マイクロサービス環境のためのマルチレイヤトレーシングの設計と実装

Yoshitani, Reina / 吉谷, 玲奈

(出版者 / Publisher)
法政大学大学院情報科学研究科
(雑誌名 / Journal or Publication Title)
法政大学大学院紀要. 情報科学研究科編
(巻 / Volume)
17
(開始ページ / Start Page)
1
(終了ページ / End Page)
6
(発行年 / Year)
2022-03-24
(URL)
https://doi.org/10.15002/00025266

マイクロサービス環境のためのマルチレイヤトレーシングの設計と実装
Design and implementation of multi-layer tracing for microservices

吉谷 玲奈
Reina Yoshitani
法政大学大学院情報科学研究科情報科学専攻

Abstract

Microservice architecture is an architectural design of the cloud services, that is already used in the major service providers, such as Amazon and Netflix. Using this pattern, a conventional monolithic service is divided into multiple microservices, then each microservice runs in the container that is light-weight virtualized technology. However, since a single system is provided by many microservices, it is difficult to understand the dependencies among microservices and to monitor traffic latency and error rates. When an abnormality occurs in a system with insufficient monitoring, it takes time to identify the cause of the abnormality and the service that is the bottleneck. To solve these problems, distributed tracing technology is used. This facilitates the analysis of increasingly complex applications. However, existing tracing technologies are unable to determine whether a large delay is actually occurring at the application layer, in the queue of the communication protocol stack, at the node driver level, or in the relay network. Therefore, in order to be able to identify system failures in more detail, it is desirable to be able to trace requests not only at the conventional application layer, but also at other layers. In this study, we propose multi-layer tracing, which enables analysis including transport and network layers.

1 序論

近年、Amazon や Netflix のサービスで見られるように、大規模な Web サービスにおけるマイクロサービスアーキテクチャは広く使われるようになっている。マイクロサービスアーキテクチャは軽量な仮想化技術を用いて運用されることが一般的で、個々のマイクロサービスをコンテナとして稼働させ、サービス同士の連携によって 1 つのサービスを提供する。この方式を用いることでサービスごとに割り当てる CPU やメモリの性能を変更したり、一部のコンポーネントのみを更新したりすることで開発や運用が容易になることから、従来のモノリシックアーキテクチャよりも変化に強い柔軟性に富んだサービスを実現することができる。しかし、多数のマイクロサービスで構成することは、マイクロサービス間の依存関係を把握などの可観測性の向上が困難となった。

マイクロサービス間の依存関係を把握するために分散トレーシングの技術が用いられている。分散トレーシングではマイクロサービスのようにアプリケーションが複数動作し、複雑化したサービスのリクエスト全体の流れの分析が容易になる。特に、マイクロサービスアーキテクチャにおいてマイクロサービス間の通信や経路を実環境の変動に対して透過的に制御するサービスメッシュでは、アプリケーションレイヤでの分散トレーシングを実現している。この通信制御のための中間層において、通信の遅延やエラー率といったメトリクス情報も取得することができるため、サービスコンポーネントの実行状況の把握に有効である。しかし、サービスメッシュのレベルで収集可能な情報では、実際に大きな遅延が発生した場合に、それがアプリケーションレイヤで発生しているのか、通信のプロトコルスタックのキューで発生しているのか、ノードのドライバレベルで発生しているのか、中継ネットワークで発生しているのかを判別することができない。

そこで、本研究では複数のレイヤにまたがる統合された分析を可能とするマルチレイヤトレーシングを提案する。トレーシングするレイヤとしてトランスポートレイヤ、仮想ネットワークレイヤ、物理ネットワークレイヤ、ネットワーク機器が存在するネットワークレイヤが挙げられる。各レイヤにおいてリクエストパケット通過時のタイミングとそのパケットがどのサービスにおけるリクエストであるか判別することでマルチレイヤトレーシングを実装する。これらのレイヤを統合的にトレーシングすることで、トレーシング対象サービス遅延時の各レイヤにおける処理時間や処理タイミングを記録することが可能になる。この時間を判別可能とすることで、サービスにおいて発生している遅延がどのレイヤが原因であるかを判別することが容易となり、問題解決のアプローチをスムーズに行うことが可能になる。これにより、問題解決までの合計時間も短縮され、サービス全体のパフォーマンスを向上させることに繋がる。

2 関連研究

Suo らはクラウドシステムのサービス間を接続する仮想化ネットワークを観測すること目的として vNetTracer を開発した [1]。vNetTracer は eBPF (extended Berkeley Packet Filter) を用いることにより、軽量かつプログラマブルにトレーシングを実現しており、　TCP 通信の場合は TCP ヘッダーのオプション部分にユニーク ID を追加し、UDP 通信の場合は UDP ヘッダーにユニーク ID を追加する。この ID を使用してネットワークレイヤにおける個々のアプリケーションを区別することを実現した。

