動的スケジューリングによるマイクロサービスの実行最適化

仮屋　郷佑[1]　坂本 龍一[2]　中村 宏[3]

概要：Web アプリケーションは同時に大量のアクセスを処理すること，また頻繁にアップデートを行うことが要求されている．また，コンテナや仮想化などの技術の登場もあり，Web アプリケーションのアーキテクチャとしてマイクロサービスが注目されている．マイクロサービスは複数のサービスから構成される．サービスのサーバーへのマッピングを静的に考えることによる実行最適化は多く行われているが，サーバーに新しくサービスをデプロイするには数分オーダーの時間が必要になる点，リクエストは急激に上昇することがある点から動的なスケジューリングによる実行最適化が必要である．本研究では共有資源を考慮する同的なレプリカ選択によるスケジューリング手法を提案する．そしてその有用性を検証するために 2 種類のデータセットを作成し，シミュレーションを行った．

1. はじめに

近年マイクロサービスアーキテクチャ [1][2] の利用が急速に進んでいる. マイクロサービスアーキテクチャではアプリケーションを小さなサービス=マイクロサービスに分割し，それぞれを異なるプロセスとして動作させる．従来の開発手法であるモノリシックでは，全ての機能がまとめて 1 つのプロセスとして同一サーバー上で実行される. 一方マイクロサービスアーキテクチャでは各サービスは独立したコンテナとして別々に存在し，それらがそれぞれの通信 API を利用して連携することにより一つのアプリケーションをなす.

マイクロサービスアーキテクチャはそれぞれのサービスごとに独立して開発を行うことができ，それぞれに適した言語，フレームワークを用いることができるため，チーム開発を促進するというメリットがある．一方で，多数のコンテナにより莫大なサーバーリソースが消費されるため，サービスは複数のサーバーに分散して存在し，同一のサーバーに配置されたサービスの間での資源の競合が課題となる.

現在マイクロサービスアーキテクチャは，Web サービス分野などで多く取り入れられており，この分野では高い負荷耐性を担保することが重要である．特に新しいサービスのリリース，イベントの開催時は要求されるリクエストが増加し，各サービスの処理量が増加し応答性能を担保することが難しい．高負荷時におけるマイクロサービスアーキテクチャでは，それぞれのサービスが行う処理内容や処理量が異なること，到着するリクエスト種別に偏りがあることなどによって，特定のサービスがボトルネックとなりリクエストの応答性能が悪化する.

これに対応するため，マイクロサービスアーキテクチャにおける高負荷時の対応として，ボトルネックとなるサービスのコンテナをレプリカとして複数作成することにより同種のコンテナを複数用意することで負荷分散を行う方法が用いられている．コンテナが複数存在するサービスにおいて，処理を実行するレプリカはラウンドロビンにより決定することにより，負荷分散を行うとする．一方で，物理サーバーに対するコンテナの初期配置の偏りや，各サービスで行われる処理内容の違いから特定のサーバーに対する資源利用の要求が偏ることがある．この結果，コンテナが利用できる資源量に制限が生じ，結果として応答性能が悪化する.

そこで本研究では，共有資源の競合を考慮した動的なレプリカ選択によってマイクロサービスアーキテクチャの高負荷時における応答時間の改善を行うことを目標とする.

2. レプリカによるスケーリングと課題

2.1 レプリカによるロードバランシング

マイクロサービスアーキテクチャによるアプリケーションは機能ごとにサービスとして分割されており，別個のプロセスとして実行され，それぞれが通信 API を用いて疎に結合される．それぞれのサービスは異なる機能を担当しているため各サービスで行われる処理内容が異なる．また各サービスが呼ばれる回数などが異なる．そのため，アプ

[1] 東京大学工学部計数工学科
[2] 東京工業大学
[3] 東京大学大学院情報理工学系研究科

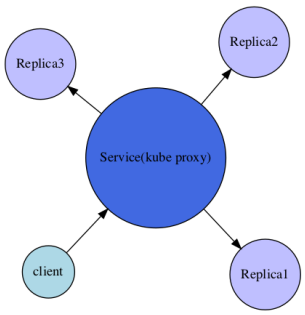

図 1 Kubernetes におけるロードバランシング

リケーションに多量のリクエストが到着した際に，特定のサービスに対する処理要求が集中する．このサービスがボトルネックとなり応答性能が著しく悪化する．

モノリシックサービスの場合はサービス全体を複数用意することでスケールしていたが，マイクロサービスアーキテクチャの場合はボトルネックとなりうるサービスのコンテナのみを複数用意することでスケールする．この複数用意されたコンテナをレプリカと呼ぶ．

レプリカを１つ以上もつサービスが呼び出された際には，複数のコンテナの中から実際に処理を実行するコンテナをラウンドロビンやランダムによって選択することにより負荷分散を行う．

マイクロサービスアーキテクチャを用いたアプリケーションを運用する際にはオープンソースコンテナプラットフォームの Kubernetes[3] が普及している．Kubernetes においてサービスにレプリカがある場合，同種類のサービスのコンテナの集合で一つの Service として一まとまりとして扱われる．そして，それぞれのコンテナが IP アドレスを持ちつつ，Service として一つの IP アドレスを保持する．これによりクライアントはバックエンドとなるコンテナを呼びだす際に Service の IP アドレスのみ知っていればよく，コンテナそれぞれのの IP アドレスを管理する必要がない． Service の IP アドレスが呼び出されると，そのサーバー上にあるプロキシがラウンドロビンやランダム選択によりコンテナの IP アドレスに変換することでロードバランシングが行われる (図 1)．

