情報処理学会論文誌 Vol.0 No.0 1–11 (??? 1959)

マイクロサービスにおけるIPアドレスの再利用に柔軟なコンテナへのユニークな識別子の付与

飯島 貴政[1,a)] 串田 高幸[1,b)]

受付日 xxxx年0月xx日, 採録日 xxxx年0月xx日

概要：マイクロサービスアーキテクチャの実装の一つである Kubernetes ではイベント (スケールアウトやコンテナのライフタイム) で IP アドレスが再利用される．再利用された IP アドレスが含まれているコンテナのログはイベントの前後の判別が不可能という課題がある．本研究では Kubenetes 環境において, 環境内で各コンテナが自立してユニークな ID(CUID) を作成, 保存, 及びアプリケーションコンテナのログに自動付与するマネジメント用のコンテナを提案する．コンテナに割り当てられる IP アドレスが再利用された場合にもログの振り分けが可能である．実際の環境で使用可能かを確認するため, 実証実験としてマイクロサービスを用いた論文検索サービスを作成し, Istio とのレスポンスタイムの比較を行った．本提案の実装では約 13.3%のみのオーバーヘッドで各ログに付与することができる．この提案によって大規模化するマイクロサービスにおいてコンテナレベルでのサービスのデバッグと障害分析の負担を軽減した．

キーワード：分散処理, 分散トレーシング, マイクロサービス, サービスメッシュ

Container identification independent of container lifetime

Takamasa Iijima[1,a)] Takayuki Kushida[1,b)]

Received: xx xx, xxxx, Accepted: xx xx, xxxx

Abstract: In Kubernetes, which is one of the implementations of microservice architecture, IP addresses are reused by events (scale-out and container lifetime). In the Kubernetes environment, IKontainer creates and stores a unique ID for each container in the environment. The CUID is automatically assigned to the application container log. The log can be sorted even when the IP address assigned to the container is reused. To confirm whether the proposed system can be used in a real environment, we created an article search service using microservices as a demonstration experiment, and compared the response time with that of Istio. In our proposed implementation, only about 13.3% of the overhead can be assigned to each log. This proposal reduces the burden of debugging and fault analysis of the service at the container level in large-scale microservices.

Keywords: Distributed systems, Distributed processing, Distributed tracing, Microservices, Service mesh

1. はじめに

1.1 背景

2000 年の初期に提唱されたソフトウェアの設計及びアーキテクチャ手法であるサービス志向アーキテクチャ (Service Oriented architecture) はアプリケーションは機能ごとに分割する [1]．近年では SOA の各機能ごとを更に独立させた Micro Service Architecture(MSA) が用いられている．このアーキテクチャはそれぞれの機能は独立しているが, ネットワークを介して通信を行っている．機能ごとの通信はそれぞれの機能同士で行うため ESB(Enterprise Service Bus) のようなリクエストを管理するモジュールは存在しない．そのため ESB のような単一障害点を回避できることが特徴の一つである．また, サーバーは機能をシャットダウ

[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
TUT, Hachioji, Tokyo 192–0982, Japan
[a)] g21210077d@edu.teu.ac.jp
[b)] kushida@acm.org

ンさせることなく動的に計算リソース (CPU, RAM) を増減できる. MSA では各機能に対してロードバランサーを事前に定義しておくことで負荷分散を行うことができる. そのためニュースで話題に取り上げられる, SNS で話題になるというイベント発生によるサービスの急激なトラフィックの増加に動的に対応できる. また, ビジネス要求に対して柔軟なソフトウェアの改修が可能なため DevOps を採用する際には MSA を用いる場合がある [2]. 実際に MSA は Netflix や Amazon で採用されている [3], [4].

MSA における障害発生時のログ調査

マイクロサービスはそれぞれが単機能かつネットワークを経由して実行されている. そのため, ネットワークのダウン, マイクロサービスを実行しているノードのリソース不足が発生した際にサービスの停止や大幅な遅延が発生する (以後, 障害と呼ぶ). MSA においては, 障害が発生した際には 2 つのタスクが順に実行される. 1 つ目はサービスを利用可能にするための復旧を行うことである. 2 つ目はサービスが復旧し, 今後同じ原因で障害が発生しないようにするための原因調査を行うことである. しかし, MSA ではそれぞれのマイクロサービス にログの形式が異なるため, 障害に関して調査する際には Fluentd や Logstash, Kafka をログ収集モジュールとして用いて, それぞれのマイクロサービスのログを調査することになる [5][6][2]. Envoy はユーザーのリクエストに対してユニークな ID をつけることで, 特定のリクエストについて複数のマイクロサービスをトレース出来る仕組みがある. MSA では利点にあったスケールアウトとスケールインで動的に IP アドレスの割当が行われる. しかし, 動的な IP アドレスの割当をした際にはそれぞれのマイクロサービスの IP アドレスが過去に使われていたサービスの IP アドレスと重複することがある.

2. 課題

MSA の障害調査時の課題には以下がある.

- ログに記載されている IP アドレスが再利用されている場合に, どのタイミングでのコンテナのログか判別できない.

現状のマイクロサービスのコンテナログからは, どの IP アドレスから実行したかが判別できる. しかしその IP アドレスが今までにも利用されたことがあれば, 同一 IP アドレスが含まれているログは過去のコンテナのログか直近のコンテナのログか判別できない. 結果, ログを IP アドレスでフィルタリングしたときには必要としているコンテナと過去に同じ IP アドレスを用いていたコンテナのログも含まれることになる.

現状のマイクロサービスではサービスのリクエストの情報をそれぞれのサービスで保持している. そのため障害が発生した際のリクエストごとのステータスを抽出することに時間がかかっている [7]. 既存の MSA におけるログ監視システムではサービスごとのそれぞれの入力や出力, 処理内容が含まれるリクエスト情報を保存していた. 大規模なクラウドアプリケーションの場合, 多くは大量のリクエストを処理するためにマイクロサービスをスケール可能な形でデプロイすることになる [8]. その際に既存の環境では同一マイクロサービス内では IP アドレスの割り当て範囲に限りがあるため, IP アドレスが意図せず重複することがあった [9].