Tkashi らはマイクロサービスの品質確保のため、eBPF を用いて作成したインフラのメトリクスを追跡するリアルタイムモニタリングシステムを提案した [2] 。このシステムはコンテナオーケストレーターからのイベントと連動し、動的にフルスタックトポロジーを作成し、メトリクス追跡に必要なセンサを再配置を行う。このシステムの優れた点としては従来の Application Performance Management (APM) ソフトウェアで測定されるサービスレベルのレイテンシからネットワークインフラのパケット遅延を抽出できる点にある。これにより、サービス遅延増加の原因をアプリケーションレイヤであるかネットワークレイヤであるかの切り分けを実現した。

本研究では既存の APM にアプリケーションレイヤにスイッチなどのネットワーク機器が存在するネットワークレイヤやトランスポートレイヤにおけるトレーシングも追加した実装をす

Supervisor: Prof. Toshio Hirotsu

る。これによってサービスの遅延発生時に原因がどのレイヤに存在するかの切り分けを容易にし、サービス全体のパケット追跡のためのマルチレイヤトレーシングが実現する。

3 eBPF

eBPF は Berkeley Packet Filter (BPF) と呼ばれるパケットフィルタの技術を拡張したものであり、Linux システムにおける汎用的な監視ツールである。ユーザー空間でユーザーが自由に作成したプログラムをカーネル空間に送り込み、独自の命令セットを持つカーネル内部の仮想マシン上で実行することができる。そのため、カーネルのソースコードを変更したり、モジュールを追加したりすることなくプログラマーの実行したい処理をすることができる。プログラムを実行するフックポイントはカーネル内やユーザー空間のアプリケーションの関数呼び出し時やパケットの NIC 到着時など多岐に及ぶ。

BPF のバイトコードは C 言語で記述でき、LLVM/Clang コンパイラによって生成される。このとき、BPF プログラムではカーネル内にハードコーディングされているヘルパー関数のみを呼び出すことができ、任意のカーネル関数を呼び出すことはできないよう制限されている。生成されたバイトコードはユーザー空間から bpf システムコールによってカーネルに渡され、BPF の検証器を使用して BPF バイトコードがカーネルをクラッシュさせることなく安全に実行可能かどうかを検証する。この検証結果が問題ない場合、Just in Time (JIT) コンパイラによってマシンコードに変換される。BPF は利用場面に応じて BPF プログラムが可能な操作や呼び出し可能な外部関数が異なる。挿し込みたい処理ごとにアタッチする BPF プログラムタイプを変更することで様々な処理をカーネル内に挿し込むことが可能になる。

4 分散トレーシング

分散トレーシング技術とはマイクロサービスアーキテクチャで構築されたサービス品質を管理するために開発された APM の一種である。マイクロサービスアーキテクチャでは 1 つ 1 つのマイクロサービスが個別のコンテナ上でデプロイされることで 1 つの大きなサービスを提供している。個々のマイクロサービスは相互に通信しながら独立して動作するため、特定のマイクロサービスに対してのみ機能を修正することができる。近年ではこの拡張性の高さや修正に対するコストの低さからマイクロサービスアーキテクチャが増々普及してきている。しかし、マイクロサービスが増加するほどネットワーク通信が発生し、システム全体のパフォーマンスが低下したり、通信のエラー発生が原因のサービス遅延が発生したりする。特に、大規模なサービスであれば数千のマシン上で動作している多量のプロセスが相互に通信し、サービスの依存関係の把握やエラー発生箇所を把握することが困難である。そのため、可観測性を高めることが非常に重要な要素である。可観測性とはシステムの状態やステータスを外部からどれくらい理解できるかを測定したものである。マイクロサービスアーキテクチャにおいて可観測性を高めるため、サービスメッシュを利用することで従来の観測データに加えてトランザクションを観測可能にするためのトレースデータを収集する。このトレースデータを用いることでアプリケーションレイヤでのトレーシングを可能にする。

