異なるコンテナのライブラリ比較による完備率の算出

山口 舜[1]　飯島 貴政[2]　串田 高幸[1]

概要：コンテナに割り当てられた CPU とメモリ容量は他のコンテナで使用できない．このため異なるコンテナにアプリケーションをコピーし，1 つのコンテナで 2 つのアプリケーションを実行させることにより，余剰した CPU とメモリ容量を活用する手法がある．しかし，この手法において異なるコンテナでアプリケーションを実行するための要件が満たされているか判断できないことが課題となる．そのため本研究では完備率という異なるコンテナでアプリケーションを実行するためのライブラリがどの程度満たされているか判断する指標を作成する．完備率は異なるコンテナで実行したいアプリケーションに依存したライブラリが何%コピー先のコンテナに含まれているかを算出した値である．実験はアプリケーションの実行に必要なライブラリのインストール時間とインストールしたデータサイズと算出した完備率を比較し評価を行う．

1. はじめに

背景

コンテナ技術を用いたマイクロサービスというソフトウェア開発技法がある [1][2]．このマイクロサービスはサービスごとにアクセス数により CPU 使用量とメモリ使用量で偏りができることがある．これにより 1 つのサービスで CPU 使用量とメモリ使用量を独占しないよう，人の手でコンテナごとに CPU とメモリ容量を割り当てる必要がある．しかし，それぞれのサービスのコンテナごとに CPU とメモリ容量を割り当てると他のサービスのコンテナで使用できない．この解決方法としてマイクロサービスごとに優先的な CPU の処理量とメモリ容量を割り当てる優先度という指標を作成する．優先度はマイクロサービスの相互通信のログとノードにおける CPU 使用率，メモリ使用量，稼働時間を取得することで算出する．優先度が高いサービスを実行環境（アプリケーションの実行に必要なライブラリ）が類似しているサービスにリクエストを分散することで，処理に必要とされる CPU の処理量とメモリ容量を類似しているサービスが補うという方法がある [3]．この異なるコンテナにリクエストを分散し処理するために類似コンテナの検索，プロクラムのコピー，プログラムの実行の 3 ステップで実行する必要がある．図 1 にコンテナ内で行う処理を別のコンテナの余剰リソース（CPU の処理量とメモリ容量）を持つコンテナを用いてリクエストを分散する例を示す．この例ではコンテナ A にリクエストが大量に来ることで割り当てた CPU 使用率が 100%になり，コンテナ B は割り当てた CPU 使用率が 20%であったとする．この場合コンテナ B にプログラム A をコピーし，コンテナ B でコンテナ A の処理を行うことでリクエストを分散できる．これにより新たにノードを追加することなくリクエストを処理できるため，ノードにおける CPU の処理量とメモリ容量を効率よく使用できる．

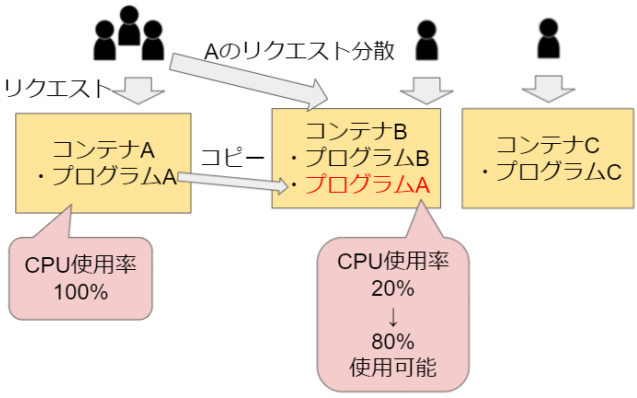

図 1　異なるコンテナでリクエスト分散

このコンテナ内で行う処理を別コンテナの余剰リソース（CPU とメモリ容量）を持つコンテナで行うため，異なるコンテナで同じアプリケーションを実行できるようにアプリケーションの実行に必要なライブラリをそろえる必要がある．図 2 のようにコンテナ A のアプリケーションである Python ファイル A をコンテナ B で動かすためには Flask をインストールする必要がある．またコンテナ C で

[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

Python ファイル A を実行するには Flask がもともとインストールされているため, Python ファイル A を実行するために新たにライブラリをインストールする必要がない. このようにコンテナの実行環境ができるだけ等しいコンテナでプログラムを実行することにより, 少ないコンテナ実行環境の変更でプログラムを実行できる.

