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

飯島 貴政[1]　串田 高幸[1]

概要：クラウドアプリケーションにおけるマイクロサービスでの導入時にはサービスの優先度をユーザーが構築ファイルに記述する必要があった．そのため, ノードの CPU が枯渇した際, ノードを新たに追加することでスケールしていた．本稿ではマイクロサービスの優先順位を計算し, 優先順位が高いサービスを低いサービスが共助することで処理に必要とされる CPU をクラスター内から再利用する．前段階として, マイクロサービスの相互通信のログやノードにおける CPU 使用率を取得する．取得した指標を提案する計算式に代入することでマイクロサービスが複数ある環境で優先度の順位付けを行う．このモデルでは異なるコンテナでの優先度の比較を行い, 優先度が高いマイクロサービスの処理を低いコンテナが共助する．評価方法は, マイクロサービスが 15 個稼働している環境にウェブサイトを用意する．ユーザーが定義したサービスの優先順位付けと 提案手法による優先順位付けを比較し， 優先順位の一致率を評価する．また, 上記の優先順位付けによるノードのリソースを効率的に使い切れているかを評価する．

1. はじめに

背景

クラウドを用いてアクセスより柔軟に対応できる近年の WEB アプリケーションを構築する際には以下の 3 つの構築方法がある．

- クラウドで利用できるコンテナを作成/ビルドして動作させる
- クラウドアプリケーションを単機能に分割し相互に連携させる
- クラウドでプログラムコードのみをおいて実行させる

1 番目はコンテナオーケストラレーションと呼ばれ, これは既に OS のカーネル部分を共通化し, その上に OS がインストールされたコンテナと呼ばれるものをカスタマイズすることで起動や再起動が高速化した．これにより構築ユーザーの負担が大きく軽減されたため, ビジネスにおけるアイデアから実装までの時間が高速化した．

2 番目の選択肢はマイクロサービスアーキテクチャと呼ばれ, コードベースを機能ごとに分割し, ビジネスの機能と開発チームとの高い頻度での連携が可能となった [1]．クラウドサービスが使われる背景として, ダイナミックに変化するトラフィックに対して可能な限り人間の介入なく柔軟なリソースの調整を期待されている [2]．しかしクラウドアプリケーションで用いられるマイクロサービスでは, スケーラビリティを担保するためにはマイクロサービスごとのリソース, ここでは CPU やメモリの使用量から 1 つあたりの pod のスペックを明示的に策定する必要があった．

3 番目はサーバレスと呼ばれ, マイクロサービスをさらに進化させたものである．構築ユーザーはプログラミング言語を選択するだけで自動的にプログラムの実行環境を構築できるようになった．プログラムの実行がより容易になる一方で構築ユーザーはどのようなタイミングでスケールするのか, ノードのリソースはどのように管理されているのかといった IaaS で管理できるパラメータとは縁遠くなる．

そのため, 現段階でのクラウドアプリケーションでは 3,4 番目の (マイクロサービス, サーバレス) の課題を解消する仕組みが必要である．

マイクロサービスおよびサーバレスではそれぞれのサービスのコードベースが独立しているがゆえにコンテナリソースの使いまわしができない．従来の研究及びクラウドアプリケーションにおけるマイクロサービスでの導入時にはサービスの優先度を人間が事前定義する必要があった．

そのため，ノードにおけるリソースの割り振りにおいても人間のオペレーションが発生する．

課題

クラウドアーキテクチャ，特にマイクロサービスおよびサーバレスを用いたクラウドアプリケーションにおいて,

[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

© 2021 Cloud and Distributed Systems Laboratory

ノードの計算リソース（割り当てられる CPU 量，メモリ量）が枯渇した際, ノードに余剰および待機している計算リソースがあるにも関わらず，ノードを追加している．以下の図 1 に示す.