2.2 物理サーバーにおけるサービスの配置と資源競合

マイクロサービスアーキテクチャでは多数のサービスにより構成されており，それぞれがコンテナとして別々に存在するため，莫大なサーバーリソースが消費される．そのため，これらのサービスは複数のサーバーにまたがって配置される．

サービスごとに処理が異なるため，実行するために必要となるリソースの種類や量が異なる．そして，それらのサービスがネットワーク帯域やメモリなどのサーバー内で共有する資源を使用する際には同一サーバー上にあるコンテナ同士で共有資源の競合が生じる．同一のサーバー上に通信の回数や量が多いコンテナが複数存在する場合や，malloc などのシステムコールを多く実行するコンテナが複数存在する場合にはシステムソフトウェアが競合しコンテナの実行効率は悪化する．

2.3 マイクロサービスアーキテクチャのアプリケーションにおける負荷分散の課題

アプリケーションに到着するリクエストの種類や量は季節，時間帯，突発的なイベントの発生などにより変動する．到着するリクエストの種類によって呼び出されるサービスの種類や回数が異なるため，それぞれのサービスの共有資源の使用量や負荷は変化することとなる．よってボトルネックとなるサービスは時間と共に変化する．

マイクロサービスアーキテクチャを用いたアプリケーションのリクエスト量が増加し性能が悪化する際の対応として，ボトルネックとなるサービスのレプリカを作成する方法や，低負荷なサービスのコンテナと高負荷なサービスのコンテナを同一のサーバーに置くようサービス配置変更を行うという手法が考えられる．一方でレプリケーションを新たに作成したりサービス配置の変更を行うためにはコンテナ生成を新たに行うことが必要となり，コンテナイメージを外部からコピーする場合では，数分を要する．よってリクエストがコンテナ生成に要する時間より短い期間に急激に上昇した際にはこれらの手法によってアプリケーションのリクエストの応答性能を改善することはできない．

3. 研究目的とアプローチ

マイクロサービスアーキテクチャを用いたアプリケーションにおいて，高負荷時における応答時間の改善を本研究の目標とする．前章において説明したように，マイクロサービスにおけるアプリケーションは複数のサーバー上で動作し，共有資源の競合が起こる点，レプリカを新規に作成する手法やコンテナのサーバーへの配置を変更する手法は少なくとも数分を要するという点から，本研究では，各サーバーの状態，リクエストの実行に必要なリソースを考慮した動的なレプリカ (コンテナ) 選択によるスケジューリングを行うことにより応答時間の改善を目指す．すなわち，サーバー上にあらかじめ十分なレプリカを生成しておき，各サーバーの資源利用量を把握したうえで最適なレプリカ選択を動的に行う．

4. 動的なレプリカ選択手法

本章ではマイクロサービスアーキテクチャのアプリケーションに対するレプリカ選択によるスケジューリング手法の概要について説明する．4.1 において本研究における資源競合モデルを説明する．4.2 において本研究においてスケジューリングする際に使用できる情報に関する仮定につ

いて述べる．4.3 においてスケジューリングをどの段階において行うかを述べる．最後に 4.4 において本研究の提案手法を具体的に説明する．

4.1 資源競合の考え方

本研究では共有資源を使用する処理として，ネットワーク等の I/O やメモリ管理等の OS が行う処理を考える．本研究では簡単のため，コンテナがサーバーにおいて共有するリソースはコアとコア以外の一種類の共有資源があるとし，サーバーは複数のコアと一つの共有資源を持つものとしてモデル化する．そして，全てのサーバーがが持つコア数と，持つコアの周波数をは同じであり，サーバーに配置されるコンテナの数はコア数以下であるとした．そして，以上のような状況におけるの資源競合を，コンテナがコアに割り当てられ，処理を実行する際の時間を独立資源使用時間と共有資源使用時間の 2 種類に分けることで表現する．

独立資源使用時間

コンテナが独立資源使用時間としてコアで処理を実行する際は，コンテナはその割り当てられたコアのみを占有する．同一サーバー上ではサーバーに存在するコア数まで同時に独立資源使用時間を実行可能とする．

共有資源利用時間

コンテナが共有資源使用時間としてコアで処理を実行する際は，コンテナはその割り当てられたコア一つを占有した上で，共有資源も占有する．本研究においては，サーバーは共有資源を一つ持つものとして考えるため，同一サーバー内で同時に共有資源使用時間を実行できるコンテナは一つのみとなる．

4.2 本研究における仮定

本研究においてサービスで実行される処理を完了するまでに必要な独立資源使用時間と共有資源使用時間はプロファイル等を用いることで事前に与えられているものと考える．さらに，各コンテナの待ちキューに存在する処理の独立資源使用時間と共有資源使用時間はモニタリングアプリ等を用いることで把握することが可能であるとする．

4.3 スケジューリングを行うタイミング

マイクロサービスアーキテクチャのアプリケーションにおいてリクエストは各サービスが自身で処理を実行しつつ，他のサービスを呼び出すことで他のサービスと連携しながら実行される．本研究のレプリカの選択はサービスが呼び出された際に行うものとする．Kubernetes における Service の IP アドレスからレプリカへの IP アドレスに変換する際の変換手法が本提案手法の実装される部分となる．

4.4 提案手法

本研究では共有資源を考慮したレプリカ選択手法を提案する．以下のように変数を定義する

$T1sum_c$: コンテナ c の待ちキュー上の処理の合計独立資源使用時間

$T2sum_c$: コンテナ c の待ちキュー上の処理の合計共有資源使用時間

$\mathbf{C_m}$: サーバー m 上にあるコンテナの集合

$\mathbf{C_s}$: サービス s を実行するコンテナの集合

m_c : コンテナ c が配置されたサーバー

さらに，次に実行する処理の独立資源使用時間を $T1next$，共有資源使用時間を $T2next$ とし，次に実行する処理を行うサービスを s_{next} とする．

