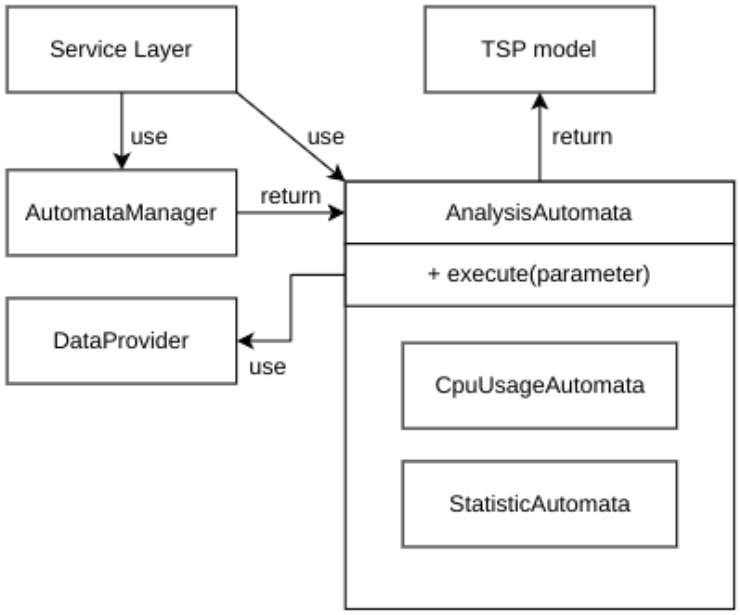

Figure 4.10 AnalysisAutomata UML

We also extended the TSP protocol to allow the coordinator to specify an AutomataID when making a request, and receive a response with the current step of the analysis. If used with a single client and analysis server, it proceeds as usual, and the client does not need to know about the presence of the automata information.

As in Figure 4.11, the coordinator implementation is composed mainly from components called aggregator. Upon receiving a request, the coordinator dispatches it to the analysis servers and wait for all the responses to return, then collects them into an array. The array of TSP responses is then passed through the aggregator registered for that specific request type.

Similar to the DataProvider in Trace Compass, the aggregators are also divided into types like XY chart...etc. This is because each type of DataProvider on the analysis server returns a TSP response in a different format and should be treated accordingly. As in listing 7, when implementing a new aggregation algorithm, developers only need to register their aggregator to a path that is composed of the type of the DataProvider, the aggregation operator, and the analysis automata step it handles. The aggregator itself is a function that accepts an array of TSP responses in the respective format. The coordinator does not enforce any format on

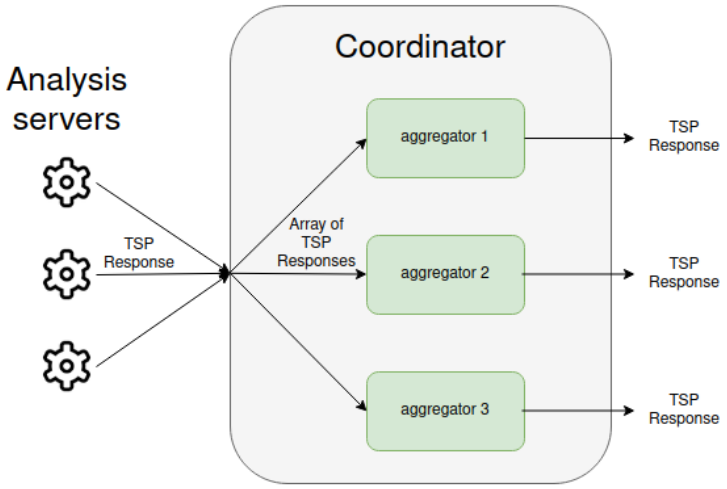

Figure 4.11 Coordinator implementation

```
cpuUsageTree = function (payload: TspXYResponse[]): TspXYResponse;

const xy_tree_aggregator = new Aggregator(
  {
    [`cpuusage.CpuUsageDataProvider/tree`]: cpuUsageTree,
    [`statistics.StatisticsDataProvider/tree/0`]: statisticsStep0,
    [`statistics.StatisticsDataProvider/tree/1`]: statisticsStep1,
    [`statistics.StatisticsDataProvider/tree`]: statistics,
  },
  `xy_tree_aggregator`,
);

aggregate = (payload: AggregatorsPayload): object => {
    switch (payload.type) {
        case AGGREGATOR_PAYLOAD_TYPE.XY_TREE:
            return xy_tree_aggregator.aggregate(payload);
        ...
    }
};
```

Listing 7 Aggregators implementation

the returned value of the aggregators.

4.4.2 Running time measurements

To evaluate our solution, we measured the running time of different analyses on a single Trace Compass server, against our model that includes one coordinator and four Trace Compass servers. Both analyse the same collection of traces. The Trace Compass server instances and the coordinator are deployed on desktop machines located on the same local network with 1Gbps Ethernet connections. All machines have Intel(R) Core(TM) i5-9400F CPU @ 2.90GHz, 16 GB of main memory and run on Fedora 34. The collection of traces used for testing is increased in size gradually, from 1.3 GB to 102 GB. This is the size of traces per analysis server. If analysed by the single Trace Compass server instance, the traces are duplicated by four (the number of analysis servers) to simulate the same set of traces. That is, if in the coordinator model each analysis server has 100 GB of traces data, the single instance will handle 400 GB of traces data.

The implementation of the trace coordinator, the Trace Compass patch, the trace duplication scripts, and the benchmark script can all be found at : https ://github.com/trace-coordinator

We ran the benchmark with different trace sizes for these scenarios of trace analysis : CPU Usage, Statistics (1-pass), Statistics (2-pass) and also measured the indexing time after the traces are imported. For each of these scenarios, we measured two types of running time, the running time of the first analysis execution after the traces are imported and the time it takes for subsequent requests of the same type of analysis. The reason is that in Trace Compass the state system database is specific to each type of analysis and thus the first run also includes the state system construction. The subsequent requests only query the already built database. Therefore, we distinguish the first and subsequent requests to see how the model behaves in different use cases. We measured 10 independent executions for each case and computed the average and standard deviation. We then divided the running time of the single Trace Compass server (TS) by the running time of our model (TC) to obtain the gain factor (TS/TC). With the first CPU Usage tree request, the measured standard deviation stood around 2 to 7% of the mean. For the rest of the scenarios, it is relatively low, ranging between 1-4%, with most of the measured running times falling within the first standard deviation. As mentionned above, the first CPU Usage tree request involves memory and disk I/O operations for reading traces data and writing the state system. As we execute it multiple times on the same machine, there should be disk management operations that took place in the background during its execution, especially in the case of the single instance Trace Server, where it almost saturates the disk space, which explains its higher measured time variation. There are no measurements for the single instance Trace Compass server at 102 GB traces size, because the available disk space on each machine is capped at 530 GB. Analysing 102

GB of traces per analysis server means that the single server must handle 408 GB of traces data, plus the generated state system written to disk that takes about the same size as the traces data itself. The results are shown in Figure 4.12.