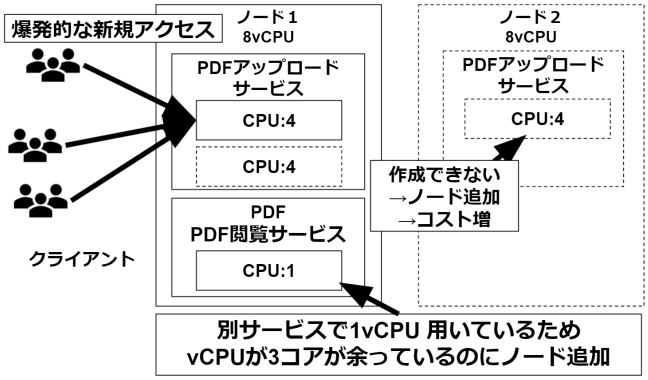

図 1　リソース枯渇による非効率なノード追加

これはノードおよびノードの集合体であるクラスター内において, クラウドアプリケーションの機能 (以降サービスと呼ぶ) に優先順位がつけられていないためである．サービスの優先順位がつけられていない場合，どのサービスの CPU 割当量を優先したらよいかわからない，言い換えればクラスター機能中は一部のサービスがインアクティブである場合すべてのサービスが重要と考えるため，サービスに割り当てられた CPU が枯渇した際にはノードを追加せざるを得なくなるためである.

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

クラウドにおけるキューへの優先度付けによるタスクスケジューリングではクラウド環境におけるタスクをキューに分割し，キューに GA(遺伝的アルゴリズム) を用いて優先度を付与した [3]．この手法ではクラウド環境における厳格なタスクの優先順位を算出した．しかし，この手法をマイクロサービスに適用した時にはそれぞれの機能がネットワークを介しているため，このモデル通りにタスクをスケジューリングすることは困難である.

クラウドコンピューティングにおける負荷分散のためのユーザー優先度ガイド付き Min-Min スケジューリングアルゴリズムではヘテロジニアスな環境が多いクラウド全体で見た CPU, メモリといったクラスターリソースの負荷分散のために，Min-Min スケジューリングアルゴリズムを用いることでクラウド全体 (以降クラスター) のリソースよりユーザーあたりが使用できるリソースの優先順位を求めている [4]．この手法ではクラスターにおいてユーザー単位, 多くは企業やグループといった単位でのリソースの管理が実現できているが，やはりこれもマイクロサービスのようなネットワークを通じて高度に機能と機能が相互通信している場合に課題が残っているとしている．また，この手法では VIP レベルと呼ばれる優先度をユーザーが手動でシステムに教えることでタスクごとの優先度を考慮した負荷分散ができるとしている．しかし，タスクやサービスの数が爆発的に増加するマイクロサービスではこれらの優先度を自動で設定すべきである.

マイクロサービスバックエンドシステムのクラウドリソースプロビジョニングのためのオンライン機械学習ではマイクロサービスに特化したアプリケーションのスケール手法を提案している [5]．この提案では機械学習を用いることでサービスごとの適切なプロビジョニングを行い，スケーリングを行っている．しかし彼らが議論で述べているように，現状彼らは CPU 使用率という一つのメトリックのみを用いている．そのため，CPU の使用量よりストレージの帯域使用量が多い時の方がメトリックとして価値のあるデータベースには対応できていない.

3. 提案方式

クラスターにおける計算リソースを余剰リソースも含めて利用するため，サービスの優先順位による計算リソースの共助モデルを提案する．複数のマイクロサービスからなるアーキテクチャにおける優先順位を定義する. 以降これを PoS(Priority of Service) と呼ぶ. 類似語として QoS がある. QoS との違いは QoS がネットワーク, ハードウェアの帯域制御を事前に定義して渋滞しないようにするのに対して PoS は”サービス”を主体に考え, 実際にクラウドアプリケーションを利用するユーザーが使用するサービスごとの目標パフォーマンスに優先順位をつけることである.