本提案手法では，処理を次に実行するサービスは

$$\min_{c \in \mathbf{C_{s_{next}}}} \max(T2next + \sum_{d \in \mathbf{C_{m_c}}} T2sum_d, T1next + T1sum_c)$$

を満たす c を選択する．すなわち次に実行する処理を行うサービスのコンテナ集合 ($\mathbf{C_{s_{next}}}$) のうち，次に実行する処理の共有資源使用時間 $T2next$ とそのコンテナの存在するサーバー上のコンテナの待ちキューに存在する処理の共有資源使用時間の合計 $\sum_{d \in \mathbf{C_{m_c}}} T2sum_d$ の和と，次に実行する処理の独立資源使用時間 ($T1next$) とそのコンテナの待ちキューに存在する処理の独立資源使用時間の合計 ($T1sum_c$) の和の max が最小となるコンテナを選択する．本手法では共有資源使用時間はサーバー上で一つのコンテナのみが実行可能であるためサーバー上のコンテナ集合での和を考えた．

5. データセットと予備評価

本研究では提案手法の効果をシミュレーターを用いて検証する．そのため，シミュレーターで使用するためのマイクロサービスアーキテクチャによるアプリケーションのデータセットを作成した．すなわち，複数のサービスから構成されるアプリの各サービスに関して，独立資源使用時間と共有資源使用時間を定義する．また，サービス間の依存関係，リクエストの種別を合わせて定義する．本章では作成した 2 種類のデータセットについて述べる．

5.1 シンプルなデータセット

本データセットは共有資源の競合が起こりやすいように設計した．本データセットのサービス構成と本アプリケーションに対するリクエストについて説明する．

5.1.1 サービスと各サービスの処理内容

本データセットは 3 種類のサービス Service1，Service2，Service3 からなり各サービスで行われる処理は一種類のみとした．全てのサービスにおける処理に必要なコアの割り

当て時間は同じであるが，Service1 と Service2 において共有資源の競合が発生するように設計した．各サービスで行われる処理の具体的な内容は表 1 に示す．

表 1 シンプルなデータセットにおける各サービスで行われる処理

サービス	独立資源使用時間 (μs)	共有資源使用時間 (μs)
Service1	5	15
Service2	5	15
Service3	20	0

5.1.2 リクエスト内容

本データセットにおけるリクストは Request1, Request2, Request3 の 3 種類用意した．Request1，Request2，Request3 はそれぞれ Service1，Service2，Service3 で処理を実行してリクエストを完了するものとした．各リクエストにおける呼び出される述べサービス数を表 2 に示す．

表 2 シンプルなデータセットにおけるリクエスト

リクエスト	呼び出されるされるのべサービス数
Request1	1
Request2	1
Request3	1

5.2 SocialMedia による実測ベースのデータセット

本データセットは実アプリケーションを計測することにより作成した．計測対象としては Social Network を用いた．本章では Social Network の概要，計測方法，本データセットのサービス構成，本アプリケーションに対するリクエストについて説明する．

5.2.1 Social Network とは

Social Network は end-to-end で実装されたマイクロサービスアーキテクチャによるアプリケーションのベンチマークである DeathStarBench[4] のうちの一つである．Social Network はソーシャルメディアアプリケーションであり，以下のリクエストが実装されている．

- テキストをポストする
- ポストを読む
- ユーザーのタイムラインを読む
- フォローするユーザーのレコメンデーションを見る
- ユーザーやポストを検索する
- ユーザーとして登録，ログインする
- ユーザーをフォロー，アンフォローする

本アプリケーションは compose-post-service，media-service，nginx-web-server，post-storage-service，social-graph-service，text-service，unique-id-service，url-shorten-service，user-mention-service，user-service，user-timeline-service，write-home-timeline-service の 12 種類のサービスから構成される．

本研究ではテキストをポストする compose-post とユーザーのタイムラインを読む read-user-timeline という 2 種類のリクエストのサービスの呼び出される順序とそれぞれサービスで行われる処理の計測を行った．

5.2.2 計測方法

本計測では compose-post， read-user-timeline，二つのリクエストの実行時間と各リクエストのみを実行した際の各サービスのシステムモードで実行した時間とユーザーモードで実行した時間を計測した．

計測に使用したツール

各リクエストの実行時間の計測には Social Network に埋め込まれていた Jaeger[5] を用いて行った．Jaeger はオープンソースの end-to-end の分散トレーシングサービスである．各サービスのシステムモードで実行した時間とユーザーモードで実行した時間は Linux の/proc/stat から取得した．

実験内容

本計測では単一のサーバー上に docker-compose を用いて Social Network を構成するサービスをレプリカなしで配置した．本計測で使用したサーバーのスペックを表 3 に示す．計測は 1)．docker-compose で Social Network をサーバー上に配置する．2)．/proc/stat で各サービスのリクエスト生成前のプロファイルをとる．3)．10 req/sec でリクエストを 20 秒生成する．4)．/proc/stat で各サービスのリクエスト処理後のプロファイルをとる．5)．Jaeger を用いて各リクエストを完了するまでに必要な所要時間を計測するという流れで行った．2 と 4 の結果を比較して各サービスで実行されるシステムモードでの実行時間とユーザーモードでの実行時間を取得した．