			Total size of analyzed traces (GB)									
			1.3 x 4	2.6 x 4	6.8 x 4	11 x 4	15 x 4	23 x 4	31 x 4	45 x 4	74 x 4	102 x 4
Trace Coordinator (TC) requests response times (ms) (4 nodes analyzing 1 trace each + coordinator)	Indexing time (ms)		80489	162635	422289	713417	969896	1473459	2037469	2933247	4055898	6042829
	CPU Usage tree	1st request	23520	52474	127987	206746	299833	445441	594097	1103247	1753030	2547295
		subsequent	42,8	60,6	141,4	169,22	248,8	308,1	447,7	602	833	1105
	CPU Usage XY	1st request	9437	13984	36093	44085	58807	89099	119064	168995	234793	347217
		subsequent	9205	13655	35425	43955	58145	87720	118723	167819	230413	331361
	Statistics (single pass)	1st request	43,8	42,1	61,8	50,9	46,5	61	69,7	80,4	85,7	94,1
		subsequent	40	36,9	47,8	42,3	44	45,1	55,2	46,8	50,6	54,7
	Statistics (multi-pass)	1st request	41,6	41,9	55,7	45,2	43,1	55,4	60,2	65,6	67,8	72,2
		subsequent	39,1	36,4	47	40,5	39,6	42,3	49,6	43,7	45	46,6
Trace Server (TS) requests response times (ms) (1 analysis node analyzing 4 traces alone)	Indexing time (ms)		320017	674629	1773194	2778007	3850587	5953175	7987124	11976823	16329527	-
	CPU Usage tree	1st request	47911	98951	257378	402834	572926	886971	1287284	2093781	3396128	-
		subsequent	36,4	55	161,6	166,5	254,1	303,2	471,4	656,1	924	-
	CPU Usage XY	1st request	8932	13357	30311	37175	52837	79761	97623	147872	228375	-
		subsequent	8321	12606	26937	35051	46052	76374	96247	130768	178510	-
	Statistics (single pass)	1st request	43,8	42,4	57,3	48,4	50,2	56,7	75,4	83,1	89	-
		subsequent	22	12,5	17,9	16,3	14,6	15,1	13,9	18,2	17	-
TS / TC (gain)	Indexing time (ms)		3,976	4,148	4,199	3,894	3,970	4,040	3,920	4,083	4,026	-
	CPU Usage tree	1st request	2,037	1,886	2,011	1,948	1,911	1,991	2,167	1,898	1,937	-
		subsequent	0,850	0,908	1,143	0,984	1,021	0,984	1,053	1,090	1,109	-
	CPU Usage XY	1st request	0,946	0,955	0,840	0,843	0,898	0,895	0,820	0,875	0,973	-
		subsequent	0,904	0,923	0,760	0,797	0,792	0,871	0,811	0,779	0,775	-
	Statistics (single pass)	1st request	1,000	1,007	0,927	0,951	1,080	0,930	1,082	1,034	1,039	-
		subsequent	0,550	0,339	0,374	0,385	0,332	0,335	0,252	0,389	0,336	-

Indexing time:	Time to read the trace + compute the state system for the default analysis (Linux kernel, TID, Callsite, Statistics analysis)
CPU usage tree	Response time for getting the tree of visible entries of the CPU usage analysis
CPU usage XY	Response time for getting the XY model of the CPU usage analysis
statistics (single pass)	Statistics analysis response times when using the single pass approach
Statistics (multi-pass)	Statistics analysis response times when using the multi-pass approach

Figure 4.12 Coordinator and Trace server response times based on traces total size

We recall from subsection 4.3.3, equation (2) that, for $T(n)$ the running time of an analysis of a single Trace Compass server, the running time of the same operation in our model is roughly

$$T(n/\alpha) + kC(n) + c \text{ with } c > 0 \quad (6)$$

With α the number of analysis nodes and k the number of steps. For the CPU Usage process tree, the results show that the gain from having multiple Trace Compass servers is most

significant for the first run, and less so for subsequent analysis requests. This is coherent because, as the running time $T(n)$ of the first analysis execution includes the state system construction, which is significantly bigger than the time it takes to only query the database, $T(n/\alpha)$ is also very big compared to $kC(n)$, thus the overhead in this case is very small compared to the gain we have from dividing the state system construction workload to multiple analysis nodes. For subsequent analyses, $T(n)$ is essentially the time used by Trace Compass server to put data points into the TSP array format, since querying the state system database is very efficient once it is built. Therefore, $T(n/\alpha)$ is in the same order of magnitude as $C(n)$, thus the overhead from $C(n)$ becomes much more important and eats up the performance gain.

The running time for querying XY chart data is similar between the first and subsequent requests, because the state system is already built after the first request that retrieves the CPU process tree. Thus, the first execution does not involve the state system construction anymore.

From the results, we can see that after the first request, our model ran slower than a single Trace Compass server for the same workload. This can be explained by the fact that with simple aggregations, or small queries, the workload is small and easily supported by a single Trace Compass analysis server. The gain from parallelizing simple requests is thus quickly offset by the latencies at the coordinator level. Most notable are the network latencies, the time for parsing the JSON data, as well as the time required time to perform the aggregation. This is characterized by a small slowdown, instead of a gain, for the requests made after the construction of the state system, in the case of the "CPU usage" analysis or the Statistical analysis, as shown by our results in Figure 4.12.

It is important to keep in mind that the main advantages of parallelizing Trace Compass are in the indexing phase as well as the construction of state systems. For those, the processing time is essentially divided by the number of analysis nodes, and therefore 4 times faster in the case of our experiments. We should also mention that the gains in terms of "execution time" represent only one part of the improvements brought by our work. Another major advantage consists in the availability of a greater quantity of storage and memory RAM resources thanks, to the combination of the resources of different analysis nodes. The impact of this was clearly shown in our experiments, in the last column of Table 4.12, where the single analysis node reached its saturation point with a set of 4 times 102 GB traces, while the coordinator was able to process the analysis on that same amount of data with no difficulties.

In figure 4.13, the time span of the aggregation function is on the right, after the coordinator fetches the responses from the analysis nodes. Figure 4.13 and 4.14 show the proportion

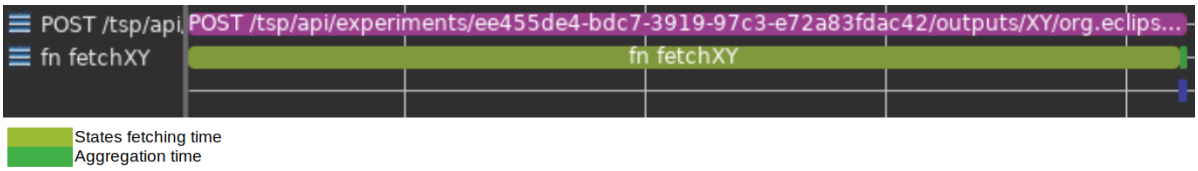

Figure 4.13 Coordinator trace during CPU usage analysis

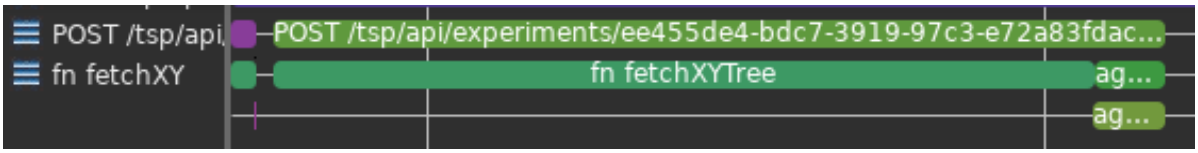

Figure 4.14 Coordinator trace during CPU usage analysis

between the aggregation operation and the time the coordinator takes to fetch the raw responses from the analysis servers. Clearly, the running time of the aggregation is reasonable compared to the total running time of each request in the coordinator. It is the function that fetches responses from the analysis servers that takes up much of the running time.

The case for Statistics analysis confirms the advantage of the multi-pass approach. It constantly performs better than the single-pass approach. As mentioned in Section 4.3.3, when running as single-pass, the coordinator iterates through the list of events to recompute its percentage based on the total number of events from all analysis nodes. Meanwhile, with multi-pass, the coordinator simply sums up the total number of event in the first pass, which runs in $\mathcal{O}(1)$. The percentage is then computed in parallel by the analysis servers directly while formatting the data. The second aggregation is then only a matter of concatenating arrays, hence the model achieves a better gain. For the subsequent requests, our model is slower than the single instance, despite the multi-pass approach. This is because in our test cases, the Statistics is a relatively lightweight analysis to run, and it is even quicker once the state system is built. Thus the gain of multiple analysis servers cannot compensate the overhead from network latency and passing through the coordinator.

We also established the running time bound for $C(n)$ in subsection 4.3.3, equation (5).

$$C(n) \leq \frac{\alpha-1}{k} T(n/\alpha) - c \quad (5)$$

If we chose an aggregation algorithm with a complexity order bigger than $T(n/\alpha)$ which is mostly linear in our prototype, the running time of $C(n)$ will quickly outgrow the constant factor $\frac{\alpha-1}{k}$ and eat up any gain that we have. As mentioned in subsection 4.3.3, the running time of our operations depends on the size of the result, which in turn depends on the request

from the client, as shown in Subsection 4.3.3. Indeed, the client can change the resolution of the XY chart through the timestamp parameter, the more timestamp intervals are provided, the more data points are returned. To see the effect of the aggregation algorithm on the performance, we ran the benchmark on the second CPU Usage Analysis that retrieves the XY chart data with a traces size of 6.8 GB, using an aggregation algorithm that is intentionally made $\mathcal{O}(n^2)$ (For every element in the array of data, we iterate through the entire array). We used two different numbers of timestamp parameter, one with 2550 intervals and the other with only 1275 intervals (same as benchmark). The result is shown in figure 4.15.

