4K-06

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

吉谷　玲奈†　　李　忠翰‡　　廣津　登志夫†

法政大学情報科学研究科情報科学専攻†，トヨタ自動車株式会社‡

1 はじめに

Amazon や Netflix のサービスで見られるように、マイクロサービスアーキテクチャは広く普及している。マイクロサービスアーキテクチャは Docker などのコンテナ技術と組み合わせて運用されることが多く、個々のマイクロサービスがそれぞれコンテナとして稼働し、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。

このようにサービスを分割させたことによってマイクロサービス間の依存関係を把握することやトラフィックのレイテンシやエラーレートなどのモニタリングをすることが困難になる。この問題を解決するためにこのような環境では分散トレーシングの技術を用いて状況のモニタリングを行う。分散トレーシングでは HTTP ヘッダーにトレース ID とスパン ID を付与し、サービス間で伝播させることで、サービス間の依存関係の把握やサービスレイヤにおけるトラフィックのレイテンシやエラーレートなどのモニタリングを行う。

分散トレーシングで測定されるレイテンシはアプリケーションレイヤで測定したものであるため、コンテナ内部の処理時間や仮想・物理ネットワークにおけるネットワーク遅延が含まれる。そのため、遅延が処理過程のどこで発生しているかを明らかにするためには、トレース ID が付与された通信パケットの通過状況を下位層で検出し、より詳細に問題の原因を特定することが必要になる。そこで、本研究ではネットワークレイヤとサービス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。

2 関連研究

Suo ら[1] は eBPF を用いて複数ノード上の仮想化ネットワークに対して軽量かつプログラマブルにトラフィックトレースが可能な vNetTracer を開発した。vNetTracer は 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 を使用して個々のアプリケーションを区別する。

vNetTracer ネットワークレイヤに着目してトレーシングを実施したが、本研究ではサービスレイヤと統合してのトレーシングを実現するためにサービスで付与されたトレース ID をネットワークレイヤでも取得できるようにする。

3 設計と実装

トレース ID の検出は eBPF という Linux システムにおける汎用的な監視ツールを用いる。eBPF はユーザー空間で作成したプログラムをカーネル空間にて実行するため、命令数やループを使用できない等の制約条件が多い。トレース ID は HTTP の拡張ヘッダーとして付与されるため、その情報が置かれる場所は他の拡張ヘッダーの長さにより変動する。そのため、それを eBPF で探索するためには HTTP ヘッダーの最初から探索をすることが必要になり探索コストの増加や、計算量的に解析可能範囲に入らないことが考えられる。この問題解決のため、トレース ID を付与するサービスを改造し、付与情報を HTTP ヘッダーの前方に移動させる（図 1）。

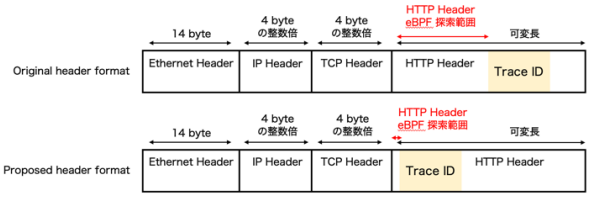

図 1 Header format 比較

本研究ではマイクロサービスの実行基盤として Kubernetes を想定し、サービス間の通信を担うサービスメッシュを Istio を用いること前提として実装を行った。Istio ではトレース ID をデータプレーンに存在する Envoy Proxy を利用して着脱する。そのため、HTTP ヘッダー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 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 ID 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 ID を付与することで HTTP ヘッダーの前方にトレース ID が移動する。

eBPF を用いたトレース ID 検出ツールは BPF

Design and implementation of multi-layer tracing for microservice environments
† Yoshitani Reina, Hosei University
‡ Chunghan Lee, Toyota Motor Corporation
† Toshio Hirotsu, Hosei University

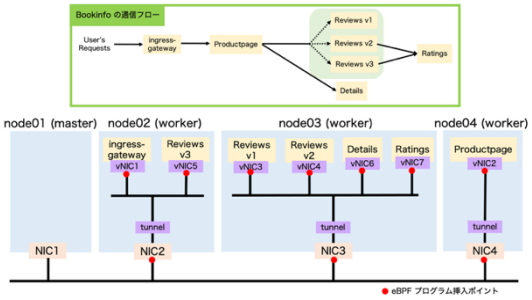

図 2 実験環境構成図

Compiler Collection (BCC) と Python を用いて作成する。この検出ツールでは IP ペイロードの先頭から HTTP ヘッダーの存在を調べ、さらに拡張ヘッダーの中からトレース ID を探す。抽出したトレース ID 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。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。

4 評価

評価環境として master 1 台、worker 1 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。

4.1 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 (*vNIC*) と Bookinfo が動作している 3 つの worker の物理インターフェース (*NIC*) 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、何個のトレース ID を検出できたかカウントし、トレース ID の取得率を算出した結果を図 3 に示す。

既存手法では *vNIC1* にてトレース ID を検出できておらず、*NIC2*, *NIC4*, *vNIC2* でも同様に ingress-gateway から送信されたリクエスト分のトレース ID の取得に失敗している。一方、提案手法では各インターフェースにおいて eBPF でトレース 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

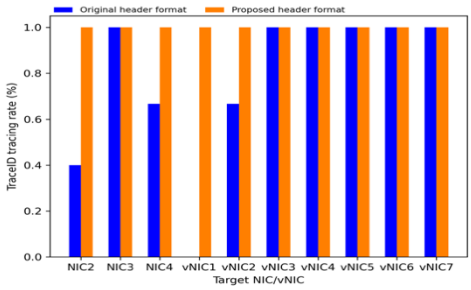

図 3 各インターフェースのトレース ID 取得率

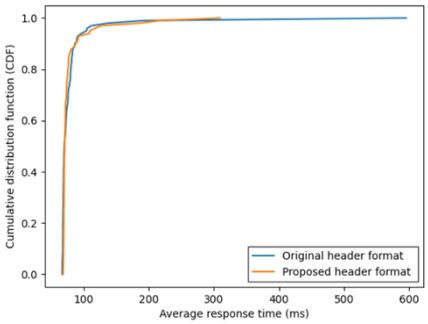

図 4 既存手法と提案手法の平均応答時間の CDF

ようになった。

4.2 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 80 リクエストを合計 5000 個送信し、リクエストを送った瞬間から Bookinfo がページを作成し、レスポンスが返却されるまでの平均時間を出力する。この計測を 100 回繰り返した結果の累積分布関数 (CDF) を図 4 に示す。提案手法の場合では既存手法の場合と比較して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。この結果からトレース ID を前方へ移動させたとしてもシステム全体として大きなオーバーヘッドは発生しないことが言える。

5 まとめ

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を実現するために HTTP ヘッダーの前方に移動させることで仮想・物理インターフェースそれぞれにてトレース ID を検出できるようにした。ネットワークレイヤにおけるトレース ID 検出には eBPF プログラムを作成し、Kubernetes 上のアプリで評価実験を行った。その結果、提案手法がトレース ID を漏れなく検出でき、そのオーバーヘッドは無視できるほど小さいことが確認できた。

参考文献

[1] K. Suo, Y. Zhao, W. Chen and J. Rao, "vNetTracer: Efficient and Programmable Packet Tracing in Virtualized Networks," 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), 2018, pp. 165-175.