Vol.2022-DPS-190 No.17
Vol.2022-CSEC-96 No.17
2022/3/10

コンテナ間のプログラムの複製および負荷分散によるノード群でのCPU使用効率の向上

飯島 貴政[1]　串田 高幸[1]

概要：マイクロサービスでのリクエストレートの急増による CPU 使用量の増加がある．既存のノードの追加手法では, 既存のノード上に未使用の CPU があるにも関わらず新規のノードを追加している．リクエストレートが急増した際にノードが追加されることでレスポンスタイムの遅延が発生する．本稿ではリクエストレートの急増時におけるレスポンスタイムを維持するために, 既存のノードにおける CPU 使用効率を向上させることでノードの追加頻度を減少させる手法を提案する．リクエストレートが CPU 使用量に与える影響を複数のマイクロサービスでリクエストレートと CPU 使用量の線形相関をそれぞれ計算し, 線形相関の値が高いマイクロサービスから負荷分散の対象とする．論文検索サービスのアクセスログをもとに生成したリクエストを定常時である 100[req/s], 急増時を 500[req/s] としてそれぞれのリクエストレートでアクセスした際のレスポンスタイムを計測した. 既存の手法では急増時のレスポンスタイムである約 120[ms] から定常時の約 50[ms] に復帰するまで約 50 秒間継続しているのに対し, 提案手法では約 10 秒で復帰することが可能となった．

1. はじめに

背景

マイクロサービスアーキテクチャではそれぞれのマイクロサービスのコードベースが独立している [1]. これにより, 一つの機能を最小限にとどめることにより柔軟かつスケーラビリティをもつアプリケーションを構築できる. 一つのマイクロサービスごとにリソースが個別に管理することができる．一方で, マイクロサービスに割り当てられた CPU, メモリはノード内での複数マイクロサービスで再利用または共有することができない．そのため, マイクロサービスがスケールする際には, 新たなコンテナを起動し，負荷分散を行う [2].

課題

本稿における課題はマイクロサービスアーキテクチャを使用している際，コンテナオーケストレーター, 例えば Kubernetes のデフォルトでのオートスケーラの設定を用いてノードを追加したとき, 追加されたノードが処理を開始するまでの間，レスポンスタイムに遅延が発生することである．例として，WEB 上で論文を検索および投稿できるクラウドアプリケーション (以降アプリケーション) を挙げる．このアプリケーションは 8[vCPU] を搭載するノード上にクライアントが論文を検索できる論文検索サービス，クライアントが論文を投稿できる論文投稿サービスがそれぞれマイクロサービスとして構成されている. 論文検索サービスは 4[vCPU], 論文投稿サービスは 3[vCPU] 必要とする．図 1 は論文検索サービスに 1 秒あたりで到達するリクエスト (以降リクエストレート) が急増した際の従来のスケール手法である．論文検索サービスに対してリクエストレートが急増し, 現状の 4[vCPU] では処理できない時は新規ノードを追加した後, 新規ノード上に論文検索サービスを追加する．図 2 にリクエストが増加した際のレスポ

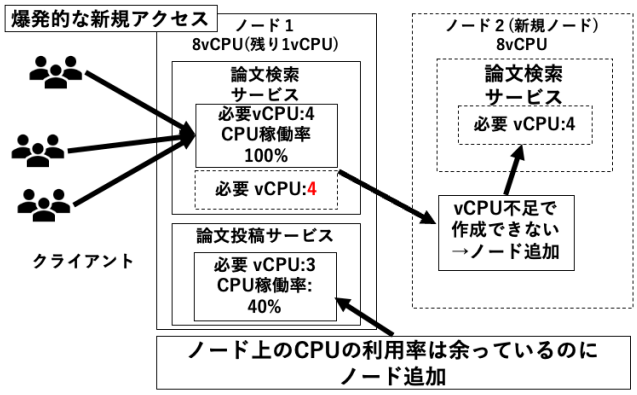

図 1　リクエストが急増した際の従来のスケール手法

[1] 東京工科大学大学院 バイオ・情報メディア研究科コンピュータサイエンス専攻
〒 192–0982 東京都八王子市片倉町 1404-1

ンスタイムを示す．ここでは実験で用いるリクエストレートとして，定常時を 100 [req/s], 急増時を 500 [req/s] とした．今回のアプリケーションは世界で最も有名な学会である IEEE の論文検索サイトである IEEE Xplore を模している.IEEE Xplore は約 500 万件の論文を掲載している [*1]．また, 毎月 25,000 件の文書が追加されている．各論文に共著者が 1 人いると仮定すると, 少なくとも著者と共著者がサイトにアクセスしチェックする必要がある．そのため, 本稿では検索ポータルへのアクセス数を 1 ヶ月に投稿される文書数の少なくとも 2 倍であると仮定している. 毎月 25,000 件の文書が投稿されているため 50,000 人が毎月のポータル閲覧ユーザーである．これは, 1 秒間に 0.019 回のリクエストに相当する．(50000/30(月間 30 日とする)x24(一日の時間)x3600(60 分 x60 秒)=0.019)．これに加えて, 毎週カンファレンスが開催されている場合, カンファレンスに参加したユーザーが論文を閲覧する．例として IEEE CCEM 2020 カンファレンスでは, 参加者は約 150 名であった．同様の規模のカンファレンスが毎週開催される場合, のべ 600 人が論文検索ポータルを利用する．また, 1 本の論文の投稿に付き, 多数の参考文献（本稿では 20 件を想定) にアクセスする．それに加えて, 引用回数が 10 回以上の論文については, 本稿では月間閲覧数を 2000 件とする．これは IEEE メトリクス解説サイトの例では, 月間閲覧数が 2000 件前後で推移しているためである．本稿では，平均ページビューアクセス数を 386.22 リクエスト/秒と定義する．ここから，定常時と急増時のピークを考慮し, 定常時 100 [req/s], 急増時 500 [req/s] とした．