近年、アプリケーションレイヤにおける分散トレーシングのアプローチとして Annotation-based schemes が主流である。Annotation-based schemes では各サービスに分散トレーシング用のメタデータを付与し、下流サービスへ伝播させることで特定のリクエストに対する分析を可能とする。実際によく利用されるトレーシングツールには Dapper[3], Zipkin[4], Jaeger[5] などがある。このアプローチにおける分散トレーシングの実装ではユーザーからのリクエストを受け付ける最初のポイントにてリクエスト毎にトレース ID というリクエスト全体を表す一意の文字列を作成する。そして、HTTP ヘッダーにトレース ID を追加し、下流のサービスへ伝播させる。このようにトレース ID と紐付けられた一連のログのまとまりはトレースと呼ばれ、アプリケーションがリクエストの処理時間とリクエストのステータスを追跡するために使用される。トレースは 1 つ以上のスパンと呼ばれる特定の期間における分散システムの論理的な作業単位から構成されている。そして、Jaeger などのトレーシングツールでは各トレース内で一意の文字列となるスパン ID をトレース ID などと共に伝播させることによって各サービス同士の親子関係を表し、サービス間の依存関係や各サービスの処理時間を表すことができる。

5 マルチレイヤトレーシング

スパンによって表される各サービスの処理時間はサービスのリクエスト開始時刻からレスポンス返却時刻の差であるため、サービスの処理時間だけではなくネットワークの通信にかかる時間やドライバのキュー待ち時間なども含まれている。これではサービス全体からどのサービスで遅延が発生しているかは判別できるが、原因の詳細を特定できない。この問題を解決するために分散トレーシングにおいてアプリケーションレイヤとトランスポートレイヤ、物理・仮想ネットワークレイヤなど、マルチレイヤにトレーシングできることが望まれる。サービス間が通信する際、通過するインターフェースやスイッチなどのネットワーク機器に関してもトレーシングを実施することでサービスに発生した遅延の原因がどのレイヤで存在するかといった原因切り分けが容易になる。

マルチレイヤにトレーシングするためには各レイヤを通過するパケットがどのようなリクエストと関係するパケットであるか判別できる必要がある。しかし、アプリケーションレイヤ以外のレイヤではアプリケーションレイヤほど自由な処理ができず、アプリケーションレイヤで追加・利用されているトレース ID をそのまま利用することができない。そのため、他レイヤにおいて扱いが容易な領域でトレース ID を扱う必要がある。ネットワークレイヤで機能を拡張するためには TCP のオプションヘッダーが準備されている。TCP オプションヘッダーは TCP の通信において、性能を向上させるために準備されている可変長な領域である。この TCP オプションヘッダーはネットワークレイヤでの扱いが容易であり、RFC 6994 以降で実験的なオプションをサポートしているため、ユーザーレベルで任意の値を操作するためには最適な領域であるといえる。そのため、アプリケーションレイヤにて付与されたトレース ID やトレース ID との関連付けが可能なユニークな値等を TCP オプションヘッダーに埋め込むことでネットワークレイヤでもアプリケーションレイヤと同等のトレーシングができる。また、このような TCP オプションヘッダーの処理はトランスポートレイヤにおける処理であるため、この追加処理実行時の処理時刻も合わせて関連付けることでトランスポートレイヤのトレーシングも実現可能となる。これにより、アプリケーションレイヤとネットワークレイヤ、トランスポートレイヤにおけるマルチレイヤトレーシングが実現できると考える。

従来のアプリケーションレイヤにおけるトレーシング結果のモニタリングは各分散トレーシングツールがサポートしている

Web UI 上で行われていることが多い。そのため、このトレーシングツールへ各インターフェースやネットワーク機器において抽出した値をトレーシングツールが処理できるようにして送信することでアプリケーションレイヤで行われていたトレーシングと一緒にネットワークレイヤのトレーシングも実施できる。これにより、従来のトレーシングに加えてネットワークレイヤも含めたマルチレイヤトレーシングの結果を Web UI 上で確認することができる。

6 設計

この節ではマルチレイヤトレーシング実現のために必要な設計について述べる。具体的には TCP レイヤにおいてユニークな ID の埋め込み、その埋め込み処理の時刻を記録することでトランスポートレイヤにおけるトレーシングを実現する。仮想・物理ネットワークではトレース ID の抽出と TCP レイヤにおいて埋め込まれた値の抽出を同時に行い、アプリケーションレイヤとトランスポートレイヤ、仮想・物理ネットワークレイヤにおけるトレーシングを関連付ける。そして、トレーシングの確認方法であるモニタリング手法について述べる。

