ファイル名およびファイルのハッシュ値の比較によるプログラムコピーを用いたスケーラビリティの向上

野木 空良[1]　飯島 貴政[2]　串田 高幸[1]

概要：マイクロサービスは，エンドユーザーの増加に伴いリクエスト数が増加すると，サービス管理者はマイクロサービス単位でスケールアウトを行う．課題として，スケールアウトによりマイクロサービスの複製を行う最中にアクセス数の増加が継続した場合，マイクロサービスの起動が間に合わず，レスポンスタイムが上昇する．本研究では，異なるマイクロサービス間で，プログラムのファイル名の比較とハッシュ値の比較を行い，同一ファイル以外のプログラムをコピーする「DocCP」を提案する．マイクロサービスどうしで DocCP を使用し，リクエストの急増時に分散先を増やすことでレスポンスタイムの上昇を抑え，スケーラビリティを向上させる．評価実験は，Kubernetes 環境を用いて web サーバーおよびロードバランサーを作成し，プログラムコピー前とプログラムコピー後の応答時間（レスポンスタイム）を比較する．結果として，プログラムコピー後は，レスポンスタイムの上昇を抑えることができた．また，ノード全体の CPU 使用量では，3500 [millicores] の削減ができた．

1. はじめに

背景

マイクロサービスアーキテクチャでは，アプリケーションを機能ごとに分割している．マイクロサービス 1 つあたりにおけるユーザーからのアクセス数が増加すると負荷が上昇し，SLO(Service Level Objective) に違反する場合がある [1]．SLO の違反発生は，マイクロサービスの核となるアプリケーションの CPU やメモリの過剰な使用により，リクエストの処理に時間がかかるのが原因である [2]．SLO 違反の対策として，単体のマイクロサービスを複製し，CPU 使用率，メモリ使用率，ディスク I/O 使用率の分散を行うスケールアウトという機能がある [3]．マイクロサービスのスケールアウト方法を図 1 に示す．「マイクロサービス A1」，「マイクロサービス B」，「マイクロサービス C」では，個々の機能をコンテナに格納し，独立したマイクロサービスとして実装している．例えば，「マイクロサービス A1」の「機能 A」の負荷が上昇した際にコンテナ単位でのスケールアウトが可能となり「マイクロサービス A2」として起動する．管理システムはマイクロサービス単位でのスケールアウトにより，リクエスト数の分散を図る．Amazon ECS ではクラスター及び，マイクロサービスのメモリと CPU 使用率をモニタリングし，リソースが不足した際に，クラスターを自動的にスケールアウトする機能がある [4]．マイクロサービス内部のメトリクスを取得し，CPU やメモリの使用率が減少し，スケールアウトをする必要がなくなった場合に自動でスケールインを行う．マイクロサービスにリクエスト数が急増した際, マイクロサービス単位でスケールアウトを行い，マイクロサービスの CPU やメモリの消費を分散できる．

図 1　マイクロサービスのスケールアウト

[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
[2] 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

課題

課題は，スケールアウトはマイクロサービスの起動に時間がかかり，リクエスト数が増加し続けた際の分散が間に合わないことである．その為，スケール時にかかる時間を考慮した閾値を設定する必要がある [5]. 例えば，リクエスト数の分散を間に合わせる為，閾値の値を CPU 全体の 10 % に達した時にスケールアウトを行うように設定する．しかし閾値を 10 %に設定したことにより，リクエストが急増し続けた場合でも急増の最中にスケーリングを行ってしまう．その結果，次のスケールアウト実行までにインターバル (AWS の場合デフォルトで 300 秒) が発生し，その間のレスポンスタイムの維持は不可能となる [6]．スケールのタイミングが遅れた場合，リクエスト分散が間に合わなくなり，レスポンスの遅延が生じる [7]．スケールアウトによるレスポンスタイムの遅延を図 2 に示す．