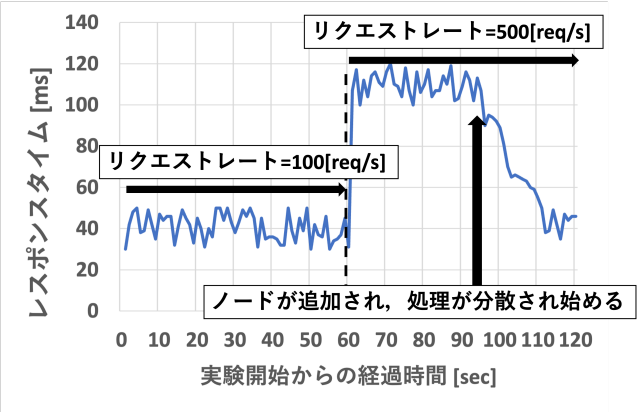

図 2 リクエストレートの急増時におけるレスポンスタイムの増加

図 2 では論文検索サービスへリクエストレートを 100 [req/s] で GET リクエストを 60 秒送信し,60 秒経過後に 500 [req/s] に増加させた際の論文検索サービスからのレスポンスタイムを示している．60 秒から 110 秒までの間の約 50 秒間は定常時に比べて，レスポンスタイムの遅延が最大 92 [ms] 発生している．その後追加されたノードおよび論文検索サービスが動作を開始し，負荷分散する．負荷分散が開始されたタイミングでレスポンスタイムは定常時に戻るように推移する既存のスケール手法ではリクエストレートが急増した際に一定期間, ここでは約 50 秒間は定常時のレスポンスタイムより大きく遅延が発生することが課題である．

[*1] IEEE Xplore https://ieeexplore.ieee.org/Xplore/home.jsp

各章の概要

第 2 章では本論文の関連研究について述べる．第 3 章では第 1 章で述べた課題を解決するシステムの提案について述べる．第 4 章では提案するシステムの実装と実験環境について述べる．第 5 章では提案するシステムの評価手法と分析手法について述べる．第 6 章では提案, 実験, 評価に関する議論について述べる．第 7 章では本研究の結果から得られた成果について述べる．

2. 関連研究

1 つのサービスあたりにマイクロサービスのインスタンスを複数個用意することでレスポンスタイムは早くなる [3]. マイクロサービスにリクエストが集中した際には平均レスポンスタイムが遅くなる. そのため, マイクロサービスを運用する際はスケールを用いてインスタンス数を増やし, レスポンスタイムを維持させる手法が一般的である．

マイクロサービスごとに割り当てられる CPU および RAM のオートスケールの閾値を強化学習によって自動設定し，Kubernetes のデフォルトのオートスケーラを利用した場合と比較して最大 20%レスポンスタイムが高速化した研究がある [4]. 本稿ではマイクロサービスごとのリクエストレートと連動して変動するメトリクスの値を取得することで各マイクロサービスにおける重要なメトリクスを特定するアプローチが近似している. この手法でマイクロサービスにおけるレスポンスタイムを 20%高速化できているものの, 閾値を自動設定するために多量の CPU 及び RAM を消費する強化学習モジュールを設置しているため, 学習を含めた場合最終的な CPU 及び RAM 消費量が Kubernetes の既存手法よりも多くなっている. 本稿においては, スケールアウト時におけるレスポンスタイムの遅延を抑制しつつ，アプリケーション全体で消費するリソースを節約するモチベーションとなっている．

3. 提案方式

本稿ではリクエストレートの急増時におけるレスポンスタイムを維持するために, 既存のノードにおける CPU 使用効率を向上させることでノードの追加頻度を減少させ, 同一ノード上のプログラムに互換性のある別のマイクロサービスで処理を代行することでレスポンスタイムを維持する手法を提案する．

本提案は以下の 5 ステップから構成される．

(1) アプリケーションにおける影響度の高いメトリクスの選定
(2) マイクロサービスにおける優先順位の算出
(3) 2 つの異なるマイクロサービスの類似度計測
(4) プログラムの共有による他サービスのプログラムの実行
(5) 他のサービスへのリダイレクトによる負荷分散機能

3.1 アプリケーションにおける影響度の高いメトリクスの選定

初めに, アプリケーションにおいてどのメトリクスがリクエストレートによる影響を受けているかを特定する．本稿では取得及び利用するメトリクスは本稿では CPU 使用量,RAM 使用量の 2 つとする. これにより, アプリケーションがリクエストレートが急増した際に最も影響を受けるメトリクスが順位付けする．ノード上にある複数のマイクロサービスの各 CPU 使用量,RAM 使用量およびメトリクスの取得時刻を毎秒取得する．これをもとにマイクロサービスごとにリクエストレートと CPU 使用量と RAM 使用量をそれぞれマッピングする. 以下の表 1 ににマイクロサービスから取得できるメトリクス (CPU 使用量) とリクエストレートごとへのマッピングを表した表を示す.