6.1 トランスポートレイヤ

ネットワーク機器などの制限が多いマシン上やトランスポートレイヤのトレーシングをアプリケーションレイヤと共に統合して実現するためにトランスポートレイヤへ値の埋め込みを行う。本研究ではネットワークの拡張を容易にするために提供されている TCP オプションヘッダーを利用する。TCP オプションヘッダーへ任意の値を追加や更新を行う標準的な手法は追加したい任意の TCP オプションを定義することである。この処理はカーネル内部で行われる処理であるため、拡張のためにはカーネルのリビルドが必要となる。しかし、カーネルのリビルドは処理が重く、汎用性が高くない上に、サービスが動作するホスト環境のカーネルが固定されてしまい、任意の環境で動作できるというマイクロサービスアーキテクチャの利点も失われる。そのため、本研究ではカーネルをリビルドせずに TCP オプションヘッダーを拡張するために eBPF を用いる。

eBPF を用いることでプログラマブルにカーネル内部の処理を行うことができるが、トランスポートレイヤにおける eBPF プログラムはアクセス可能なメモリ領域が限られており、HTTP ヘッダーのあるメモリ領域をロードすることができない。そこで、TCP オプションヘッダーでは重複しないランダムな値を追加し、別レイヤにおいてこのランダムな値とトレース ID を関連付けるよう設計する。このとき、ランダム値を埋め込む処理を実行する時刻も記録することでトランスポートレイヤにおいてもトレーシングを実現できるようになる。

6.2 データリンクレイヤ

マルチレイヤトレーシングを実現するためには各レイヤを通過するパケットがどのトレースに含まれているリクエストであるかを判別できる必要がある。そのためにはアプリケーションレイヤでサービスの区別に利用しているトレース ID をアプリケーションレイヤ以外においても利用可能にし、各レイヤそれぞれが処理にかかる時間をそのリクエストで発生するトレース ID と関連付けて集計する必要がある。現在主流の分散トレーシングでは HTTP ヘッダーにトレース ID などのトレースコンテキストを付与することでトレーシングを行っている。HTTP ヘッダー自体は可変長であり、GET 文や HTTP バージョンなどの定形文を除いてはアプリケーション次第で自由に着脱可能なものである。そのため、HTTP ヘッダーのどのあたりにトレースコンテキストが存在しているかということは予め判別可能な情報ではない。

パケットの処理もカーネル内部で行われる処理であるため、カーネルリビルド等の問題を避けるために eBPF を利用する。検出において利用する eBPF は仮想・物理ネットワークレイヤで動作させることでパケット全体のメモリにアクセスできるため、TCP オプションヘッダーの値も HTTP ヘッダーの値も読み取り可能である。そこで、仮想・物理ネットワークレイヤにおける抽出処理において 6.1 節で埋め込まれた TCP オプションヘッダーのランダム値とトレース ID の関連付ける。

ただし、このように関連付けるためには HTTP ヘッダーを探索し、トレース ID を検出する必要がある。特に、トレース ID が後方にある場合や HTTP ヘッダーが非常に大きい場合は探索できる可能性が低い。また、読み取れたとしても常に可変長探索をすることはコストが高く、パフォーマンスの低下を招く。そのため、この探索量を減少させるためにトレース ID を前方に付与するようにアプリケーションレイヤにおいて処理する。これらの作業から、アプリケーションレイヤ、トランスポートレイヤ、仮想・物理ネットワークレイヤのマルチレイヤトレーシングを実現できる。

6.3 モニタリング

アプリケーションレイヤで動作しているトレーシングツールにて、ネットワークレイヤで動作しているインターフェースやネットワーク機器などのマシンから送信されたトレースコンテキストと対象パケットの通過時刻情報も共に処理することでモニタリングをする。一般的なトレーシングツールではトレーシングをモニタリングするために Web UI をサポートしている。そのため、リクエストパケットから抽出したトレース ID とスパン ID、インターフェースを通過する時刻を 1 つのスパンとしてトレーシングツールへ送信することで、サービス間のトレーシングに各インターフェースやネットワーク機器の通過状況などのスパンが追加されることとなり、Web UI 上でマルチレイヤトレーシングを確認することができる。

7 実装

本節ではまず、トランスポートレイヤにおいて TCP オプションヘッダーに値を追加する方法について説明する。次にアプリケーションレイヤにおけるトレースコンテキストの移動方法と抽出方法について説明し、最後に仮想・物理インターフェースにおいて取得したトレース ID とスパン ID をトレーシングツールに送信方法について述べ、最後にモニタリングする方法について説明する。

7.1 トランスポートレイヤ

TCP オプションヘッダーに関する eBPF 追加機能を用いるためにアタッチする BPF プログラムタイプは BPF_PROG_TYPE_SOCK_OPS であり、カスタム TCP ヘッダーオプションなどを含むカーネル TCP スタックの動作を調整可能にする。TCP オプションは 1 バイト のオプション番号、1 バイトのオプション全体の長さと値の 3 要素から成り立つ。TCP ヘッダーの最大サイズは 60 バイトであり、送信元ポート番号などの通信において決められた値は 20 バイト存在するため、TCP オプションヘッダーの最大バイト数は 40 バイトとなる。そのため、値部分には 40 バイト分から元々の通信において追加されているタイムスタンプの値等の領域を差し引いた分を最大で割り当てることができる。本研究では初めにランダム値を記録するために 4 バイト分の値領域を持つ TCP