			Number of requested times intervals	
			1275 intervals	2550 intervals
Trace Coordinator (TC) response time (ms) (4 nodes each analyzing one 6,8 GB trace)	**CPU Usage tree**	1st request	125383	119983
		subsequent	140,1	138,5
	CPU Usage XY	1st request	35540	96244
		subsequent	35283	93792
Trace Server (TS) response time (ms) (1 node analyzing alone 4 traces of 6,8 GB each)	**CPU Usage tree**	1st request	257378	233170
		subsequent	161,6	163
	CPU Usage XY	1st request	30311	40957
		subsequent	26937	39690
TS / TC (gain)	**CPU Usage tree**	1st request	2,053	1,943
		subsequent	1,153	1,177
	CPU Usage XY	1st request	0,853	0,426
		subsequent	0,763	0,423

Figure 4.15 Coordinator and Trace server response based on requested times number when using quadratic aggregation

With this aggregation algorithm, we see the parallizing gain offset by the aggregation cost for the XY chart data. If we combine its running time with the process tree request (since both requests are often made together), the first analysis execution, where the state system construction takes much longer to finish than the aggregation, we still have some running time gain. For subsequent requests, the performance is reduced, because of $C(n)$ growing in quadratic order. As we increase the number of timestamp parameters (thus the TSP result size), the overhead of $C(n)$ also increases drastically.

The objective of the following paragraphs is to shed light on the distribution of the time between the different processes and operations that run before the coordinator returns the final result of the aggregation. As we can see in figure 4.16 below, which depicts the trace data of a coordinator during its execution, the time is split between :

— The processing time of the trace-server : this time corresponds to the time for a trace-compass server node to process a request received from the coordinator, and since this time could vary between the different nodes, we take the longest duration, because it is the one that decides when the coordinator can start aggregating.
— Aggregation time : it is the time needed for the coordinator to gather all the responses received from the different Trace Compass server nodes into a single global view.
— JSON parsing : this is the time needed to convert a server response into JSON format.
— JSON Stringify : this is the time required to convert JSON data into String format.
— Network time and latency : corresponds to the time required to exchange packets (sending/receiving) between the coordinator and its various Trace Compass server nodes.

On the figure 4.17 below, we show the average distribution of the times mentioned above, based on some requests made with the coordinator on the states of CPU usage analysis. As we can see, the average processing time for the trace compass servers is around 11.5%, while the aggregation time is less than 1%, which confirms that the aggregation overhead is very low. Those results show also where most of the time is consumed, between the JSON parsing that represents around 35%, as well as the time it takes to send the data on the network which is greater than 49%. This confirms our explanation concerning the weak gains obtained with simple queries.

From these results, we can say that significant gains could be obtained through a more efficient encoding than JSON, or by parallelizing on multiple threads JSON encoding and decoding. It is also important to remember that the network overhead we are seeing here is already partly compensated by the fact that we do not need to transmit traces data from multiple nodes to a single trace server.

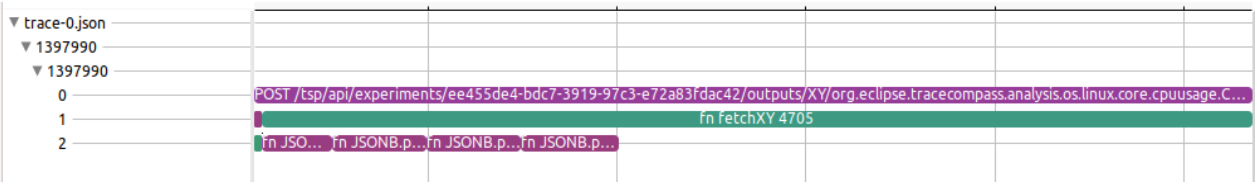

Figure 4.16 A coordinator trace showing the time distribution

As we saw, the overhead mainly comes from the network latency and the JSON parsing time of the TSP responses coming from the servers. This overhead depends mainly on the size of the responses returned by the server. They are especially important in the case of the requests where the response size is significant compared to the time it takes the server to

Operations	Times (ms)	Percentage From request total time (%)
Aggregation	79	0,75
JSON parsing	3765	35,68
JSON Sringify	285	2,70
Trace server Time	1213,59	11,50
Network time & latency	5210,40	49,37
Request total time	10553,00	100,00 %

Figure 4.17 Coordinator request time distribution

process them. This is the case for the CPU Usage XY chart data, for example, where the size of the response is approximately 35 MO.

CPU Usage XY execution time with 12 GB traces per trace server	3 loops	6 loops	9 loops
Trace Coordinator (TC) (ms)	22067	23886	24414
Trace Server (TS) (ms)	29591	47215	61482
TS / TC (gain)	1.341	1.977	2.518

Figure 4.18 Coordinator and Trace server response times with complex algorithm simulation server-side

It should be noted that the bandwidth of the computers used as analysis servers for our tests is limited to 65 MO/s (1Gbit/s network adapters). This is a significant limitation when each analysis node must submit a 35 MB response to the coordinator. That being said, even with a very limited bandwidth, our model will still benefit applications in the cases where complex algorithms with long execution times are required, provided that the response size remains fairly constant (e.g. determined by the chart size and resolution). Since the size does not change, the overhead remains the same, which means that its proportion of the total time is reduced when the total execution time increases. This was evident with the state system construction that is costly and saw an almost perfect gain close to 4 for 4 parallel servers.

We simulated a complex operation by looping over the results on each analysis server, before they are actually sent over the network. This simulates the behaviour of a complex algorithm. We increased the number of loop iterations gradually to see the effect of this added complexity

on the gain factor. As in the benchmark, we ran the test with a model of four analysis servers and one coordinator, in order to compare the results afterwards to a single instance analysis server that handles the same workload. The trace size used during this test is of approximately 12 GB per server. The result is shown in Figure 4.18.

As we can see from the table, the gain factor increases proportionally with the number of loop iterations. This implies a greater CPU time that the single analysis trace server must handle alone. However, in the case of the coordinator, we have at our disposal a greater amount of CPU resources that we exploit, thanks to the parallel architecture of the coordinator, reducing the processing time and consequently increasing the gain. On the coordinator and 4 parallel servers, we can see that the JSON parsing time remains the same, but its proportion of the total time becomes smaller.

4.4.3 Limitations

The benchmark for statistics analysis is for the purpose of demonstrating the advantage of the multi-pass approach and does not reflect a realistic use case. Notably, in the 2-pass execution, the second aggregation in our test only concatenates the result arrays together. Similar event types are typically used across traces from different analysis nodes. Statistics from equivalent event types should be aggregated together, similar to how processes are aggregated in the CPU Usage analysis example. However, as previously mentioned, matching equivalent processes and event types is a separate topic, out of the scope of this article.

NodeJS is not an ideal choice for a production version of the coordinator node. The aggregation often consists of intensive manipulations on big arrays of data. These operations could often be multithreaded, which is difficult to achieve in NodeJS.

As mentioned earlier, it is possible to have an organization with multiple levels of coordination and aggregation. This could be interesting when the number of analysis nodes becomes extremely large. It was not investigated in this article.

4.5 Conclusion and Future Work

In this paper, we proposed a distributed tracing architecture that is suitable for various HPC and heterogeneous systems tracing. The platform is inspired by existing scalable frameworks and uses Trace Compass server as analysis server to provide flexibility and extensibility to tracing applications. It offers the capability of running aggregation operations on the results from the server side, similar to how Prometheus supports operations through its query language PromQL. Trace Compass however provides more sophisticated analyses, hence requires

more sophisticated aggregation operators. Thus, we introduced the analysis automata paradigm to facilitate the development of new aggregation algorithms, and to make trace analysis algorithms easier to integrate with the multi-pass approach.