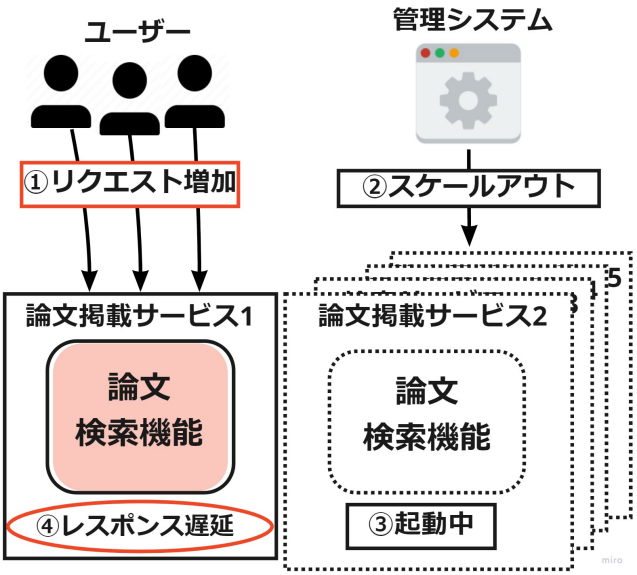

図 2 スケールアウトによるレスポンスの遅延

図 2 は論文掲載サービスの例である．Web サービスではユーザー数が不定期に増減する．例えば，Twitter や Facebook といった SNS でアプリケーションが人気になると，Web サービスに対してユーザーのアクセス数が増加する．リクエスト数の急激な増加は図 2 の例である論文掲載サービスも同様であり，管理システムはマイクロサービス単位のスケールアウトを行う．しかし，スケールアウトによって追加された論文掲載サービスは，起動している最中のリクエストの処理を受け付けない為，その間はレスポンスの遅延が生じる．レスポンスの遅延は，web サイトの直帰率の原因により，アクセスするユーザー数の減少につながる [8]．したがって，スケールアウトによるサービス起動中のレスポンスタイム維持が必要である．以下にマイクロサービスに対してリクエストが急増した際の，スケールアウトによるレスポンスの遅延が発生するまでの流れを示す．

① ユーザー数の増加により，論文掲載サービスのリクエスト数が急増する．
② 論文掲載サービスの管理システムはリクエスト数増加に伴い増加させる数 n を決定し，スケールアウトを行う．
③ 決定した論文掲載サービス n 個 (n=2，3，4...) の起動する．
④ 論文掲載サービスの起動中または，スケールアウトのインターバルによりレスポンスが遅延する．

各章の概要

2 章では，本研究と関連した既存研究について述べる．次に，本研究の提案を 3 章で述べる．4 章では，実装と実装環境について述べる．5 章では，基礎実験と，提案内容の実験結果を定量的に比較し，示している．6 章では，本研究の提案をもとに議論する．最後に，本論文の取り組みと貢献について簡潔に述べる．

2. 関連研究

MYSQL や PostgreSQL といったデータベース管理システム（DBMS）を対象とした，負荷の上昇に対応するスケールアウトの指標を示す研究がある [9]．DBMS はステートフルシステムなため，スケールアウトによって増加させた DBMS との整合性を維持する必要がある．しかしスケールアウトやスケールインによって別ノードへ DBMS を増減させた場合，元の DRBS からデータを転送する必要がある為，ディスク I/O 使用率や CPU 使用率が上昇する．この研究では，DBMS 内のデータをパーティション分割により種類分けし，分割した一部のみをスケールアウトにより複製する事で，データ転送量の削減を行っている．しかし DBMS のパーティション分割は，データが分割されている為，通常のテーブルに比べ，構成が複雑になる．これにより，データを読み込む際の処理時間がオーバーヘッドに繋がる可能性がある為，データ読み込みまでのレスポンスタイムの上昇が課題となる．

オートスケールによるサーバーの複製のタイミングを動的に変更する研究がある [10]．Kubernetes の場合，クライアントが手動で Pod（サーバー）のスケールアウトを行う閾値を手動で決定している．この研究では，メトリクスサーバーによってスケールアウトを行うサーバーの状態を監視し，スケールアウトの閾値を動的に変更することで，オートスケールの実行のタイミングを決定している．しかしメトリクスサーバーによる CPU 使用率やメモリ使用率の取得は，タイムラグがある為，負荷が上昇してから，閾値の決定までに差が発生する為，レスポンスタイムを維持できない．

モノリシックサービスを自動的に分解してマイクロサー

ビスとして運用する研究がある [11]．この研究ではモノリシックサービスのアクセスログと，機械学習を用いて，CPU やメモリの消費量が同様のアプリケーションを識別し，URI ごとに機能を分割している．分割した URI ごとにマイクロサービスとして自動で作成を行い，スケールアウト機能を付与している．これによりマイクロサービスごとにスケールアウトを行い，リクエストの分散をレスポンスタイムを改善している．しかし，マイクロサービスアーキテクチャは機能ごとに CPU やメモリの消費量が異なるため，最大限に CPU やメモリを使用することは困難である．クラウドプロバイダは従量課金制を取り入れている為，リクエスト数に対して機能ごとの消費量に合わせなければレスポンスタイムの増加につながる．

3. 提案

本稿では，マイクロサービスの主機能が実装されているプログラムを，機能が違う複数のマイクロサービスにコピーし，1 つのマイクロサービスにリクエスト数が急増した際に，リクエスト数を分散するソフトウェア「DocCP」を提案する．DocCP の起動のタイミングは既存研究である PoS を用いる [12]．PoS とはコンテナの CPU 使用率をもとに，どのマイクロサービスが重要かをランク付けするシステムである．PoS は既存のアプリケーションに新機能のマイクロサービスが追加されたタイミングで実行される．PoS の実行と同時に DocCP を起動し，ランク付け上位のマイクロサービスのプログラムを下位のマイクロサービスにコピーすることで，CPU 使用率の高いコンテナを共助することが可能となる．DocCP のシステムアーキテクチャを図 3 に示す．