オプションを定義し、この拡張処理時刻を記録するため 8 バイト分の値領域を持つ TCP オプションも定義する。次にポート番号などの情報を利用してトレーシング対象パケットを制限し、最後に追加処理を行う。

追加処理はサービスから別サービスへアクティブな接続が確立時に行う。オプションを追加する前に追加したいオプション領域として bpf_reserve_hdr_opt 関数を用いて 6 バイトと 10 バイト分の領域を確保する。そして、bpf_get_prandom_u32 関数を用いて 4 バイトのランダムな値を生成し、bpf_ktime_get_ns 関数を利用してこの追加処理を行う時刻を取得する。取得したこれらの値を bpf_store_hdr_opt を用いて TCP オプションとして追加する。

7.2 データリンクレイヤ

本研究では Kubernetes を実行基盤とし、サービスメッシュに Istio を使用しているシステムを対象として実装を行った。Istio ではトレースコンテキストをデータプレーンに存在する Envoy Proxy を利用して着脱している。ネットワークレイヤにおいてトレース ID を検出するために BPF Compiler Collection (BCC) と Python を用いて作成する。この検出ツールはフィルタリング機能を提供する BPF_PROG_TYPE_SOCKET_FILTER にアタッチし、HTTP ヘッダーの存在を調べ、HTTP ヘッダーの先頭からトレース ID を探す。ヘッダーの構造はヘッダー名: 具体的な値 + 改行コードとなっている。また、抽出したいトレース ID のヘッダー名はトレーシングツールごとに固有のものであるため、改行コードの次の値が対応するヘッダー名であればトレースコンテキストであると判定でき、load_byte 関数を利用してトレース ID を検出できる。このとき、7.1 節にて追加した TCP オプションを関連付けるため、トレース ID の検出時に TCP オプションを同時に検出する必要がある。検出したトレース ID 等は eBPF プログラム間やユーザー・カーネルスペースのプログラム間でのデータのやりとりに利用可能な map 上に格納する。作成した eBPF プログラムを Python からロードし、任意のインターフェースにアタッチすることでそのインターフェースを通過するすべてのパケットをフィルタリングし、トレース ID を検出することができる。このとき、トレース ID の位置が判別できているため、スパン ID の位置も同様に判別でき、検出できるようになっている。

このように検出処理を行う場合、必ず HTTP ヘッダーの可変長探索が発生する。HTTP ヘッダーはアプリケーションが自由にカスタマイズできるため、トレース ID 等が後方に存在する可能性がある。そのため、HTTP ヘッダーの内部情報を移動させるために Envoy Proxy のソースコードに手を加える。Envoy Proxy では付与するヘッダーをリストで管理している。新しいヘッダーを追加する際にはこのリストに push_back をする。そのため、トレースコンテキストを前方に移動させるためには付与するタイミングでリストの先頭にトレースコンテキストを付与することで HTTP ヘッダーの前方にトレースコンテキストを移動させることができる (図 1)。

トレースコンテキストをリストに追加する処理は injectContext 関数が呼び出されることで行われる。injectContext 関数は Envoy Proxy がサポートしている分散トレーシングツールの中からアプリケーションが伝播したいトレースコンテキストを使用するツールの管理する injectContext 関数にコールバックし、setByReferenceKey 関数を呼び出すことでヘッダーと値を紐付けてリストに追加する。setByReferenceKey

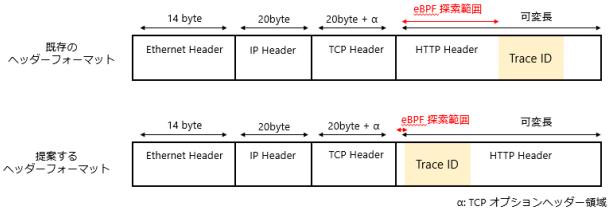

図 1　Header format の比較

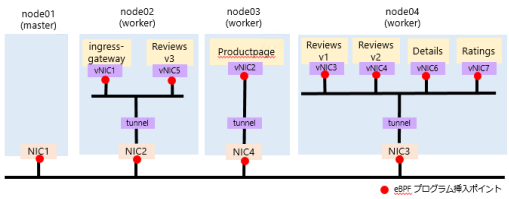

図 2　実験環境構成図

関数は既存の手法において insertByKey 関数を呼び出すことでリストに push_back している。本研究では insertByKey の代わりにリストの先頭にトレースコンテキストを挿入する関数として insertTraceContext 関数を新規追加する。