To evaluate our proposition, we discussed the running time analysis of the model and how it behaves with different numbers of analysis nodes. We also established a running time "budget" for the aggregation algorithm, based on those variables. We then implemented a prototype of our proposition, using Trace Compass as the analysis server, and measured its performance against a single Trace Compass server instance in different scenarios. The results show that our proposition works well in a distributed environment. They also showed that beyond a certain trace size, a single trace server node reaches saturation due to insufficient storage and RAM memory resources. However, by using the Trace coordinator combined with several trace server nodes, the overall amount of storage and available RAM memory is much greater, thus allowing to handle very large traces. This represents a major contribution to the scalability of Trace Compass. This provides a proof of concept of how Trace Compass can work in a distributed/parallel environment, and an understanding of its advantages and shortcomings.

Future work will refine the multi-pass approach further with more use cases, for example to allow powerful combinations of aggregation algorithms and efficiently reusing the intermediate results across different operations. It will also be interesting to study organizations with multiple levels of coordination and aggregation, as mentioned earlier, to achieve an even better scalability.

CHAPITRE 5 DISCUSSION GÉNÉRALE

5.1 Compatibilité

L'avantage d'utiliser TSP comme protocole de communication pour le coordonnateur est que les clients peuvent communiquer avec le coordonnateur de la même manière qu'ils communiquent avec un serveur de Trace Compass. Le client n'a donc pas besoin de connaître la nature exacte du composant avec lequel il communique, tant que c'est un composant qui est capable de comprendre les requêtes TSP. Une interface de Trace Compass dans Theia IDE, par exemple, est parfaitement compatible avec notre modèle.

En outre, on peut faire en sorte que le client puisse changer de "serveur" d'analyse dynamiquement. Il pourra donc passer du coordonnateur vers un des serveurs de Trace Compass utilisés pour l'analyse et vice versa. Cela permet à l'utilisateur par exemple d'étudier la vue globale (agrégée) d'une analyse à partir du coordonnateur puis, une fois qu'il détecte un problème sur une des machines tracées, il pourra communiquer directement avec le serveur de Trace Compass qui s'est chargé de traiter la trace, sans passer par le coordonnateur. Ceci lui permettra d'avoir une vue d'analyse (non agrégée) avec seulement les informations qui correspondent à la machine problématique.

5.2 Extensibilité

Cette architecture permet d'avoir une extensibilité en termes de nombre de serveurs d'analyse et aussi en terme des opérations d'agrégation. Une augmentation de nombre de serveurs d'analyse implique une augmentation d'usage de mémoire du coordonnateur. Cependant, l'usage de mémoire du coordonnateur est due principalement aux phases d'agrégation, d'où le seuil supérieur est la limitation d'affichage du client (l'interface graphique). C'est-à-dire qu'il est probablement inutile d'envoyer plus l'informatique que ce que l'interface est capable d'afficher (la résolution maximale de l'écran, etc.). Tandis que l'usage de mémoire pour la construction de systèmes d'état est proportionnel à la taille des traces. L'extensibilité de l'architecture vient du fait qu'on a distribué la surcharge de travail de cout variable en ajoutant une charge de travail de cout "fixe".

L'architecture du coordonnateur facilite grandement l'ajout des nouvelles opérations d'agrégation. La séparation des requêtes, la collection des réponses et l'agrégation des résultats sont bien séparées en différents modules, avec la classe *Coordinator* qui prend le rôle d'un tuyau pour garantir que le flux de données est conforme à ce que demande chaque module.

Autrement dit, il existe un "protocole" interne, ce qui est une extension de TSP, utilisé pour la communication entre les agrégateurs et la collection des réponses des serveurs d'analyses. L'ajout d'un nouvel opérateur d'agrégation nécessite seulement l'implémentation d'une fonction qui est capable de traiter les données dans le format de ce "protocole", aucune modification du code source dans le reste du coordonnateur est nécessaire. Le coordonnateur utilise le nom d'analyse et le nom d'opération comme chemin pour trouver la fonction d'agrégation appropriée. Avec cette architecture du coordonnateur, la combinaison des opérateurs est simplement un "currying" des fonctions (l'exécution des fonctions l'une après l'autre, la sortie d'une fonction est l'entrée de la suite). Éventuellement, l'implémentation des agrégateurs est inspirée par le patron "middleware", qui est utilisé pour permettre la personnalisation de transformation de données dans les cadriciels JavaScript de gestion de données comme "Redux" ou les cadriciels de serveur HTTP (Fastify, Express, etc.). Puisque les effets des agrégateurs sont isolés du reste du coordonnateur (les agrégateurs n'ont pas d'effet secondaire) et son protocole est bien défini, on pourrait penser à supporter sécuritairement la création dynamique des opérateurs à travers des scripts de l'utilisateur. Le script peut être un sous-ensemble d'un langage populaire comme Python ou JavaScript et ensuite exécuté dans un environnement isolé, la seule partie du système exposée est ce qui est à agréger. D'ailleurs, ce même principe pourrait aussi être appliqué pour les serveurs d'analyse pour supporter dynamiquement les opérateurs d'agrégation local.

5.3 Identification des processus similaires

Comme mentionné dans les parties précédentes, l'agrégation des résultats de certaines analyses nécessite une identification des processus similaires à travers la grappe de calcul et notre travail n'inclut pas une solution pour ce problème, nous avons présumé que l'identifiant créé par le serveur Trace Compass est l'identifiant unique du processus, ce qui n'est pas le cas. Il faut remarquer qu'une solution générique n'est pas triviale, mais dans plusieurs cas on pourrait retomber sur une approche plus simple. Notamment, le développeur du programme peut directement donner un nom spécifique au processus, le coordonnateur ensuite utilise ce nom comme l'identifiant unique. L'intervention de l'utilisateur est aussi une piste potentielle, comme ce qui a été conçu pour permettre aux utilisateurs de choisir les événements de réseau pour la synchronisation de trace dans Trace Compass. Le nombre de choix est cependant beaucoup trop grand dans le cas d'une grappe de calcul. Pour aider les utilisateurs, plusieurs stratégies pourraient être appliquées pour donner un indicateur utilisé pour le classement des processus en termes de ses similarités.

5.4 Protocole d'interrogation

Nous avons considéré les langages d'interrogation comme PromQL ou GraphQL pour le protocole de communication. Dans la majorité des cas, les données de l'analyse occupent une grande partie dans la réponse retournée par le serveur Trace Compass sous la forme d'un tableau de séries de temps. Ces valeurs ne sont pas identifiables par une clé, le mécanisme de "schéma" de GraphQL n'est donc pas une approche appropriée pour filtrer nos données. De plus, puisque le reste de l'information dans ces réponses a une taille relativement petite comparée aux données de l'analyse, utiliser GraphQL pour filtrer ces informations, même dans des cas nécessaires, ne vaut pas le coût de la complexité ajoutée par la librairie.

Un langage de requête comme PromQL est beaucoup plus convenable. Pourtant, c'est une solution surpuissante en ce moment pour notre modèle. Avec PromQL, en plus de pouvoir préciser les intervalles souhaités pour récupérer les données, nous sommes capables également de faire des combinaisons d'opérations très puissantes, directement sur ces données. Or, nous ne possédons pas encore beaucoup d'opérateurs pour bénéficier de cette capacité. Pour spécifier de simples opérateurs, avec de petites extensions, TSP est sémantiquement capable de couvrir notre besoin. Il serait tout de même intéressant de considérer PromQL dans le futur, lorsque nous aurons besoin d'opérateurs plus complexes, car il pourrait être, à ce moment-là, un candidat intéressant pour notre cas d'usage.

5.5 Retour sur les résultats

5.5.1 Mesure du temps d'exécution

Dans certains cas, notre modèle s'est exécuté plus lentement qu'un seul serveur de Trace Compass, pour une même quantité de travail, pour les requêtes après la première exécution. Ceci s'explique par le fait, qu'avec des agrégations très simples, ou bien de petites requêtes, il est normal qu'il y ait peu ou pas de gain. En effet, il s'agirait d'une petite charge de travail qu'un serveur d'analyse de Trace Compass peut indivituellement prendre en charge. Cela signifie que le gain qu'on pourrait avoir en parallélisant les requêtes simples, sera vite réduit par les latences que nous pouvons avoir au niveau du coordonnateur, notamment les latences du réseau, le temps écoulé durant l'extraction des données JSON, ainsi que le temps nécessaire pour effectuer l'agrégation. Cela se caractérise par un ralentissement au niveau des requêtes simples effectuées après la construction du système d'état. Ces requêtes incluent le cas de l'analyse "CPU usage" ou encore l'analyse Statistique, comme l'ont montré nos résultats de la figure 4.12.