提案方式

課題を解決するために以下の 3 つを提案する

- マイクロサービスにおける優先順位の算出
- 2 つの異なるマイクロサービスの類似度計測
- コードもしくはプログラムの共有による他サービスプログラムの実行

マイクロサービスにおける優先順位の算出ではマイクロサービスアーキテクチャ上のネットワークを複数の指標を用いて順位を策定する．マイクロサービスにおける優先順位の算出では先述した優先順位のうち共助できるコンテナを探すために異なる 2 つのマイクロサービスの類似点をコ

© 2021 Cloud and Distributed Systems Laboratory

ンテナ内にコピーされたプログラムのコード及びコンテナのレイヤを用いて比較する．比較した結果, 類似していると認められる場合にはコードもしくはプログラムの共有による他サービスプログラムの実行がテストされ, 実行に備える．以下の図 2 に開発しているマイクロサービスである論文検索アプリケーション，doktor に提案手法を用いた際の共助部分を構成図を示す．

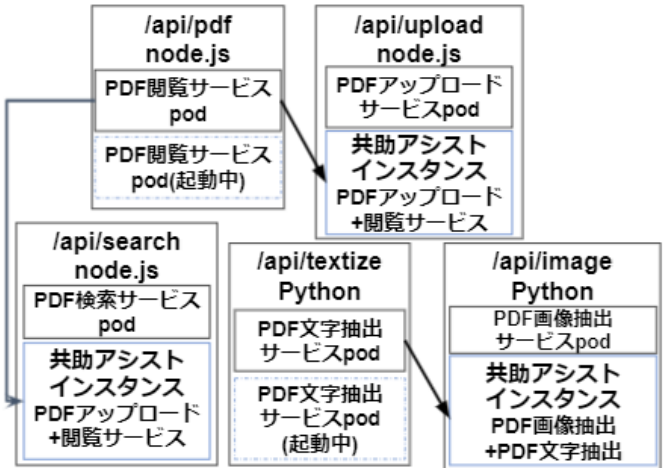

図 2　共助構成図

図 2 に示された doktor ではサービスが 5 つ存在する．それぞれのサービスのエンドポイントは/api/から始まり，以降の URL がそれぞれ pdf,upload,search,textize,image となっている．以下にそれぞれのサービスの説明を示す．

- /api/pdf doktor システムのデータベースから PDF を一覧表示するマイクロサービス.node.js で構築されている．
- /api/upload PDF を doktor システムのデータベースにアップロードするマイクロサービス.node.js で構築されている．
- /api/search PDF を doktor システムのデータベースからユーザーが入力したキーワードで論文を検索するサービス．node.js で構築されている．
- /api/textize upload サービスでアップロードされた PDF から文字部分を抽出するサービス．Python で構築されている．
- /api/image upload サービスでアップロードされた PDF から画像部分を抽出するサービス．Python で構築されている．

これら機能とは別に，各サービスで割り当てられている CPU が余剰しているとき，共助アシストインスタンスを作成する．共助アシストインスタンスは node.js で構築されているものは他の node.js のサービスを，Python で構築されているものは他の Python のサービスの処理を代行することができる．

3.1　マイクロサービスにおける優先順位の算出

マイクロサービスにおける優先順位の算出は各マイクロサービス間での通信を監視し, 以下のような指標を用いて算出する．またこれらの指標は以下の式で取り扱うため, 変数をおく．

CPU 使用率の平均を取得する.CPU の使用率は各サービスの 1 コンテナを起動するのにかかる時間を取得間隔とする． これは，各サービスの CPU 使用率はコンテナの数により大きく変化するためである．１コンテナを起動するのにかかる時間を取得間隔とすることで，各サービスのより実態に沿った CPU 使用率が取得できる．また，クラスター全体のスペックから取得した値を割ることで占有率が求められる．以下に式を示す. 平均を示す関数は avg() と示す．また式中に用いたメトリクスを以下のように定める．