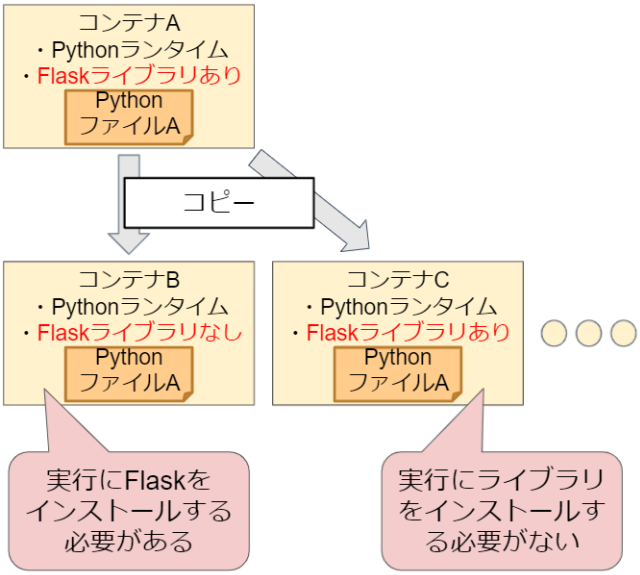

図 2 コンテナ内実行環境の違い

課題

異なるコンテナでアプリケーションを実行するための実行環境（アプリケーションを実行するのに必要なライブラリ）が完備されているかを示す指標が存在しない. そのためどのコンテナどうしの実行環境が類似しているか判断できないことが課題である. そのためコンテナで実行しているアプリケーションを異なるコンテナにコピーした際, アプリケーションを実行できるか判断できない. 例として図 3 のようにコンテナ A で実行できるアプリケーションである Python ファイル A をコンテナ B またはコンテナ C にコピーして実行できるかわからない. これはアプリケーションを実行するために必要なライブラリが完備されているか判断できないためである. また実行できなかった場合コンテナ B とコンテナ C のどちらの実行環境が似ているか判断する指標が存在しない. そのためコンテナ B とコンテナ C のどちらのコンテナのほうが実行に必要なライブラリをインストールする数が少なく Python ファイル A を実行できるのか判断できないことを課題とする.

各章の概要

第 2 章では本研究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方式とユースケースについて述べる. 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.

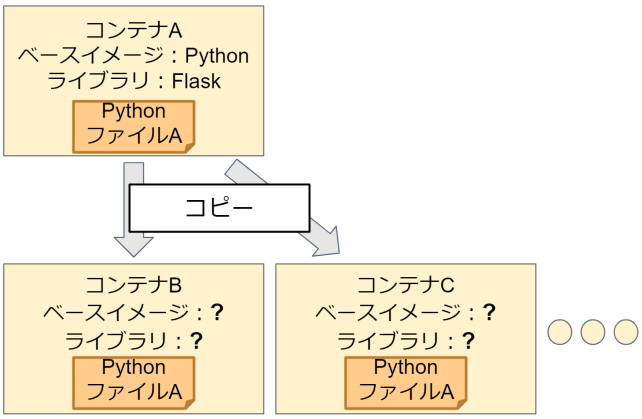

図 3 コンテナ実行環境のブラックボックス

2. 関連研究

Shripad Nadgowda らはコンテナ状態の完全な移行についての研究を行った [4]. この研究は Open Container Initiative（OCI）の原則にしたがって設計された Voyager（ジャストインタイムのライブコンテナ移行サービス）を紹介する. Voyager は一貫したフルシステム移行を提供するファイルシステムにとらわれず, ベンダーにとらわれない新しい移行サービスである. これにより Voyager は CRIU ベースのメモリ移行とユニオンマウントのデータフェデレーション機能を組み合わせることで移行のダウンタイムを最小限に抑えることができる. しかし, この移行ソフトウェアでは移行のしやすさは考えられていないため, どの環境からどの環境へ移行がしやすいかがわからず, 移行のしやすさについての指標は作成されていない.