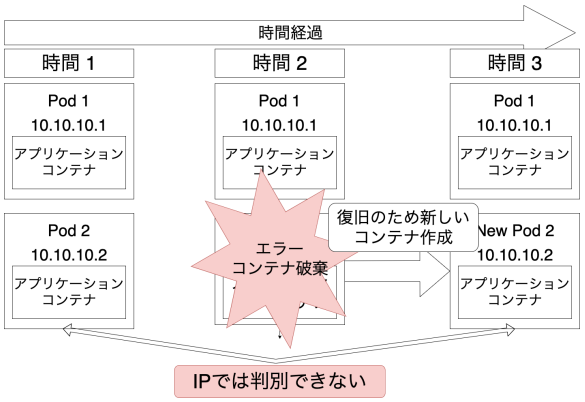


図 1 サービスのコンテナスケール時の IP アドレス割り当て

マイクロサービス内の Pod では, スケールアウトとスケールインを繰り返すため同一の IP アドレスでも同一の Pod が発行したとは限らない. 図 1 は Kubernetes システムを用いた時のスケールアウトとスケールイン時のサービスのコンテナへの IP アドレス割り当てを示した図である. 図 1 では 10.10.10.2 の IP アドレスが別のコンテナであるにもかかわらず 10.10.10.2 が割り振られているため, New Pod 2 が判別できない状態である.

アプリケーションを実行するコンテナが動的に増減する状況において障害が発生した時, ログを調査して障害の原因を探す際には過去に存在した同一 IP のサービスのコンテナを調査する. つまり直接原因とは関係ないサービスのコンテナを調査する. 結果, オペレーターが障害の原因を調査する時間を増加させる. また, MSA ではサービスとサービスは連携している. そのため, 該当サービスのコンテナを通過したリクエストを周辺マイクロサービスに及ぼした影響を調査することにかかる時間も増加する. クラウドサービスにおける原因調査 (Root Cause Analysis) は次の障害を発生させないためには欠かせないことである. ログの例としてクラウド上でプロキシ型のロガーとして機能する Envoy で作成されたログを以下のソースコード 1 に示す.

ソースコード 1 では 172.30.146.73 にて実行された http-bin のログである. リクエストの送信元は 172.21.13.94 であり, 172.30.146.82 がリクエストの送信先であることがわかる. 前述のとおり, 172.30.146.73 がスケールされたサービスのコンテナだと仮定した場合, これらの情報は

ソースコード 1 Envoy のログコード例

```
1 [2019-03-06T09:31:27.354Z] "GET␣/status/418␣HTTP
    /1.1" 418 - "-" 0 135 11 10 "-" "curl/7.60.0
    "␣"d209e46f-9ed5-9b61-bbdd-43e22662702a"␣"
    httpbin:7000"␣"172.30.146.73:70"␣outbound
    |7000||httpbin.default.svc.cluster.local␣-
    ␣172.21.13.94:7000␣172.30.146.82:60290-
```

172.21.13.94 に紐図いており, 今後別のコンテナに同じ IP アドレスが割り振られた場合はログメッセージだけではどのタイミングでの 172.21.13.94 であるかを明確にすることができない.

3. 関連研究

NetFilx は Simian Army を使用して MSA に障害を注入して構成全体の復元力をテストしている[*1].

Envoy を用いた Meina Song らの研究では, 障害検知の際に実装部分である Application Layer と Envoy がデプロイされる Proxy Layer, 新たに Zipkin や Jeager を用いた Data Analyzer layer を作成し, Envoy から得られるデータを用いて障害検知を行う仕組みが提案されている [10]. この方法では Istio とは別に Data Analyzer layer を用いる. しかし, 管理用のレイヤーを設置することに別の管理用の環境を用意する必要がある.

サービスメッシュ上の監視及び制御トラフィックについての研究では, MSA におけるサービス間での相互監視においてデータにおいて監視及び制御トラフィックについてはアプリケーションテナントとは分離してデプロイすることは困難であり, これによりセキュリティ上の問題が発生する可能性があるとしている [11]. これに対して監視および制御用のデータに関しては Envoy 側に同期させるが, アプリケーションテナントは経路に入らないデザインを提案している.

この提案はクラウドサービス上のマイクロサービスの監視・制御にかかる技術者の時間を短縮し, 新たなクラウドサービスを設計する際に, 実装を抽象化してパッケージ化することで各サービスの状態を共有する. 適切なリソースをクラウドに格納できるシンプルな仕組みを提供している. 実際のシナリオでは, デプロイは運用コストに制約される [12]. クラウドリソースをリアルタイムで管理することで, その制約の中でサービスが展開され, リソースに負荷がかかってしまった場合のダウンタイムやプリリクエストの失敗を軽減することができる. ハイブリッドクラウドやマルチクラウド環境では, コスト削減策は各ベンダーに依存する [13]. クラウド資源の節約は, 信頼性が最も重視されるサービス以外のサービス（バックアップや災害対応サイト）では, 経常的なコストを削減できるので有用である. そのため, ベンダーに依存しない形でリソースを削減することが望ましい. また, マイクロサービスに障害が発生した場合, サービスの平等性を確保するために, リクエストを再試行し, リクエストレベルでの各マイクロサービスの状態をネットワークと共有しなければならない. 現在のマイクロサービス間のデータ共有方法では, 特定のノード間でのみ通信を行っているが, よりシステムの透明性を高めるためには, すべてのノードがサービスの状態にアクセスできるようにする必要がある.

サービスメッシュ上の監視・制御トラフィックに関する研究では, MSA におけるサービス間の相互監視において, アプリケーションのテナントとは別のデータに監視・制御トラフィックを展開することが困難であることが示されており, セキュリティ上の問題が発生する可能性があることが示されている. これに対して, Envoy 側で監視制御データを同期させながら, アプリケーションテナントがパスに入らない設計を提案している [11].

しかし, 構成要素がサービスから独立している各サービスの Envoy から Istio のコントロールプレーンにこれらを報告するモデルをサービスメッシュに統合することで, 外部との通信を追加実装する必要がない.

Envoy のプロキシによるロギング手法は既存の Kubernetes 環境の追加のコンフィグが最小限でパフォーマンスを損なわないため, 本研究においても同じサイドカーモデルを用いてクラウド上にデプロイすることとする.

4. 提案

同じ IP アドレスのログを判別するために各コンテナユニークの ID(以降 CUID) を作成する. コンテナで実行されたログに付与するプログラムが搭載されたコンテナ (以降 IKontainer) を作成する. IKontainer と CUID によってマイクロサービスにおける単体のコンテナをログから判別することで可能になる. CUID はそれぞれの IKontainer で作成し, 集中管理はされない. CUID には衝突の確率が低いアルゴリズムで作成した一意の識別子を付与する.