7.3 モニタリング

各インターフェースにて取得できたトレースコンテキストを OpenTelemetry を用いて Istio にて動作しているトレーシングツールに送信することでマルチレイヤトレーシングを実現する。OpenTelemetry とはアプリケーションの動作やパフォーマンスの全体像を把握するためにトレースやメトリクスなどのテレメトリデータの作成・管理を目的に設計されたオープンソースプロジェクトである。Python, Java, Go などの様々な言語用に SDK, API, ライブラリが準備されており、テレメトリデータの収集に利用可能である。本研究では 7.2 節で述べた通り Python から eBPF プログラムをロードすることで ID の取得をしているため、OpenTelemetry の実装も Python を使用する。任意のインターフェースにて取得したトレース ID とスパン ID を使用して新たな Span を作成し、トレーシングツールへ送信することでリクエストがインターフェースを通過する時刻が記録できる。

8 評価

評価環境として master 1 台、worker 3 台からなる Kubernetes 環境を用意した (図 2)。各マシンの Kernel バージョンは 5.4.0-91-generic 、Kubernetes は v1.21.0 を使用した。この基盤上で、6 つのマイクロサービスで構成された Bookinfo というアプリケーションを稼働させ、Istio 及び作成したトレース ID 検出ツールを用いてトレース ID を検出するという処理に対して、トレース ID の取得率とオーバーヘッドを計測した。

8.1 トレース ID 取得率

今回は 7 つのマイクロサービスが稼働するコンテナそれぞれの仮想インターフェースと Bookinfo が動作している 3 つのノードの物理インターフェースの合計 10 箇所に eBPF プログラムを挿入した。そして、10000 個のリクエストを送信し、検出できたトレース ID の個数をカウントし、トレース ID の取得率を算出した結果を図 4 に示す。このとき、検出ツールの実装としてトレース ID が取得時にスパン ID も取得できること

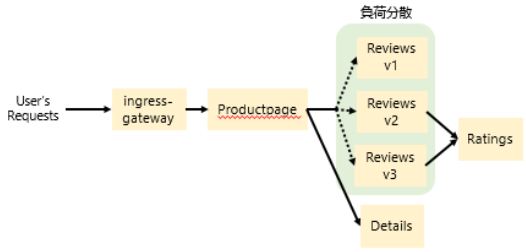

図 3　Bookinfo のリクエスト処理図

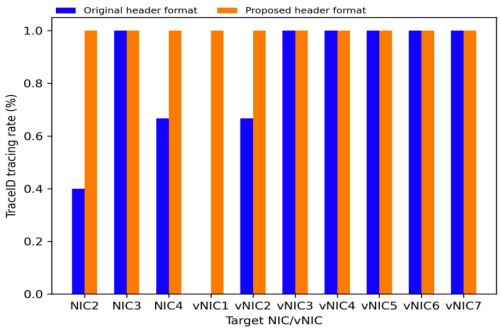

図 4　各インターフェースのトレース ID 取得率

を保証している。そのため、この実験ではトレース ID に着目して取得率を調査する。

既存手法では vNIC1 にてトレース ID の検出率が 0% であり、NIC2, NIC4, vNIC2 でも同様に一部のリクエストが持つトレース ID の取得に失敗している。図 3 より、ingress-gateway は Productpage にリクエストを送信する。経路としては vNIC1, NIC2, NIC4, vNIC2 の順である。この時 10000 リクエスト送信しているため、発生するトレース ID は 10000 個存在する。その後、Productpage HTTP ヘッダーの変更を行い、Reviews と Details それぞれに 10000 個のリクエストを送信する。そのため、vNIC2 と NIC4 では合計 30000 個のリクエストが通過し、そのうち ingress-gateway からのリクエストに含まれているトレース ID を検知できていないため、リクエスト全体の 66% のトレース ID を eBPF プログラムが検知できたこととなる。NIC2 では Productpage から Reviews に送信されるリクエストのうち約 33% と Reviews v3 が Ratings に送信するリクエストに含まれるトレース ID の検知ができているため、リクエスト全体の約 40% のトレース ID を eBPF プログラムが検知できている。このことから既存手法では ingress-gateway から送信されたリクエスト分のトレース ID が検知することができず、提案手法では各インターフェースにおいてトレース ID をすべて検出することができた。これらの結果からトレース ID を前方に移動させることで他の拡張ヘッダー次第で取得できていなかったトレース ID を漏れなく取得できるようになった。