Van Tran らはクラウドへのアプリケーションの移行についての研究を行った [5]. 一部のソフトウェアはクラウド専用にゼロから作成されているが, 多くの組織は既存のアプリケーションをクラウドプラットフォームに移行することを望んでいる. しかし, プログラミングモデルやデータストレージ API などのソフトウェア環境の違いがあるためいくつかの変更を加える必要がある. そのためクラウドプラットフォームへの移行は簡単ではない. このことから関連する移行タスクの分類法を提案し, タスクのカテゴリ間のコストの内訳を示している. また移行タスクのコストに影響を与える重要な要因について示している. この手法ではアプリケーションの移行について処理内容を分類することにより処理を分け, 分類したそれぞれの処理で移行作業をすることによりコストを算出している. しかし, 移行の分類ごとで移行のしやすさについては調査されていない.

Kunal Mahajan らはコンテナの類似性を利用したコン

テナ展開のコールドスタート対処についての研究を行った [6]．サーバーレスコンピューティングの「コールドスタート」という現象は, 既存の仮想化展開システムをサポートするコンピューティング, ストレージ, およびネットワークリソースの使用率に関して固有の効率を低下させる．そのため展開されているアプリケーション間のデータの類似性を特定を行う．これらに基づいてピアツーピアネットワーク, 仮想ファイルシステム, およびコンテンツアドレス可能ストレージの上に構築された新しい展開システムを提案した．この類似性分析手法ではアプリケーション間でソースコードの類似性をブロックしており, GitHub として使用される 20 の最も一般的な PyPI パッケージを特定しているため 20 個のライブラリでしか分析がされていない．

3. 提案方式

本研究で述べた課題を解決する提案方式とユースケースシナリオについて説明する．

提案方式

課題の解決方法として異なるコンテナでアプリケーションを実行するために必要なライブラリが完備されているかの指標として完備率という新たな指標を作成する．完備率の算出式を以下に示す．

$$C = \frac{R_s}{R_a} \tag{1}$$

変数 C は完備率, 変数 R_s は比較元のライブラリの中で比較先にあるライブラリ数, 変数 R_a は比較元のライブラリ数とする．この完備率を用いることにより, どのコンテナの実行環境が同一ライブラリを多くインストールされているかを数値から明確に判断できる．この明確な判断とは完備率が高いほどコンテナ内で実行するために必要なライブラリの差が少なく同一ライブラリが多いコンテナであることを示す．また完備率から異なるコンテナでアプリケーションを実行できるか判断できるようにすることで本研究の課題を解決する．

この完備率を考えるうえで比較する 2 つのコンテナのコンテナイメージレイヤのベースイメージに同じベースイメージを使用する．例として比較する 2 つのコンテナのベースイメージに python:bullseye を用いる．また比較するコンテナで実行されるアプリケーションは Python で作成されたアプリケーションを用いることを前提とする．完備率の算出方法は異なるコンテナ 2 つから同一なライブラリを比較することで求め, 完備率算出式を用いて 0 ≦完備率≦ 1.0 の値で算出する．なお前提としてベースイメージを固定しているため Python の標準ライブラリは同一のものを使用しているため, 比較するライブラリは外部ライブラリのみとなる．また比較するライブラリはバージョンも等しいときのみ同一のものとして定める．これはバージョン違いによりアプリケーションが実行できない可能性があるためである．これよりライブラリの比較にはバージョンを含めた文字列で比較する．2 つのコンテナから取得したライブラリをもとに算出する完備率の算出式は式（1）となる．図 4 に 2 つのコンテナのライブラリ比較の例を示す．完備率の算出式を用いて, 比較元のライブラリの中で比較先にあるライブラリ数が Flask と dateutil の 2 つなので Rs は 2, 比較元のライブラリ数が Flask と pillow と dateutil と Numpy の四つなので Ra は 4 となるので $\frac{2}{4} = 0.5$ となり完備率は 0.5 となる．