この方式では CUID を用いることでログの検索時に Pod 単位で絞りこむことができる. Pod 単位でログを絞りこめることのメリットについては後述のユースケースシナリオで説明する.

Stackrox 社の調査によると Kubernetes 以外のマイクロサービス構築アーキテクチャ管理ソフトにおいてシェアを得ているものは, Amazon 社の Amazon EKS, Microsoft 社の Azure AKS, RedHat 社の Openshift がある. それらにおいても Pod の概念は Kubernetes の物を同様に扱っているため, AKS, Openshift においても提案手法のアプローチを用いることで同様のメリットを享受できる [14][15][16][17]. そのため, 本論文ではもっとも基本的で

[*1] SimianArmy: https://github.com/Netflix/SimianArmy

シェアが高い CNCF[*2]が OSS(Open Source Software) として開発, 公開している Kubernetes[*3]システムを利用した場合の提案方式について記述する.

4.1 IKontainer:アプリケーションコンテナとは別のマネジメント用コンテナ

アプリケーションコンテナとはマイクロサービスにおいて主目的を実現するために実装されるコンテナを指す. 例として Web サイトを構築するときのアプリケーションコンテナは http リクエストが来たら http レスポンスを返すという機能が実装されているプログラムが構築されたコンテナである. Web サイトの実際のユーザーがアクセスする部分以外 (以下バックエンドサービスと呼ぶ) の例としては, 入力値に応じて計算するコンテナや, データベースからリクエストされたクエリへのレスポンスを返すコンテナが想定される.

今回本論文の提案手法は上記のアプリケーションコンテナとは別のマネジメント用コンテナ (以下 IKontainer) に実装する.

4.2 コンテナのユニーク ID:CUID の生成

MSA ではコンテナの管理の一環としてコンテナが動作不能になった際に自動的に同機能のコンテナを立て直すオートヒーリング, 単体のコンテナでは処理しきれない量のリクエストが来た際に同機能のコンテナを追加で複数起動することで正常に処理を実行させるオートスケーリングがメリットとして挙げられる [18][19]. オートヒーリングやオートスケーリングで作成されるコンテナは同機能ではあるが, 別のコンテナが作成され, 実行されている. そのため, オートヒーリング前にエラーを発生して異常な動作を起こしたコンテナのログを調べるためにはオートヒーリング後のコンテナのログから調査を始める. しかしオートヒーリングやスケーリングからすでに時間が経過している際には大量のログからコンテナの再起動された点を探すことが必要となる. 本論文のアプローチはコンテナが起動した際には機能, 状態に関わらず一意の識別子である Container Unique IDentifer(以後 CUID) を作成する. CUID は同一の IP アドレスが振られる環境で重複しないものを作成する. 同一の IP アドレスで同一の CUID を作成された場合は識別が不可能になるためである. 本提案の実装ではユニークな ID を作成するため universally unique identifiers (UUIDs) を用いる. UUID は ITU-T X.667 および ISO/IEC 9834-8 でユニークな ID 生成方法として標準化されている [20].

また IKontainer はそれぞれのコンテナが独自に CUID を作成する. これは CUID を管理するサーバーを設置したとき, 単一障害点が発生することを防止するためである.

CUID は各コンテナにおいてそのコンテナが終了するまで保持する. また CUID は常に規定のフォーマットでログ末尾に追記する. ここではコンテナに紐づいている CUID を”bccf1e60-994d-44c4-8b78-d33535f31af4”としたときのログを以下のソースコード 2 に示す.

ソースコード 2 IKontainer が CUID を追記したログ

```
1 {"ikontainer_CUID":"bccf1e60-994d-44c4-8b78-
      d33535f31af4", "svc":"svc.namespace.
      cluster"}
```

4.3 コンテナのログへのユニーク ID の割り当て

CUID はコンテナが起動している間においてすべてのアプリケーションログとコンテナログの書き込みが行われたらログの末尾に CUID を付与する. 一般的なコンテナを想定した際の IKontainer の動作を以下の図 2 に示す.

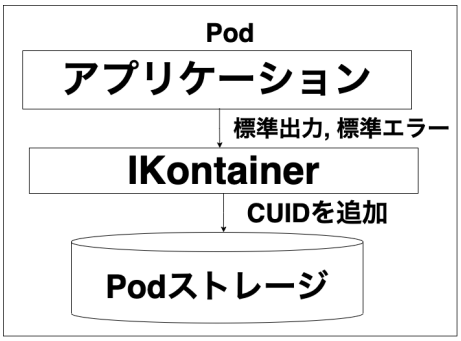


図 2 IKontainer の設計

ここでは Pod を作成した際にアプリケーションコンテナが標準出力と標準エラーを保存するボリュームにアクセス, 監視をすることで新しいログが発生した際にすべてのログの書き換えを IKontainer が行う. それにより Pod 内で出力されるログはすべて前述の CUID が記載されたログを持つ.

ソースコード 3 ではログに CUID が付与されているため. 同一のサービスのコンテナでのログを調査する際には CUID でフィルタリングすることでログの調査ステップが削減できる.

ソースコード 3 IKontainer が CUID を追記したログ

```
1 {"type":"log","@timestamp":"2021-01-17T12
      :52:39Z","tags":["status","plugin:spaces@7
      .4.1","info"],"pid":7,"state":"green","
      message":"Status␣changed␣from␣red␣to␣green
      ␣-␣Ready","prevState":"red","prevMsg":"[
      data]␣Elasticsearch␣cluster␣did␣not␣
      respond
2 with␣license␣information."}{"ikontainer_CUID":
      "bccf1e60-994d-44c4-8b78-d33535f31af4", "
      svc":"svc.namespace.cluster"}
```

*2 CNCF: https://www.cncf.io/
*3 Kubernetes: https://kubernetes.io/

4.4　ユースケースシナリオ

前提となるシステム

以下に簡単な Web サイトにおけるマイクロサービスの障害において, IKontainer が CUID を持つことで調査ステップが減少する事例を示す. 前提として, 一つの Pod において 1 秒間に処理できるリクエスト数を 500 回とする. これはシステム設計者が事前に負荷テストを行うことで取得した値とする. Web サイトに対して急激な処理増加に対応するため, 同機能の Pod をオートスケール対応のレプリカとして配置し, ロードバランサがリクエストを割り振るとする. このユースケースではクラウドの運用コストの面から Pod を 3 つ以上のスケールアウトはできないものとする. すなわちこの 2 つの Pod で処理できる最大のリクエストの理論値は 1000 回である. この 2 つのポッドが多量のリクエストを処理できずに応答なし状態, その後コンテナがハングアップしたとする. MSA ではこのコンテナ分の処理を継続するために 2 つ目のコンテナを一度削除し, 作成し直す (オートヒーリング). ここで作成されるコンテナは同一機能であり, 同一のグループ, Kubernetes においては同じデプロイメントに属することになる. そのため, IP アドレスの重複が発生し得る. 以下の図 3 に示す. 図 3 では, 実装で用いている CUID は 32 桁であるが, 図の表記上 CUID を 5 桁とする.