時刻 (s)	CPU 使用量 (milicore)	リクエストレート ([req/s])
1	1400	100
2	700	300
3	110	150
4	400	600
5	1000	140

表 1 毎秒メトリクスを取得

アルゴリズム 1 はメトリクス及びリクエストレートを取得した時間 time_t を用いてマイクロサービスから取得した複数メトリクスをそれぞれレクエストレートに紐付ける処理である.

入力としてメトリクスの種類, 例えば CPU 使用量,RAM 使用量が格納されたリスト ML, システム上での時間とリクエストレートのハッシュマップである R, システム上での時間と個別メトリクスの値をハッシュマップである M が用意されている．出力はメトリクスごとに計測されたリクエストレートと同一時刻に取得されたメトリクスを格納したハッシュマップ I とする．関数 $METRICS\_MAPPING\_BY\_REQ\_RATE$ ではじめに R から時刻 $time\_t$ とリクエストレートである req_rate を R の要素分取得する．その後，各メトリクスごとに $time\_t$ をキーに M からリクエストレートごとのメトリクスの取得値を I に挿入する．以下の表 2 にリクエストレートごとにマッピングされたメトリクス (CPU 使用量) の表を示す.

アルゴリズム 1 マイクロサービスから取得した複数メトリクスをそれぞれレクエストレートに紐付ける処理

Input:
ML: メトリクスの種類が格納されたリスト
R:(時間 t(time_t): リクエストレート (req_rate) のハッシュマップ,
M:(時間 t(time_t): 各メトリクスの取得値) のハッシュマップが全てのメトリクス分格納されているハッシュマップ
Output: I : {}: メトリクス名ごとにリクエストレート (req_rate) に紐付いたメトリクスの取得値を格納するハッシュマップ

```
1: function METRICS_MAPPING_BY_REQ_RATE(ML, R, M)
2:     ▷ R に格納されている時間をキーにメトリクスの取得値を
3:     ▷ リクエストレートと紐付ける
4:     for all time_t, req_rate ← R do
5:         for all metrics ← ML do
6:             I.insert(metrics, {req_rate: M[metrics][time_t]})
7:         end for
8:     end for
9:     return I
```

リクエストレート ([req/s])	CPU 使用量 (milicore)
100	1400
140	1000
150	110
300	700
600	400

表 2 リクエストレートごとにメトリクスを取得

3.1.1 相関係数および回帰直線の算出によるメトリクスの順位付け

アルゴリズム 2 はアルゴリズム 1 で取得した I を用いて, 各マイクロサービスの複数メトリクスから相関係数及び回帰直線を用いてマイクロサービスにおけるメトリクスの重要度を算出し，リスト PoM に保存する．その後算出した重要度をもとに昇順ソートし，最終的な各マイクロサービスのメトリクスの優先度となるリスト PoM_Final を作成する.

入力としてアルゴリズム 1 で作成したリクエストレートごとのメトリクスの値が格納されたリスト I がある，出力はマイクロサービスのメトリクスの優先度が昇順に保存されているリスト PoM_Final である．1 行目の function $SELECT\_METRICS$ を以下に説明する．3,4 行目では I よりメトリクスごとにリクエストレートをキー，各メトリクスの値をバリューを再帰的に取得する．また，各メトリクスの処理中におけるリクエストレートの集合を x, メトリクスの値の集合を y と示す．各メトリクスとリクエストレートにおいて x および y を用いて相関係数 ρ を計算する. このホワイトペーパおよびクラウドのメトリックの監視の研究では, 相関係数を用いて強い相関のみを得るためには相関係数 0.8 以下の相関を無視するべきとしている [5],[6]. 相関係数を 5 段階に人為的に分類した先行研究と同様の分類を行い ρ^2 が 0.64 以上のものを相関があるものとみなし,

Vol.2022-DPS-190 No.17
Vol.2022-CSEC-96 No.17
2022/3/10

アルゴリズム 2 各マイクロサービスの複数メトリクスから相関係数及び回帰直線を用いてマイクロサービスにおけるメトリクスの重要度を算出する

Input:
I: アルゴリズム 1 で作成したリクエストレートごとのメトリクスの値が格納されたリスト
Output: PoM_Final: メトリクスの重要度順位リスト

function SELECT_METRICS(I)
▷ 各メトリクスごとにリクエストレートとの相関係数を算出する
  for all $metrics \leftarrow I$ **do**
    for all $req\_rate, metrics\_value \leftarrow metrics$ **do**
      $x(array) \leftarrow req\_rate$
      $y(array) \leftarrow metrics\_value$
      ▷ 相関係数を算出する $\rho = \dfrac{\mathrm{cov}(X,Y)}{\sigma_x \sigma_y}$
      ▷ 相関係数の 2 乗が 0.64 以上のものを有効な相関とみなす
      if $\rho^2 > 0.64$ **then**
        ▷ 回帰直線を算出する
        $r = \dfrac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$
        $PoM.insert([metrics, r])$
      end if
    end for
  end for

  function GET_METRICS_RANK(PoM)
    $PoM\_Final = \{\}$
    $max = 0$
    for all $selected\_metrics_r \leftarrow PoM$ **do**
      for all $metrics, r\_value, index \leftarrow PoM\_Final$ **do**
        if $PoM\_Final.Length()is0$ **then**
          $PoM\_Final.insert([metrics, r\_value])$
        else$selected\_metrics\_r > r\_value$
$PoM\_Final[index-1].insert([metrics, r\_value])$
        end if
      end for
      return PoM_Final