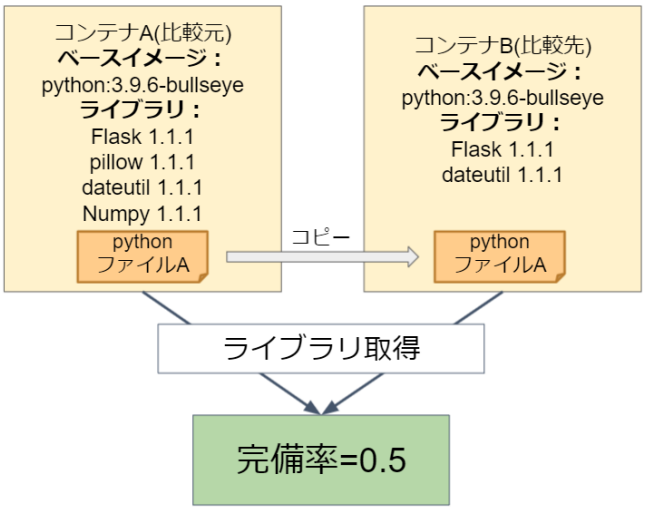

図 4 ライブラリ比較例

この完備率をビルドされているすべてのコンテナで算出することで, もっとも同一ライブラリの多いコンテナを見つけることができる．また算出した完備率によりアプリケーションが実行できるか判断できる．比較元と比較先のコンテナ内にあるライブラリが一致しているときは完備率を 1 となり, 異なるコンテナでアプリケーションが実行できると判断できる．また比較元と比較先のコンテナ内にあるライブラリが異なるとき完備率を 0 以上 1 未満となり, 足りないライブラリを比較先のコンテナにインストールすることでアプリケーションを実行できると判断できる．

ユースケース・シナリオ

ユースケースとして論文掲載サービスである doktor[*1] というマイクロサービスを用いる．doktor には search, upload, website, pdf-image, pdf-textize, pdf-thumbnail があり, それぞれ別のコンテナで動いている．そのため完備率を用いて類似しているコンテナを見つけることによりコンテナを少ない変更量でリクエストを分散できる．図 5 のように search コンテナ, upload コンテナ, website コンテナ,

[*1] https://github.com/cdsl-research/doktor (参照 2021-09-06)

pdf-image コンテナ, pdf-textize コンテナ, pdf-thumbnail コンテナがあるとき, search コンテナの割り当てられた CPU 使用率が 100%になり, 処理を補いたい場合がある. この場合提案手法により完備率を算出したとき, 比較元を search コンテナ, 比較先を upload コンテナとして完備率を算出したとき完備率が 0.8 となり, 比較先が website コンテナのとき 0.5, 比較先が pdf-image コンテナ, pdf-textize コンテナ, pdf-thumbnail コンテナのとき 0.1 となった場合がある. 算出した完備率により search コンテナと完備率の高い upload コンテナが類似していることがわかり, upload コンテナで処理を補うことで upload コンテナを少ない変更で search コンテナのアプリケーションである main.py の実行ができるようになる.

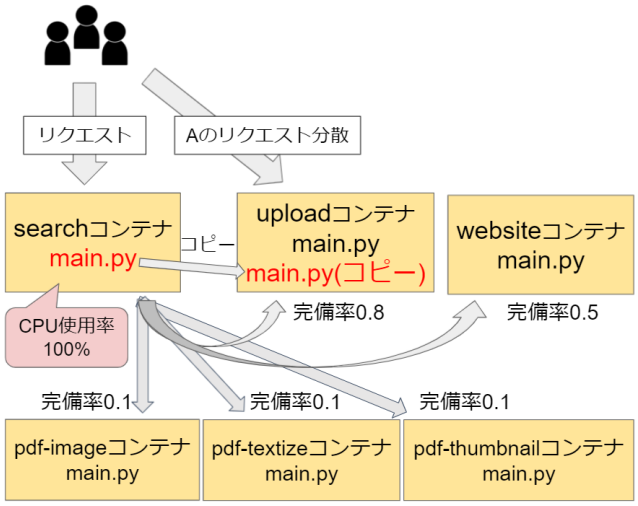

図 5 完備率使用例

4. 実装と実験方法

本研究で述べた提案方式をもとに開発したソフトウェアの実装と実験方法について説明する.

実装

実装は本研究の提案内容を行うソフトウェアとして ComCal という名前のソフトウェアを新たに作成する. このソフトウェアの機能として比較元のコンテナを 1 つ定めることで, 同じマシン上に立てられたすべてのコンテナと完備率計算を行い, もっとも同一ライブラリが多いコンテナを求める. 完備率算出するための Docker で作成されたコンテナの外部ライブラリの取得方法として docker exec コマンドを用いてコンテナの中で pip freeze コマンドを用いることで取得する [7]. 取得した外部ライブラリを比較元と比較先で比較することで同一のライブラリを取得する. これらの情報をもとに提案で述べた完備率算出式を用いて完備率を算出する. 図 6 に ComCal で完備率を算出する際の例を示す. コンテナ A（ID:915508fdb9f2）を比較元, コンテナ B（ID:a7f40ba2daf0）を比較先のコンテナとすると, コンテナ A とコンテナ B からライブラリを取得することにより完備率を算出する.