- S_CPU(milicore),S_Memory(MiB)
  サービスが使用している CPU 使用量．単位はそれぞれ milicore,MiB である．
- C_CPU(milicore),C_Memory(MiB)
  クラスター全体の CPU, メモリ搭載量．単位はそれぞれ milicore,MiB である．

$$PoS(CPU) = \frac{\text{avg(S\_CPU)}}{\text{C\_CPU}} \tag{1}$$

$$PoS(Memory) = \frac{\text{avg(S\_Memory)}}{\text{C\_Memory}} \tag{2}$$

しかしこの指標だけを用いた場合にはログや, 死活監視用のトラフィック (以降ログサービスのトラフィック) が存在するときにアプリケーション本体の優先順位よりログサービスのトラフィックが重要視されることになる．これを回避するため, 独自の優先順位用の指標として接続されているサービス数を取得する．

これは現状のリクエストとは関係なく, 今までにアクセスされたユニークなサービスの個数を取得する．これにより, 直近のデータだけではなく, 長期的なアクセスに基づき, PoS を補正することができる．

上記に示した指標を用いて以下のように PoS を計算する．

PoS(CPU) 及び PoS(メモリ) の値が同じ時には接続されているサービス数を用いて順位付けを行う．

最後にこれらの PoS 値をランク付けし, 1 から m 番の値を順番に振っていく．m はマイクロサービスの数に応じて変化する．クラスターにおけるマイクロサービスが 1 から 100 の場合は m=100, 1000 以上の場合はサービス数を 10 で割った段階とする．これはマイクロサービス全てにおいての順位付けを行うと, PoS の計算専用のリソースを用意する必要が出てくることを回避するためである．

3.2　異なる 2 つのマイクロサービスの類似度計測

マイクロサービスにおける優先順位の算出では，先述した優先順位のうち共助できるコンテナを探すために異なる

2 つのマイクロサービスの類似点をコンテナ内にコピーされたプログラムのコードをコンテナのレイヤを用いて比較する．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．

コンテナレイヤの比較

異なるサービスにおいて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ソースコードやプログラムを実行するため，ID が全てにおいて同じであることは少ないと言える．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．

3.3 ソースコードもしくはプログラムの共有による他マイクロサービスプログラムの実行

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．

ユースケースシナリオ

ユースケースシナリオとして複数のマイクロサービスからなる論文検索サイト, doktor において PoS を用いてリソースの有効活用とレスポンスタイムの高速化を確認できる．

ここでは doktor.com に二つのサービスがある．ここで，海外の研究者が doktor に着目し，急激に多量のトラフィックが増加したとする．その時に pdf サービスはスケール待ち状態となる．ここで提案する PoS とプログラム共有を行うことにより，クラスター内の余剰リソースを有効活用することができる．