表 3 評価環境

CPU	Intel(R) Xeon(R) Gold 6240R CPU
メモリ	DDR4 64GB
OS	Ubuntu 20.04

5.2.3 サービスと各サービスの処理内容

本データセットは Social Network に含まれる compose-post-service，media-service，nginx-web-server，post-storage-service，social-graph-service，text-service，unique-id-service，url-shorten-service，user-mention-service，user-service，user-timeline-service，write-home-timeline-service の 12 種類のサービスの実行時間内訳から構成される．Jaeger で計測した時間は独立資源使用時間と共有資源使用時間共に含む形で計測される．さらに，各サービスで実行される処理は複数存在する．よって，それぞれのサービスで行われる処理の独立資源使用時間と共有資源使用時間は処理に必要な時間，サービスのシステムモードで実行した時間，ユーザーモードで実行した時間から以下のように計算した．

$$\text{独立資源使用時間} = \frac{T * S_s}{S_s + U_s}$$
$$\text{共有資源使用時間} = \frac{T * U_s}{S_s + U_s}$$

T : Jaeger で計測された処理に必要な時間

S_s : 処理が行われるサービスのシステムモードでの実行時間

U_s : 処理が行われるサービスのユーザーモードでの実行時間

実測データセットを構成する各サービスにおける処理の独立資源使用時間と共有資源使用時間のグラフを図 2 に示す.

5.2.4 リクエスト内容

本データセットにおけるリクエストは compose-post と read-user-timeline の 2 種類を作成した．各リクエストにおける呼びだされるのべサービス数を表 4 に示す.

表 4 実測データセットにおけるリクエスト

リクエスト	呼びだされるのべサービス数
read-user-timeline	8
compose-post	34

compose-post と read-user-timeline の概形をそれぞれ図 3，図 4 に示す．後続のサービスで行う処理は，一つ前の処理の完了時にサービスが呼びだされるものとした.

6. シミュレータの設計

本研究では提案手法を検証するため，複数サーバー上で動作するマイクロサービスアーキテクチャのアプリケーションにおける，リクエストの応答時間をシミュレートするシミュレーターを作成した.

6.1 シミュレータの入力

シミュレーションを行う際の入力は以下とした.

- サーバーの数と各サーバーのコア数
- サービスのコンテナの数
- 各サーバーへのコンテナの配置
- サービスで行われる処理の独立資源使用時間と共有資源使用時間
- リクエストを実行する際の処理の順序
- 到着するリクエストの種類と分布
- シミュレーションを行う時間

6.2 シミュレータの出力

シミュレーションの出力は以下とした.

- クエリの開始時間と終了時間
- 各時間における各コンテナの待ちキューにある処理の独立資源使用時間と共有資源使用時間ごとの合計

Algorithm 1 シミュレーターの流れ

```
while t < t_limit do
  リクエストの生成
  全ループに終了した処理の後続の処理するサービスを呼び出だす
  for m ∈ M do
    m 内にあるコンテナをコアにマップする
    for k ∈ K_m do
      if F_m and 実行中の処理が共有資源を必要とする then
        k で共有資源使用時間をΔt 実行
        F_m = False
      else
        k で独立資源使用時間をΔt 実行
      end if
    end for
    F_m = True
  end for
  t+ = Δt
end while
```

6.3 シミュレータの内部

シミュレータの内部の動作をアルゴリズム 1 に示す．ここで各変数は以下とする.

t : 経過時間

t_{limit} : シミュレーション時間

Δt : 更新時間

$\mathbf{M}$: シミュレーション環境に存在するサーバーの集合

$\mathbf{K_m}$: サーバー m 上のコアの集合

F_m : サーバー m 上の共有資源が使用可能か

コンテナのコアへのマッピングをラウンドロビンで行うことで共有資源を使用する優先権がコンテナ間で偏りのでないようにした.

7. 実験内容

本研究ではに 2 つのデータセットにおいて，ラウンドロビン，提案手法それぞれでレプリカを選択した際のリクエストの応答時間を計測する実験を行った．本章ではそれぞれの実験について説明する.

7.1 シンプルなデータセット

7.1.1 サービスのサーバーへのマッピング

Service1，Service2，Service3 の各サービスにレプリカを一つずつ作成し (表 5)，コア数が 2 のサーバーを 3 台用意した．アプリケーションを頑健なものとするために，同一サービスのコンテナは同じサーバーに配置しないことから配置は表 6 のように配置した.