8.2　オーバーヘッドの測定

オーバーヘッドの測定には、autobench を使用し、毎秒 50・80 リクエストを合計 5000 個送信し、リクエスト送信時から Bookinfo がページを作成し、レスポンスが返却時までの平均時間を出力する。この計測を既存手法、トレースコンテキストを移動させた提案手法、トレースコンテキスト移動させた上で eBPF プログラムで TCP オプションヘッダーにランダム値を

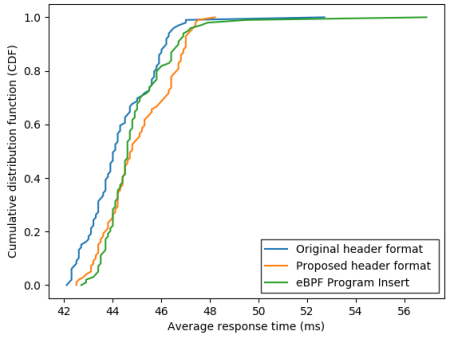

図 5　既存手法と提案手法の平均応答時間の CDF (毎秒 50 リクエスト)

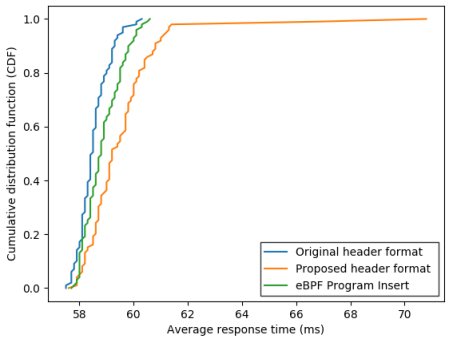

図 6　既存手法と提案手法の平均応答時間の CDF (毎秒 80 リクエスト)

追加する処理を追加した場合の 3 パターンで実施する。それぞれ 100 回繰り返した結果の累積分布関数 (CDF) を図 5 と図 6 に示す。eBPF プログラムを挿し込まない提案手法では既存手法と比較してどちらのレートでも数 % 程度の遅延が確認できた。また、eBPF プログラムを挿し込んだ場合は大きなオーバーヘッドが発生していないことが確認できた。

8.3　マルチレイヤトレーシングの確認

Bookinfo が動作しているノードの物理インターフェースと Bookinfo の仮想インターフェースに eBPF プログラムを動作させ、OpenTelemetry を使用して取得したトレース ID とスパン ID を付与して作成したスパンを Zipkin に送信した結果を確認する。そして、提案したマルチレイヤトレーシングの Web UI 上の表示と既存のトレーシングの Web UI 上の表示を比較する。図 8 はマルチレイヤトレーシングの結果であり、表示されている仮想インターフェースのフォーマットはサービス名_仮想インターフェース名となり、物理インターフェースのフォーマットは node 名_物理インターフェース名となっている。この結果から、各サービスが持つ仮想インターフェースとサービス間の通信で通過する物理インターフェースが各サービスの処理ごとにまとめて記録できていることがわかる。

9　考察

本研究ではマルチレイヤトレーシングの実現に向けてアプリケーションレイヤにおいて付与されたトレースコンテキストをネットワークレイヤである物理と仮想、両方のインターフェースを通過時に取得し、Istio 上で動作している Jaeger において通過タイミングを記録することでマルチレイヤトレーシングを実現した。提案手法が効果的であることを示すために 3 つの