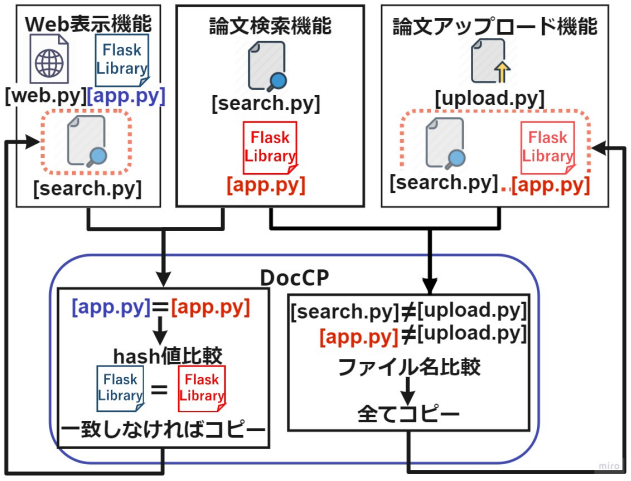

図 3 ソフトウェア「DocCP」によるプログラムコピー手法

図 3 のシステムアーキテクチャでは，ホームページを表示する「Web 表示機能」，ユーザーが検索した論文を表示する「論文検索機能」，ユーザーが論文をアップロードできる「論文アップロード機能」が実装されている．本稿では前提として，PoS 実行の結果「論文検索機能」を「Web 表示機能」「論文アップロード機能」にコピーを行うシナリオを示している．DocCP を起動した際に「論文検索機能」と「Web 表示機能」，「論文検索機能」と「論文アップロード機能」のそれぞれでプログラムの比較を行う．

3.1 「論文検索機能」と「Web 表示機能」の比較

「論文検索機能」には，ユーザーが検索したリクエストを処理する「search.py」と，検索結果を Web 上に表示する為に Python のモジュールである Flask を用いて実装している．また「Web 表示機能」では，アプリケーションにアクセスするリクエストを処理する「Web.py」と，Web サーバーを立てる為に Python のモジュールである Flask をインストールして実装している．DocCP を起動した際に，「論文検索機能」と「Web 表示機能」内で実行されているプログラムの名前を比較する．しかし，Flask のライブラリにある「app.py」がそれぞれのマイクロサービスに存在していた場合，プログラム名が一致している為判別が出来ない．プログラム名が一致していた場合は，プログラムのハッシュ値を取得し，比較する．ハッシュ値も一致した場合は，機能が同じプログラムと判断し，プログラムのコピーを行わない．また，ハッシュ値が一致しない場合はコピーを行う．プログラムのコピーを行った際のオーバーライドを避けるため，専用ディレクトリを作成しコピーを行う．

3.2 「論文検索機能」と「論文アップロード機能」の比較

「論文アップロード機能」では，ユーザーが論文のアップロードをリクエストした際に処理する「upload.py」を実装している．「論文検索機能」と「Web 表示機能」の比較と同様に，各機能のプログラム名を比較する．各機能に存在する全てのプログラム名を比較し，一件も一致しなかった場合，「論文検索機能」の全てのプログラムのコピーを行う．ファイル名が一致しない場合は，別機能のプログラムと決定し，ハッシュ値の比較を行う処理を省略する．これによりプログラムの比較処理速度が速くなる．

DocCP のアルゴリズムを図 4 に示す．また DocCP 実行の順序を以下に示す．

① PoS の実行と同時に DocCP を起動する．
② アプリケーション内にある各マイクロサービスどうしでプログラム名を比較する．
③ プログラム名が一致しているかの確認をする．
④[No]「論文検索機能」に存在する全てのマイクロサービスを「Web 表示機能」と「論文アップロード機能」にコピーする．
④[Yes] アプリケーション内にある各マイクロサービスどうしでプログラムのハッシュ値を比較する．

⑤ ハッシュ値が一致しているの確認をする．
⑥[No] ハッシュ値が一致しなかった場合プログラムをコピーをする．
⑥[Yes] ハッシュ値が一致した場合コピーしない．