7.1.2 到着するリクエスト

Request1，Request1，Request1 共に 500μs おきに同一のポアソン分布

$$P(X = k) = \frac{e^{-\lambda}\lambda^k}{k!}$$

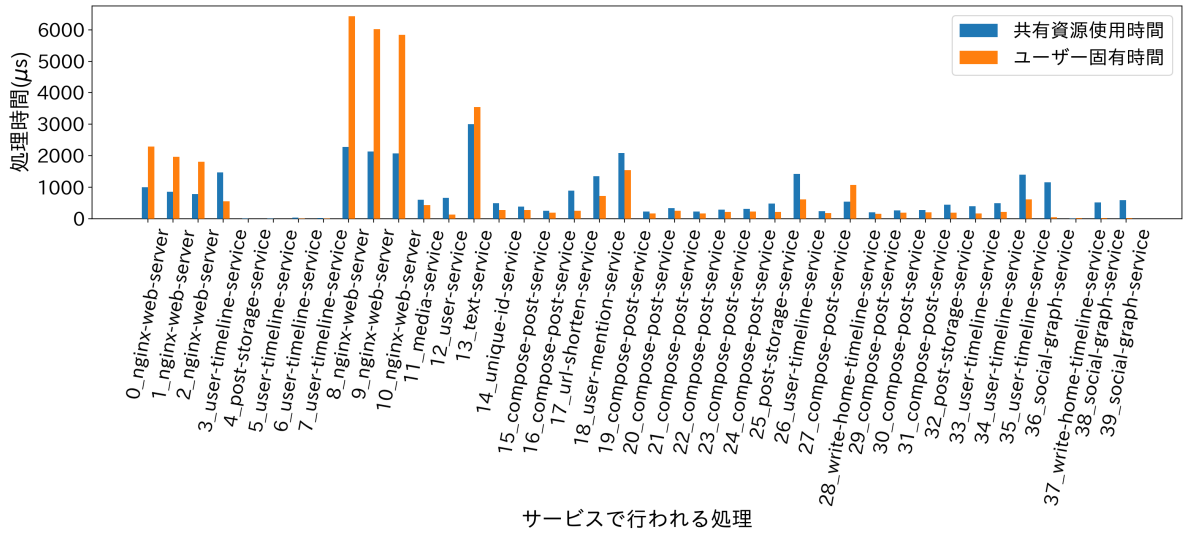

図 2 実測データセットにおける各サービスで行われる処理の独立資源使用時間と共有資源使用時間

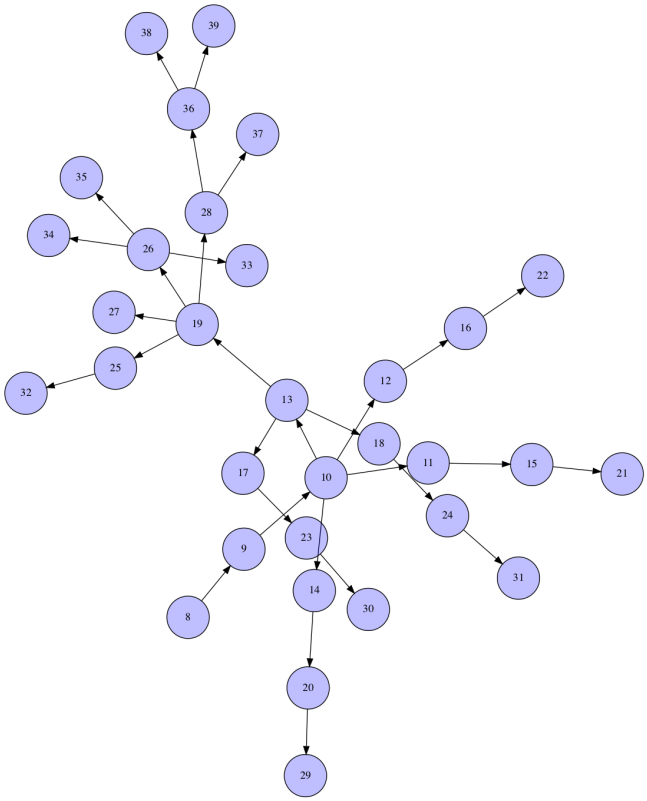

図 3 compose-post の処理の流れ

表 5 シンプルなデータセットにおける各サービスのコンテナ数

サービス	レプリカ数
Service1	2
Service2	2
Service3	2

に従って到着するものとした．

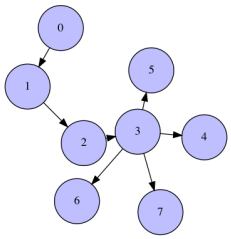

図 4 read-user-timeline の処理の流れ

表 6 シンプルなデータセットにおける各コンテナの配置

サーバー	サービス
サーバー 1	Service3, Service1
サーバー 2	Service1, Service2
サーバー 3	Service2, Service3

7.1.3 実験の詳細

シミュレータの $\Delta(t)$ を $1\mu s$ に設定し，複数の λ に対して $50{,}000\mu s$ 秒間シミュレーションを行い，リクエストの平均応答時間を求めた．

7.2 実測ベースのデータセット

7.2.1 レプリカ作成

実測ベースのデータセットによるアプリケーションでは，レプリカを作成するため，各リクエストに対して次の操作を複数回行った．一つのリクエストを完了するまでにサービス s で実行する処理時間 (共有資源使用時間と独立資源使用時間の和) の合計を T_s とし，サービス s のコンテナの数を R_s とする．このデータセットに存在するサービス全ての集合を $\mathbf{S}$ とする．リクエストに対し，サービスのコンテナを増やした際に

$$\max_{s \in \mathbf{S}} \frac{T_s}{R_s}$$

を最小値を更新するようなサービスのコンテナを追加する．compose-post-service を完了するまでに呼び出されるサービスは 12 種類より 6 回，read-user-timeline を完了するまでに呼び出されるサービスは 3 種類のため 2 回行った．結果を表 7 に示す．

表 7 実測データセットにおける各サービスのコンテナ数

サービス	コンテナ数
nginx-web-server	6
user-timeline-service	3
compose-post-service	3
text-service	2
post-storage-service	2
user-service	1
unique-id-service	1
write-home-timeline-service	1
media-service	1
social-graph-service	1
user-mention-service	1