相関メトリクスリスト PoM に挿入する．ここで作成された PoM を用いて $GET\_METRICS\_RANK$ 関数を実行する．$GET\_METRICS\_RANK$ 関数では，PoM に保存されたメトリクスを回帰直線の傾きの大きさ順にソートする．このソートを行うことにより，メトリクスの値がリクエストレートに影響を与えている度合いによって順位付けする．1 位には 3 ポイント,2 位には 2 ポイント,3 位には 1 ポイントとメトリクスにポイントを付与する．この順位に基づくポイントは後のマイクロサービス全体でのメトリクスの優先度に用いられる．

全てのマイクロサービスでメトリクスの順位付けが終わり次第，順位付けが行われたもののポイントを加算し，アプリケーションでリクエストレートに影響を受けるメトリクス名を順位付けし 1 位のメトリクスの相関係数をアプリケーションが開始してからの各マイクロサービスのリクエストレートの平均に掛けることでアプリケーション内でのマイクロサービスの優先度を決定する．

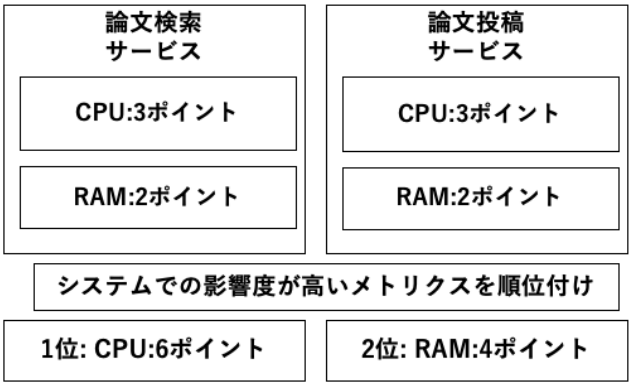

図 3 アプリケーション全体でメトリクスの順位付け

マイクロサービス	優先度	メトリクス
商品	110	RAM
在庫	80	Storage
アカウント	20	CPU
履歴	30	Storage
ポイント	15	RAM
トレンド	10	RAM

メトリクス別, 優先度順に共助

マイクロサービス	優先度	メトリクス
アカウント	20	CPU

マイクロサービス	優先度	メトリクス
商品	110	RAM
ポイント	15	RAM
トレンド	10	RAM

マイクロサービス	優先度	メトリクス
在庫	80	Storage
履歴	30	Storage

図 4 リクエストレートとメトリクスとの相関関係例:CPU の場合

3.2 2 つの異なるマイクロサービスの類似度計測

ここでの目的は 2 つのマイクロサービスを構成するコンテナを比較することでプログラムが互換している部分に関しての処理を代行が可能であるかどうかを判断できるようにすることである．異なるマイクロサービスにおいてマイクロサービスを構成する最小単位であるコンテナを比較することで 2 つの異なるマイクロサービスの類似度を計測する．コンテナイメージが同一の場合，コードやプログラムの共有なしで処理が代行できる．それぞれのコンテナには ID が振られている．コンテナ ID が全ての文字列において同一の場合は処理を代行できる．しかし実運用環境において，コンテナは開発者によって独自のプログラムを開発されるため，コンテナ ID が全一致するケースはは少ないと言える．ここでコンテナを構成しているレイヤ (以降コンテナレイヤと呼ぶ) に着目する．コンテナレイヤは一つのコンテナイメージの読み込み専用のイメージ部分と，読み書きが可能なイメージ部分を分けることで OS，カーネルの部分を共通化し，開発ユーザー独自のソースコードがイメージに上書きできるようになっている．コンテナの比較についての詳細は実装の章に記す．

3.3 プログラムの共有による他マイクロサービスのプログラムの実行

ユーザー独自のソースコード部分以外のレイヤが同一で

あるならば，ソースコードやプログラムを共有すれば実行できる可能性が高いと言える．ソースコードやプログラムの共有手法については次章で述べる．

また，ユーザー独自のソースコード部分以外のレイヤが同一であった場合，より類似度の精度を高めるための検証をする．ユーザー独自のソースコード部分プログラムのソースファイルをテキストベースで比較する．

3.4 他のサービスへのリダイレクトによる負荷分散機能

類似のマイクロサービスを検索した結果をもとに優先度の高いマイクロサービスから低いマイクロサービスへのプログラムコピーを行う．その後，プログラムコピーを送った先の Pod に負荷分散するためのロードバランサーを設置する．

