マイクロサービスにおけるサービス間メッシュ型アーキテクチャ上でのキューレベルでのサービスステータス管理手法

飯島 貴政[1,a)]　串田 高幸[1]

概要：マイクロサービスの欠点としてあげられるロールバックの処理時間を縮小を図るため, データ通信レイヤーを拡張し，各サービスのキューの状況とリソースの残り状況をサービスメッシュのネットワークを利用して共有することで, すべてのマイクロサービス全体の冗長なロールバックとリトライを減少させリソースの効率化を図る.

1. はじめに

1.1 背景

昨今の IT サービスでは物理的なサーバの管理を Amazon や Google などのベンダーに任せ, ユーザー側では物理的な問題を気にする必要のないクラウドサービスが主流になりつつある [1].Kubernetes や Istio というクラウドアーキテクチャに関わるサービスがオープンソースで開発されている. また, クラウド上でのインフラストラクチャが主流になるにつれて, 従来までの 1 つのサーバーで全ての処理を行うモノリシックなサービスから一つ一つが細かく別れるマイクロサービスへとシステムのアーキテクチャが変容してきた [2]. しかし, 一つ一つのサービスが細かくなったことで, 問題の発見は容易になったがそれぞれの実装の仕方や実装に用いるプログラム言語が違ったり, ネットワークのアクセス手法が従来のモノリシックなサービスに比べてデータの一貫性の確保や個別のリソース管理を配慮する必要がある. マイクロサービスにおける一貫性についてはデータが巨大化する際に回避できない問題である [3][4][5]. 本論文では, ２つのギャップを埋める手法を研究し, 高可用性やスケーラビリティなどに優れているクラウドサービスの欠点を少なくする研究をする. クラウドサービスにおいてそれぞれのマイクロサービスの統合監視のアーキテクチャにサービスメッシュがある. [6] これは近年のマイクロサービスのデプロイにはよく用いられる. これはそれぞれのマイクロサービス間における通信をソフトウェアの実装とは関係なく導入することができる. サービスメッシュのミドルウェアとして知られる Istio では Lyft が開発した Sidecar-proxy である Envoy と連携することで Kubernetes を用いてデプロイされている環境で Control plane と data plane をサービスに上乗せする形で実装することができる [7] . Istio で提供される機能を以下に挙げる.

- マイクロサービス間のネットワークのルーティング
- 通信の暗号化
- メッシュ内のサービスの監視
- ロードバランシング

マイクロサービス間のネットワークのルーティングにおいてはこれまでのマイクロサービスはそれぞれのサービスは疎結合であったがサービスごとに連携するサービスを手動で入力する. 例として，商品を販売する EC サイトを構築したとする. 既存のマイクロサービスにのデプロイ例を図 1 に示す.

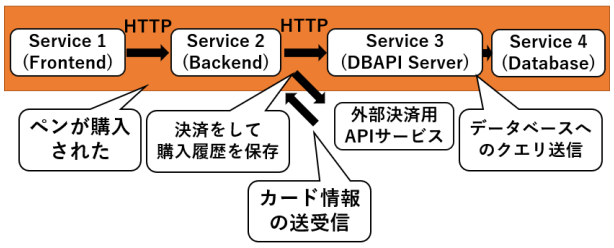

図 1　Micro Service Architecture

サービスは大きく 4 つと外部 API に別れている.

- 商品を購入するページ/フロントエンド部分を担当するサービス 1
- フロントエンドから送信された処理を API に転送するサービス 2
- 決済の処理を実行する外部 API
- API Server 送信された処理を実行するサービス 3
- 購入履歴をデータベースに保存するサービス 4