7.2.2 サービスのサーバーへのマッピング

コア数が 4 のサーバーを 6 台用意した．用意したサーバー全ての集合を $\mathbf{M}$，サーバー m 上で動くコンテナの集合を $\mathbf{C_m}$，コンテナ c のサービスの種類を s_c とする．

$$\max_{m \in \mathbf{M}} \sum_{c \in \mathbf{C_m}} \frac{T_s}{R_{s_c}}$$

を最小にするマッピングを考え，これの解を貪欲方で近似した．結果として得られたマッピングを表 8 に示す．

7.2.3 到着するリクエスト

compose-post， read-user-timeline 共に 200000μs おきに同一のポアソン分布

$$P(X = k) = \frac{e^{-\lambda}\lambda^k}{k!}$$

に従って到着するものとした．

7.2.4 実験の詳細

シミュレータの $\Delta(t)$ を 1μs に設定し，複数の λ に対して 10,000,000μs 秒間シミュレーションを行い，リクエストの平均応答時間を求めた．

8. 実験結果と考察

本章では実験の結果を示す．結果のうち，シミュレーション中に生成された全リクエストのうち 20% 以上のリクエストを完了できていない λ は各グラフにおいて除いて表示した．

表 8 実測データセットにおける各コンテナの配置

サーバー	サービス
サーバー 1	text-service, nginx-web-server, user-service, user-timeline-service
サーバー 2	text-service, nginx-web-server, unique-id-service, user-timeline-service
サーバー 3	compose-post-service, write-home-timeline-service, nginx-web-server
サーバー 4	compose-post-service, nginx-web-server, url-shorten-service, post-storage-service
サーバー 5	compose-post-service, nginx-web-server, media-service, user-timeline-service
サーバー 6	social-graph-service, user-mention-service, nginx-web-server, post-storage-service

8.1 シンプルなデータセット

λ の値を変化させた際の Request1 の平均応答時間の変化を図 5 に示す．横軸は到着率 λ，縦軸は平均応答時間である．この結果，ラウンドロビンの場合 $\lambda = 27$ までしかリクエストを返すことができないが，提案手法では $\lambda = 31$ までリクエストを返すことができることが確認できた．また，各サービスに滞留している累積リクエスト量を確認しリソース不足となっているサービスの有無を判断する．$\lambda = 23$ においてラウンドロビンと提案手法それぞれによりレプリカを選択したときのサーバーごとの累積した処理量（共有資源使用時間と独立資源使用時間それぞれに分けて表示している）とシミュレーションの経過時間のグラフをそれぞれ図 6，7 に示す．本グラフの縦軸は各サービスごとの処理待ちリクエスト量を示し，横軸は経過時間を示している．図 6 に示すラウンドロビンの場合経過時間に沿ってサービスに対する処理待ちリクエスト量が増加することが確認できる．よって，2 つのサービスが資源不足となり，リクエストを正常にさばけない状態になっていることが確認できる．提案手法である図 7 では処理待ちリクエスト量に増加は見られないため，資源を有効に利用することができ，リクエストを正しくさばけていることが確認できる．

8.2 実測ベースのデータセット

λ の値を変化させた際の compose-post と read-user-timeline それぞれの平均応答時間の変化を図 8，9 に示

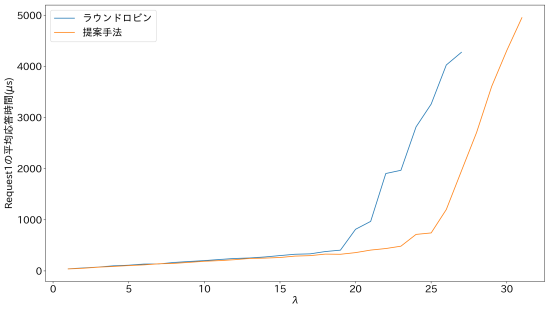

図 5 シンプルなデータセットにおいて提案手法とラウンドロビンそれぞれによりレプリカを選択した際の Request1 の平均応答時間

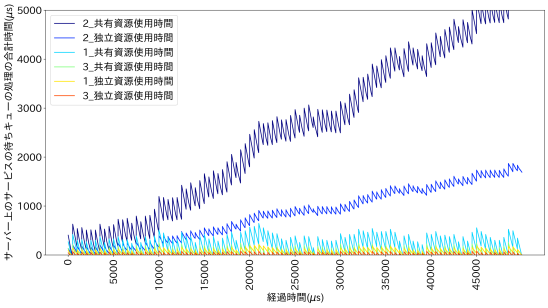

図 6 シンプルなデータセットにおいて $\lambda = 23$ とした際にラウンドロビンによりレプリカ選択をした際の各サーバーの累積処理量

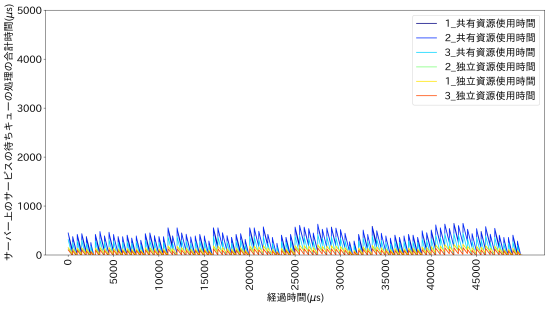

図 7 シンプルなデータセットにおいて $\lambda = 23$ とした際に提案手法によりレプリカを選択した際の各サーバーの累積処理量

す．こちらのデータセットにおいては応答時間の改善は見られなかった．$\lambda = 36$ の際の累積した処理量（共有資源使用時間と独立資源使用時間それぞれに分けて表示している）とシミュレーションの経過時間のグラフを図 10，11，に示す．図 10 はコンテナごとに表示し，シミュレーションの終了時に累積時間が多いものを上から 15 個抽出した．図 11 はサーバーごとに累積した時間を算出した．図 10 から全てのコンテナにおける nginx-web-server の独立資源使用時間，共有資源使用時間が共に累積していることがわかる．そして図 11 から全てのサーバーにおいて独立資源使用時間，共有資源使用時間共に累積していることがわかる．

