Vol.2021-CSEC-93 No.20
Vol.2021-IOT-53 No.20
2021/5/14

Kubernetesとサービスメッシュを用いたエッジ基盤における近接エッジ協調型負荷分散手法の一検討

古澤 徹[1,a)] 阿部 博[1] 小林 野愛[2]

概要：エッジコンピューティングは，端末とクラウドの中間に位置するエッジで処理を行うことで低遅延応答や中継トラフィック削減が可能となる新しいパラダイムとして着目を集めている．近年コンテナ仮想化技術や Kubernetes，およびサービスメッシュを用いたマイクロサービスアーキテクチャ (MSA) が普及しているが，エッジにおいても同様に MSA を導入することで，エッジ基盤やエッジアプリケーションライフサイクルの効率的な管理，オートスケールの実行が可能になると期待される．しかし，エッジは利用可能なコンピューティングリソースが限られているため，個々のエッジ内でオートスケールを実行させても処理能力には限界がある．特定のエッジに処理限界を超える過負荷が発生した場合，エッジの処理能力が劣化し，大規模な遅延やサービス停止が発生しうる課題がある．本研究では，コンテナ仮想化技術や Kubernetes，およびサービスメッシュを用いたエッジ基盤における協調型負荷分散の実装手法を提案する．エッジのアプリケーションへの単位時間あたりリクエスト数をモニタリングし，エッジの処理限界を超えるリクエストをリソースに余力ある近接エッジまたはクラウドに転送するようにサービスメッシュ設定を動的に変更するコントローラを実装する．実験により，過負荷発生時のアプリケーションの平均処理時間が改善されることを示す．

A Cooperative Load Balancing Method for Edge Computing Resources Using Kubernetes and Service Mesh

Abstract: Edge Computing is attracting attention as a new paradigm that enables low latency response and reduction of relay traffic by performing processing at the edge, which is located between the device and the cloud. In the cloud, microservice architecture (MSA) using container virtualization, Kubernetes, and service mesh is becoming popular. MSA is expected to enable efficient management and auto-scale of edge infrastructure and edge application lifecycle. However, since the edge has limited computing resources available, there is a limit to the processing capacity even if auto-scale is executed within an individual edge. When a heavy load that exceeds the processing limit occurs at a particular edge, the processing capacity of the edge is degraded, which can cause large-scale delays and service outages. In this article, we propose an implementation method of cooperative load balancing on edge infrastructure using container virtualization, Kubernetes, and service mesh. Specifically, we implement a controller that monitors the number of requests per unit time to applications at the edge, and dynamically changes the service mesh configuration to forward requests that exceed the processing limit of the edge to the neighboring edge or the cloud, which has more resources. Through the experiments, we show that the average processing time of the application is improved during heavy load occurrence.

1. はじめに

エッジコンピューティングの実用化に向け様々な研究が進められている [1]．エッジコンピューティングはクラウドと端末の中間に位置するエッジサーバで処理を行う手法を指し，低遅延処理や中継トラフィック削減等が実現可能になる新しいパラダイムとして期待される．特に，コネクティッドカーはエッジコンピューティングの適用が期待される領域の 1 つであり，コネクティッドカー向けエッジコンピューティングの実現に向け様々な研究が進めら

[1] トヨタ自動車株式会社
Toyota Motor Corporation, Chiyoda, Tokyo 101–0004, Japan
[2] 株式会社アイ・アイ・エム
IIM Corporation
[a)] toru_furusawa@mail.toyota.co.jp

ⓒ 2021 Information Processing Society of Japan

れている [2]．2018 年には Automotive Edge Computing Consortium (AECC) が設立され，コネクティッドカー向けエッジコンピューティングの実用化に向けてユースケースや要件，アーキテクチャ定義が進められている [3].

エッジコンピューティングの課題の 1 つに，エッジサーバが地理的に分散された環境に多数配置されるため個々のエッジサーバはクラウドやオンプレミス環境と比べて利用可能なリソースが限られることがある．将来のアクセス負荷やリソース需要が十分に予測可能な場合は各地域に計画的に必要なリソースを持つエッジサーバを配備すれば良いが，予測を上回るアクセス負荷やリソース需要が特定のエッジサーバに発生する場合，過負荷やオーバプロビジョニングが発生し処理性能の劣化や長時間のサービス停止を引き起こす可能性がある．特にコネクティッドカー向けエッジコンピューティングにおいては，交通トラフィックの変動に伴い時空間的に需要が大きく変動するためこれは大きな課題となる．

このような特定のエッジサーバに対する過負荷を分散する手法の 1 つとしてエッジコンピューティングリソースの協調型負荷分散手法がある [4]．協調型負荷分散は，エッジサーバとして動作する 2 つの近接するデータセンタが協調して動作する．一方のデータセンタに対して一時的に多量のリクエストが発生し過負荷発生が見込まれる場合，その後のリクエストをリソースに余力ある他方のデータセンタに転送して処理を行うことで，平均サービス遅延時間を改善することが可能となる．