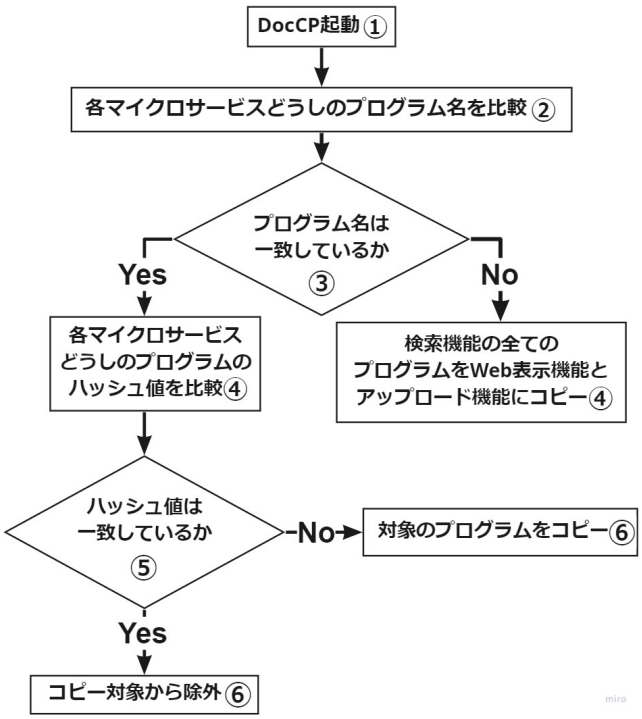

図 4 DocCP のアルゴリズム

3.3 ユースケースシナリオ

本研究では，論文掲載サービスである「Doktor[*2]」をユースケースとしている．ユースケースシナリオを図 5 に示す．「Doktor」では論文を検索するサービス，Web を表示するサービス，論文を PDF としてアップロードするサービスの 3 つで構成されている．「Doktor」を利用しているユーザーが検索機能に集中した際に，本提案の DocCP によりユーザーのリクエストを分散する．各ユーザーは論文の検索をリクエストしているが，他機能である Web サービスと PDF アップロードサービスが，論文検索の処理を受け付けることで，論文検索サービスがスケールアウトを行うまでの共助を行う．論文検索サービスのスケールアウトを行うまでのレスポンスを共助することでユーザーまでのレスポンスタイムを維持し，離脱率減少を図る．

4. 実装と実験環境

4.1 実装

プログラムコピーソフトウェア (DocCP)

本研究の提案である「DocCP」のソフトウェアアーキテクチャを図 6 に示す．またプログラムのハッシュ値の取得

*2 https://github.com/cdsl-research/doktor

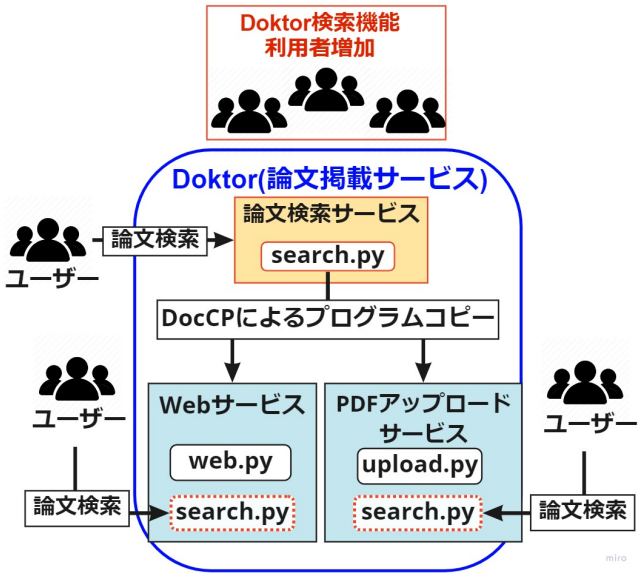

図 5 ユースケースシナリオ

からコピーまでの流れを以下に示す．

① 各 Pod で Python ファイルのハッシュ値を取得する．
② 取得したハッシュ値とプログラム名のリスト化をする．
③ リスト化したファイルを DocCP 内に送信する．
④ 各プログラムのファイル名の比較，ハッシュ値の比較を行う．
⑤ 比較した結果から，コピー先のマイクロサービスにプログラムをコピーする．

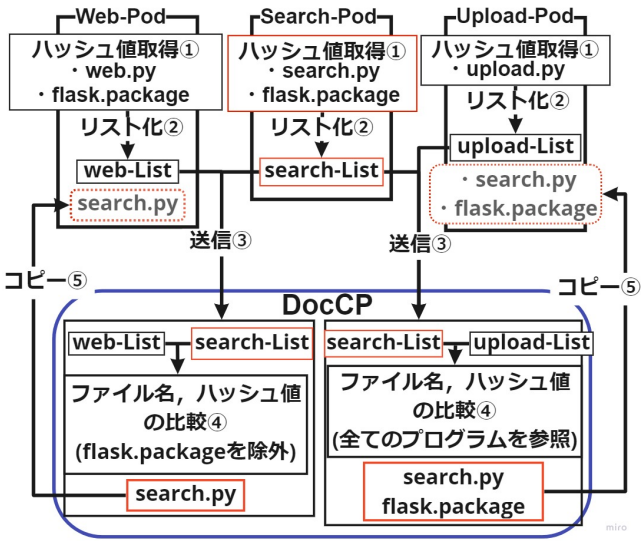

図 6 ソフトウェアアーキテクチャ

本実装ではマイクロサービスを Kubernetes を用いて構築している．また，DocCP は Python を用いて作成し，ファイル名の比較とハッシュ値の比較，プログラムのコピーを行う．ハッシュ値の取得は，md5 コマンド[*2]を用いて，各 Pod 内で行う．md5 コマンドは実行した際のトラ

*2 https://datatracker.ietf.org/doc/html/rfc1321.html

ンザクションが少ないため，マイクロサービスのようなプログラム数の多いアプリケーションに適している．取得したハッシュ値とプログラム名を csv ファイルとしてリスト化し，リスト化したファイルを DocCP に送信する．「search-List」と「web-List」を比較した場合，Flask のパッケージにあるファイル名が一致している為，ハッシュ値の比較まで行う．ハッシュ値が一致していた場合，「search.py」のみを「web-Pod」にコピーする．また，「web-List」と「upload-List」を比較した場合，ファイル名が一致しているプログラムが存在しない為，「Search-Pod」に存在する全てのプログラムを「Upload-Pod」にコピーする．プログラムのコピーの手法として，Kubernetes のコマンドである「kubectl cp」を用いる．「kubectl cp」はネットワークを通して Pod 内のプログラムの取得が可能である．DocCP が kubectl コマンドを実行することでプログラムのコピーを行う．