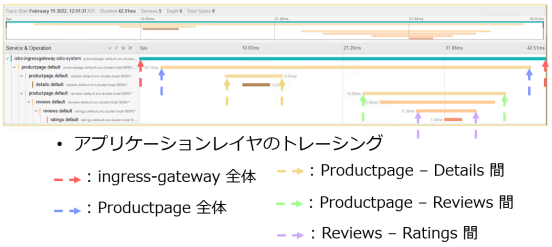

図 7　Jaeger Web UI におけるアプリケーションレイヤのみのトレーシング

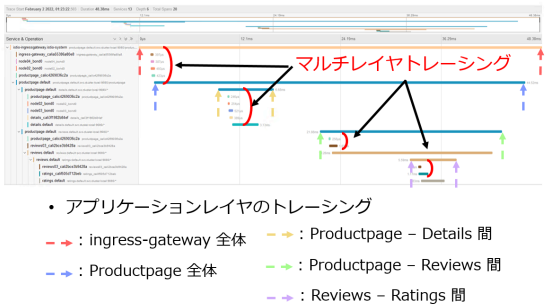

図 8　Jaeger Web UI におけるマルチレイヤトレーシング

評価を実施した。まず 1 つ目の評価としてアプリケーションレイヤにおけるトレースコンテキスト移動によるネットワークレイヤでのトレース ID 取得率の変化についてである。この評価結果からトレースコンテキストが既存の位置に存在する場合、一部のトレースコンテキストを検出できていないことがわかった。これは Envoy がトレースコンテキストを付与する前に付与された別のヘッダー情報量が多く、検出ツールとして用いている eBPF の探索可能範囲から外れてしまっていることが原因だと考えられる。具体的な値として x-envoy-peer-metadata の値に非常に大きな値が設定されている。特に、今回の実験時に損失が発生した ingress-gateway から発生するパケットの x-envoy-peer-metadata の値は 1000byte を超えており、非常に大きい値であることがわかった。eBPF では無制限のループがサポートされていない上にプログラムのサイズや命令数等に限界がある。そのため、これ以上何か処理を追加すると検出器が読み取り負荷のエラーを発生させる。別の機能追加などの拡張性のためや探索によって発生するコストを抑えるためにも for 文は少ないことが望ましいため、アプリケーションレイヤにおいてトレースコンテキストを前にすることは必要である。

2 つ目の評価としてトレースコンテキストを前方に移動させた際の Istio システム全体のオーバーヘッドの測定である。この評価結果からトレースコンテキスト移動におけるオーバーヘッドの評価結果からトレースコンテキストを前方へ移動させる処理自体にはオーバーヘッドが発生しないが、移動させたことによるその他の処理で数 % 程度のオーバーヘッドが発生している。また、今回の実験におけるリクエストは図 2 の node01 から送信されており、実験環境と同一のネットワーク上に存在している。そのため、一般的な HTTP リクエストにおいて発生するようなインターネット上の遅延は発生していない。インターネット上の遅延も加味すると今回のレスポンスタイムよりも一連の処理に長い時間がかかるが、今回発生した遅延の量は変わらないため、全体のレスポンスタイムから求められる遅延の量は少なくなると考えられる。また、eBPF プログラムを用いて TCP オプションヘッダーへランダム値を追加する処理を追加した際のオーバーヘッドは計測結果からほとんど存在していないと言えるため、提案手法は大きなオーバーヘッドを発生させないと言える。

3 つ目の評価はモニタリングである。この評価結果から任意のインターフェースを通過した時点を分散トレーシングツールを使用して同時に確認することができていることがわかる。また、各サービス間のやり取りにおいて同一ノード上にサービスが存在する場合は各サービスの仮想インターフェースのみが通過点として記録されており、別ノード上にサービスが存在する場合は各サービスの持つ仮想インターフェースとそのサービスが所属するノードの物理インターフェースが記録されていることも確認できた。これにより、例えば ingress-gateway から Productpage への通信では node04 と node02 という異なるノードを通過していることがわかり、Reviews v03 から Ratings への通信では node の物理インターフェース情報がないため、この 2 つのサービスが同一ノード内に存在していることが Kubernetes 上でサービスの配置状況を確認することなく判別できるようになった。

10　まとめ

本研究ではマイクロサービス環境におけるマルチレイヤトレーシングの実装を行った。提案手法では Envoy Proxy のソースコードに手を加え、トレースコンテキストを HTTP ヘッダーの前方へ移動させた。そして、インターフェースにて eBPF を利用してトレース ID 等を取得し、アプリケーションレイヤで動作しているトレーシングツールに取得した ID を使用してスパンを送信することでマルチレイヤトレーシングを実現した。また、TCP オプションヘッダーへランダムな値を埋め込み、トレース ID 等の抽出時に同様に抽出して関連付けた。これにより、トランスポートレイヤやスイッチなどの処理の制限が多いネットワーク機器においても追加したランダム値を用いてトレーシングを行うことで、アプリケーションレイヤのトレーシングと同等のトレーシングを実現できる。

文　献

[1] Kun Suo, Yong Zhao, Wei Chen, and Jia Rao. vnettracer: Efficient and programmable packet tracing in virtualized networks. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pp. 165–175, 2018.

[2] Takashi Shiraishi, Masaaki Noro, Reiko Kondo, Yosuke Takano, and Naoki Oguchi. Real-time monitoring system for container networks in the era of microservices. In *2020 21st Asia-Pacific Network Operations and Management Symposium (APNOMS)*, pp. 161–166, 2020.

[3] Benjamin H. Sigelman, Luiz AndréBarroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[4] *Zipkin*. `https://zipkin.io/`.

[5] *Jaeger Architecture*. `https://www.jaegertracing.io/docs/1.30/architecture/`.