しかし，基本的なスキームや数理モデル，およびシミュレーションによる評価は [4] で述べられているが，エッジコンピューティング基盤上での具体的な実装事例は無いため，実用化に向けて実装手法の具体化が求められる．

筆者らは，今後のエッジコンピューティング基盤には，Docker や Kubernetes(K8s) をはじめとするコンテナ仮想化技術とコンテナオーケストレーションソフトウェア，そして Istio に代表されるサービスメッシュを用いたマイクロサービスアーキテクチャ (MSA) が普及していくと考えている．MSA は近年の IT システム開発で注目を集めており，クラウド領域では既に広く使われている技術である．今後はこの MSA がエッジコンピューティング基盤にも導入され，クラウド領域と連携して利用されるアーキテクチャを提唱する研究が近年増加している．例えば，[5] では Osmotic Computing が提唱される．Osmotic Computing はエッジコンピューティング基盤とクラウド基盤を相互接続し，エッジサーバとクラウド両方の基盤にまたがってマイクロサービスのデプロイメント自動化を実現する新しいパラダイムである．

本研究では，コンテナ仮想化技術，K8s，およびサービスメッシュを用いて MSA を実現するエッジコンピューティング基盤において，複数のエッジ間で協調して負荷分散を行う協調型負荷分散の実装手法を提案する．具体的には, 各エッジサーバのアプリケーションへの RPM(Request Per Minute) をモニタリングし，各エッジサーバの処理限界を超える RPM が検知された場合は，処理限界を超えるリクエストをリソースに余力ある別のエッジサーバに転送するようにサービスメッシュを動的に設定するコントローラを実装する．

以降の構成は以下の通りである．続く第 2 章では関連研究を紹介する．第 3 章では，サービスメッシュの基本技術を踏まえ，近接エッジ協調型負荷分散の実現に向けた課題を抽出する．第 4 章では，本研究で提案するサービスメッシュを用いた近接エッジ協調型負荷分散の実装方法について述べる．第 5 章では，実験により本手法の実現性を評価し，今後の課題を考察する．最後にまとめと今後の展望を述べる．

2. 関連研究

エッジコンピューティング基盤で MSA を実現する研究は多く存在する [6]．代表的な例として Osmotic Computing が知られている [5]．Osmotic Computing はエッジコンピューティング基盤とクラウド基盤にまたがりマイクロサービスを動的に最適配置するアーキテクチャである．Osmotic Computing を実現する手法として，エネルギー消費量を考慮した動的リソースマネジメントを行うフレームワークの提案や [7]，Mobile Augmented Reality Networks (MARN) 向けのサービスマイグレーションとリソーススケジューリング手法 [8] 等が提案されている．Osmotic Computing におけるマイクロサービスの実行環境としては主にコンテナ仮想化技術が想定されているが，K8s 等を用いたコンテナオーケストレーションの具体的な実装については指定されていない．

エッジコンピューティング基盤にコンテナ仮想化技術と K8s を用いる研究はいくつか先行研究が存在する．[9] では，エッジサーバが広域に多数分散配置されるエッジコンピューティング基盤において，コンテナと K8s を用いたエッジアプリケーションの効率的なデプロイメントとオーケストレーション手法が提案される．カスタムスケジューラを用いて，標準の K8s スケジューラと比較してより高速かつ CPU 計算量の少ないコンテナのスケジューリングを実現している．[10] では Fog Computing 環境向けに，拠点間の通信遅延に基づき指定場所に近い Worker Node に優先的にコンテナを配置する K8s スケジューラの実装方法が提案されている．

エッジコンピューティング基盤へのサービスメッシュ適用については，今後研究が期待される分野ではあるものの [11]，研究報告は少ない．[12] では，IoT 機器向けのディープラーニングアプリケーションシステムのフレキシブルな開発を実現する手法として，コンテナ，K8s および

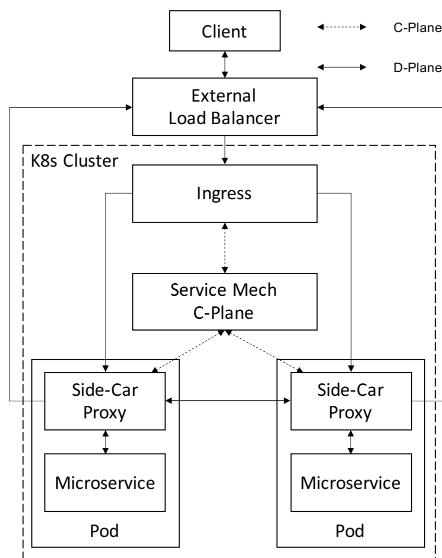


図 1 サービスメッシュアーキテクチャの概要
Fig. 1 Overview of Service Mesh Architecture

サービスメッシュの代表的なオープンソースである Istio を用いた実績的なアプローチが示される．Istio を用いた負荷分散を行う点で本研究に類似するが，[12] では 1 つの K8s クラスタ内での負荷分散を行っているのに対し，本研究はエッジサーバごとに独立した K8s クラスタを構築しエッジサーバ間の協調型負荷分散を実装している点で異なる．