Il est important de rappeler que les avantages principaux de la parallélisation de Trace Compass se trouvent dans la phase de l'indexage ainsi que la construction des systèmes d'états. Dans ce cas, le temps de traitement est pratiquement divisé par le nombre de nœuds d'analyse, et donc 4 fois plus rapide dans le cas des expériences que nous avons effectuées avec 4 nœuds. Il est à noter également que les gains en temps d'exécution ne représentent qu'une seule partie des améliorations qu'apporte notre travail, car un autre avantage majeur consiste en la disponibilité d'une plus grande quantité de ressources de stockage et de mémoire RAM, grâce à la combinaison des ressources des différents nœuds d'analyse. Cela a été clairement montré dans nos expériences à la dernière colonne du tableau de la figure 4.12, où un seul nœud d'analyse avait atteint sa saturation avec une quantité de traces équivalentes à 4 fois la trace de 102 Go, alors que le coordonnateur a pu effectuer la même analyse sans difficulté. Ainsi, tant que la construction des fichiers d'index et du système d'état sont effectués à avec un bon gain (la partie la plus coûteuse en termes de CPU, de mémoire RAM et de stockage), avoir un léger ralentissement sur les requêtes envoyées au système d'état (qui sont à la base très rapide) n'est pas un grand inconvénient.

5.6 Comparaison avec les solutions existantes

L'idée d'un composant qui s'occupe de l'envoi des traces est inspiré de Jaeger. Du côté de l'analyse, notre proposition ressemble plus à l'architecture actuelle de Prometheus, car Jaeger ne définit pas de manière distincte le composant d'analyse de trace. Il utilise Cassandra pour sauvegarder directement les traces de manière distribuées et donne l'accès à cette base de données au composant de vue. Il y a quelques procédures de Spark qui exécutent le post-traitement sur ces données, mais globalement l'analyse de trace de Jaeger est très simple. Le nœud d'analyse dans notre modèle joue un rôle assez similaire au serveur Prometheus. Il sauvegarde les événements sous forme d'un tableau de séries de temps, une sorte de base de données similaire au système d'état de Trace Compass. Cette base de données n'est pas distribuée comme celle de Jaeger mais est sauvegardée localement sur le nœud d'analyse et exposée par le serveur à travers PromQL. Ceci est semblable à la méthode avec laquelle nous exposons le résultat d'analyse avec le protocole TSP. Cependant, avec plusieurs instances d'analyse, Prometheus prévoit la possibilité d'avoir deux serveurs qui communiquent directement entre eux en mode "cross-service", lors de la connexion du client sur le premier serveur, le deuxième se comporte comme un supporteur qui fournit ses événements au premier serveur. Le premier serveur analyse ensuite les deux collections de séries de temps. Nous pouvons couvrir un cas d'usage similaire dans notre modèle en imaginant le coordonnateur comme une couche qui réside directement sur le nœud d'analyse. Le mode Prometheus "hiérarchie"

avec un "coordonnateur" central est aussi différent de celui de notre proposition. Le "coordonnateur" Prometheus ne fait pas une agrégation pour chaque requête, mais une agrégation globale sur la base de données des serveurs Prometheus de niveau plus bas. Le résultat est ensuite sauvegardé lui-même comme une base de données dans le "coordonnateur" et aussi exposé à travers PromQL. Pour une analogie, ce serait comme si notre coordonnateur faisait une agrégation sur le système d'état au lieu de la faire sur les réponses des requêtes de l'analyse.

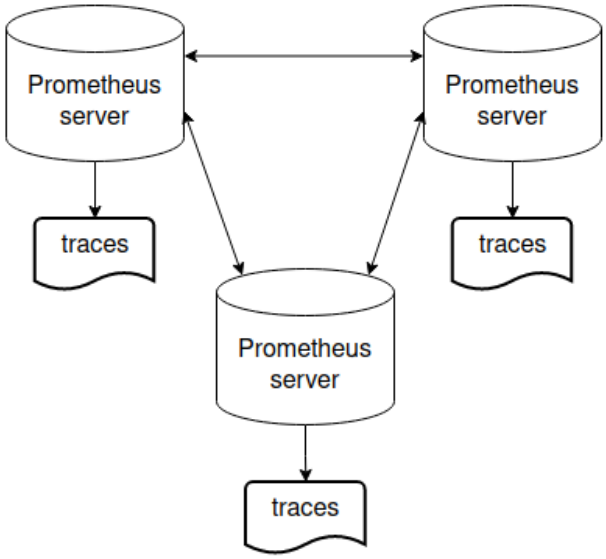

Figure 5.1 Les instances Prometheus en fédération "cross-service"

Il est important de noter que Jaeger et Prometheus sauvegardent les événements presque "brut" en base de données. C'est pourquoi ces plateformes sont fortement couplées avec le format de trace qu'elles supportent. Les requêtes faites à la base de données sont aussi fortement couplées avec le format de trace. De plus, "l'analyse" au sens de "Trace Compass" est éventuellement réalisée dans le côté client de ces outils. Le fait d'avoir les analyses exécutées du côté serveur permet à Trace Compass d'avoir le système d'état comme une couche d'abstraction du format de trace. Grâce à cela, notre proposition pourrait être adoptée pour plus de types de système.

CHAPITRE 6 CONCLUSION

Ce chapitre conclut le mémoire. Nous commençons d'abord par présenter une synthèse de nos travaux de recherche. Nous discutons ensuite des limitations de notre proposition et enfin les améliorations possibles à effectuer dans le futur.

6.1 Synthèse des travaux

La proposition d'une nouvelle architecture pour l'analyse distribuée et parallèle de traces, le modèle d'exécution pour les opérations d'agrégation d'analyse de trace, et l'évaluation de sa performance et de son efficacité sont les principales contributions de ce mémoire.

Pendant la durée de ce projet de recherche, nous avons étudié en détail plusieurs outils d'analyse de trace existants. L'outil Trace Compass offre des analyses très sophistiquées et aussi supporte de manière flexible plusieurs formats de trace différents. Il est aussi facile pour les développeurs de rajouter le support pour un nouveau type de trace ou une nouvelle analyse. Nous avons ensuite étudié le comportement et la performance de Trace Compass en l'utilisant dans le contexte de traces de très gros volume. Nous avons défini l'objectif de la recherche qui consiste à proposer une architecture permettant d'utiliser Trace Compass pour analyser les traces des systèmes distribués et parallèles. Par la suite, nous avons vérifié l'état de l'art des outils dans le domaine du traçage distribué. Les outils comme Jaeger et Prometheus ont été développés récemment pour répondre à ce besoin, mais ils possèdent des limitations qui rendent leur adaptation parfois difficile, notamment le fort couplage entre les différents aspects dans son écosystème. Cependant, leur architecture est bien utilisée dans les environnements distribués. C'est pourquoi nous nous en sommes inspirés pour proposer une architecture avec plusieurs instances de serveur de Trace Compass et un coordonnateur central, qui communiquent entre eux à travers le réseau. La proposition initiale consiste simplement en un niveau de serveurs Trace Compass parallèles, et un coordonnateur qui regroupe les résultats des analyses.

Prometheus permet aux utilisateurs d'effectuer des opérations sur les données récupérées, par exemple pour agréger une vue d'ensemble sur les traces de grande taille. Inspirés par ce concept, nous proposons d'utiliser le coordonnateur pour effectuer des agrégations sur le résultat d'analyses lorsqu'on les regroupe ensemble. Nous avons ensuite regardé les protocoles de communication dans les outils de traçage pour en choisir un qui serait adéquat pour notre cas d'usage. Nous avons aussi défini un modèle d'exécution et de programmation pour

faciliter la séparation des analyses de trace et ses agrégations en plusieurs étapes. Ceci permet d'accélérer leur exécution dans certains cas d'usage. En ayant un modèle assez complet, avec les composants nécessaires, nous avons fait des analyses théoriques sur l'impact des opérations d'agrégation et de séparation de requêtes sur le temps d'exécution du programme. À travers cette analyse, nous avons pu aussi définir un budget pour les opérations d'agrégation en fonction du temps d'exécution original d'une analyse. Cela aide au processus de choisir un algorithme d'agrégation adéquat.