4.2 実験環境

システムアーキテクチャを図 7 に示す．ハードウェア上に VMware ESXi をインストールし，Kubernetes 環境を構築した MicroK8s ノードと，実験で使用する負荷試験ノードを作成する．MicroK8s ノードでは Pod を 3 つ作成し，「Pod1」にアクセスが集中したマイクロサービスと仮定し，スケールアウト機能を構築する．また，プログラムのコピーを行う DocCP ソフトウェアを作成しプログラムのコピーを行った後に，評価実験を行うため，NGINX ロードバランサー[*3]を作成する．負荷試験ノードでは，ユーザーのアクセスを想定して HTTP リクエストを GET で送信する Locust[*4]を用いる．本実験で使用した MicroK8s ノードのハードウェア構成を以下に示す．

- CPU：10 [vCPUs]
- メモリ：8 [GB]
- ハードディスク：40 [GB]

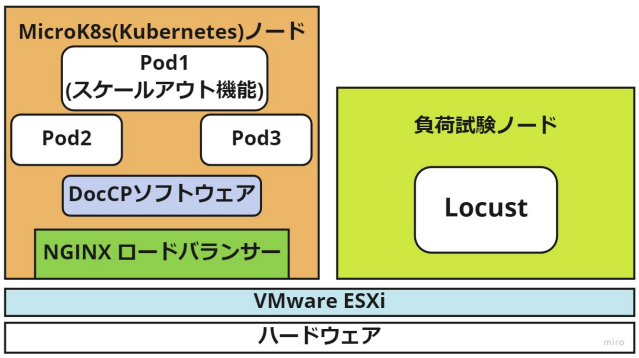

図 7 システムアーキテクチャ

[*3] http://nginx.org/en/docs/http/load_balancing.html
[*4] https://docs.locust.io/en/stable/

4.3 基礎実験

本研究では，Kubernetes の機能である既存のスケールアウト方式のインターバルによるレスポンスの遅延を検証する為に基礎実験を行った．Python のモジュールである Flask を用いて Web サーバーを立てた Pod を用いて負荷実験を行った．Web サーバー内では 1 リクエストにつき 2 の 6 乗を計算し出力する．2 の 6 乗の計算を行うことで，CPU 使用率を増加させ，処理数が増加した際のレスポンスの変化を測定している．また本研究の実験環境で 2 の 7 乗の計算を行うと計算処理の増加により 1Pod あたりの最大 CPU 使用量を超えてしまい 503（Bad Gateway）エラーが返ってくる．本基礎実験の目的はレスポンスの遅延を再現することである為，エラーを避けるため，2 の 6 乗の計算を使用した．既存のスケールアウト方式を用いた Pod と既存のスケールアウト方式をしない Pod のレスポンスタイムのグラフを図 8 に示す．横軸は実験開始時間 [s]，縦軸は Locust がリクエストを送信してから Pod からレスポンスが返ってくるまで（レスポンスタイム [ms]）を表している．

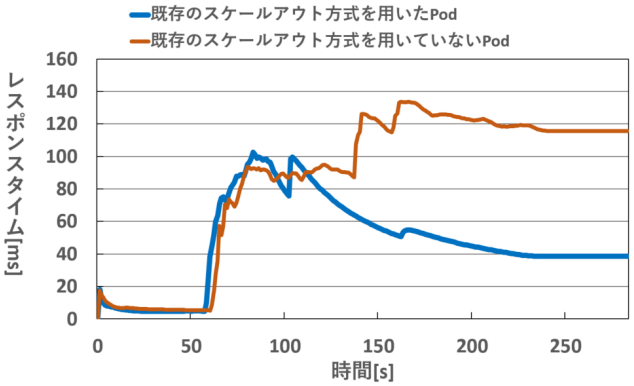

図 8 既存のスケールアウト方式の実行有無によるレスポンスタイムの変化

基礎実験では，HTTP リクエストを GET で送信する Locust を用いた．実験開始から 60 [s] は 10 [req/s] 送信している．理由として，Pod の処理開始時は web サーバーの起動によりレスポンスタイムが数秒だけ増加する為，レスポンスタイムが一定の状態から，リクエスト数を増加させている．実験開始から 60 [s] 以降は，500 [req/s] を最大とし，50 [req/s] ずつ上昇させている．500 [req/s] 以上のリクエストを送信すると，1Pod 当たりの CPU 使用量を超えるため，リクエストを処理できず 503（Bad Gateway）エラーが返ってくる．そのため最大リクエスト数を 500 [req/s] としている．図 8 では既存のスケールアウト方式をしない Pod に比べ，既存のスケールアウト方式を行う Pod の方が最終的に約 80 [ms] 程レスポンスが早い結果となった．これは Pod の CPU 使用率の上昇から既存のスケールアウト方式により Pod の数を増やしているためである．しかし，60 [s] から 105 [s] まではレスポンスタイムは減少していな