3. サービスメッシュを用いた近接エッジ協調型負荷分散手法の検討

本章では，サービスメッシュの基本的な特徴をふまえ，近接エッジ協調型負荷分散に有用なサービスメッシュの基本機能を特定し，実現に向けた実装上の課題を議論する．

3.1 サービスメッシュの概要

サービスメッシュは MSA におけるマイクロサービス間通信を制御するプラットフォームである [11]．多数のマイクロサービスが複雑なトポロジー構成で接続される場合においてもサービス間の安定した通信基盤を提供する．

一般に，サービスメッシュは図 1 に示すように Control Plane と Data Plane に分離したアーキテクチャが用いられる．Data Plane は，各マイクロサービスがデプロイされる Pod(コンテナ) に配置されるサイドカー・プロキシ群から構成される．全てのマイクロサービスは，サイドカー・プロキシを介して他のマイクロサービスや外部クライアントとの通信を行う．Control Plane は，サイドカー・プロキシ群を集中的に制御，管理する役割を担い，Data Plane のメトリクス収集や，Data Plane へのポリシーや設定の適用を行う．また，K8s クラスタの Ingress のルーティング制御も行う場合が多い．

サービスメッシュは主に以下の 5 つの機能を提供する．

表 1 代表的なサービスメッシュソリューション
Table 1 Representative Service Mesh Solutions

Name	Data Plane	Open Source
Istio	Envoy	Yes (Apache 2.0)
Linkerd2	linkerd-proxy	Yes (Apache 2.0)
Consul Connect	Envoy	Yes (MPL 2.0)
AWS App Mesh	Envoy	No
Traefik Mesh	Traefik Proxy	Yes (Apache 2.0)
Kuma	Envoy	Yes (Apache 2.0)
Open Service Mesh	Envoy	Yes (MIT License)

- Service Discovery: サービスインスタンスの実行場所を特定しリクエストを適切なサービスインスタンスへ送信する
- Load Balancing: サービスへのリクエスト種別毎にトラフィックのロードバランシングを行う
- Observability: メトリクス（レイテンシやエラー率等）の収集，ログ収集，および分散トレーシング機能を提供する
- Resilience: マイクロサービス毎のタイムアウト値設定やリトライ数設定，およびサーキットブレーキング機能を提供する
- Security: サービス間通信の暗号化とアクセス制御を行う

表 1 に代表的なサービスメッシュソリューションを示す．代表的なサービスメッシュソリューションである Istio をはじめ，多くがオープンソースソフトウェアとして開発されており，Data Plane には Envoy が用いられている．

3.2 サービスメッシュにおける負荷分散手法

3.1 で述べたように，負荷分散 (Load Balancing) はサービスメッシュの担う代表的な機能の 1 つである．ここでは，代表的なサービスメッシュである Istio における負荷分散の設定例を紹介する．

Istio には様々なトラフィック管理機能が実装されているが，マイクロサービス宛トラフィックの負荷分散を設定するには主に VirtualService を用いる手法と DestinationRule を用いる手法がある．VirtualService の設定例と DestinationRule の設定例を図 2 に示す．

VirtualService は，トラフィックのマッチ条件とそれに対応する宛先を指定しルーティング先を設定する．図 2 の例では，$svc1$ アドレス宛のリクエストのうち 25%のリクエストをバージョン $v2$ の $svc1$ Service へ，75%のリクエストをバージョン $v1$ の $svc1$ Service へ転送する設定を行っている．

DestinationRule は，VirtualService によるルーティングによって特定されたサービスホスト宛のリクエストに対して設定されるポリシーを定義する．図 2 の例では，$svc1$ Service 宛トラフィックを，Round-Robin アルゴリズムに

```
1  apiVersion: networking.istio.io/v1alpha3
2  kind: VirtualService
3  metadata:
4    name: svc1
5  spec:
6    hosts:
7    - svc1.prod.svc.cluster.local
8    http:
9    - route:
10     - destination:
11         host: svc1.prod.svc.cluster.local
12         subset: v2
13       weight: 25
14     - destination:
15         host: svc1.prod.svc.cluster.local
16         subset: v1
17       weight: 75
```

(a) VirtualService の設定例

```
1  apiVersion: networking.istio.io/v1alpha3
2  kind: DestinationRule
3  metadata:
4    name: svc1
5  spec:
6    host: svc1
7    trafficPolicy:
8      loadBalancer:
9        simple: ROUND_ROBIN
```