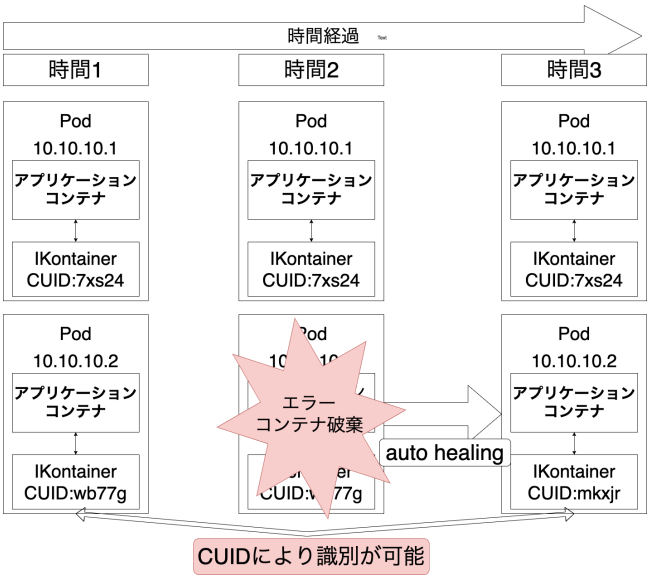


図 3　CUID を用いることによる同一 IP の Pod 識別例

障害に関する情報を取得するためにログを検索する際には計 3Pod のログを確認することになる. その際, 障害が発生し Warn や Fatal Error のログから検索をかけた際にはコンテナの障害原因を探るためにそこから異常ログが発見されたコンテナのログだけを閲覧することでいつどのように障害が発生したかが把握できる. IKontainer の CUID で絞りこむことで実行時のコンテナのみのログが閲覧できるようになるため, ログの検索利便性が向上する.

また, MSA では複数のサービスがそれぞれ別の API を経由しているため, Envoy の一部の拡張機能ではマイクロサービス内でのリクエストについてリクエストユニーク ID である X-requests-id(以下リクエスト ID) を付与している. リクエスト ID と IKontainer の CUID を両方用いてログの検索結果をフィルタリングすることで, 障害の直前にまったく同じコンテナを通過したリクエストについてその後の処理についてもトレース可能になる.

検索ステップ数の比較

障害が発生した際には, どのコンテナが原因となっているのか, 他のサービスに影響を与えているのかを調査する必要がある. このケースでは, 図 4 の最下ログの Payment サービスからアラートが発生したとする.

現在の解決策は, IP アドレスからチェックして, サービスが通過したリクエストをチェックしている.

name	ip	request id
Gateway	10.10.1.2	46BB
Gateway	10.10.1.2	966D
Payment	10.10.11.2	46BB
Mail	10.10.12.1	46BB
Payment	10.10.11.1	966D
Store	10.10.13.1	46BB
DB	10.10.100.1	46BB
Gateway	10.10.1.2	7AD0
Gateway	10.10.1.2	F02F
Payment	10.10.11.2	7AD0

図 4　IP アドレスベースでのログ調査ステップ

Name	ip	cuid	request id
Gateway	10.10.1.2	643B	46BB
Gateway	10.10.1.2	643B	966D
Payment	10.10.11.2	2F52	46BB
Mail	10.10.12.1	82B8	46BB
Payment	10.10.11.1	7D53	966D
Store	10.10.13.1	CC62	46BB
DB	10.10.100.1	17E1	46BB
Gateway	10.10.1.2	CD85	7AD0
Gateway	10.10.1.2	CD85	F02F
Payment	10.10.11.2	4A23	7AD0

図 5　CUID ベースでのログ調査ステップ

10.10.11.2 のシステムでは, その IP アドレスから 2 つのログを見つけることができる. 2 つのログのリクエスト ID を確認すると, 46BB と 7AD0 がある. そして, それぞれのリクエスト ID を確認することで, どのサービスがこのリクエストに影響を与えたのかを明確にすることができる. 7 つの log のメッセージは存在するが, リクエスト 46BB はすべて正常に処理されている. また, そのコンテナはスケー

ル前のものとは別のコンテナを使用しているため, これらの 5 つのログは必要ない. そのため, これら 5 つのログは不要な調査である.

CUID を用いたアラート一覧を図 5 に示す. CUID を含めたアラートが出ている時は, そのサービスがどのサービスに影響を与えたのかを明確にする必要がある. どのコンテナが使用されたかを追跡することができるが, この場合, そのコンテナのログは 1 つのみである.

また, 各リクエストの ID を確認すると, 2 つのログがあり, 両方ともそのコンテナに関連したリクエストである. これは調査が効率的であることを意味する.

5. 実装

提案手法であるマイクロサービスにおけるログマネジメントコンテナ:IKontainer の設計と実装を示す.

5.1 全体構成

アプリケーションコンテナと IKontainer を別にすること手法も用いる実装を言語に処理を実装するライブラリと比較したものを以下の表 1 に示す.

要件	コンテナ	ライブラリ
メンテナンスの容易さ	○	△
既存環境への導入難易度	○	×
複数の言語/コンテナへの対応	○	×
処理速度	△	○

表 1 マネジメントコンテナ:コンテナ方式とライブラリ方式の比較

表 1 で示すメンテナンスの容易さとは, アプリケーションコンテナおよび IKontainer そのものアップデートの際に既存のコードに対する変更の少なさのことである. 前述のとおり, IKontainer ではアプリケーションコンテナとは別のコンテナで動作する. すなわちアプリケーションコンテナと IKontainer は異なるコードベースで実行されている. IKontainer を用いる際には Docker hub にて公開しているコンテナイメージを用いることで使用するユーザーは手元の IKontainer をバージョンアップごとに最適なパッケージのダウンロード, IKontainer 内で用いるコードや環境の依存関係の把握をせずとも継続的にメンテナンスが可能になる. 具体的にはイメージ IKontainer 自身のアップデートは IKontainer コンテナイメージが docker hub で公開している状況においては, デプロイするサービスや Pod 単体のマニフェストファイルにおけるイメージ名を変えるのみでアップデートを適用することができる. 対してライブラリとしてアプリケーションコンテナ内にコードを実装した場合は, アプリケーションコンテナの変更時にはライブラリの重複する読み込みによる処理時間の増加が発生する.