い．図 8 の既存のスケールアウト方式を行った Pod のレスポンスタイムをもとに，どの頻度で Pod を複製しているかを示したグラフを図 9 に示す．図 9 では横軸を実験開始時間 [s]，縦軸を Locust が HTTP リクエストを送信してから Pod からレスポンスが返ってくるまで（レスポンスタイム [ms]）と Pod のレプリカ数を示している．本稿ではレプリカ数とは実験環境である Kubernetes の Pod の合計数を指す．

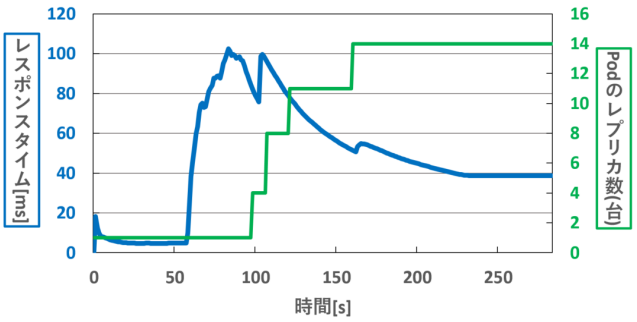

図 9 既存のスケールアウト方式による Pod の起動数とレスポンスタイム

Pod の Deployment の設定として，CPU の使用上限を 500 [millicores] とし，最大 20 個の Pod が起動する．本実験環境の仮想マシンが最大 10000 [millicores] まで使用可能なため，上限を 500 [millicores] とし最大起動数を 20 個としている．また，既存のスケールアウト方式を行う閾値を CPU の 10 %を超えた時と設定することで，CPU 使用率が上昇を始めた時点で Pod の複製を行いレスポンスタイムの減少を図った．しかし，レスポンスタイムが上昇し始めてから，30 [s] までは既存のスケールアウト方式を行っていない．このことから，CPU 使用率の上昇を検知してから Pod をスケールするまでに 30 [s] かかり，その間のレスポンスは遅延することが分かる．

5. 評価と分析

本実験では，Locust がリクエストを送信してからレスポンスが返ってくるまでの時間をレスポンスタイムと定義し，実験を行った．評価では，プログラムコピー後にリクエストを分散させたマイクロサービスとプログラムコピー前のマイクロサービスのレスポンスタイムを比較する．評価実験方法を図 10 に示す．

本実験では，プログラムコピー後にリクエストを分散させる方法として，NGINX のロードバランサーを使用した．ロードバランサーの分散アルゴリズムは，NGINX サーバーがリクエストを受け取った時点でのコネクション数が最も少ない Pod に振り分ける方式を用いている．NGINX は Apache httpd といった他のロードバランサーに比べ，同時処理能力が高い [13]．よってロードバランサーがレスポンスタイムのボトルネックにならない為，負荷試験に適して

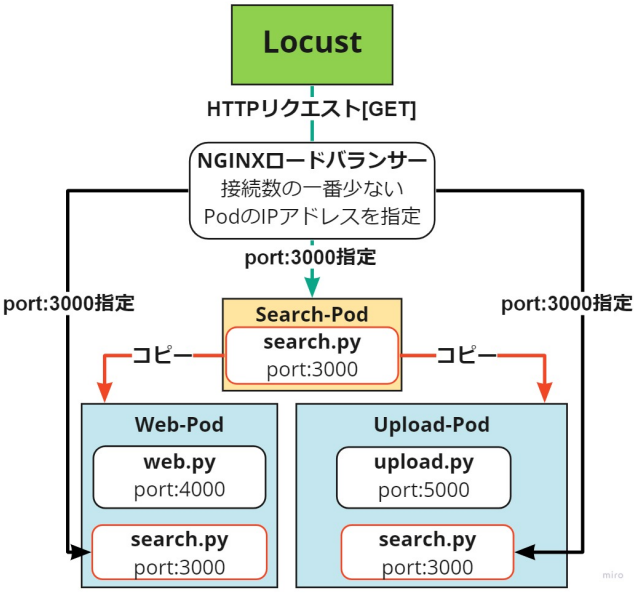

図 10 ロードバランサーを用いた評価実験方法

いる．各 Pod のプログラムは Python の Flask を用いており Web サイトを立てている．リクエストを NGINX サーバーに送信することでロードバランサーが各 Pod の IP アドレスと，各 Pod にコピーした Flask サーバーの port を指定し，リクエストを分散させ，レスポンスタイムを測った．レスポンスタイムの計測方法は Locust が HTTP リクエストを送信してから処理結果が返ってくるまでの時間を 1 秒単位で取得している．既存のスケールアウト方式を用いたプログラムコピー前の Pod と，DocCP によるプログラムコピーをした後の Pod のレスポンスタイムを比較を図 11 に示す．図 11 の縦軸と横軸は検証実験の図 8 と同様である．また提案方式の実験ではレスポンスタイムの増減幅を比較する為 1400 [s] の実験を行った．