(b) DestinationRule の設定例

図 2 Istio における負荷分散の設定例
Fig. 2 Example of Load Balancing Configuration in Istio

基づき各サービスインスタンスへ分散する設定がなされる．

3.3 近接エッジ協調型負荷分散の実装に向けた課題

本研究で想定する近接エッジ協調型負荷分散の概要を図 3 に示す．動作の概要は以下の通りである．各エッジサーバではそれぞれ独立した K8s クラスタが動作しており，同一のアプリケーションが動作するコンテナ (Pod) が各エッジサーバの K8s クラスタで動作している．各端末は近接するエッジサーバ上のアプリケーションにリクエストを送り処理を行う．単位時間リクエスト数が測定されており，各エッジサーバが安定してアプリケーションを処理可能な単位時間リクエスト数が閾値として事前に設定される．測定される単位時間リクエスト数が閾値を上回り，かつ近接するエッジサーバの単位時間リクエスト数は少なく処理能力に余裕がある場合に限り，閾値を上回る分のリクエストを近接するエッジサーバ上のアプリケーションに転送し処理を行う．

このような動作をサービスメッシュを用いて実現するには以下に述べるような課題が存在する．3.2 で述べたように，Istio におけるトラフィックの負荷分散手法は VirtualSerivce によるトラフィック転送先の割合を指定する手法と，DestinationRule による負荷分散アルゴリズムを指定する手法がある．本研究で実現を目指す近接エッジ協調型

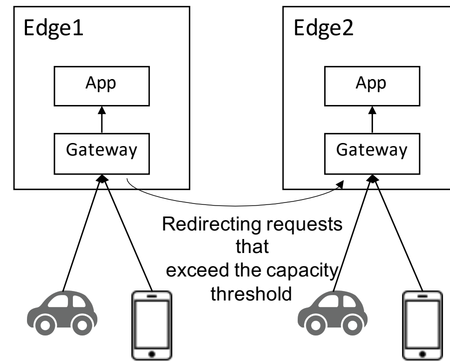


図 3 近接エッジ協調型負荷分散の概要
Fig. 3 Overview of Cooperative Load Balancing among Neighboring Edge Servers

負荷分散シナリオでは各エッジサーバが独立した K8s クラスタである想定であることから，サービスルーティングを行う VirtualService によるトラフィックの分散を行う手法が適していると考えられる．しかし，近接エッジサーバへ転送するトラフィック割合は単位時間リクエスト数に従い変化し一定では無いことから，固定的な重み値を指定する VirtualService をそのまま適用することはできないため，単位時間リクエスト数にしたがって動的に VirtualService の重みを変更する手法が求められる．

4. 提案手法

本章では，本研究で提案する K8s とサービスメッシュを用いたエッジコンピューティング基盤における，近接エッジ協調型負荷分散の実現手法について述べる．

4.1 動作環境とシナリオ

本研究で想定する動作環境を以下に述べる．あるエッジサーバ ($Edge1$ とする) と，$Edge1$ に近接する 1 つのエッジサーバ ($Edge2$ とする) それぞれが独立した K8s クラスタとして動作しており，それぞれのクラスタ上でサービスメッシュとして Istio が動作する．各クラスタ上で，クライアントからの接続を受け何らかの処理を行うアプリケーションが Pod として動作し，Horizontal Pod Auto-scaler(HPA) 機能によって負荷に応じて動的にオートスケールが実行される．各端末は位置情報等を用いて近接エッジサーバを特定可能であり，近接エッジサーバ上の Ingress Gateway を介してアプリケーションへリクエストを送信する．

本研究で想定する近接エッジ協調型負荷分散の動作シナリオは以下の通りである．$Edge1$ 上のアプリケーション Pod が HPA によって最大限スケールアウトした環境で安定して処理可能な最大 RPM(RPM_{max} とする) が事前に測定されていることを前提とする．$Edge1$ に対する端末からの RPM 値が RPM_{max} を下回る場合は全てのリクエストを $Edge1$ 上の Pod で処理するが，RPM 値が RPM_{max} を

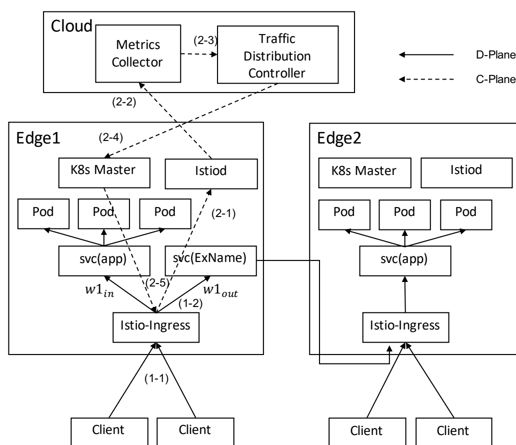


図 4　近接する 2 つのエッジサーバ協調型負荷分散の実装

Fig. 4 Implementation of Cooperative Load Balancing among Neighboring Two Edge Servers