[1] 東京工科大学コンピュータサイエンス学部
〒 192–0982 東京都八王子市片倉町 1404-1
[a)] C0116023

上記のサービスの例ではサービス 1,2,3,4 が一方的な流れで HTTP API を通して通信している．そのため，サービス 2 は 1 のサービスを，同様にサービス 3 は 2,1 の状態を調べることはできない．Istio はサービスの入り口と出口にプロキシを挟み，HTTP でサービスのイベントを送信する際に Envoy に判別できるヘッダーを付与することができる．これにより実際にサービスのイベントが発生した際に情報を付加することが可能になる．また，中身の処理を空にして Envoy をトリガーさせるヘッダーのみを送信することが可能になった．サービスの監視についてはそれぞれのサービスの内側で測定する仕組みを作る必要があったが Istio を導入することにより，Envoy が Kubernetes クラスター上のサービスに上乗せする形でクラスターが実行される．これにより上記に示したサービス間における通信がネットワーク経由になったことによるデメリットが軽減された．図 1 のサービスに Istio を導入した際のアーキテクチャを以下の図 2 に示す．

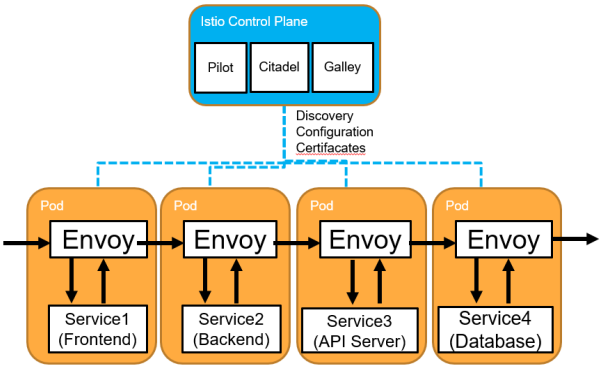

図 2　Istio Architecture

1.2　課題

Envoy ではプロキシという特性上，サービスのステータスの共有タイミングが外部サービスとの通信タイミングになる. 図 1 に示されるように，通信時に必ずネットワーク経路で Envoy を通過するため，サービスが他のサービスとの通信を行う際にリソースの状況をサービスメッシュに共有することが可能である．長時間実行するジョブなどではリソースの状況などは次のキューに入るまで共有されない. 長時間実行するジョブなどではクラウドリソースを長時間拘束することもあるため，ジョブを実行した後にサービスのリソースを随時確認する必要がある．また，一つのサービスで障害が発生した際にその処理に関わるキューすべてが正しく行われているかを確認する必要がある．本研究の目的はクラウドサービス上のマイクロサービスを維持する上でかかるエンジニアの監視や制御に要する時間を削減し，実装を抽象化してパッケージする事で新しくクラウドを設計する際にサービスごとにステータスを共有する仕組みを簡単に入れる事で適切なリソースのプロビジョニングによる維持する事へのコストダウンを図る．実際のシナリオではデプロイメントが運用コストによって制約されることが挙げられている [8]．クラウド上のリソースの状態をリアルタイムで取得し，管理することはそれらの制約内にサービスをデプロイした際にリソースに負荷がかかる際でもダウンタイムの緩和や，障害の予防に効果を発揮する．ハイブリットクラウドやマルチクラウド環境下ではコストの削減策がそれぞれのベンダーに依存する形になる [9]．クラウドリソースの節約は信頼性を最重要に据えるサービス (例: バックアップ, 障害対応サイト) 以外では経常のコストを削減できるため有用である．そのため，ベンダーに依存しない形でのリソース削減が望ましい．また，マイクロサービス内のサービスにおいて障害が発生した際にイベントのリトライを実行する機能を見直し，各マイクロサービスごとのイベントレベルのでのステータスをネットワークに共有することでサービスにおける冪等性を担保する．現状のマイクロサービス間のデータの共有方法では特定のノード間についてのみ通信することが多いがより透過性を持たせるためにサービスのステータスを全ノードからアクセスさせる必要がある．