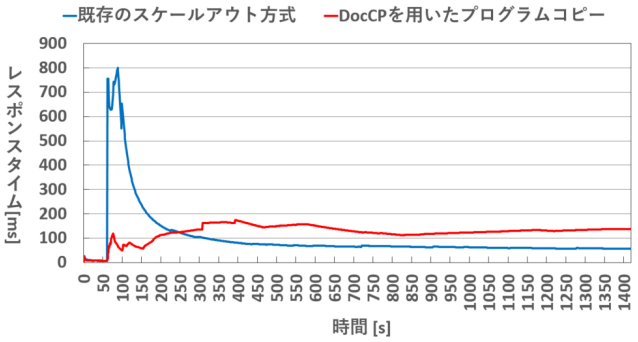

図 11 Locust で Pod にリクエストを送信して返ってくるまでのレスポンスタイム

図 11 では，基礎実験の条件と同じく，実験開始から 60 [s] は 10 [req/s]，以降は最大 500 [req/s] に到達するまで 50 [req/s] ずつ上昇させている．DocCP によるプログラムコピーをした後のレスポンスタイムは，約 100 [ms] まで上昇した後，既存のスケールアウト方式のようにレスポンスタイ

ムの急上昇はせず約 100 [ms] を保つ結果となった．これはリクエストが上昇した場合でも，Web-Pod と Upload-Pod がレスポンスを共助していることにより，1Pod あたりの処理数が減少し，CPU 使用率が減少しているためである．しかし 250 [s] 以降の DocCP を用いたプログラムコピーのレスポンスタイムは，既存のスケールアウト方式のレスポンスタイムより上昇する結果となった．これはプログラムコピーをした Pod がレスポンスを共助したことにより Search-Pod のレプリカ数が減少した為であると考える．プログラムコピー前とプログラムコピー後のノード内の合計 Pod 起動数を示したグラフを図 12 に示す．図 9 の縦軸と横軸は検証実験の図 12 と同様である．

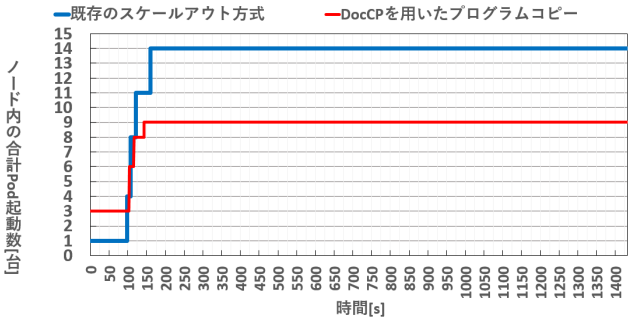

図 12 ノード内の合計 Pod 起動数

Pod のスケールアウトの設定は基礎実験と同様であり，Pod の requests と limit の値を CPU の 500 [millicores] とし，最大レプリカ数を 20 個と設定している．本実験環境の仮想マシンが最大 10000 [millicores] まで使用可能なため，上限を 500 [millicores] とし最大起動数を 20 個としている．ノード内の合計 Pod 起動数は DocCP を用いた場合 100 [s] まで 3 [台] となった．これは Search-Pod のプログラムを，Web-Pod と Upload-Pod にコピーした為，Search-Pod をのプログラムを実行している数が増加している．また，プログラムコピー前の最終的な Pod の起動数は 14 個なのに対し，プログラムコピー後は 9 個という結果になった．これはプログラムコピーにより，リクエストの分散先が増えたことで Search-Pod の CPU 使用率が減少しスケールアウトの閾値（50 [mcores]）以下になったためであると考える．1Pod の CPU 使用量が 500 [millicores] であることからプログラムコピー前のノードの CPU 使用量が，7000 [millicores] に対して，プログラムのコピー後は 3500 [millicores] となり 3500 [millicores] の差があった．このことからプログラムコピー後はノード全体の CPU 使用量を削減出来ていることが分かる．本実験の結果として，リクエストが上昇した際でも，レスポンスタイムを維持しながら，ノード全体の CPU 使用量を削減できることが分かった．

6. 議論

提案方式の実験を行った際に，図 11 で，プログラムコピー後の 250 [s] 以降ではレスポンスタイムが既存のスケールアウト方式より上昇した．これは，プログラムのコピー先である Web-pod と Upload-Pod では別の Flask サーバーが立っているため，処理できるレスポンス数が Search-Pod と比べ少ないのが原因であると考える．解決策として，負荷を振り分けるロードバランサーで重みづけを行い，Search-Pod の優先度を高くすることで，レスポンスタイムの上昇を抑えることができる．

また、DocCP を起動する前に，新しい機能を持ったマイクロサービスの追加を条件として，PoS システムを実行し，マイクロサービスのランク付けを行う．さらに，プログラムコピーを行う対象は，ランクの一番高いマイクロサービスのプログラムを同一ノード内にある全てのマイクロサービスにコピーする．これにより PoS によって割り出された CPU 使用率が高いマイクロサービスのリクエストを分散することが可能となる．しかし新しい機能を持ったマイクロサービスを続けて追加し，ランクの一番高いマイクロサービスが更新される可能性がある．その際，更新前にランクが一番であったマイクロサービスのプログラムが，ノード内の全てのマイクロサービスに存在することでメモリの使用率が上昇しボトルネックになる．解決策として，ランクの一番高いマイクロサービスが更新された際は，更新前にコピーしたプログラムを削除する．削除の方法として，DocCP がランクの更新を検知し，更新前に作成したプログラムコピー用のディレクトリごとに上書きすることで，コピーされた既存のプログラムは削除される．