上回る場合は最大 RPM を超えるリクエストが $Edge2$ に転送され，$Edge2$ 上の Pod で処理されるようにトラフィックが分割される．

4.2　提案アーキテクチャ概要

本研究で提案する，近接エッジ協調型負荷分散を実現するアーキテクチャについて述べる．アーキテクチャ概要と制御フローを図 4 に示す．端末からのリクエストがアプリケーションが動作する Pod に到達するまでの Data Plane の動作フローを以下に示す．

- (1-1) 端末が近接するエッジサーバ ($Edge1$ とする) の Ingress Gateway に HTTP リクエストを送信する．
- (1-2) Ingress Gateway は，設定された重み値に基づき，w_{in}%のリクエストを $Edge1$ 内の Service を介して Pod に転送し，w_{out}%のリクエストを External Name Type の Service を介して $Edge2$ の Ingress Gateway に転送する．

$Edge1$ の Ingress Gateway における w_{in} と w_{out} の動的設定を行うための Control Plane の動作フローを以下に示す．

- (2-1) $Edge1$ の Ingress Gateway における RPM が Istiod に送信される．
- (2-2) 収集された RPM が Istiod から Cloud 環境に配置される Dynatrace に送信される．なお，Dynatrace は SaaS として提供される Application Performance Management(APM) 製品である．
- (2-3)Traffic Distribution Controller は Dynatrace を介して RPM を収集し，以下の条件に基づき w_{in} と w_{out} の値を算出する．
  - ($RPM > RPM_{max}$ の場合)
    $w_{in} = 100 \times RPM_{max}/RPM$
    $w_{out} = 100 \times (RPM - RPM_{max})/RPM$
  - ($RPM \leq RPM_{max}$ の場合)
    $w_{in} = 100, w_{out} = 0$
- (2-4) K8s API Server に Ingress Gateway の w_{in} と w_{out} 値の設定変更をリクエストする．
- (2-5) リクエストに従い，Ingress Gateway の VirtualService における Weight 値の設定を変更する．

5.　実験

提案手法の実現性と課題の抽出を目的とし，以下の実験を行う．

5.1　実験構成

AWS のマネージド K8s サービスである EKS を用いて，エッジサーバとクラウド環境を想定した K8s クラスタをそれぞれ構築し，クラスタ上に Istio と実験用アプリケーションを動作させる．また，Metrics Collector には Dynatrace サービスを用いる．表 2 に各 K8s クラスタで動作させるアプリケーションを示す．各 K8s クラスタは全て以下の通りリソースを割り当てる．

- インスタンスタイプ: t3.large
- vCPU: 2
- メモリ: 8GB
- ディスク: 100GB
- OS: AmazonLinux2

実験に用いる各ソフトウェアのバージョンを表 3 に示す．

表 2　実験に用いる K8s クラスタ環境

Table 2　K8s Cluster Environment for the Experiment

Cluster Name	Running Applications
Edge1	EdgeApp, Istio
Edge2	EdgeApp, Istio
Cloud	CloudApp, Istio, Traffic Distribution Controller
Client	ClientApp

表 3　実験に用いるソフトウェアとバージョン

Table 3　Software and Version for the Experiment

種類	バージョン
Kubernetes	1.18
Istio	1.9.0
Dynatrace SaaS	1.211.90
Dynatrace OneAgent	1.209.154
Dynatrace ActiveGate	1.205.144

以下の 3 種類のアプリケーション (Pod) をクラスタ上にデプロイする．

- ClientApp: 多数端末からのリクエスト送信を模擬するアプリケーション．EdgeApp の REST-API に対して一定間隔で 1KB の JSON データを送信する．

ⓒ 2021 Information Processing Society of Japan

- EdgeApp: Edge1 クラスタと Edge2 クラスタに配置するアプリケーション．ClientApp から JSON データを受信後，JSON データ内の文字列のハッシュ値計算を 10000 回繰り返し，同一の JSON データを CloudApp に送信する．なお，本処理はエッジサーバでの CPU 負荷を増加させる目的で用意したダミーアプリケーションである．
- CloudApp: Cloud クラスタに配置するアプリケーション．EdgeApp から JSON データを受信後，RDB にデータを格納する．

いずれの Pod も，リソースの要求値と制限値として以下の値を割り当てる．

- CPU: 250m
- Memory: 500MiB

ClientApp から EdgeApp への RPM を変化させたときの，処理時間の変移と近接エッジサーバへの転送割合の変化を確認する．

5.2 RPM_{max} の決定

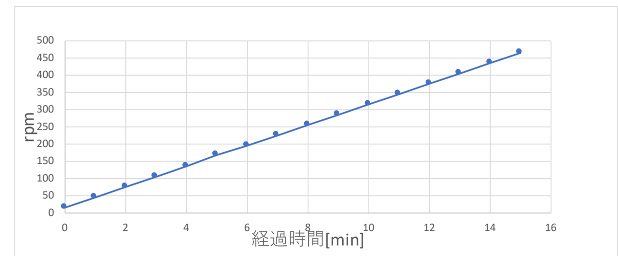
RPM_{max} を決定するため，以下の予備実験を行う．EdgeApp のオートスケールを無効化し 1 つの Pod だけ動作させる状態において，ClientApp から EdgeApp に対するリクエストの RPM を初期値を 10 として毎分 30 ずつ上昇させていき，応答時間とエラー率の変化を Metrics Collector で測定する．データ集計は 1 分間隔で行う．経過時間に伴う RPM の変化を図 5(a) に，応答時間の変化を図 5(b) に，エラー率の変化を図 5(c) に示す．RPM が 180 を超える頃から応答時間とエラー率の上昇が徐々に始まり，250 を超える頃から上昇率が急増することが確認される．