2.　関連研究

Envoy を用いた Meina Song らの研究では，障害検知の際に実装部分である Application Layer と Envoy がデプロイされる Proxy Layer, 新たに Zipkin や Jeager を用いた Data Analyzer layer を作成し，Envoy から得られるデータを用いて障害検知を行う仕組みが提案されている．この方法では Istio とは別に Data Analyzer layer を用いることで，Istio の強みである導入への難易度が高くなってしまっている [10]．本研究ではそれぞれのサービスの状況を監視し，関連するサービスを見に行く Repoter を Istio の Control plane, Envoy の一部にネイティブで組み込むことに繋がった．サービスメッシュ上の監視及び制御トラフィックについての研究では，マイクロサービスアーキテクチャにおけるサービス間での相互監視においてデータにおいて監視及び制御トラフィックについてはアプリケーションテナントとは分離してデプロイすることは困難であり，これによりセキュリティ上の問題が発生する可能性があるとしている [11]．これに対して監視および制御用のデータに関しては Envoy 側に同期させるが，アプリケーションテナントは経路に入らないデザインを提案している．本研究においてはイベントデータを取得するという特性上，完全にアプリケーションテナントと監視/制御用テナントを分離することはできなかった．本論文ではイベントの生データから Proxy 側で抽出するのではなくアプリケーション側でイベントのステータスを予め定義することで監視/制御用のデータを暗号化可能かつ既存のパフォーマンスを損なわないこのモデルを使用した．クラウドサービスにおけるリソースプロビジョニングに対する研究は Cloud broker を

用いてプロビジョニングを行う手法, スケジューラーを用いてプロビジョニングを行う手法がある [12][13]．しかし，監視及びデプロイするモジュールをサービスネットワークの外に実装しているため外部のプロビジョニング計算サーバにアクセスすることを考慮する必要がある．しかしこれらをサービスとはコンポーネントが独立している各サービスの Envoy から Istio の Control plane に報告するモデルをサービスメッシュに統合することで，外部との通信の追加実装が不必要となる．ネットワーク内でのプロビジョング意思決定を行うことでより素早い意思決定と外部サービスに依存しないプロビジョニング体制を構築できる．

3. 提案

3.1 サービスメッシュ内でのリソースの共有

従来の Istio ではサービス間での通信を Envoy を経由することで Control plane にリソースの状況を共有していた [14]．しかしこの方法ではサービスが複数あった際に，処理は依存していないが，リソースの使用状況について関連がある場合に検出することが難しい．また，処理が行われていない際のリソースについてのステータスを確認することができない．また現状サービスからサービスに一度 API を通して通信してしまうとイベントごとのリソース消費量は一貫して取得することは困難である．本論文ではイベントまたはジョブキューレベルでのマイクロサービスのそれぞれのリソース使用量を共有するプログラムを提案する．データ通信レイヤーとは別に各サービスのキューの状況とリソースの残り状況とは別のネットワークとしてメッシュ状に作成されたネットワークで共有することで，すべてのマイクロサービス全体の冗長なロールバックとリトライを減少させリソースの効率化を図る．

3.1.1 手法 1 サービスメッシュにおけるリソース共有レイヤーを新たに作成

この手法では，従来の Istio,Envoy とは別のネットワークレイヤーを新規作成することで 1 秒間に 1 回など，高頻度なリソース共有においてもメインのデータ通信ネットワークに影響を与えることなくそれぞれのサービスのリソース状況を得ることができる．アーキテクチャのイメージを以下の図 2 に示す．この手法の課題であり後述する手法に劣るところは Istio の持つ導入の容易さを失うことである．Envoy は Application が実装されているコンテナと同じ Pod に配置され，入力の port を envoy を通してから envoy によって実装部分へ転送されたのち，実装部分の出力から envoy を通して別サービスに出力される．これらは同じ Pod, ネットワークで構成されていることから既存の Kubernetes の環境を変えずに実装できた．しかしサービスと同じネットワーク上でサービスのイベントやジョブのキューレベルのリソース状況を共有することは既存のネットワークの帯域幅の減少, 及びピーク時には輻輳も考えられる．このため，メインサービスのデータの通信層とはネットワークを分離させる必要がある．ネットワークを分離したことでメインサービスのネットワークの帯域幅を減少させる恐れはなくなったが，ステータス共有レイヤーを作成するには新たなネットワーク，リソースを監視するノードを用意する必要があり，既存の環境とほとんど変わらずデプロイできる Istio と Envoy の利点が失われてしまう．この提案の全体のアーキテクチャを以下の図 3 に示す．