既存環境への導入難易度は, アプリケーションコンテナの実装がすでに完了している環境に対して, 追加でロギングシステムを追加するときに IKontainer ならば既存のアプリケーションコンテナのログを変更させることなく実装できる. 一方ライブラリ方式では, アプリケーションコンテナ内のログシステムに介入する必要があり, 多数存在するログの種類すべてに CUID を付与する変更が必要であるため実装時間が IKontainer に比べ長くなる.

MSA ではその特性上, それぞれのマイクロサービスでは異なるプログラミング言語やフレームワークから構築されていることが一般的である [4]. そのため, 言語に強く依存するライブラリ方式では各アプリケーションコンテナごとにログの実装をする必要がある. 一方, IKontainer ではログの保存時に CUID を付与するアプローチであるため, どのようなアプリケーションであっても導入に際しては手順が変わらない. 複数のプログラミング言語が存在するサービスにおいてもアプリケーションコンテナには依存しないことから導入への障壁を軽減する.

処理速度においては, ライブラリ方式が言語や実行環境に適したコードを記述した場合に外部コンテナとの通信を挟まない点から IKontainer よりも高速であるといえる. しかしアプリケーションコンテナ内の呼び出し回数が複数回にわたる場合にはライブラリが何度も呼び出しされることで処理能力の低下, リソースの圧迫がある.

以上の点より IKontainer はライブラリ方式での設計はせず, 独立したマネジメント用コンテナとして設計する.

IKontainer は単体で実行可能な形態にするため, Docker イメージにパッケージした. これは Docker イメージにすることでどの Kubernetes においても環境に適応するための追加設定がパッケージマネージャー, およびソースの直接適用をするよりも減少するためである [21]. また, 既存のマイクロサービス環境に導入する際には, 既存のアプリケーションコンテナのマニフェストファイルを変更することなくローリングアップデートできることもメリットである. IKontainer のベースイメージは Alpine Linux[*4]を用いた. コンテナを各 Pod にデプロイする際には Docker イメージのサイズが大きいことでマシンの性能およびストレージキャパシティを使用する実行するコマンドの要求以上に用いることで, スペックのオーバーなプロビジョニングの要因となる. 監視やログ生成の IKontainer に必要な機能ではないリソース量のオーバーなプロビジョニングを行った際には既存環境の性能の低下, ストレージ容量の必要以上の消費, また前項による維持コストの増加, アップデート時間の増加によるダウンタイムが増加する要因となる [22], [23]. 今回使用する Alpine Linux は musl libc と busybox を組み合わせた軽量かつ以降説明する IKontainer で利用するコマンドがパッケージマネージャーにて管理可能な LinuxOS であることから選定した. また, コンテナ内の Volume は再起動時に消去されるため, Log Agent を用

[*4] Alpine Linux: https://AlpineLinux.org/

いて Pod 外の永続ボリュームおよびログ検索エンジンに転送する．IKontainer の全体構成図を以下の図 6 に示す．

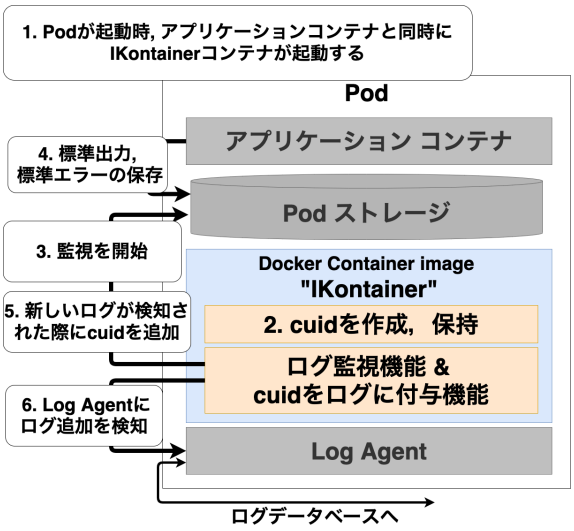


図 6　全体構成図

(1) Pod が作成された際にはアプリケーションコンテナと同時に, IKontainer と Log Agent が同一 Pod に作成される．同時に IKontainer の初期動作として CUID を発行する．

(2) IKontainer が Pod の共通 Volume(以後 Pod ストレージ) にアクセス, 指定ディレクトリ内の監視を開始する．

(3) アプリケーションコンテナから標準出力と標準エラーが Pod ストレージに書き込まれる．

(4) IKontainer の監視によりステップ 3 のログファイルの変更が検知される．検知されたファイルの各行の末尾に 4.2 章の方法にて作成した CUID 文字列を追記する．

(5) CUID を追加したのちに, Log Agent にイベントを通知し, ログを Pod 外の永続ボリュームおよびログ検索エンジンに転送させる．

本設計では Pod 内の通信は Volume を介して行われるため, ネットワークの帯域幅を圧迫することがない．

IKontainer が搭載する機能のシーケンス図を図 7 に示す．

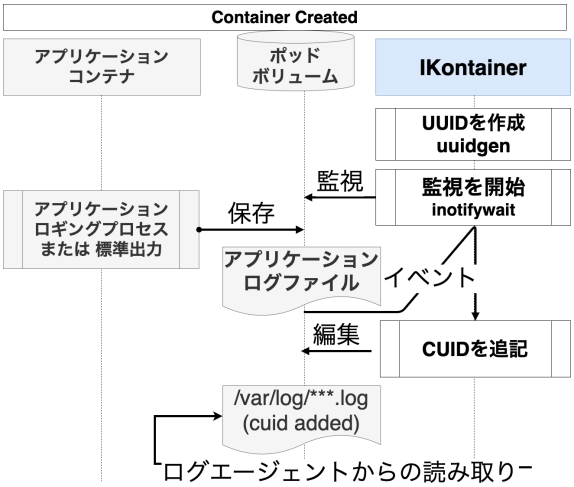


図 7　各コンテナのタイムライン

アプリケーションコンテナの出力を Log Agent が Pod 外部のストレージに保存する前に CUID を追加する処理を入れることで, アプリケーションコンテナ側, および外部のログ検索エンジンの変更をせずに導入が可能である．