この結果をふまえ，次の節では 1Pod あたりの RPM_{max} を 180 とする場合と，130 とする場合の 2 通りについて実験を行う．前者は，応答時間とエラー率が上昇し始めたら近接エッジサーバへの転送を開始する閾値設定を，後者は，応答時間とエラー率が上昇する前に余裕をもって近接エッジサーバへの転送を開始する閾値設定を想定している．

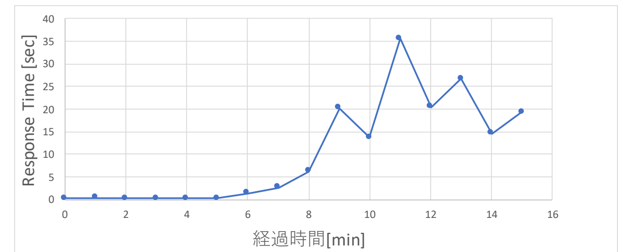
5.3 近接エッジサーバへの負荷分散

次に，本研究で提案する近接エッジ協調型負荷分散実装の評価を行う．ClientApp から EdgeApp に対するリクエストの RPM を変化させた時の応答時間とエラー率，およびリクエストの実行場所を測定する．Edge1 と Edge2 におけるオートスケールによる最大 Pod 数は 3 としていることから，RPM_{max} の値は $180 \times 3 = 540$ の場合と $130 \times 3 = 390$ の場合の 2 通りについて実験を行う．また，ベースラインとして近接エッジサーバへの負荷分散を行わない場合の応答時間とエラー率も測定する．

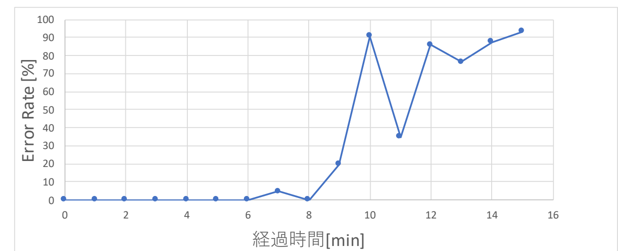
RPM は実験開始後からの経過時間を t[min] とし，以下の条件で RPM を変化させる．



(a) RPM



(b) 応答時間



(c) エラー率

図 5 1Pod 環境での RPM と性能
Fig. 5 RPM and Performance in one Pod

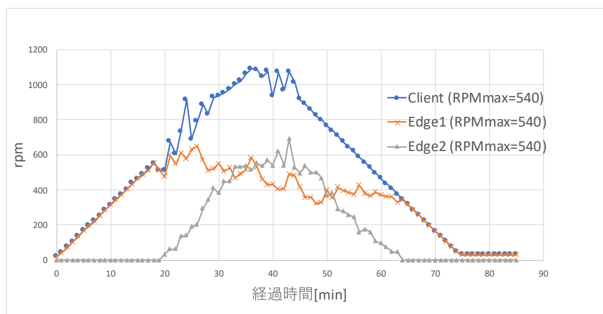
- ($0 \le t < 36$ の場合) $RPM = 30t$
- ($36 \le t < 41$ の場合) $RPM = 1080$
- ($41 \le t < 77$ の場合) $RPM = 1080 - 30(t - 41)$

実験結果を図 6 に示す．(a) と (b) はそれぞれ 1 分あたりの以下の値を測定している．

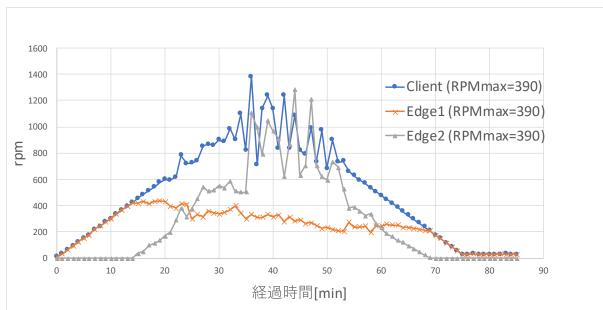
- Client: ClientApp から送信されたリクエストが EdgeApp で処理され，EdgeApp からレスポンスを受信する数．
- Edge1: *Edge*1 クラスタの EdgeApp が受信する Client からのリクエスト数
- Edge2: *Edge*2 クラスタの EdgeApp が受信する Client からのリクエスト数