Finalement, nous avons implémenté un prototype basé sur notre proposition. L'implémentation est conçue comme un petit cadriciel qui permet d'ajouter facilement les autres types d'agrégation. Les mesures de temps de performance ont ensuite été menées pour évaluer notre solution. Nous avons étudié les résultats obtenus et analysé les cas qui diffèrent de ce qui est décrit en théorie. Ce travail est une preuve de concept de l'utilisation de Trace Compass de manière distribuée à grande échelle. Nous concluons que nous avons atteint les objectifs de recherche présentés au chapitre 1.

6.2 Limitations de la solution proposée et améliorations futures

Pendant notre recherche, en essayant d'appliquer l'agrégation sur certains types d'analyse, nous avons constaté que l'agrégation effectuée sur le résultat de TSP est peut-être une solution non-optimale. Tel que mentionné dans les sections précédentes, le TSP expose le modèle de vue. C'est avantageux lorsque ces données sont directement utilisées pour l'affichage. Cependant, pour effectuer une agrégation, ce modèle de vue peut manquer un peu de sémantique de l'analyse qu'il présente. En faisant l'agrégation sur le modèle de vue, nous limitons peut-être les possibilités. De plus, le format de données qui est pratique pour l'affichage peut être inefficace pour l'agrégation. On peut imaginer comme alternative d'effectuer les agrégations directement sur les données du système d'état, avant qu'elles ne soient transformées en format TSP. Cela nécessiterait d'exposer le système d'état sur le réseau, pour que le coordonnateur puisse faire des requêtes à distance. La couche DataProvider de Trace Compass est pourtant très avantageuse. Dans ce cas, on pourrait penser à déplacer cette couche dans un service à part, qui serait responsable uniquement de convertir les données en format TSP pour le client final, ou encore elle pourrait être intégrée dans le coordonnateur. Néanmoins, cette approche ajoutera potentiellement beaucoup de complexité. En exposant le système d'état au lieu du modèle de vue, la communication entre le coordonnateur et le serveur Trace Compass nécessitera un protocole autre que le TSP. Cela implique, d'un point de vue du client final, un changement considérable dans le rôle du coordonnateur ainsi que du serveur Trace Compass, parce que le client ne pourra plus vraiment communiquer directement avec un serveur Trace

Compass. Une solution possible serait de distribuer la couche DataProvider comme une librairie du côté client, similaire au module d'analyse de Perfetto. Cela nécessiterait tout de même un effort pour supporter la librairie en plusieurs langages de programmation. Ceci dit, les algorithmes d'agrégation dépendent beaucoup du type d'analyse et du besoin de l'utilisateur. Il est donc important de déterminer des cas d'usage concrets, qui seront utilisés en réalité, pour évaluer si ces gros changements rapporteraient réellement un bénéfice.

Il y a un autre détail important qui n'est pas vraiment investigué dans ce mémoire concernant l'agrégation des données d'analyse de trace. Les analyses sophistiquées de Trace Compass relèvent souvent de nombreux détails en ce qui concerne les processus et/ou les flux d'exécution. Imaginons un système hétérogène qui exécute une tâche en parallèle et nous souhaitons analyser l'usage de ressources de cette tâche d'un point de vue global du système. Trace Compass peut donner ces détails sur un processus à partir de la trace du système, mais comment peut-on corréler ces processus ensemble, de plusieurs machines différentes, pour calculer sa moyenne ? Ce n'est pas viable de forcer un même PID ou TID pour tous les processus qui sont les instances parallèles d'un même programme. Puisque l'arbre des processus est aussi potentiellement différent, dû aux branchements du programme exécuté, on ne peut pas se baser sur le processus parent de manière fiable pour ce but non plus. Que fait-on si ce n'est pas un système hétérogène et les traces sont complètement indépendantes ? Il existe plusieurs techniques pour déterminer les processus similaires dans la trace, grâce à une combinaison d'informations, mais le sujet de comparaison de trace est un autre projet de recherche. Nous supposons dans ce projet que, dans ces cas, les processus similaires peuvent être identifiés avec l'identifiant fourni dans la réponse TSP, ce qui n'est pas encore le cas en réalité au moment de ce travail.

RÉFÉRENCES

[1] Google. Perfetto documentation. [En ligne]. Disponible : https://perfetto.dev/docs

[2] Y. Shkuro. (2017) Uber engineering blog, evolving distributed tracing at uber engineering. [En ligne]. Disponible : https://eng.uber.com/distributed-tracing/

[3] Prometheus Authors. (2014) Prometheus documentation. [En ligne]. Disponible : https://prometheus.io/docs/introduction/overview/

[4] M. Desnoyers et M. Dagenais, "Os tracing for hardware, driver and binary reverse engineering in linux," *CodeBreakers Journal*, vol. 1, nº. 2, 2006.

[5] B. Gregg. (2014) strace wow much syscal. [En ligne]. Disponible : https://www.brendangregg.com/blog/2014-05-11/strace-wow-much-syscall.html

[6] D. Toupin, "Using tracing to diagnose or monitor systems," *IEEE Software*, vol. 28, nº. 1, p. 87 – 91, 2011, debug ;Look-forward ;Monitor system ;Multicore technology ;Open sources ;Trace ;. [En ligne]. Disponible : http://dx.doi.org/10.1109/MS.2011.20

[7] P.-M. Fournier et M. R. Dagenais, "Analyzing blocking to debug performance problems on multi-core systems," vol. 44, nº. 2, 2010, p. 77 – 87, kernel instrumentation ;Multi-core systems ;New approaches ;Parallel software ;Performance bugs ;Performance problems ;Production system ;Total elapsed time ;. [En ligne]. Disponible : http://dx.doi.org/10.1145/1773912.1773932

[8] H. Daoud et M. R. Dagenais, "Performance analysis of distributed storage clusters based on kernel and userspace traces," *Software - Practice and Experience*, vol. 51, nº. 1, p. 5 – 24, 2021, distributed storage ;Distributed storage system ;Efficient data structures ;Performance analysis ;Performance issues ;Performance problems ;Process scheduling ;Real-world scenario ;. [En ligne]. Disponible : http://dx.doi.org/10.1002/spe.2889

[9] R. Fahem, "Points de trace statiques et dynamiques en mode noyau," mémoire de maîtrise, Dép. de génie informatique, École Polytechnique de Montréal, Montréal, QC, 2012. [En ligne]. Disponible : https://publications.polymtl.ca/849/

[10] 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.

[11] V. Prasad *et al.*, "Locating system problems using dynamic instrumentation," dans *2005 Ottawa Linux Symposium*. Citeseer, 2005, p. 49–64.

[12] T. Bird, “Measuring function duration with ftrace,” dans *Proceedings of the Linux Symposium*, vol. 1. Citeseer, 2009.