5.2　実装

IKontainer 前述の提案, 設計を実現するために以下の実装を行った．初めに実装構成図を以下の図 8 に示す．下から順に Docker イメージを構築している要素である．

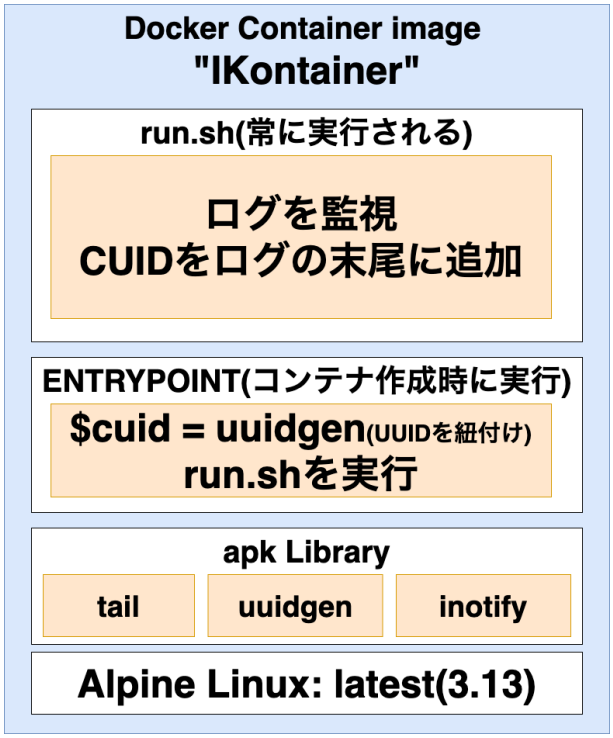


図 8　実装構成図

イメージのベース OS は Alpine Linux3.13 である．そこに CUID 用のユニークな ID を作成するため Universally Unique IDentifiers (UUIDs) を用いる．UUID は ITU-T X.667 および ISO/IEC 9834-8 でユニークな ID 生成方法として標準化されている [20]．Alpine Linux 上で使用するため apk パッケージマネージャーから uuidgen をインストールする．IKontainer がログを書き換える際には Pod Volume を経由して実行される．ファイルの変更検知方法においては inotify-tools [*5] を用いる．ディレクトリの中を再帰的に監視する．inotify-tools 内の監視プログラム,”inotifywait”においては複数のイベントタイプがあるが, 今回はファイルが変更されたことを示す”MODIFY”が検知された際に IKontainer の CUID 追記プログラムが実行される．uuidgen 同様 apk パッケージマネージャーから inotify-tools をインストールする．ログファイルの直近行を読み込むため, tail コマンドを使用する．Docker コンテナが立ちあがった際の動作を ENTRYPOINT に示している．

[*5] inotify-tools: https://github.com/rvoicilas/inotify-tools

uuidgen を用いた CUID の作成, 変数への割り当てを行い, コンテナが立ち上がっている間継続して動作する run.sh を呼び出している. run.sh では inotifywait コマンドによる Pod Volume の監視する機能, アプリケーションコンテナによってログが追記されたときに CUID を追加する機能および CUID を追記したのちに Log Agent に転送する機能の 3 つがある. Pod Volume の監視は inotifywait コマンドの”-e modify”オプションをつけることでログの追記を検出できる. また, ”-r”オプションを用いて指定ディレクトリ以下のすべてのファイルおよびディレクトリのトラッキングに対応した. そのため, Pod 内で発生するあらゆるログについて検出および CUID の追記が可能となった. 最後に Log Agent に対するイベントの発行機能がある. これは CUID を追記し終えた際に Log Agent に対し, 前回の変更からのアップデートを要求する. 今回実装に用いた Log Agent は fluentd とした. これは実験において Kubernetes 環境にて IKontainer をデプロイするのに先立ち, Kubernetes.io のロギングアーキテクチャ[*6]に用いるログエージェントとして fluentd が用いられているためである.

5.3 実験構成

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 Envoy および IKontainer を導入する. IKontainer のオーバヘッドが Envoy 単体をインストールしたときと比較することで IKontainer が実用に耐えることを実証する. そのバックエンド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.

サービス名	役割
Web サーバー	ユーザは 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
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 2 実装構成表

以上の構成を元に作成した実験構成図を以下の図 9 に示す.

5.4 各サービスの実装詳細

5.5 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 4 に示す

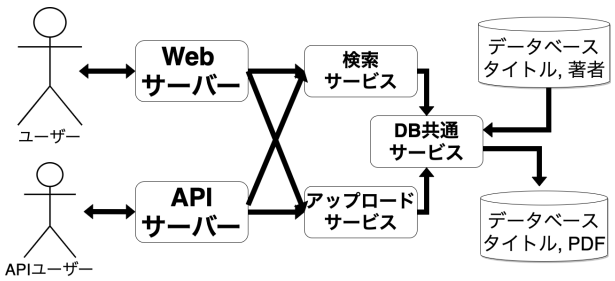


図 9 実験概要図

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

表 3 ESXi サーバーのスペック

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

5.6 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.

実験においては負荷テストツールである locust から WEB サーバーに多数のリクエストを同時に行うことで, CPU 使用率の変化を確認する. 今回の実験では最大同時接続数を 100 および 500 に増加させる. 各アプリケーションソフトは python コンテナより, ENTRYPOINT として Flask を起動している. それぞれのアプリケーションコンテナはログのデバッグレベルを”info”に設定した. ”info”に設定することでログの件数を増やすだけではなくアプリケーションコンテナ負荷や Kubernetes サービスによってロードバランシングした結果の Pod へのリクエストの割り振り結果も取得する. Elasticsearch や Kibana はクラウドアプリケーションを構築する際は導入することでログ管理が容易になるため本実験, 評価でも導入する [24]. Log database/検索エンジンとして Elasticsearch, 取得したログの統計処理, 視覚化, 検索ステップを測るために Kibana をそれぞれ

[*6] Kubernetes Logging Architecture: https://kubernetes.io/docs/concepts/cluster-administration/logging/