その後優先度が高いものから低いものに対してリクエストをリダイレクトする．

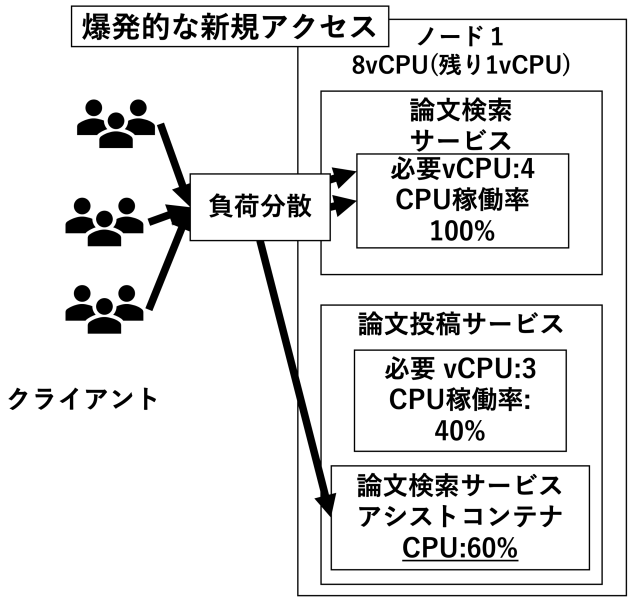

図 5 同一ノード内の使用していない CPU を用いることでノードの CPU 使用率を向上させる．

4. 実装と実験方法

実装

実験環境として実際のクラウドサービスとして論文のポータルサイトを作成した．実験ではこのサービスに共助アシストインスタンスシステムを導入する．

本提案を利用する環境の前提としてサービスメッシュを用いているものとする．マイクロサービスでのトラフィックは全て Envoy を経由することで監視し，datadog を用いてメトリクスを取得できるものとする．

ユーザーのリクエストをシミュレーションするためテストツールである Locust から構築したマイクロサービス，ここでは決済サービスに GET リクエストを送信する．決済サービスがにアクセスが集中し,CPU の使用率がしきい値,ここでは 90%を超えた際に,PoS-LB がゲートウェイ API にイベントを発火し，決済サービス宛のリクエストを一度 PoS-LB に送信するようゲートウェイ API の状態を変化させる．以後決済サービスに対してのリクエストは PoS-LB を通じて，検索サービスおよびリソースを提供している論文投稿サービスに転送する．

PoS-Calc はサービス間での類似度を計測し，別マイクロサービスのプログラムが実行できるか確認する．ここでは私が開発した論文検索サイトを模したクラウドアプリケーションでの例を示す．今回着目するのは以下に示す 2 つの異なるサービスである．

- 論文検索サービス Python により処理されている．API で受け取ったリクエストによって論文を表示するマイクロサービス．
- 論文投稿サービスアップロードされた論文ファイルをストレージにに保存するマイクロサービス．Python により処理されている．

ここではこの 2 つがどのようにコマンドを比較し，類似度の精度を向上するか説明する．以下の図 6 では論文検索サービスと論文投稿サービスでそれぞれ実行されたコードを history コマンドで比較した時，これらは全て一致したと仮定する．すなわちプログラムに使用されるコマンドは全て一致した事になる．そのためこの 2 つのコンテナはソースコード及びプログラムを共有することで実行できる可能性，すなわち類似度が高いと言える．

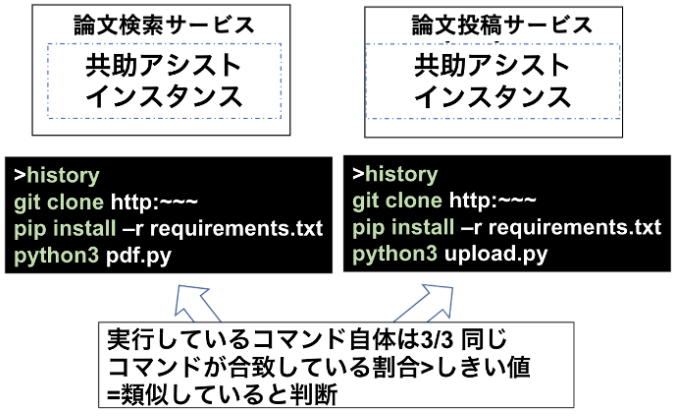

図 6 共助できる例

以下の図 7 では論文検索サービスと api サービスでそれぞれ実行されたコードを history コマンドで比較した時，1 つしかコマンドがなかった場合にはこの二つのコンテナで同じプログラムを実行することは難しいといえる．

4.1 Kubernetes クラスターの構築

はじめに，ノード VM を作成するための仮想環境として表 3 のマシンに ESXi をインストールする．次に

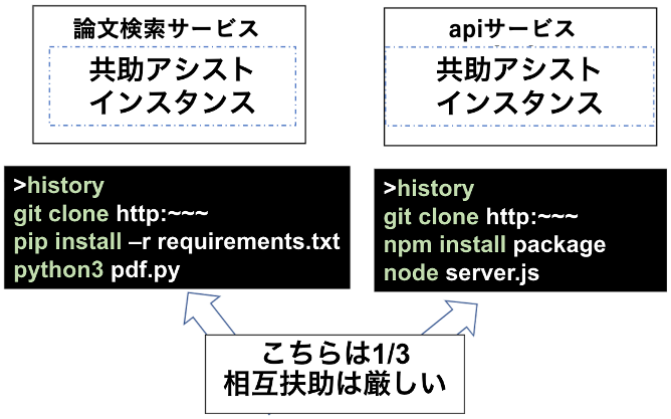

図 7 共助できない例

CPU	AMD Ryzen 3950X
RAM	128GB
SSD	NVMe 2TB
NIC	Intel 1Gbps

表 3 ESXi サーバーのスペック

bernetes クラスターを構築するため, 以下の表 4 に示すスペックの VM を 3 ノード構築した OS はデフォルトで Kubernetes,Docker モジュールがプリインストールされている microk8s を使用した.