(c) は ClientApp がリクエストを送信してからレスポンスを受信するまでの応答時間を，(d) は ClientApp 送信したリクエストに対するレスポンスがエラーとなる割合を示す．

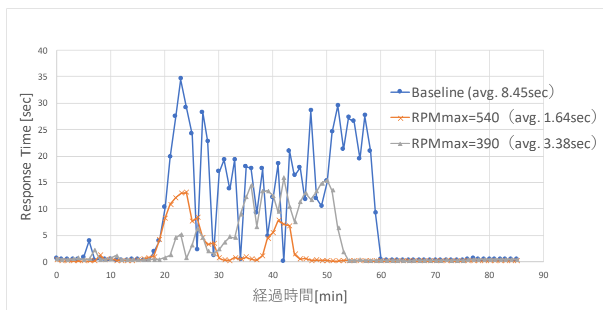
グラフの (c) および (d) から確認できるように，経過時間 17 分頃までは，ベースライン環境，$RPM_{max} = 540$ の環境，および $RPM_{max} = 390$ の環境いずれにおいても応答時間とエラー率がともに安定して動作している．しかし，以降はベースライン環境は応答時間とエラー率が両方とも急激に悪化しており，経過時間 25 分にはエラー率



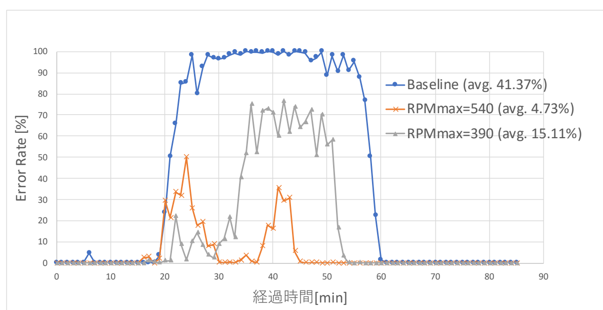
(a) RPM ($RPM_{max} = 540$)



(b) RPM ($RPM_{max} = 390$)



(c) 応答時間



(d) エラー率

図 6 性能とリクエスト転送先の変化

Fig. 6 Changes in Performance and Request Destinations

が100%近くまで上昇している．一方，$RPM_{max} = 540$ の環境と $RPM_{max} = 390$ の環境はともにベースライン環境と比較して応答時間とエラー率が改善している．ベースラインと比較して，$RPM_{max} = 540$ の環境は応答時間の平均値が19%に，エラー率の平均値が11.4%に削減され，$RPM_{max} = 390$ の環境は応答時間の平均値が40%に，エラー率の平均値が36.5%に削減される．本結果から，本研究で提案する近接エッジ協調型負荷分散によって応答時間とエラー率が改善されることが確認できる．

5.4 考察

5.2の予備実験結果を踏まえると，*Edge*1 クラスタ単独ではRPM=540まではEdgeAppが安定動作し，RPMが540を超える場合もリクエストが *Edge*2 に転送されることでシステム全体では引き続き安定動作すると期待される．しかし，図6(a)と(b)から確認されるように，$18 \leq t \leq 20$ の区間では $RPM_{max} = 540$ の環境はベースライン環境とほぼ同じ上昇率で応答時間とエラー率が悪化している．これは，RPMが540を超えてもしばらくの間全てのリクエストが *Edge*1 のEdgeAppに送られ続け，*Edge*1 の負荷が高まっているためである．Metrics Collectorが1分間隔でデータを集計しており応答時間やエラー率が悪化してから検知するまでタイムラグが発生するため，Traffic Distribution Controllerがリクエストの転送割合を変更するのに最大1分程度の時間を要することに起因する．Metrics Collectorのデータ集計間隔を短縮することでリクエスト転送割合の設定変更に要する時間を早めることができるが，Metrics Collectorのデータ収集負荷が大きくなるトレードオフがある．

また，図6(a)と(b)から確認されるように，$18 \leq t \leq 29$ の区間では $RPM_{max} = 390$ の環境の方が $RPM_{max} = 540$ の環境よりも応答時間とエラー率が小さいが，$30 \leq t$ の区間では $RPM_{max} = 540$ の環境の方が $RPM_{max} = 390$ の環境よりもこれらの値が小さくなっている．これは，$RPM_{max} = 390$ の環境ではRPMが比較的小さい段階から *Edge*2 へのリクエスト転送が開始されているため $18 \leq t \leq 29$ の区間では *Edge*1 への過負荷発生が避けられているが，$30 \leq t$ の区間に入ると *Edge*2 への過負荷が発生しているためである．*Edge*1 は $RPM = 540$ まで安定して処理可能であるが，$RPM = 390$ を超えるリクエストは全て *Edge*2 に転送されるため，*Edge*1 はRPM=150相当の空きリソースを抱えた状態で *Edge*2 に過負荷が発生していることになり，$RPM_{max} = 540$ の環境と比べてリソースの利用効率が低い状態である．要求されるリソース利用効率と性能，および予想されるRPMの変動に基づき RPM_{max} の値を設定する必要がある．