また本研究では，一つのマイクロサービスに対してリクエストが急増した時を想定している．しかし，必ずしも一つのマイクロサービスへリクエストが集中するとは限らない為，仮に別機能のマイクロサービスにリクエストが上昇した際でも，DocCP を実行させる必要がある．解決策として，マイクロサービスの CPU 使用率，メモリ使用率，ディスク I/O 使用率を毎秒取得し，合計値が一番高いマイクロサービスへコピーする．これにより，リクエストが増加したマイクロサービスを検出し，動的にプログラムコピーを行うことが可能となる．

7. おわりに

本研究では，マイクロサービスにおけるリクエスト数が急増した際のレスポンスの遅延を課題とした．既存の手法として，サーバー台数を増やす，スケールアウト機能があるが，マイクロサービスの起動時間を考量する必要があり，その間のレスポンスタイムは維持することはできない．そこで，異なるマイクロサービス同士のプログラムコピーを行い，リクエストが上昇するマイクロサービスの共助を行

い，リクエスト数の分散を行うソフトウェア「DocCP」を提案した．マイクロサービスどうしで DocCP を使用し，リクエストの急増時に分散先を増やすことでレスポンスタイムの上昇を抑え，スケーラビリティの向上を図った．実験の結果，プログラムコピー前に比べプログラムコピー後は，リクエストが上昇した際でもレスポンスタイムの上昇を抑えることが出来た．また，ノード全体の CPU 使用量を，プログラムコピー後は 3500 [millicores] の削減が出来た．このことから本提案でレスポンスタイムの維持をしながら，ノード全体の CPU 使用量を削減できることを示した．

参考文献

[1] Avritzer, A., Ferme, V., Janes, A., Russo, B., van Hoorn, A., Schulz, H., Menasché, D. and Rufino, V.: Scalability assessment of microservice architecture deployment configurations: A domain-based approach leveraging operational profiles and load tests, *Journal of Systems and Software*, Vol. 165, p. 110564 (2020).

[2] Saurabh, H. Q. S. S. B., Kalbarczyk, J. Z. T. and Iyer, R. K.: FIRM: An Intelligent Fine-Grained Resource Management Framework for SLO-Oriented Microservices, *ile*, Vol. 400, p. 800.

[3] Filho, R. R., de Sá, M. P., Porter, B. and Costa, F. M.: Towards emergent microservices for client-tailored design, *Proceedings of the 19th Workshop on Adaptive and Reflexive Middleware*, pp. 1–6 (2018).

[4] Jung, M., Mallering, S., Dalbhanjan, P., Chapman, P. and Kassen, C.: Microservices on AWS, *Amazon Web Services, Inc., New York, NY, USA, Tech. Rep* (2016).

[5] Bunch, C., Arora, V., Chohan, N., Krintz, C., Hegde, S. and Srivastava, A.: A pluggable autoscaling service for open cloud PaaS systems, *2012 IEEE Fifth International Conference on Utility and Cloud Computing*, IEEE, pp. 191–194 (2012).

[6] Podolskiy, V., Jindal, A. and Gerndt, M.: Iaas reactive autoscaling performance challenges, *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, IEEE, pp. 954–957 (2018).

[7] Kanagala, K. and Sekaran, K. C.: An approach for dynamic scaling of resources in enterprise cloud, *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, Vol. 2, IEEE, pp. 345–348 (2013).

[8] Dolma, Y., Kalani, R., Agrawal, A. and Basu, S.: Improving Bounce Rate Prediction for Rare Queries by Leveraging Landing Page Signals, *Companion Proceedings of the Web Conference 2021*, pp. 1–6 (2021).

[9] Minhas, U. F., Liu, R., Aboulnaga, A., Salem, K., Ng, J. and Robertson, S.: Elastic Scale-Out for Partition-Based Database Systems, *2012 IEEE 28th International Conference on Data Engineering Workshops*, pp. 281–288 (online), DOI: 10.1109/ICDEW.2012.52 (2012).

[10] Rossi, F., Cardellini, V. and Presti, F. L.: Self-adaptive Threshold-based Policy for Microservices Elasticity, *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, IEEE, pp. 1–8 (2020).

[11] Abdullah, M., Iqbal, W. and Erradi, A.: Unsupervised learning approach for web application auto-decomposition into microservices, *Journal of Systems and Software*, Vol. 151, pp. 243–257 (2019).

[12] 飯島貴政，串田高幸：マイクロサービスにおけるメトリクスによるサービスの優先順位および計算リソースの共助モデル，CDSL-TR-060, 東京工科大学 コンピュータサイエンス学部 クラウド・分散システム研究室 (2021.Sep.4).

[13] Chi, X., Liu, B., Niu, Q. and Wu, Q.: Web load balance and cache optimization design based nginx under high-concurrency environment, *2012 Third International Conference on Digital Manufacturing & Automation*, IEEE, pp. 1029–1032 (2012).