CPU	4vCPU
RAM	16GB
Storage	200GB
OS	Ubuntu 20.04

表 4 各ノードのスペック

4.2 各サービスの実装

実験は Kubernetes システム上で以下のアーキテクチャで行った.

- Python/Locust: ユーザーと API ユーザーの代用として自動でリクエストを送信するツール
- Flask(Scalable max3): 論文検索サービス
- Flask(Scalable max3): 論文投稿サービス

各サービスごとのレプリカを設定, ロードバランスをする Kubernetes Service をデプロイし, 各 Kubernetes Service にアクセスする. 本稿での実験概要図を以下の表 2 に示す. ユーザーのリクエストをシミュレーションするためテストツールである Locust から構築したマイクロサービス，ここでは検索サービスに GET リクエストを送信する. 検索サービスがにアクセスが集中し,CPU の使用率がしきい値, ここでは 90%を超えた際に,PoS-LB がゲートウェイ API にイベントを発火し，検索サービス宛のリクエストを一度 PoS-LB に送信するようゲートウェイ API の状態を変化させる. 以後検索サービスに対してのリクエストは PoS-LB を通じて，検索サービスおよびリソースを提供している論文投稿サービスに転送する.

以上の構成を元に作成した実験構成図を以下の図 6 に示す.

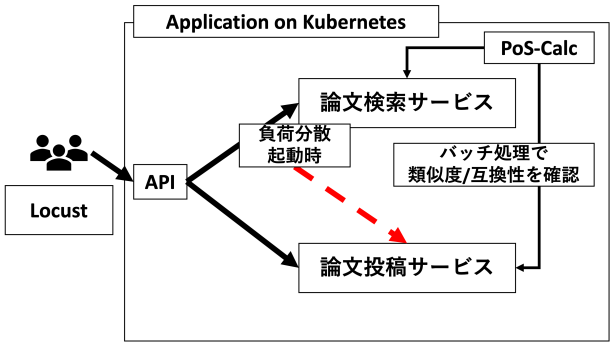

図 8 実験概要図

5. 評価手法と分析手法

評価として, マイクロサービスが 2 個稼働している環境に API を用意し以下のリクエストレートの条件にそってアクセスする. その際にスケールアウトが発生したときの既存手法および，本提案手法でのレスポンスタイムを比較する. 今回の実験では恒常リクエストレートを 100 [req/s] とし，ピークを想定したリクエストレートを 500 [req/s] とした. また対象リクエストは Web サーバーに対する POST リクエストに制限した実験時間はノードの使用率を算出する実験は 180 秒，サービスに対し初めてリクエストを送った際の不要なスパイクを実験から除くため，100 [req/s] を 60 秒間送信し，安定した状態から実験を行った. 60 秒でサービスのレスポンスタイムが安定してから，すなわち計測開始から 60 秒間は継続して 100 [req/s] を送信した. 60 秒後，リクエストの増加をシミュレーションするため，リクエストレートを 500 [req/s] に上昇させた. 各サービスは各ノードに 1Pod のみ実行できるものとし，スケールアウトする際には別ノードの拡張が必要とした. また，オートスケールのしきい値は 80%とした. また，計測のばらつきを抑えるため，試行回数は 50 回および 1000 回とした.

また, 上記の順位付けを用いてノード内で CPU が効率的に割り当てられたかを確認する. 今回は全ノードでの CPU 量を足したものを総 CPU 使用量として既存手法と提案手法で比較した.

また，API からのレスポンスタイムを本研究モデルの導入前後で比較する. その際に PoS が高いサービスはリクエストが急増しても従来のクラスターよりもレスポンスタイムが増加しないことが想定される. この実験の際に PoS が低いサービスでのレスポンスも確認し，考察する. また，クラスターの利用効率が上がることによるノードの追加頻度の減少を確認する.

Vol.2022-DPS-190 No.17
Vol.2022-CSEC-96 No.17
2022/3/10

6. 評価

6.1 PoS システムを導入したときのノードの効率的な利用

図 9 は実験を行った際の初めから用意しておいたノードにおける CPU 使用率を示したものである．青色の線が Kubernetes の純正のオートスケール，赤色が本研究の PoS システムを導入した結果のノードの CPU 使用率である．