6. おわりに

本研究では，コンテナ仮想化技術，K8sおよびサービスメッシュを用いたエッジ基盤において，協調型負荷分散の実装手法を提案した．エッジのアプリケーションへのRPSをモニタリングし，エッジの処理限界を超えるRPSが発生した場合に近接するリソースに余力あるエッジサーバまたはクラウドへリクエストを転送するようにサービスメッシュ設定を動的に変更するコントローラを実装した．実験により，アプリケーションの応答時間，エラー率が改善さ

ⓒ 2021 Information Processing Society of Japan

れることが確認された．

6.1 今後の課題

本研究では２つのエッジサーバだけを想定し，トラフィック転送先の近接エッジサーバは常に十分なリソースがある前提で負荷分散手法の提案と実装評価を行った．しかし，エッジサーバが近距離に多数配置される場合は，近接エッジサーバが複数存在しうる．また，近接エッジサーバ自体も周辺端末からリクエストを受信しリソースが多く消費されている可能性がある．今後は，近接サーバが複数存在し各エッジサーバの空きリソースが異なる状況においても，それぞれの近接エッジサーバの負荷 (リクエスト数) と空きリソースを収集しリクエストの転送先を決定する手法が求められる．

また，本研究の提案手法では，エッジサーバの RPM を Traffic Distribution Controller が取得してからエッジサーバの Ingress における負荷分散割合の設定変更が反映されるまでにタイムラグが発生することが確認された．タイムラグを短縮するには RPM 値の取得間隔を短くする必要があるが，取得間隔を短くするほどモニタリングシステムへの負荷が大きくなる．取得間隔を短縮しつつシステムへの負荷集中を避ける分散型モニタリングシステムが期待される．

加えて，本研究では RPM 値を用いた動的負荷分散手法を提案したが，アプリケーションや用途によって負荷分散に用いるべき指標は異なる．RPM 値の他にも，応答時間やエラー率を指標とした分散手法の適用が今後の課題である．

参考文献

[1] Shi, W., Pallis, G. and Xu, Z.: Edge computing [scanning the issue], *Proceedings of the IEEE*, Vol. 107, No. 8, pp. 1474–1481 (2019).

[2] Liu, S., Liu, L., Tang, J., Yu, B., Wang, Y. and Shi, W.: Edge computing for autonomous driving: Opportunities and challenges, *Proceedings of the IEEE*, Vol. 107, No. 8, pp. 1697–1716 (2019).

[3] Automotive Edge Computing Consortium: General Principle and Vision White Paper Version 3.0, `https://aecc.org`.

[4] Beraldi, R., Mtibaa, A. and Alnuweiri, H.: Cooperative load balancing scheme for edge computing resources, *2017 Second International Conference on Fog and Mobile Edge Computing (FMEC)*, IEEE, pp. 94–100 (2017).

[5] Villari, M., Fazio, M., Dustdar, S., Rana, O. and Ranjan, R.: Osmotic computing: A new paradigm for edge/cloud integration, *IEEE Cloud Computing*, Vol. 3, No. 6, pp. 76–83 (2016).

[6] Aslanpour, M. S., Toosi, A. N., Cicconetti, C., Javadi, B., Sbarski, P., Taibi, D., Assuncao, M., Gill, S. S., Gaire, R. and Dustdar, S.: Serverless edge computing: vision and challenges, *2021 Australasian Computer Science Week Multiconference*, pp. 1–10 (2021).

[7] Kaur, K., Garg, S., Kaddoum, G., Ahmed, S. H. and Jayakody, D. N. K.: En-osco: energy-aware osmotic computing framework using hyper-heuristics, *Proceedings of the ACM MobiHoc Workshop on Pervasive Systems in the IoT Era*, pp. 19–24 (2019).

[8] Sharma, V., Jayakody, D. N. K. and Qaraqe, M.: Osmotic computing-based service migration and resource scheduling in Mobile Augmented Reality Networks (MARN), *Future Generation Computer Systems*, Vol. 102, pp. 723–737 (2020).

[9] Ogbuachi, M. C., Reale, A., Suskovics, P. and Kovács, B.: Context-Aware kubernetes scheduler for edge-native applications on 5g, *Journal of Communications Software and Systems*, Vol. 16, No. 1, pp. 85–94 (online), DOI: 10.24138/jcomss.v16i1.1027 (2020).

[10] Santos, J., Wauters, T., Volckaert, B. and De Turck, F.: Towards network-aware resource provisioning in Kubernetes for fog computing applications, *2019 IEEE Conference on Network Softwarization (NetSoft)*, IEEE, pp. 351–359 (2019).

[11] 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)*, IEEE, pp. 122–1225 (2019).

[12] Xiaojing, X. and Govardhan, S. S.: A Service Mesh-Based Load Balancing and Task Scheduling System for Deep Learning Applications, *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, IEEE, pp. 843–849 (2020).