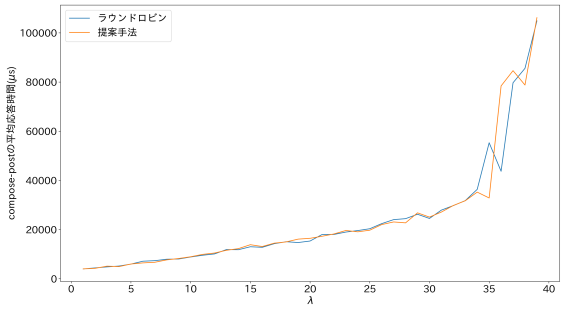

図 8 実測ベースのデータセットにおいて提案手法とラウンドロビンそれぞれによりレプリカを選択した際の compose-post の平均応答時間

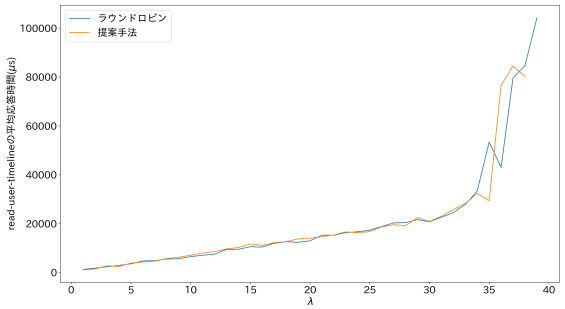

図 9 実測ベースのデータセットにおいて提案手法とラウンドロビンそれぞれによりレプリカを選択した際の read-user-timeline の平均応答時間

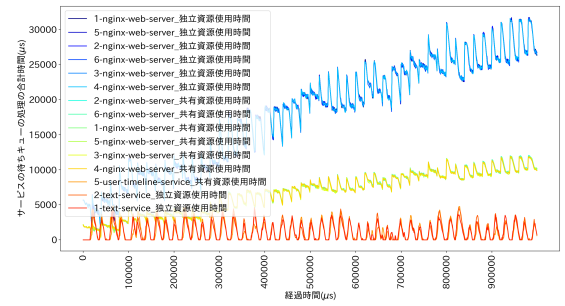

図 10 実測ベースのデータセットにおいて $\lambda = 36$ とした際の提案手法によりレプリカを選択した際の各コンテナの累積処理量

これらの結果は nginx-web-server における処理は他の処理に比べて共有資源使用時間，独立資源使用時間共に長い点と表 8 において全てのサーバーに nginx-web-server が配置している点に整合する．全てのサーバーが混雑しているため，レプリカ選択することに効果がなく，本提案手法が応答時間の改善に至らなかったと考えられる．

9. 追加実験

前章より，サーバー間で使用するリソースに偏りが生じないような際には，提案手法により応答性能の改善がに至

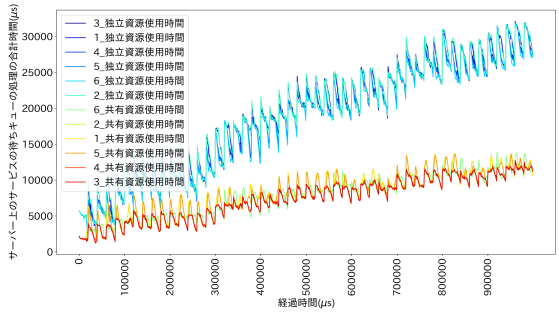

図 11 実測ベースのデータセットにおいて $\lambda = 36$ とした際の提案手法によりレプリカを選択した際の各サーバーの累積処理量

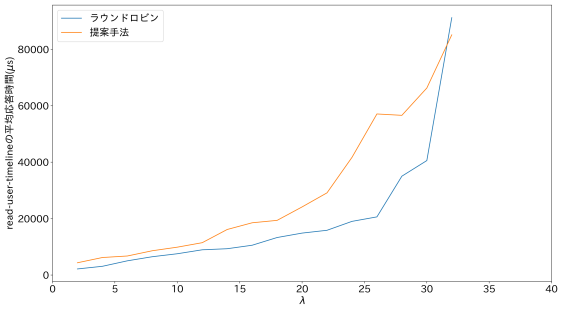

図 12 λ を変化させたアプリケーションにおいて提案手法とラウンドロビンそれぞれによりレプリカを選択した際の read-user-timeline の平均応答時間

らないことがわかった．これをうけて追加実験を行った．

9.1 実験内容

2 つの独立した Social Media をデプロイし，片方の Social Media には連続して多量のリクエストが到着するものとし，もう一方の Social Media に対するリクエストを徐々に増やすよ実験をおこなった．すなわち，リソース利用率が 2 つの Social Media 間で異なる場合の実験を行った．コア数が 4 のサーバーを 12 台用意し，実測データセットにより作成したアプリケーションを 2 つ作成した．それに対して 8.2 と同様の手法でレプリカの生成，サーバーへの配置を行った．リクエストは全て 8.2 と同様の分布に従うようにしたが，一方のアプリケーションでは $\lambda = 40$ に固定し，もう一方の λ を変化させた．

9.2 実験結果

それぞれのアプリケーションの read-user-timeline の応答時間をそれぞれ図 12，13 に示す．図 13 における横軸 λ は，もう一方のアプリケーションのリクエストの分布における λ である．シミュレーション中で生成された全リクエストのうち 20% 以上のリクエストを完了できていない λ は結果から除いた．結果から λ が変化させているアプリケーションでは平均応答時間は提案手法の方がラウンドロビンに比べて遅いが，80% 以上のリクエストを返すことのできる最大の λ の値は同じとなった．一方で，$\lambda = 40$ に固定したアプリケーションは提案手法によって応答性能が改善していることがわかる．図 12 において提案手法の方が応答時間が悪化しているが，これは，$\lambda = 40$ と固定したアプリケーションにおける処理に共有資源を多く使用したためと考えられる．