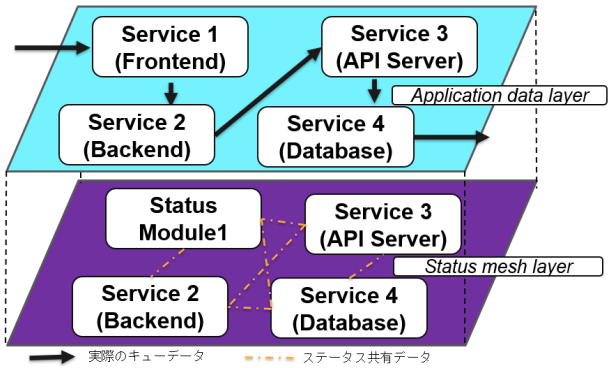

図 3 ステータス共有レイヤーを追加したアーキテクチャ

また，この手法ではノード数を 1 つ増やした際にステータス共有用のコンテナも増えるため，サービスが一つ増えた場合にはネットワークの経路が既存のサービス数分増加してしまう．後述の手法と異なるところはサービス本体のネットワークとは別のネットワークでステータス共有をおこなうため，その分のネットワークの構築の手順が複雑化する．そのため，クラウド上にデプロイするサービスメッシュではこの手法を使用すべきではない．しかし，この手法で用いたサービスのデータ層とは別のネットワークでイベント情報を取得する手法はステータスの共有回数を増やしたとしてもアプリケーションの負荷を全く増大させないので大規模システムには導入を検討すべきである．

3.1.2 手法 2 サービスメッシュにおけるステータス共有レイヤーを Envoy を拡張して実装

この手法では既存の Envoy を拡張し，サービスのイベントにおけるイベントやキューレベルでのロギングとメッシュネットワークへのイベント処理状況の共有を行う．これにより，ステータス共有専用のネットワークやコンテナを用意する必要はなくなるためサービスへの導入にあたり Istio とは別に用意する必要はなくなる．この手法のメリットを以下に挙げる．

- 既存の構成を変更させずにステータスを共有可能
- マイクロサービス間におけるにネットワークを再利用できる

この提案の全体のアーキテクチャを以下の図 4 に示す．

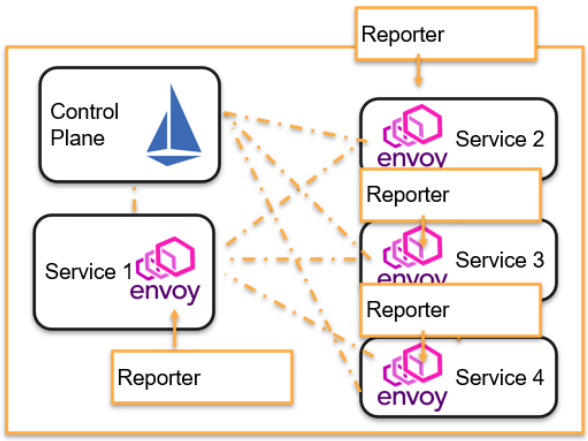

図 4　Envoy を拡張するアーキテクチャ

3.2　Envoy 拡張

既存のサービスの実行時に行われる通信の出力のヘッダーにイベントのジョブ進行状況とリソース状況を挿入する．出力の本体部分 (body) には当該イベントの詳細情報をサービスが次のサービスへ API のリクエストを送信する前に Envoy を通過する際，Header を Istio の Control plane に送信する．また，サービスのイベントが新規に 1 分以内に発生していない場合，Envoy の Repoter が定期的ジョブ監視データを Control plane に送信する．その際にイベントごとに ID を付与し，関連するサービスを各サービスの Envoy の Reporter からの報告で Control plane がイベント状況やサービス全体のキューの流れを自動で把握することができる．すなわち，Reporter を起動して初期の段階で数回イベントを発生させるだけで Reporter からの情報をもとに時系列データ本システムがサービスの流れを記憶することができる．障害が発生した際にもクラウドサービスにおいてどこの段階でエラーが起こっているのでその後の処理を実行せずにロールバックを始めるトリガーとして作用することが可能となる．各サービスの通信イメージを以下の図 5 に示す．