Service でデプロイする．これによりユーザー側からいつでも実験環境の内部ログを統合して取得できる．IKontainer と Log Agent である fluentd は Kubernetes DaemonSet にてデプロイした．この 2 コンテナについては各アプリケーションコンテナのマニフェストに IKontainer と fluentd を記載せずとも新規で立ち上がったポッドに対して当該 Pod の PodVolume をマウントした状態で起動されることを確認した．

5.7　検証方法

locust からマイクロサービスの Gateway エンドポイントに対してリクエストを送信し，リクエストに対するレスポンスが返答されるまでの時間を計測する．

- Istio がインストールされた Kubernetes 環境
- Istio + IKontainer を追加した Kubernetes 環境

実験は以下の手順で行う．

- locust から WEB サーバーに向けて HttpRequest を一定のレートでリクエストする．
- locust に最終ステータスコードをレスポンスとして返されるため，時間を計測する．

6.　評価

6.1　レスポンスタイムの比較

1 秒間 100 及び 500 リクエストから 60 秒間，トップページに対して行うアクセスした際におこる．オーバーヘッドを計測するため IKontainer と Istio，IKontainer を用いた環境での比較を行う．実験では 1 分から 60 分まで計測したが，有意な差は見られなかったため，グラフでの横軸はリクエストの秒数ごとで表示できる 60 秒としている．それぞれ 5 回測定した結果の中央値を用いている．以下は秒間 100 ユーザーがシステムにアクセスした際のレスポンスタイムの計測結果である．

各秒数のレスポンスタイムの結果を以下の図 10 に示す．

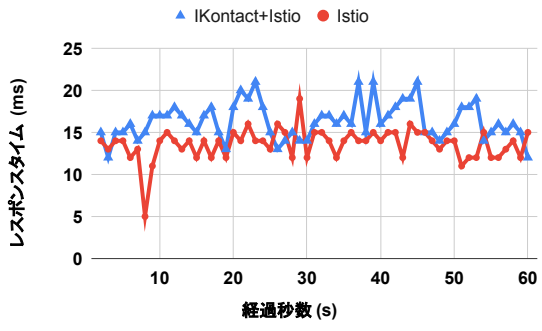


図 10　100 リクエストを 1 分間継続した際のレスポンスタイム

秒間 100 回のリクエストでは Ikontainer を入れた環境においてもレスポンスタイムは 20ms を超えることは少ない．

IKontainer を導入した際にもシステムが急激に不安定になることはなかった．このことから IKontainer を入れることによって，既存環境に IKontainer を追加しても WEB サーバーのレスポンスへの影響が少ないといえる．ここでは各環境における最頻値を以下の図 11 に示す．

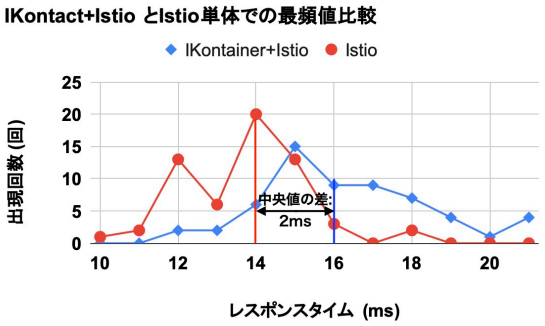


図 11　10ms - 20ms での出現頻度

Envoy でのレスポンスの中央値は 14ms であり，IKontainer を追加した際には中央値は 16ms となった．

秒間に 500 回リクエストした際のレスポンスタイムを以下の図 12 に示す．IKontainer+Istio でのレスポンスタイムの中央値は 955.3ms，Istio では 1082.6ms となった．1 秒毎に 500 リクエストしたときは IKontainer のオーバヘッドは 13.3%となった．

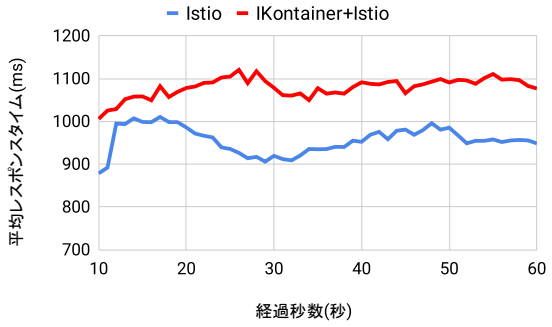


図 12　500 リクエストで実験を行った際の結果

6.2　分析

上記のレスポンスタイムの遅延の要因としては IKontainer がノードに割り当てられているメモリをアプリケーション本体部分から使っていることから，スケールする際のノード全体のメモリの枯渇が見られた．すなわちアプリケーション本体に割り当てられるメモリの量を減少させている．これは IKontainer がログをストリームさせる際にメモリを消費していたからである．

7.　議論

ログに追加で付与するパラメータについて，IKontainer が作成した CUID とクラスターのドメイン名のみを追加した．同様のアプローチとして Kubernetes の環境上で uid を作成する際には pod のオブジェクトに紐づいている uid を用いて取得する方法がある．しかしオブジェクトの値を

取得するためにはコンテナ内部から kubectl にアクセスする必要がある. 新たに立てたコンテナで UUID 作成する手法との処理速度の比較が必要である.

また, Kubernetes の Pod のログ保存に関わるところでオブジェクト ID の付与ができる場合, Kubernetes 本体のロギングの本論文と前段に述べた手法よりも高速化が可能である. Kubernetes にビルドインされることになるため, Daemonset やサイドカーを用いたスペックの必要以上のプロビジョニングを防止できる. すなわち冒頭に述べたリソース不足によるコンテナの動作不良がなくなる.

今回作成した IKontainer ではリクエストのオーバーヘッドを 13.3%発生した. Envoy をデプロイした上に Ikontainer を重ねてデプロイしているため, 現状ではコンテナのランタイムを別のイメージとして動作させているためである.

8. 結論

本研究ではサービスの中が動的にスケールするマイクロサービスにおけるログの共有の収集に用いる手法について提案した. CUID をコンテナにつけることでコンテナに割り振られた IP アドレスに関係なく正確に同一コンテナのログの識別が可能になる. IKontainer では上記の識別を 13.3%のレスポンスタイムのオーバヘッドで実現した. これにより, マイクロサービスの欠点であったサービス全体でのデータのログをスケールレベルで管理することが可能になる. そのため, マイクロサービスにおけるログデータのトレーサビリティが増加する.

本研究及び提案については MSA ー (MSA) の導入の障壁となる, 障害が発生した際のオペーレーション削減及び MSA を用いた大規模サービスにおける通常時のログのトラッキング, 分析について貢献する.