[13] F. Wininger, N. Ezzati-Jivan et M. R. Dagenais, “A declarative framework for stateful analysis of execution traces,” *Software Quality Journal*, vol. 25, nº. 1, p. 201 – 229, 2017, analysis frameworks ;Application execution ;Application problems ;Declarative debugging ;Execution trace analysis ;Multi-core systems ;Sample applications ;Software debugging ;. [En ligne]. Disponible : http://dx.doi.org/10.1007/s11219-016-9311-0

[14] L. Prieur-Drevon, R. Beamonte et M. 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, dynamic analysis tools ;External memory ;Improved structures ;Orders of magnitude ;Stateful analysis ;Trace visualization ;Tracing ;Tree ;. [En ligne]. Disponible : http://dx.doi.org/10.1016/j.jss.2017.09.023

[15] Ericsson. Trace compass developer guide. [En ligne]. Disponible : https://archive.eclipse.org/tracecompass/doc/stable/org.eclipse.tracecompass.doc.dev/Generic-State-System.html#ITmfStateProvider_.2F_AbstractTmfStateProvider

[16] Y. Chen Kuang Piao, “Nouvelle architecture pour les environnements de développement intégré et traçage de logiciel,” mémoire de maîtrise, Dép. de génie informatique, École Polytechnique de Montréal, Montréal, QC, 2018. [En ligne]. Disponible : https://publications.polymtl.ca/3282/

[17] F. Reumont-Locke, “Méthodes efficaces de parallélisation de l’analyse de traces noyau,” mémoire de maîtrise, Dép. de génie informatique, École Polytechnique de Montréal, Montréal, QC, 2015. [En ligne]. Disponible : https://publications.polymtl.ca/1899/

[18] M. Martin, “Analyse détaillée de trace en dépit d’événements manquants,” mémoire de maîtrise, Dép. de génie informatique, École Polytechnique de Montréal, Montréal, QC, 2018. [En ligne]. Disponible : https://publications.polymtl.ca/3248/

[19] D. Drews, “Implementing a mobile app for object detection,” mémoire de maîtrise, TECHNICAL UNIVERSITY OF MUNICH, 2021. [En ligne]. Disponible : https://mediatum.ub.tum.de/1633375

[20] N. Matloff, “Programming on parallel machines,” *University of California, Davis*, 2011.

[21] Y. Shkuro, *Mastering Distributed Tracing : Analyzing performance in microservices and complex systems.* Packt Publishing Ltd, 2019, p. 64–65.

[22] B. Burns, *Designing Distributed Systems : Patterns and Paradigms for Scalable, Reliable Services.* " O’Reilly Media, Inc.", 2018, p. 41–45.

[23] ——, *Designing Distributed Systems : Patterns and Paradigms for Scalable, Reliable Services.* " O'Reilly Media, Inc.", 2018, p. 45–49.

[24] ——, *Designing Distributed Systems : Patterns and Paradigms for Scalable, Reliable Services.* " O'Reilly Media, Inc.", 2018, p. 59–80.

[25] ——, *Designing Distributed Systems : Patterns and Paradigms for Scalable, Reliable Services.* " O'Reilly Media, Inc.", 2018, p. 93–106.

[26] Y. Shkuro. (2019) Jaegertracing, jaeger and opentelemetry. [En ligne]. Disponible : https://medium.com/jaegertracing/jaeger-and-opentelemetry-1846f701d9f2

[27] D. A. Tamburri, M. Miglierina et E. D. Nitto, "Cloud applications monitoring : An industrial study," *Information and Software Technology*, vol. 127, 2020, cloud applications ;Empirical research ;Industrial adoption ;Industrial monitoring ;IT infrastructures ;Level of automations ;Monitoring strategy ;Monitoring technologies ;. [En ligne]. Disponible : http://dx.doi.org/10.1016/j.infsof.2020.106376

[28] J. Renita et N. E. Elizabeth, "Network's server monitoring and analysis using nagios," vol. 2018-January, Chennai, India, 2017, p. 1904 – 1909, application monitoring ;Memory consumption ;Nagios ;Network administrator ;Network Monitoring ;Open source tools ;Real-time statistics ;Server monitoring ;. [En ligne]. Disponible : http://dx.doi.org/10.1109/WiSPNET.2017.8300092

[29] A. Ciuffoletti, "Beyond nagios design of a cloud monitoring system," vol. 2, Rome, Italy, 2016, p. 363 – 370, on demands ;Open cloud computing interfaces ;Resource monitoring ;REST Paradigm ;Websocket ;. [En ligne]. Disponible : http://dx.doi.org/10.5220/0005778303630370

[30] Nagios. (2019) Monitoring architecture solutions for large organizations. [En ligne]. Disponible : https://assets.nagios.com/downloads/general/docs/Monitoring_Architecture_Solutions_For_Large_Organizations.pdf

[31] N. Sukhija *et al.*, "Event management and monitoring framework for hpc environments using servicenow and prometheus," Virtual, Online, United arab emirates, 2020, p. 149 – 156, computational components ;Heterogeneous systems ;High performance computing ;Lawrence Berkeley National Laboratory ;Monitoring frameworks ;Operational needs ;Operations management ;Proactive Monitoring ;. [En ligne]. Disponible : http://dx.doi.org/10.1145/3415958.3433046

[32] G. Reback. (2021) How to build a scalable prometheus architecture. [En ligne]. Disponible : https://logz.io/blog/devops/prometheus-architecture-at-scale/

[33] H. Na *et al.*, "Hpc software tracking strategies for a diverse workload," Virtual, Atlanta, GA, United states, 2020, p. 1 – 9, batch jobs ;Computing

software ;Data pool ;Key elements ;License servers ;Performance computing ;Server logs ;Tracking strategies ;Usage data ;User communities ;. [En ligne]. Disponible : http://dx.doi.org/10.1109/HUSTProtools51951.2020.00008

[34] S. K. Dewangan, S. Pandey et T. Verma, "A distributed framework for event log analysis using mapreduce," Ramanathapuram, Tamil Nadu, India, 2016, p. 503 – 506, customer behavior analysis ;Distributed framework ;Effective analysis ;Evaluation results ;Hadoop ;Log file ;Map-reduce ;Sessionization ;. [En ligne]. Disponible : http://dx.doi.org/10.1109/ICACCCT.2016.7831690

[35] V. Bhosale *et al.*, "Hadoop in action : Building a generic log analyzing system," Pune, India, 2018, analyzing system ;Data processing applications ;Distributed processing ;Hadoop ;Log analysis ;Open source frameworks ;Traditional systems ;Unstructured data ;. [En ligne]. Disponible : http://dx.doi.org/10.1109/ICCUBEA.2018.8697687

[36] X. Li *et al.*, "A parallel host log analysis approach based on spark," Hangzhou, China, 2018, p. 301 – 305, abnormal behavior ;Cloud environments ;Computing capability ;Data dimensions ;Host-based intrusion detection system ;Large data volumes ;Log analysis ;Rapid growth ;. [En ligne]. Disponible : http://dx.doi.org/10.1109/CIS2018.2018.00073

[37] S. Blanas *et al.*, "A comparison of join algorithms for log processing in mapreduce," Indianapolis, IN, United states, 2010, p. 975 – 986, analytics ;Experimental comparison ;hadoop ;Join processing ;Join strategies ;Map-reduce ;Mapreduce frameworks ;Reference data ;. [En ligne]. Disponible : http://dx.doi.org/10.1145/1807167.1807273

[38] B. Debnath *et al.*, "Loglens : A real-time log analysis system," vol. 2018-July, Vienna, Austria, 2018, p. 1052 – 1062, event-driven ;Industrial environments ;Log analysis ;Log Parsing ;Log Sequence Violation ;Operational problems ;Search capabilities ;Unsupervised machine learning ;. [En ligne]. Disponible : http://dx.doi.org/10.1109/ICDCS.2018.00105

[39] A. C. Turcato *et al.*, "Introducing a cloud based architecture for the distributed analysis of real-time ethernet traffic," Roma, Italy, 2020, p. 235 – 240, cloud-based architectures ;Diagnostic systems ;Distributed analysis ;Industrial communication protocols ;Industrial networks ;Maintenance tools ;Real time Ethernet ;Traffic sampling ;. [En ligne]. Disponible : http://dx.doi.org/10.1109/MetroInd4.0IoT48571.2020.9138288

[40] I. M. de Diego *et al.*, "Scalable and flexible wireless distributed architecture for intelligent video surveillance systems," *Multimedia Tools and Applications*, vol. 78, n°. 13, p. 17 437 – 17 459, 2019, distributed systems ;Intelligent surveillance systems ;Intelligent video surveillance ;Intelligent video surveillance systems ;Robot operating system ;Robot operating systems (ROS) ;Wireless multimedia sensor

network ;Wireless multimedia sensor network (WMSNS) ;. [En ligne]. Disponible : http://dx.doi.org/10.1007/s11042-018-7065-3

[41] Ericsson. Trace server protocol. [En ligne]. Disponible : https://eclipse-cdt-cloud.github.io/trace-server-protocol/

[42] ——. Trace server protocol openapi specification. [En ligne]. Disponible : https://raw.githubusercontent.com/theia-ide/trace-server-protocol/master/API.yaml