4. 実装と実験方法

実装

以下の 3 つのソフトウェアを実装する．

- 優先順位 POS を自動計算するソフトウェア

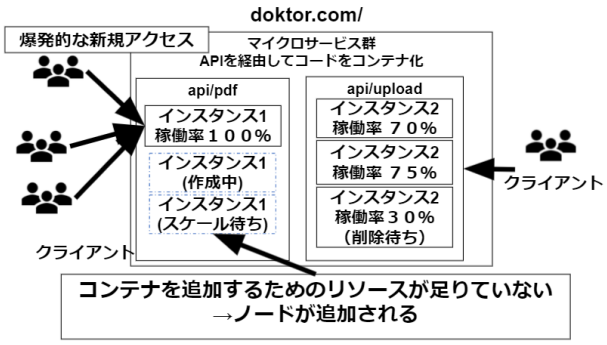

図 3 ユースケース

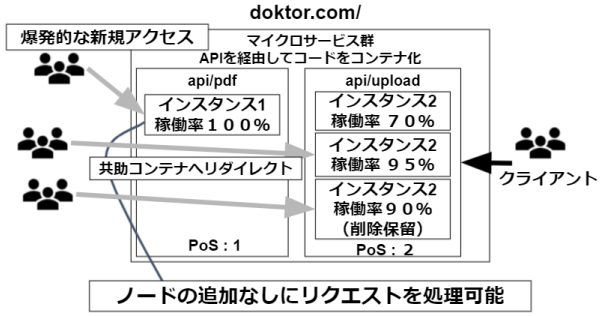

図 4 ユースケース 2

- 異なるふたつのマイクロサービスのコンテナを比較するソフトウェア
- 異なるコードベースのサーバレス機能ごとのコードをシェアリングするソフトウェア

実験環境

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．

これらのサービスの構成表を以下の表 2 に示す．これらは以下のマイクロサービス (図内では μS と表記) で構成される．

以上の構成を元に作成した実験構成図を以下の図 3 に示す．

4.1 各サービスの実装詳細

4.2 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 Kubernetes クラスターを構築するため, 以下の表 3 に示すスペックの VM を 2 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

サービス名	役割
Web サーバー	ユーザーは WEB から登録, 検索をする
API サーバー	直接バックエンドサービスに リクエストを転送する
アップロードサービス	論文の登録を管理する μS
検索サービス	PDF をタイトルによって検索する$ μS
論文ファイル textize	PDF のファイルから文字を抽出する$ μS
DB 共通サービス	DB へのアクセスを管理する μS
タイトル/著者用 DB	タイトルと著者を保存するデータベース
タイトル/PDF DB	タイトルと PDF を保存するデータベース

表 1 実装構成表

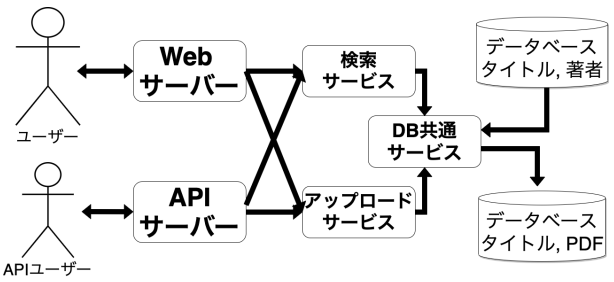

図 5 実験概要図

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

表 2 ESXi サーバーのスペック

いる microk8s を使用した.

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

表 3 各ノードのスペック

4.3 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 max2):Web サーバー
- Flask(Scalable max3):アップロードサービス
- Flask(Scalable max3): 検索サービス

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.

5. 評価手法と分析手法

評価として, マイクロサービスが 15 個稼働している環境に API を用意しアクセスし, ユーザーが優先順位付けしたサービスと. 本研究にプログラムが出力する順位付けが正しいか評価する. また, 上記の順位付けを用いてノード内で CPU がメモリが効率的に割り当てられたかを確認する. ここではクラスターを構成している各ノードの計算リソースの使用量を以下のように求める.

$$\text{CPU 効率} = \frac{\text{ノードで使用されている CPU 量}}{\text{ノードに搭載されている CPU 量}} \quad (3)$$

$$\text{メモリ効率} = \frac{\text{ノードで使用されているメモリ量}}{\text{ノードに搭載されているメモリ量}} \quad (4)$$

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.

6. 議論

PoS の算出方法ではトラフィック量と他サービス及び外部からの接続数を 1 つの指標として利用した. 一方で単純なトラフィック量と接続数のみで判断すべきでないリソースも存在する. ここでは現状使用している指標以外で判断すべき 1 つの例としてログについて説明する. コンピュータシステムにおいて, ここではクラウド環境上において障害管理の観点からログを取得することは重要とされている [6]. そのためクラウド環境上では多数のログを取得するサービス (以降, ログサービス) が実行されている [7]. しかし, PoS にサービスにおける優先順位の内にログサービスを入れることは望ましくない. ログサービスはクラウドアプリケーションのアプリケーション開発時及びエラー時のデバッグや停電, 過電流や CPU ストレージの破損やハードウェアの故障による復旧の際に極めて重要な情報群であるからだ. そのため, ログサービスに関わるリソースの効率化やレスポンスタイムの向上は, クラウドアプリケーションとは別に独立した形でするべきである. しかし, これを実現するためにはクラウドアプリケーションとログサービスの自動検出及び判別, そしてグルーピングが必要である. また, ログはアプリケーションと深く関わっているため, 単純な方法ではプログラムを判別できない可能性がある. 例えば, fluentd や Elasticsearch といったログエージェント, ログ検索エンジンはトラフィックの内容やコンテナに用いられるイメージ名からルールベースで検出することが可能である. しかし, クラウドアプリケーションを作成している企業が独自のログサービスを構築した場合, ルールベースでは検出が難しい. そのため, サービス間での通信のうち, 日付や共通の単語が多い時にはログと判断し, アプリケーション側の PoS の対象外, もしくは別の基準の採用を利用する. これにより, クラウドアプリケーション側の PoS は