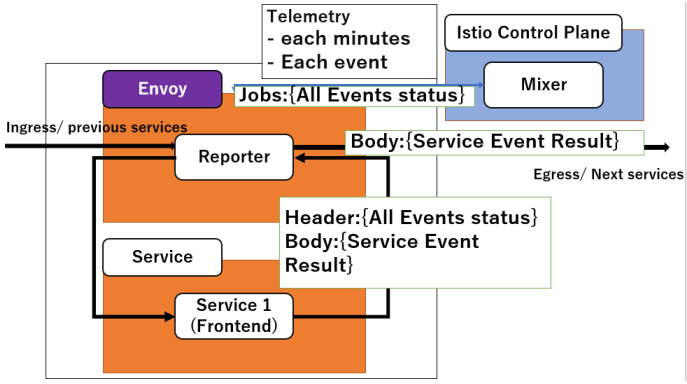

図 5　コンポーネント同士の通信イメージ

4.　実装と評価

4.1　実装

Envoy の拡張として C++で実装する．Istio 及び Envoy は OSS であり，ローカルでのコード編集が可能であるため，手元で拡張した Envoy を Istio の Envoy に追加もしくはオーバーライドし，Kubernetes クラスター上で実行させる．Envoy は proxy として既存のコンテナのアプリケーションへの入力と出力に介在する形でコンテナ同士の通信が可能になる．Istio の Cloud Discovery により，同一ネットワーク上に存在しているすべてのコンテナをリストとして取得できる．つまりすべてのサービスついてもれなくデータが得られることになるためである．実験環境は ESXi を用いる仮想環境の上で動作する．以下に動作させるコンテナの一覧を示す．

4.2　実験環境

- Service1 (Web サーバ):nginx container, CPU x4,RAM 8GB
- Service2 (バックエンドサーバ):CPU x2,- RAM 8GB,VM(Java,spring)
- Service3 (DB 中継 API サーバ):CPU x2,- RAM 8GB,container(python)
- Service4(DB) 50gb (mongodb)

以上に拡張した Envoy を用いる設定にし，クラウドに Istio 導入済み Kubernetes クラスターをデプロイする．

4.3　評価

JMeter を用いたストレステストを行い．以下の観点から結果を評価する [15]．本研究と同じ課題を扱っている，Data Analyzer を用いた提案の評価では提案機能の導入前後のサービスの平均レスポンスタイムを評価している [10]．本研究においては平均レスポンスタイムに加え，本研究のアーキテクチャと Istio のデフォルト設定でのそれぞれの各サービスの CPU 使用率，メモリ使用率，ネットワークの輻輳, テスト完了時間を比較する．

5.　議論

今回の提案ではマイクロサービスにおける一貫性をサービスメッシュを用いて実装する手法を提案した．将来的にこれらのモデルはテストされるべきであり，既存の手法とのパフォーマンスの差を測定する必要がある．メッシュ内におけるプロビジョニングに関しては Istio でデプロイした場合には Control plane コンテナが各サービスからのリソース状況を高頻度で受信しつつモデルの計算を行うため，高負荷になることが推測される．そのため，サービスメッシュ上のイベント数による負荷テストを予めデプロイ

後に自動で実行し，どのタイミングでどのサービスの負荷が増大するかをテンプレートとして保存しておくことで，計算時間の短縮につながると考える．また Control plane 部分での各サービスの Repoter からのログをもとにした関連性のあるサービスの検出および失敗時における早期ロールバック，次のサービスへの断片的なリクエストの防止については機械学習アルゴリズムである ADS を導入することでより高度なログ分析や異常検知を実装できると考えられる [16]. しかし，その際にかかる ADS 用に拡張されたコンテナのコストを計算し，導入すべきかを検討する必要がある．

6. おわりに

本研究ではマイクロサービスにおけるサービス間でのイベントレベルでのステータス共有手法について提案した．これにより，マイクロサービスの欠点であったサービス全体でのデータの整合性をイベントレベルで管理すること一貫性が向上すると考えられる．また，1 つのイベントが原因のマイクロサービスの障害についてはサービスごとにイベントの報告を Control plane に共有することでマイクロサービスにおいてどのステップで処理が失敗したかをトレースできるようになると考えられる．本研究及び提案についてはマイクロサービスアーキテクチャー (MSA) の導入の障壁となる，障害が発生した際のオペーレーション削減及び MSA を用いた大規模サービスにおける通常時のログのトラッキング，分析について貢献すると考えられる．

参考文献

[1] Dillon, T., Wu, C. and Chang, E.: Cloud computing: issues and challenges, *2010 24th IEEE international conference on advanced information networking and applications*, Ieee, pp. 27–33 (2010).