[43] M. Khouzam. (2021) Investigate the use of bigint or as string in the tsp. [En ligne]. Disponible : https://github.com/theia-ide/theia-trace-extension/issues/41#issuecomment-942471842

[44] P. Maréchal. (2021) Support bigint values. [En ligne]. Disponible : https://github.com/theia-ide/tsp-typescript-client/pull/37

[45] Q.-H. Tran. (2021) when-json-met-bigint. [En ligne]. Disponible : https://github.com/haoadoresorange/when-json-met-bigint

[46] S. L. Vadlamani *et al.*, "Can graphql replace rest ? a study of their efficiency and viability," Virtual, Online, 2021, p. 10 – 17, architectural style ;Facebook ;ITS applications ;Its efficiencies ;Open sources ;Practical experience ;Representational state transfer ;Software developer ;. [En ligne]. Disponible : http://dx.doi.org/10.1109/SER-IP52554.2021.00009

[47] Facebook. Graphql. [En ligne]. Disponible : https://graphql.org/

[48] A. Brito *et al.*, "Why and how java developers break apis," vol. 2018-March, Campobasso, Italy, 2018, p. 255 – 265, aPI Evolution ;Breaking Changes ;Field studies ;Java developers ;Java library ;Software systems ;. [En ligne]. Disponible : http://dx.doi.org/10.1109/SANER.2018.8330214

[49] L. Xavier *et al.*, "Historical and impact analysis of api breaking changes : A large-scale study," Klagenfurt, Austria, 2017, p. 138 – 147, aPI Usage ;Backwards Compatibility ;Behavioral changes ;Client applications ;Higher frequencies ;Large-scale analysis ;Large-scale studies ;Software Evolution ;. [En ligne]. Disponible : http://dx.doi.org/10.1109/SANER.2017.7884616

[50] M. Amundsen. (2021) Api versioning : What is it and why is it so hard ? [En ligne]. Disponible : https://blog.container-solutions.com/api-versioning-what-is-it-why-so-hard

[51] G. Brito, T. Mombach et M. T. Valente, "Migrating to graphql : A practical assessment," Hangzhou, China, 2019, p. 140 – 150, aPIs ;GraphQL ;Grey literature ;In-depth understanding ;Key characteristics ;Migration Study ;REST ;Web based ;. [En ligne]. Disponible : http://dx.doi.org/10.1109/SANER.2019.8667986

[52] M. Cronin. (2019) What is the n+1 problem in graphql ? [En ligne]. Disponible : https://medium.com/the-marcy-lab-school/what-is-the-n-1-problem-in-graphql-dd4921cb3c1a

[53] J. Sayago Heredia, E. Flores-Garcia et A. R. Solano, "Comparative analysis between standards oriented to web services : Soap, rest and graphql," vol. 1193 CCIS, Quito, Ecuador, 2020, p. 286 – 300, application development ;Appropriate technologies ;Comparative analysis ;GraphQL ;Java ;REST ;Systematic mapping ;Transfer capability ;. [En ligne]. Disponible : http://dx.doi.org/10.1007/978-3-030-42517-3_22

[54] D. A. Hartina, A. Lawi et B. L. E. Panggabean, "Performance analysis of graphql and restful in sim lp2m of the hasanuddin university," Makassar, Indonesia, 2018, p. 237 – 240, community services ;GraphQL ;Performance analysis ;Performance calculation ;Performance parameters ;RESTful ;WEB application ;Web information services ;. [En ligne]. Disponible : http://dx.doi.org/10.1109/EIConCIT.2018.8878524

[55] E. Lee, K. Kwon et J. Yun, "Performance measurement of graphql api in home ess data server," vol. 2020-October, Jeju Island, Korea, Republic of, 2020, p. 1929 – 1931, data server ;Energy storage systems ;Management applications ;Mobile data ;Performance measurements ;Restful api ;. [En ligne]. Disponible : http://dx.doi.org/10.1109/ICTC49870.2020.9289569

[56] Y. Guo, F. Deng et X. Yang, "Design and implementation of real-time management system architecture based on graphql," vol. 466, n°. 1, Nanjing, China, 2018, design and implementations ;Management systems ;Performance problems ;Proposed architectures ;Real time management systems ;Real time monitoring system ;Real time requirement ;Web service architecture ;. [En ligne]. Disponible : http://dx.doi.org/10.1088/1757-899X/466/1/012015

[57] Cloud Native Computing Foundation. (2015) The open tracing project. [En ligne]. Disponible : https://opentracing.io/

[58] Prometheus. Prometheus query language. [En ligne]. Disponible : https://prometheus.io/docs/prometheus/latest/querying/basics/

[59] Google. Perfetto trace processor documentation. [En ligne]. Disponible : https://perfetto.dev/docs/analysis/trace-processor

ANNEXE A DÉPLOIEMENT DES SERVEURS DE TRAÇAGE AVEC UN COORDONNATEUR

Pour pouvoir déployer ce projet, il faut compiler le serveur Trace Compass avec le code source trouvé au lien `https://github.com/trace-coordinator/`

Avant de compiler le projet, assurez-vous d'avoir la version 11 de Java installée et l'outil Maven pour la compilation.

Téléchargez les codes sources de Trace Compass et Trace Compass Incubator en exécutant les instructions suivantes :

git clone https ://github.com/trace-coordinator/org.eclipse.tracecompass.git

git clone https ://github.com/trace-coordinator/org.eclipse.tracecompass.incubator.git

Ces deux répertoires contiennent le patch nécessaire pour fonctionner avec le coordonnateur. Maintenant, compilez le serveur Trace Compass en exécutant

cd org.eclipse.tracecompass && \

mvn clean install -DskipTests=true && \

cd ../org.eclipse.tracecompass.incubator && \

mvn clean install -DskipTests=true

Une fois la compilation terminée, vous allez trouver le binaire de serveur de Trace Compass sous le répertoire `org.eclipse.tracecompass.incubator/trace-server/org.eclipse.trace\compass.incubator.trace.server.product/target/products/traceserver/linux/gtk/x86_64/trace-compass-server`. Vous pouvez maintenant lancer le serveur Trace Compass dans les machines souhaitées, veuillez référer à la documentation de Trace Compass pour la configuration des portes TCP.

Ensuite, téléchargez le code source du module trace-coordinator en exécutant l'instruction suivante

git clone https ://github.com/trace-coordinator/trace-coordinator.git

Assurez-vous avoir au minimum NodeJS version 14.18.12, l'outil "yarn" doit être aussi installé pour la compilation de trace-coordinator. Sous le répertoire du code source, vous pouvez en-

suite lancer la commande *yarn && yarn build* pour compiler le projet. Une fois la compilation terminée, vous pouvez lancer le trace-coordinator avec la commande *yarn start*.

Vous devez configurer les adresses IPs des noeuds de serveur Trace Compass. La configuration se trouve dans le fichier *package.json*, sous la section "trace-coordinator" (Exemple en 8)

```
{
  "trace-coordinator": {
        "port": 8080,
        "trace-servers": [
            {
                "url": "http://localhost:8081"
            }
        ]
    }
}
```

Listing 8 Configuration de trace-coordinator

Vous pouvez aussi configurer le port TCP sur lequel le module trace-coordinator écoute. Une fois lancé, vous pouvez faire des requêtes sur le module trace-coordinator à l'adresse `http://0.0.0.0:$PORT`. Veuillez vous référer à la documentation de Trace Server Protocol pour le format exact des requêtes.

```
{
  "parameters": {
    "uris": [
      "/path/to/traces1",
      "/path/to/trace2",
    ],
  }
}
```

Listing 9 Paramètre pour la requête /dev/createExperimentsFromTraces

Le trace-coordinator supporte aussi un endpoint expérimental au `/tsp/api/dev/create\ExperimentsFromTraces` qui permet de créer un "experiment" à partir des traces sur les serveurs Trace Compass. Il suffit d'appeler cet URL avec une opération POST et un paramètre

qui contient le chemin du répertoire de trace sur les machines de serveur de Trace Compass. Éventuellement, les traces doivent déjà se trouver localement sur les serveurs (sur le même chemin dans tous les serveurs de Trace Compass).

Le script utilisé pour mesurer le temps d'exécution de trace-coordinator se trouve aussi dans ce projet, en `src/benchmark.ts`.