また図 6 の例のコンテナで ComCal を使用した出力例を以下に示す.

python3 ComCal.py 実行結果

```
比較元のコンテナイメージ名またはコンテナイメージ ID を入力してください
>> 915508fdb9f2
比較先のコンテナイメージ名またはコンテナイメージ ID を入力してください
>> a7f40ba2daf0
比較元のライブラリ一覧
click==8.0.1
Flask==2.0.1
itsdangerous==2.0.1
Jinja2==3.0.1
MarkupSafe==2.0.1
numpy==1.21.2
python-dateutil==2.8.2
six==1.16.0
Werkzeug==2.0.1
比較先のライブラリ一覧
numpy==1.21.2
比較元ライブラリ数：9
比較元の中で比較先と同じライブラリ数：1
完備率：0.1111
```

この出力結果からコンテナ A には click 8.0.1, Flask 2.0.1, itsdangerous 2.0.1, Jinja2 3.0.1, MarkupSafe 2.0.1, numpy 1.21.2, python-dateutil 2.8.2, six 1.16.0, Werkzeug 2.0.1 の 9 個のライブラリを持ち, コンテナ B には numpy 1.21.2 の 1 個のライブラリを持っていることが出力される. この取得したライブラリから完備率は 0.1111 であることを出力する.

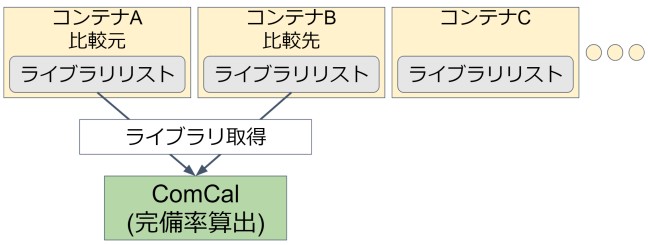

図 6 ライブラリ取得ソフトウェア

実験環境

実験環境は仮想マシン（VM）上に Docker をインストール

した環境を用いる [8][9]. VM の作成に ESXi を用いる [10]. VM の構成情報を下記に示す.

- VM 構成情報
  OS　Ubuntu-20.04.2
  vCPU　4 コア
  RAM　8GB
  HDD　50GB

また VM に実装で作成した ComCal を入れて実験する. これらの実験に用いる環境を図 7 に示す. Docker を用いることで実験で使用するコンテナを作成し, ComCal を使用することで完備率の算出を行う.

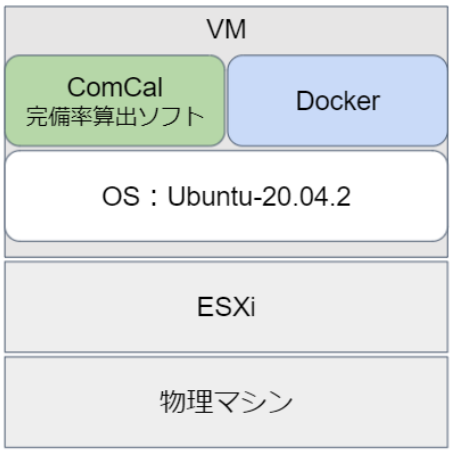

図 7　実験環境

5.　評価手法と分析手法

まず実験に使用するコンテナを用意する. 本研究ではベースイメージを python:3.9.6-bullseye で行うことを前提としているため, ベースイメージに python:3.9.6-bullseye を用いて外部ライブラリをインストールしたコンテナをビルドする. インストールするライブラリの例として pickle, dateutil, calendar, Numpy, matplotlib, pandas, Pillow, flask を用いる. また Docker Hub からベースイメージに python:3.9.6-bullseye を用いたコンテナイメージをダウンロードしビルドする. 実際に使用するコンテナの例として python:3.9.6-bullseye のベースイメージを用いた pip install Flask を行った Python ファイルを持つコンテナイメージを用いる. ビルドしたコンテナから完備率算出で使用する比較元のコンテナを 1 つ定め, 比較元のコンテナと他のコンテナで提案手法に基づいて完備率を算出する.