[2] Jamshidi, P., Pahl, C., Mendonça, N. C., Lewis, J. and Tilkov, S.: Microservices: The Journey So Far and Challenges Ahead, *IEEE Software*, Vol. 35, No. 3, pp. 24–35 (2018).

[3] Fan, W., Han, Z., Zhang, Y. and Wang, R.: Method of Maintaining Data Consistency in Microservice Architecture, *2018 IEEE 4th International Conference on Big Data Security on Cloud (BigDataSecurity), IEEE International Conference on High Performance and Smart Computing, (HPSC) and IEEE International Conference on Intelligent Data and Security (IDS)*, pp. 47–50 (2018).

[4] Zhao, W., Melliar-Smith, P. and Moser, L. E.: Fault tolerance middleware for cloud computing, *2010 IEEE 3rd International Conference on Cloud Computing*, IEEE, pp. 67–74 (2010).

[5] Jhawar, R., Piuri, V. and Santambrogio, M.: Fault tolerance management in cloud computing: A system-level perspective, *IEEE Systems Journal*, Vol. 7, No. 2, pp. 288–297 (2012).

[6] Li, W., Lemieux, Y., Gao, J., Zhao, Z. and Han, Y.: Service Mesh: Challenges, State of the Art, and Future Research Opportunities, *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pp. 122–1225 (2019).

[7] Klein, M.: Lyft's Envoy: Experiences Operating a Large Service Mesh, San Francisco, CA, USENIX Association (2017).

[8] Yang, Z., Nguyen, P., Jin, H. and Nahrstedt, K.: MIRAS: Model-based Reinforcement Learning for Microservice Resource Allocation over Scientific Workflows, *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pp. 122–132 (2019).

[9] Van den Bossche, R., Vanmechelen, K. and Broeckhove, J.: Cost-Optimal Scheduling in Hybrid IaaS Clouds for Deadline Constrained Workloads, *2010 IEEE 3rd International Conference on Cloud Computing*, pp. 228–235 (2010).

[10] Song, M., Liu, Q. and E, H.: A Mirco-Service Tracing System Based on Istio and Kubernetes, *2019 IEEE 10th International Conference on Software Engineering and Service Science (ICSESS)*, pp. 613–616 (2019).

[11] Kang, M., Shin, J. and Kim, J.: Protected Coordination of Service Mesh for Container-Based 3-Tier Service Traffic, *2019 International Conference on Information Networking (ICOIN)*, pp. 427–429 (2019).

[12] Chaisiri, S., Lee, B. and Niyato, D.: Optimization of Resource Provisioning Cost in Cloud Computing, *IEEE Transactions on Services Computing*, Vol. 5, No. 2, pp. 164–177 (2012).

[13] Bi, J., Zhu, Z., Tian, R. and Wang, Q.: Dynamic Provisioning Modeling for Virtualized Multi-tier Applications in Cloud Data Center, *2010 IEEE 3rd International Conference on Cloud Computing*, pp. 370–377 (2010).

[14] Sheikh, O., Dikaleh, S., Mistry, D., Pape, D. and Felix, C.: Modernize Digital Applications with Microservices Management Using the Istio Service Mesh, *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*, USA, IBM Corp., p. 359–360 (2018).

[15] Abbas, R., Sultan, Z. and Bhatti, S. N.: Comparative analysis of automated load testing tools: Apache jmeter, microsoft visual studio (tfs), loadrunner, siege, *2017 International Conference on Communication Technologies (ComTech)*, IEEE, pp. 39–44 (2017).

[16] Sauvanaud, C., Kaâniche, M., Kanoun, K., Lazri, K. and Silvestre, G. D. S.: Anomaly detection and diagnosis for cloud services: Practical experiments and lessons learned, *Journal of Systems and Software*, Vol. 139, pp. 84–106 (2018).