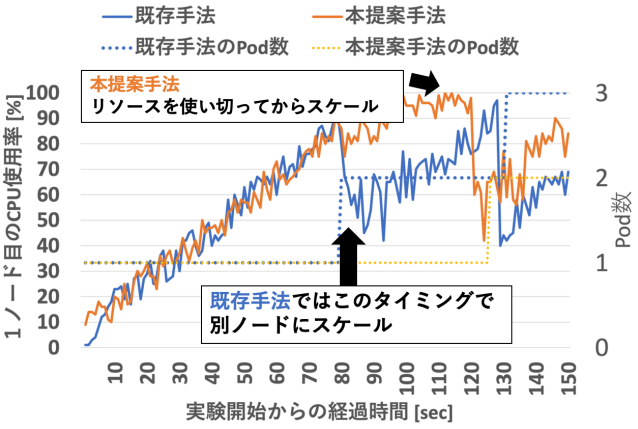

図 9 2 手法での 1 ノード目で計測した CPU 使用率

評価開始から 60 秒間は 100 [req/s] でリクエストがサービスに到達しており，60 秒まではほとんど 2 つのシステムに差はないといえる．その後 Kubernetes のオートスケール機能が動作したことにより，70 秒時点ほどで大きく CPU 使用率が低下している．ここでの CPU 使用率の低下は，Kubernetes により別ノードに同一サービスが展開され，負荷分散したことによる．70 秒時点では CPU 利用率を 10% 程度残した段階でのスケールであるため，ノードの資源を有効に使い切れていないことがわかる．これと比較し，PoS システムを導入した結果，70 秒時点では元のサービスでは処理しきれなくなったため，類似コンテナに負荷分散したことにより，CPU の使用率を極限まで使用してからスケールを行っている．この点に関して本研究の提案する PoS システムは Kubernetes のオートスケーラにくらべ, 約 40 秒ノードの高効率な利用に貢献しているといえる．

しかし，130 秒のあとの CPU 使用率はスケールをしているのにも関わらず CPU 使用率が Kubernetes のオートスケーラほどに低下していない．これは PoS システムが極限までシステムのリソースを使い切った結果，別ノードにスケールしたときと Kubernetes が余裕を持ってスケールを行った際では，システムの受けているリクエストの数が前者の方が多いからである．

図 10 は最大 3 ノードまでのデプロイを可能にしたときの CPU 使用量の累計の折れ線グラフである．

斜線で表された部分は既存の Kubernetes の Scaling よりも PoS システムが CPU 使用率を削減できている部分である．試行回数を 1000 回に増加させたときの 1 つ目のノードの CPU 使用率を示したものが図 11 である．

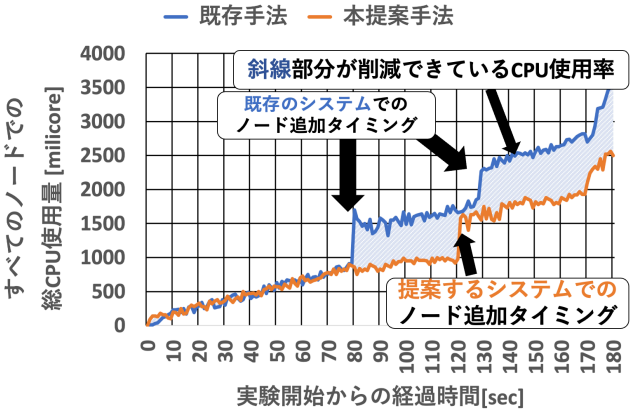

図 10 すべてのノードにおける CPU 総使用量

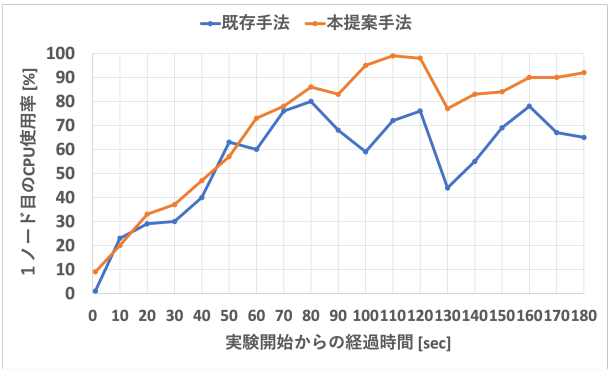

図 11 試行回数を 1000 回にした際の 1 つ目のノードの CPU 使用率の推移

試行回数を増加させた場合でも，既存手法と本提案手法は実験開始から約 70 秒のタイミングで既存手法が新規ノードを追加することにより大きく CPU 使用率は下がっている．既存手法でのノード追加タイミングと本提案手法でのノード追加タイミングの差は平均 39.6 秒であり，これは同じアプリケーション内に互換性を持つマイクロサービスの数が増えれば増えるほどこの差は広がっていくと考えられる．

また，これを試行回数を 1000 回にしたものを以下の図 12 に示す．試行回数を増加させた場合でも，本提案手法は既存手法に比べてノードの追加頻度が低くなっていることがわかる．

6.2 PoS システムを導入したときのレスポンスタイムの維持

図 13 はリクエスト開始からリクエストレートの増加を行った際の Kubernetes のオートスケーラ，または PoS システムの共助プロセスが開始された際のレスポンスタイムを示している．60 秒を超えた段階でリクエストレートが 100 [req/s] から 500 [req/s] に増加したことにより，両シス