評価は比較元のコンテナのアプリケーションを比較先のコンテナで実行できるのかアプリケーションとアプリケーションの依存ライブラリをインストールし実行する. 実際にアプリケーションを実行することができたかと, 算出した完備率から提案手法に基づいてアプリケーションが実行できるか判断した結果があっていたかを異なるライブラリを持つコンテナで行い, どの割合で完備率から予測した結果が当たっていたのか評価する. また実験結果から完備率が 0 以上 1 未満の場合, プログラムを実行するためにライブラリをコンテナにインストールする必要がある. このプログラムの実行に必要なライブラリのインストール時間とインストールしたデータサイズと算出した完備率を比較する. これにより完備率がコンテナ内の実行環境を変更する際に参照するのに適した値を算出しているか評価する.

6.　議論

本研究の完備率算出の際にライブラリのバージョンが一致しているときのみ同一のライブラリとしているため, バージョンが異なる場合でもプログラムを実行できる可能性がある. そのためバージョンが異なる場合でも同じライブラリとみなし, 完備率に反映させる方法として 2 つの方法がある. 1 つはセマンティックバージョニングに定められている互換性を用いて完備率に適用させる方法 [11]. もう 1 つの方法は実際にアプリケーションを実行させて互換性があるかを確かめて完備率に適用させる方法がある. 1 つ目の方法はセマンティックバージョニングを適用しているライブラリのみ使用できる. セマンティックバージョニングを用いたバージョンナンバーは x.y.z で表し, x をメジャーバージョン, y をマイナーバージョン, z をパッチバージョンとしている. バージョンを上げる際に API の変更に互換性のない場合はメジャーバージョンを上げる. 後方互換性があり機能性を追加した場合はマイナーバージョンを上げる. 後方互換性を伴うバグ修正をした場合はパッチバージョンを上げるという形式が取られている. 図 8 にセマンティックバージョニングを用いたバージョンの互換性について示す. 図 8 の例ではライブラリである Flask をセマンティックバージョニングに基づいて互換性があるかを示している. 図 8 の一番上のパッチバージョンが異なる場合は互換性があるとする. 図 8 の真ん中のマイナーバージョンが異なる場合は新しいバージョンが古いバージョンを互換している後方互換性があるとする. 図 8 の一番下のメジャーバージョンが異なる場合は互換性がないとする.

またもう 1 つの方法としてバージョンが異なるライブラリでアプリケーションを実行することで, 実行できたらバージョン違いでも互換性があるとして同一のライブラリとしてみなし, 実行できなかった場合同一のライブラリとは見なさず完備率を算出するという方法がある.

また同じ完備率の場合どちらのコンテナの実行環境がより類似しているか判断できないという問題がある. 例としてある Python ファイルを実行したいとき, ライブラリの Flask がない場合とライブラリの dateutil がない場合では, 同じ差が 1 つで完備率が同じでも実行環境構築の処理量は Flask のデータサイズが 10.6MB と dateutil のデータサイズが 1.01MB なので Flask をインストールするほうがコ