参考文献

[1] MacKenzie, C. M., Laskey, K., McCabe, F., Brown, P. F., Metz, R. and Hamilton, B. A.: Reference model for service oriented architecture 1.0, *OASIS standard*, Vol. 12, No. S 18 (2006).

[2] Balalaie, A., Heydarnoori, A. and Jamshidi, P.: Microservices architecture enables devops: Migration to a cloud-native architecture, *Ieee Software*, Vol. 33, No. 3, pp. 42–52 (2016).

[3] Thönes, J.: Microservices, *IEEE Software*, Vol. 32, No. 1, pp. 116–116 (2015).

[4] Jamshidi, P., Pahl, C., Mendonça, N. C., Lewis, J. and Tilkov, S.: Microservices: The Journey So Far and Challenges Ahead, *IEEE Software*, Vol. 35, No. 3, pp. 24–35 (2018).

[5] Dunning, T. and Friedman, E.: *Streaming architecture: new designs using Apache Kafka and MapR streams*, " O'Reilly Media, Inc." (2016).

[6] Indrasiri, K. and Siriwardena, P.: Microservices for the enterprise, *Apress, Berkeley* (2018).

[7] Manouvrier, M., Pautasso, C. and Rukoz, M.: Microservice Disaster Crash Recovery: A Weak Global Referential Integrity Management, *Computational Science – ICCS 2020* (Krzhizhanovskaya, V. V., Závodszky, G., Lees, M. H., Dongarra, J. J., Sloot, P. M. A., Brissos, S. and Teixeira, J., eds.), Cham, Springer International Publishing, pp. 482–495 (2020).

[8] Hasselbring, W. and Steinacker, G.: Microservice Architectures for Scalability, Agility and Reliability in E-Commerce, *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pp. 243–246 (2017).

[9] Burns, B., Grant, B., Oppenheimer, D., Brewer, E. and Wilkes, J.: Borg, omega, and kubernetes, *Queue*, Vol. 14, No. 1, pp. 70–93 (2016).

[10] Song, M., Liu, Q. and E, H.: A Mirco-Service Tracing System Based on Istio and Kubernetes, *2019 IEEE 10th International Conference on Software Engineering and Service Science (ICSESS)*, pp. 613–616 (2019).

[11] Kang, M., Shin, J. and Kim, J.: Protected Coordination of Service Mesh for Container-Based 3-Tier Service Traffic, *2019 International Conference on Information Networking (ICOIN)*, pp. 427–429 (2019).

[12] Villamizar, M., Garces, O., Ochoa, L., Castro, H., Salamanca, L., Verano, M., Casallas, R., Gil, S., Valencia, C., Zambrano, A. et al.: Infrastructure cost comparison of running web applications in the cloud using AWS lambda and monolithic and microservice architectures, *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, IEEE, pp. 179–182 (2016).

[13] Van den Bossche, R., Vanmechelen, K. and Broeckhove, J.: Cost-Optimal Scheduling in Hybrid IaaS Clouds for Deadline Constrained Workloads, *2010 IEEE 3rd International Conference on Cloud Computing*, pp. 228–235 (2010).

[14] StackRox: kubernetes-adoption-security-and-market-share-for-containers (2021). https://www.stackrox.com/kubernetes-adoption-security-and-market-share-for-containers/.

[15] AWS: Getting started with Amazon EKS – AWS Management Console and AWS CLI (2021). https://docs.aws.amazon.com/eks/latest/userguide/getting-started-console.html.

[16] Azure, M.: Kubernetes core concepts for Azure Kubernetes Service (AKS) (2021). https://docs.microsoft.com/en-us/azure/aks/concepts-clusters-workloads.

[17] Openshift, R. H.: Pods and Services - Core Concepts (2020). https://docs.openshift.com/enterprise/3.0/architecture/core_concepts/pods_and_services.html.

[18] Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R. and Gil, S.: Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud, *2015 10th Computing Colombian Conference (10CCC)*, pp. 583–590 (2015).

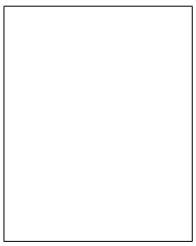
[19] Toffetti, G., Brunner, S., Blöchlinger, M., Dudouet, F. and Edmonds, A.: An architecture for self-managing microservices, *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud*, pp. 19–24 (2015).

[20] Leach, P., Mealling, M. and Salz, R.: A universally unique identifier (uuid) urn namespace (2005).

[21] He, S., Guo, L., Guo, Y., Wu, C., Ghanem, M. and Han, R.: Elastic application container: A lightweight approach for cloud resource provisioning, *2012 IEEE 26th International Conference on Advanced Informa-*

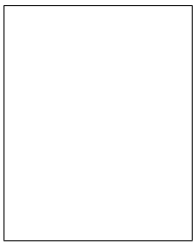
tion Networking and Applications, IEEE, pp. 15–22 (2012).
[22] Chang, C.-C., Yang, S.-R., Yeh, E.-H., Lin, P. and Jeng, J.-Y.: A kubernetes-based monitoring platform for dynamic cloud resource provisioning, *GLOBECOM 2017-2017 IEEE Global Communications Conference*, IEEE, pp. 1–6 (2017).
[23] von Kistowski, J., Eismann, S., Schmitt, N., Bauer, A., Grohmann, J. and Kounev, S.: Teastore: A microservice reference application for benchmarking, modeling and resource management research, *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, IEEE, pp. 223–236 (2018).
[24] Bagnasco, S., Berzano, D., Guarise, A., Lusso, S., Masera, M. and Vallero, S.: Monitoring of IaaS and scientific applications on the Cloud using the Elasticsearch ecosystem, *Journal of physics: Conference series*, Vol. 608, No. 1, IOP Publishing, p. 012016 (2015).

著者



飯島 貴政 （学生会員）

東京工科大学バイオ・情報メディア研究科修士課程在学. 研究対象は SOA, マイクロサービスおよびサービスメッシュにおけるデバッグ, トレーサビリティ.



串田 高幸 （正会員）

2003 年岩手県立大学ソフトウェア情報学研究科後期博士課程修了. 日本アイ・ビー・エム株式会社東京基礎研究所にてクラウド, 分散システムとネットワークの研究に従事. 2019 年より東京工科大学コンピュータサイエンス学部教授. IEEE, ACM 各会員. 2018-2020 年本会監事. 本会フェロー.

謝辞

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