10. 関連研究

[6] ではサービスとサーバーの物理配置を考慮した配置最適を提案した．アプリケーションのプロファイル情報を基に通信コストと計算コストを考慮し物理サーバーと各サービスの配置割り当てを静的に提案し応答性能を改善する手

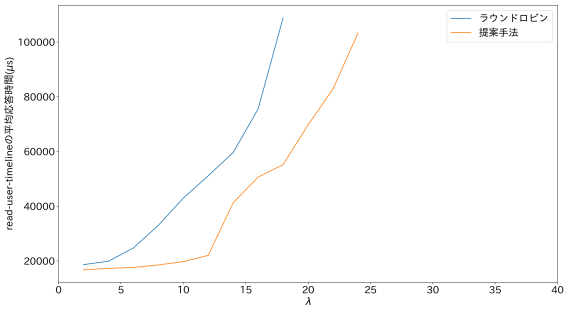

図 13 $\lambda = 40$ に固定したアプリケーションにおいて提案手法とラウンドロビンそれぞれによりレプリカを選択した際の read-user-timeline の平均応答時間

法を提案した．[7] では各サービスの電力特性とリクエスト要求を考慮し，各サービスを高性能ノードと低性能ノードに配置することで省電力化を行う手法を提案した．また，[8] では，各サービスと各サーバーの配置を全体で最適化したのち，各サーバー内でも細粒度な制御を行うことで省電力化を行う手法が提案されている．[9][10] では各サービスの実行時間，メモリ帯域，LLC キャッシュのセンシティブティを事前に分析し，ノード配置や CPU 構成の最適化を静的に行う手法を提案している．本研究では各サービスの特性を静的に分析したうえで，時々刻々と変化するリクエスト要求に対して動的に最適なサービスの選択を動的に行う点において新規性がある．

11. まとめ

本研究ではマイクロサービスアーキテクチャのアプリケーションの高負荷時の応答時間の改善を目的とした．アプローチとして，リクエストの分布は時間により変動すること，コンテナの配置の変更，新規作成は数分オーダーの時間がかかることより，サーバーのリソース使用量を考慮した動的なレプリカ選択手法を提案した．その後，データセットを 2 種類作成し，それらのデータセットを用いて本提案手法の効果をシミュレーターを用いて検証した．

結果として，シンプルなデータセットにおいては提案手法により応答性能を改善させることができたが，実測データセットでは改善が見られなかった．これは，一部のサービスに負荷が偏っており，全てのサーバーに高負荷となるサービスが配置されているため，サーバー間でのリソース使用量に偏りがなかったからであると考えられる．これを受けて，リクエスト量の異なるアプリケーションにおいて提案手法を検証したところ，混雑しているアプリケーションにおいて提案手法は応答性能を改善した．

本研究では実測ベースのデータセットを一種類のみ用意したが他のデータセットにおける検証も必要であると考える．また，本研究ではサーバーのコアの数，周波数は全て同一として考えたが，ヘテロな環境に対応す手法の提案も必要となる．さらに，今回は共有資源として OS によって管理されるもののみを考えたが，CPU キャッシュなどのリソースを考えることも必要となる．

謝辞　本研究の一部は JST さきがけ JPMJPR19M3 の助成を受けたものである．

参考文献

[1] Thönes, J.: Microservices, *IEEE software*, Vol. 32, No. 1, pp. 116–116 (2015).
[2] Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R. and Safina, L.: Microservices: yesterday, today, and tomorrow, *Present and ulterior software engineering*, Springer, pp. 195–216 (2017).
[3] Authors, K.: Production-Grade Container Orchestration (2020). `https://kubernetes.io/`.
[4] Gan, Y., Zhang, Y., Cheng, D., Shetty, A., Rathi, P., Katarki, N., Bruno, A., Hu, J., Ritchken, B., Jackson, B. et al.: An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, pp. 3–18 (2019).
[5] Authors, J.: Jaeger. `https://www.jaegertracing.io/`.
[6] 横山遼 and 坂本龍一 and 中村宏：通信経路を考慮したマイクロサービスの高速化検討，研究報告システムソフトウェアとオペレーティング・システム（OS），IPSJ, pp. 1–9 (2017).
[7] Hou, X., Liu, J., Li, C. and Guo, M.: Unleashing the Scalability Potential of Power-Constrained Data Center in the Microservice Era, *Proceedings of the 48th International Conference on Parallel Processing*, ACM, p. 10 (2019).
[8] Hou, X., Li, C., Liu, J., Zhang, L., Hu, Y. and Guo, M.: ANT-man: towards agile power management in the microservice era, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ACM, pp. 1–14 (2020).
[9] Li, Q., Li, B., Mercati, P., Illikkal, R., Tai, C., Kishinevsky, M. and Kozyrakis, C.: RAMBO: Resource Allocation for Microservices Using Bayesian Optimization, *IEEE Computer Architecture Letters*, Vol. 20, No. 1, pp. 46–49 (online), DOI: 10.1109/LCA.2021.3066142 (2021).
[10] Sriraman, A., Dhanotia, A. and Wenisch, T. F.: SoftSKU: optimizing server architectures for microservice diversity @scale, *Proceedings of the 46th International Symposium on Computer Architecture*, ACM, pp. 513–526 (2019).