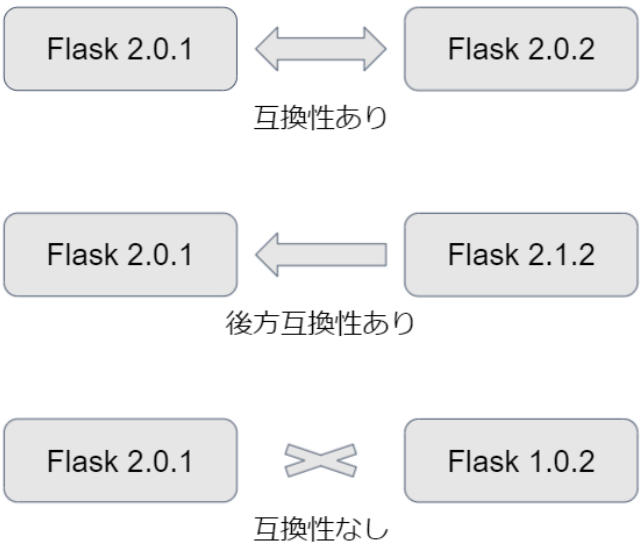

図 8 バージョン互換性

ンテナの変更量が多くなる．そのためライブラリのインストールすることによるストレージ使用量で同じ 1 つでも重さを加えるという方法がある．

他にも本研究ではベースイメージを固定し, アプリケーションを作成する実行ファイルを Python とすることを前提としているためベースイメージ依存になる Python の標準ライブラリは同じになる．しかし, 同じ Python ファイルを実行したい場合でもベースイメージはたくさんの種類があるため, ベースイメージが同じものが 1 つのマシン上に複数ビルドされていることはほとんどない．また Python の標準ライブラリはベースイメージ依存となる．そのためベースイメージが異なり標準ライブラリが異なる場合でも完備率を算出する方法が求められる．このベースイメージが異なる場合の完備率の算出方法として本提案では外部ライブラリのみ完備率算出に用いたが, 標準ライブラリも完備率算出のためのライブラリ数に加える方法が考えられる．またもう 1 つの完備率算出方法として実行ファイルに使用されているライブラリのみを算出に使用するライブラリ数とする方法が考えられる．この場合比較元の Python ファイルを解析し, 使用しているライブラリが何個比較先のライブラリで使用できるかを取得することで算出できる．

7. おわりに

本研究の背景にマイクロサービスはサービスごとにアクセス数, CPU, メモリ使用量に偏りができることがある．しかし, それぞれのサービスのコンテナごとに割り当てられた CPU とメモリ容量を他のサービスで使用できないという問題がある．この解決方法として異なるサービスのコンテナにアプリケーションをコピーし, 1 つのコンテナで 2 つのアプリケーションを実行させることにより, 異なるサービスのコンテナにリクエストを分散するという方法がある．この異なるサービスのコンテナでアプリケーションを実行するにはコンテナの実行環境（アプリケーションの実行に必要なライブラリ）を同じにする必要がある．このことから本研究の課題は異なるコンテナでアプリケーションを実行するための実行環境が完備されているか判断できないこととした．そのため完備率という 2 つのコンテナの実行環境が完備されているか判断する指標を作成することにより課題の解決を行った．また完備率を用いて異なるコンテナでアプリケーションを実行するためのライブラリがどのくらい完備されているかを示す．実験はアプリケーションの実行に必要なライブラリのインストール時間とインストールしたデータサイズを算出した完備率と比較する．これにより完備率がコンテナの実行環境を変更する際に参照するに適した値を算出しているか評価する．この完備率を用いてコンテナを比較する手法を提案することによりコンテナ内の実行環境が類似しているコンテナを見つけられるようになり, ユースケースで使用することによりノードにおけるリソース効率よく処理できる．

参考文献

[1] Sofia, Z.: Container technologies, *Hypatia*, Vol. 15, No. 2, pp. 181–201 (2000).
[2] 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).
[3] 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.
[4] Nadgowda, S., Suneja, S., Bila, N. and Isci, C.: Voyager: Complete container state migration, *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, IEEE, pp. 2137–2142 (2017).
[5] Tran, V., Keung, J., Liu, A. and Fekete, A.: Application migration to cloud: A taxonomy of critical factors, *Proceedings of the 2nd international workshop on software engineering for cloud computing*, pp. 22–28 (2011).
[6] Mahajan, K., Mahajan, S., Misra, V. and Rubenstein, D.: Exploiting content similarity to address cold start in container deployments, *Proceedings of the 15th International Conference on emerging Networking EXperiments and Technologies*, pp. 37–39 (2019).
[7] Bernstein, D.: Containers and cloud: From lxc to docker to kubernetes, *IEEE Cloud Computing*, Vol. 1, No. 3, pp. 81–84 (2014).
[8] Goldberg, R. P.: Survey of virtual machine research, *Computer*, Vol. 7, No. 6, pp. 34–45 (1974).
[9] Potdar, A. M., Narayan, D., Kengond, S. and Mulla, M. M.: Performance evaluation of docker container and virtual machine, *Procedia Computer Science*, Vol. 171, pp. 1419–1428 (2020).
[10] Walters, J. P., Younge, A. J., Kang, D. I., Yao, K. T., Kang, M., Crago, S. P. and Fox, G. C.: GPU passthrough performance: A comparison of KVM, Xen, VMWare ESXi, and LXC for CUDA and OpenCL applications, *2014 IEEE 7th international conference on cloud computing*, IEEE, pp. 636–643 (2014).

[11] Decan, A. and Mens, T.: What do package dependencies tell us about semantic versioning?, *IEEE Transactions on Software Engineering* (2019).