© 2021 Cloud and Distributed Systems Laboratory

ログを排除したものになるため, より PoS の検出の精度が上昇する.

また，ユーザーの任意で PoS の計算より除外できる仕組みが必要である．構築ユーザーがセキュリティの観点といった意図的に PoS の計算より除外したいサービスが発生することが想定される．そのため，以下のような構築ユーザー側で明示的に PoS の動作を制御するコードを導入する．以下のソースコード 1 に示す.

ソースコード 1 明示的に PoS システムへの参加を拒否するコード

```
apiVersion: v1
kind: Service
metadata:
  name: pdf-textize
  pos: False #POS による自動優先順位の決定を許可
  beHelper: False # True にすることで他サービスが
  #このサービスのリソースを使うことを許可
spec:
  type: NodePort
  ports:
  - port: 80
    targetPort: 5000
    name: http
  selector:
    app: flask
```

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 [8]. 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.

7. おわりに

従来の研究及びクラウドアプリケーションにおけるマイクロサービスでの導入時にはサービスの優先度は人間が事前定義する必要があった. そのため, ノードのリソースが枯渇した際, どのマイクロサービスを優先的に処理するかを動的に策定する手法が存在していなかった．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．これによりコンテナの類似度計算及びプログラム共有を行うことでクラウドを構築するノードのリソース (CPU, メモリ, ストレージ領域) を効率的に利用することができた.

参考文献

[1] 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*, pp. 195–216 (2017).

[2] Lehrig, S., Eikerling, H. and Becker, S.: Scalability, elasticity, and efficiency in cloud computing: A systematic literature review of definitions and metrics, *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures*, pp. 83–92 (2015).

[3] Keshanchi, B., Souri, A. and Navimipour, N. J.: An improved genetic algorithm for task scheduling in the cloud environments using the priority queues: formal verification, simulation, and statistical testing, *Journal of Systems and Software*, Vol. 124, pp. 1–21 (2017).

[4] Chen, H., Wang, F., Helian, N. and Akanmu, G.: User-priority guided Min-Min scheduling algorithm for load balancing in cloud computing, *2013 National Conference on Parallel Computing Technologies (PARCOMPTECH)*, pp. 1–8 (online), DOI: 10.1109/ParCompTech.2013.6621389 (2013).

[5] Alipour, H. and Liu, Y.: Online machine learning for cloud resource provisioning of microservice backend systems, *2017 IEEE International Conference on Big Data (Big Data)*, pp. 2433–2441 (online), DOI: 10.1109/BigData.2017.8258201 (2017).

[6] Chuvakin, A., Schmidt, K. and Phillips, C.: *Logging and log management: the authoritative guide to understanding the concepts surrounding logging and log management*, Newnes (2012).

[7] Khan, S., Gani, A., Wahab, A. W. A., Bagiwa, M. A., Shiraz, M., Khan, S. U., Buyya, R. and Zomaya, A. Y.: Cloud log forensics: foundations, state of the art, and future directions, *ACM Computing Surveys (CSUR)*, Vol. 49, No. 1, pp. 1–42 (2016).

[8] 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).

© 2021 Cloud and Distributed Systems Laboratory