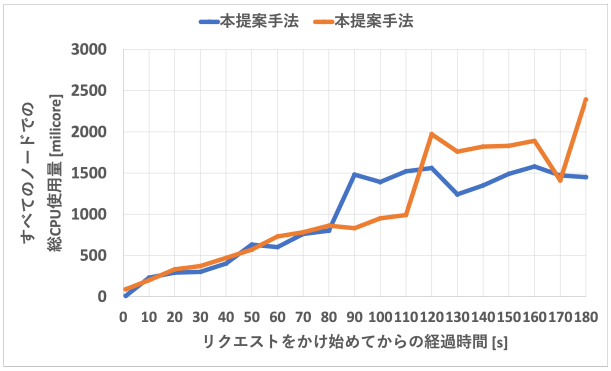

図 12 試行回数を 1000 回にした際の総 CPU 使用量の推移

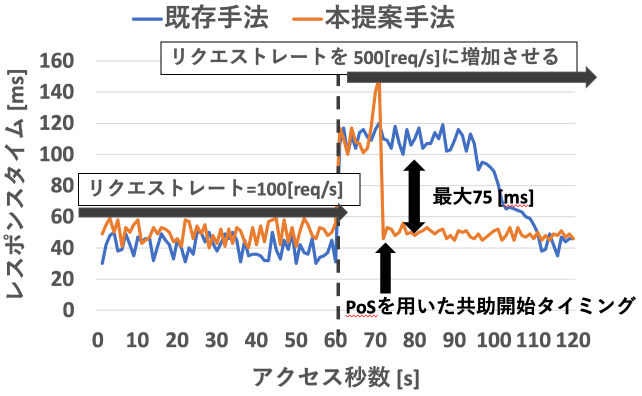

図 13 PoS システムを用いたときのレスポンスタイムの維持

テムともに一時的にレスポンスタイムが飛躍的に増加している．しかしその後 PoS システムを用いたときは 10 秒ほどでサービス間の共助により負荷分散され，レスポンスタイムがリクエストレートが 100 [req/s] の時に近づくように減少したのに対して，Kubernetes のオートスケーラでは新たなノードにサービスを展開するまでのタイムラグから PoS システムが共助開始したタイミングから 40 秒ほど最大 75 [ms] 差が見られる．

7. 議論

実際のトラフィックでの測定として，評価方法としてクラウドアプリケーションを提供している開発者と PoS 計測システムが出力するサービスの順位付けを比較した．また，レスポンスタイムを用いてクラウドアプリケーションのレスポンス高速化を評価した．しかし，実際にマイクロサービスを用いたクラウドアプリケーションでは 500 以上のマイクロサービスを使用することもある [7]．今回の実験の 50 倍以上のマイクロサービスが使用されている環境での評価，及び問題点の抽出が必要である．しかし実際に顧客が存在していて，SLA が定義されていた場合にはこのような研究のための実験は困難である．そのため trainticket のようなマイクロサービス用のベンチマークソフトウェアを拡張し，研究分野でも 100 以上のマイクロサービスでテストできる環境を開発する必要がある [8]．

8. おわりに

本稿ではマイクロサービスアーキテクチャを採用した際，ノード追加を伴うスケーリング時におけるレスポンスタイムの増加を課題とした．論文検索サービスのアクセスログをもとに生成したリクエストを定常時である 100[req/s]，急増時を 500[req/s] としてそれぞれのリクエストレートでアクセスした際のレスポンスタイムを計測した．既存の手法では急増時のレスポンスタイムである約 120[ms] から定常時の約 50[ms] に復帰するまで約 50 秒間継続しているのに対し，提案手法では約 10 秒で復帰することが可能となった．この研究では自動的に稼働中のマイクロサービスより優先度を算出するメカニズムを提案した．これによりノードの追加時におけるレスポンスタイムの遅延を抑制することが可能となった．

謝辞

本研究は，JSPS 科研費 JP20K11776 の助成を受けたものである．

参考文献

[1] Cerny, T., Donahoo, M. J. and Trnka, M.: Contextual understanding of microservice architecture: current and future directions, *ACM SIGAPP Applied Computing Review*, Vol. 17, No. 4, pp. 29–45 (2018).
[2] Dragoni, N., Lanese, I., Larsen, S. T., Mazzara, M., Mustafin, R. and Safina, L.: Microservices: How to make your application scale, *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, Springer, pp. 95–104 (2017).
[3] Akbulut, A. and Perros, H. G.: Performance Analysis of Microservice Design Patterns, *IEEE Internet Computing*, Vol. 23, No. 6, pp. 19–27 (online), DOI: 10.1109/MIC.2019.2951094 (2019).
[4] Khaleq, A. A. and Ra, I.: Intelligent Autoscaling of Microservices in the Cloud for Real-Time Applications, *IEEE Access*, Vol. 9, pp. 35464–35476 (online), DOI: 10.1109/ACCESS.2021.3061890 (2021).
[5] Evans, J. D.: *Straightforward statistics for the behavioral sciences.*, Thomson Brooks/Cole Publishing Co (1996).
[6] Wang, T., Xu, J., Zhang, W., Gu, Z. and Zhong, H.: Self-adaptive cloud monitoring with online anomaly detection, *Future Generation Computer Systems*, Vol. 80, pp. 89–101 (online), DOI: https://doi.org/10.1016/j.future.2017.09.067 (2018).
[7] Zhou, X., Peng, X., Xie, T., Sun, J., Ji, C., Li, W. and Ding, D.: Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study, *IEEE Transactions on Software Engineering* (2018).
[8] Zhou, X., Peng, X., Xie, T., Sun, J., Xu, C., Ji, C. and Zhao, W.: Poster: Benchmarking microservice systems for software engineering research, *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, IEEE, pp. 323–324 (2018).