コンテナ環境における名前空間の設計と分散型名前解決機構の提案

片岡 拓海[1] 篠田 陽一[2]

概要：
クラウドコンピューティングや分散システム環境において、コンテナ型仮想化技術の利用が増加している。多くのコンテナ型仮想化技術ではコンテナ環境内部に専用のネットワークを構築し、その名前空間は既存の名前空間と互換性がない。一方、コンテナ内部で稼働するサービスへのグローバルアクセスの要求は多い。そのため、コンテナ外部からコンテナ内部で稼働するサービスを把握するために透過的な名前解決が求められている。本研究ではコンテナ環境内部で稼働するサービスに対して外部から柔軟にアクセスできるようになるためのサービスディスカバリシステムを提供することを目的とする。本稿では、まず、コンテナ環境におけるネットワーク技術やサービスディスカバリ機構に関連する技術について調査した。次に、本研究で最適な設計手法の検討を行い、DDNS を用いたサービスディスカバリシステムの設計と実装を行った。そして、SOCKS プロキシを用いた実証実験の結果、透過的なサービス名解決が可能になることを確認した。今後の検討項目として DNS リゾルバ API の拡張実装や大規模なサービス展開時の挙動を調査する必要があることが明らかになった。

Design of Namespaces in Container Environments and Proposal of a Distributed Name Resolution Mechanism

Takumi KATAOKA[1] Yoichi SHINODA[2]

1. 背景

クラウドコンピューティングや分散システム環境において、仮想化技術の一つであるコンテナ型仮想化技術の利用が増加している。コンテナ型仮想化技術とは単一の OS 上で複数の OS が動作しているように見せる仮想化技術である [1]。

コンテナは従来の VM 型仮想化と比較してさまざまなメリットがある。コンテナは、起動・終了をはじめとする動作が軽量である。また、他の環境への環境の統一が容易に可能である。そして、システム全体のオーバーヘッドが減らすことが可能となり計算資源の集約率の向上につながる。

コンテナの本番環境での利用例としてヤフー株式会社 [2] では、2020 年 6 月時点で約 13 万個、その 6 ヶ月後の 2020 年 12 月時点では 20 万個以上のコンテナが稼働している。6 ヶ月という短期間にコンテナ数が約 1.6 倍にも増加している。このようなことから、今後もコンテナの利用増加、コンテナ数の増加が考えられる。

コンテナ数が増加し、その中でさまざまなファンクション・コンポーネントが提供される中、サービスディスカバリの重要性が増している。サービスディスカバリとはサービスのネットワーク上の位置や名前を参照・解決することである。ドメイン名と IP アドレスの対応関係を保持・解決する Domain Name System [3,4] はインターネット上におけるサービスディスカバリの一例である。

コンテナの実装として様々なものがあるが、代表的なソフトウェアとして Docker [5] が挙げられる。この Docker の登場により、アプリケーションを素早く実行・展開することが可能になった。

[1] 北陸先端科学技術大学院大学先端科学技術研究科
Graduate School of Advanced Science and Technology, Japan Advanced Institute of Science and Technology
[2] 北陸先端科学技術大学院大学情報社会基盤研究センター
Research Center for Advanced Computing Infrastructure, Japan Advanced Institute of Science and Technology

© 2022 Information Processing Society of Japan

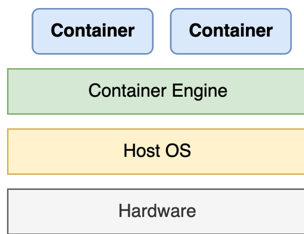


図 1　コンテナ型仮想化技術の概要図

Docker コンテナ利用時に作成される bridge ネットワークでは、ホスト内部に独自のネットワークを構築するため、ホスト外部からコンテナ内部で稼働するサービスに対して容易に発見することができない。そのため、コンテナ内部で稼働するサービスに対し柔軟にアクセスできるためのサービスディスカバリ機構が必要である。

本研究の目的はコンテナ環境内部で稼働するサービスを柔軟に発見できるようなサービスディスカバリを行うシステムを提供することである。派生する効果として、コンテナ内部で稼働するサービスに対してアクセスする時にネットワーク上の位置を意識することなく透過的なアクセスを行うことが可能となる。

本論文の構成について説明する。本論文は、本章を含め、7 章から構成する。第 2 章では、コンテナに関する基本事項や関連知識について説明する。第 3 章では、本研究において取り組むコンテナ環境におけるサービスディスカバリに関する課題と要求事項を整理する。第 4 章では、さまざまなサービスディスカバリの事項を説明する。第 5 章では，グローバルなサービス解決を行うための提案手法について説明し、その設計と実装について述べる。第 6 章では、第 5 章で述べた提案手法の効果について実証実験を通して明らかにする。第 7 章では、本研究をまとめ、本研究の総括と今後の展望について述べる。

2.　コンテナ型仮想化技術

本章ではコンテナ型仮想化技術についての基本事項について述べる。

2.1　概要

コンテナ型仮想化は単一の OS 上で複数の OS が動作しているように見せる仮想化技術である [1]。コンテナ型仮想化を利用することで、コンテナ内部で稼働するアプリケーションに OS 内部の資源を占有しているように見せかけることを可能にしている。従来の VM 型仮想化と比較してゲスト OS を仮想化しないので、起動・停止などが高速に行えることが特徴である。

2.2　Docker

Docker [5] とはコンテナを使いやすくするためライフサイクルを提供するプラットフォームである。Docker を用いることでコンテナ実行のためのコンテナイメージの作成・共有・実行といった一連のサービスを提供している。Docker コンテナの実現には Linux Namespace や Control Groups のようなカーネルの機能を最大限に利用する。

表 1　Docker における Network の種類

名前	概要
bridge	Linux カーネルの bridge ネットワークを使用
host	ホストマシンの NIC を直接使用
none	ネットワーク接続をしない

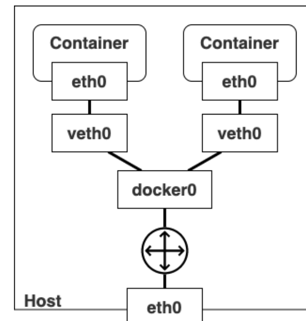


図 2　Docker の bridge 利用時の概念図

2.3　ネットワーク

Docker が接続されるネットワーク種別は主に三種類あり、表 1 に示す。bridge が利用された時の概念図を図 2 に示す。

bridge が用いられる場合は、Docker コンテナに接続される内部のネットワークは NAT を用いてプライベート空間で構成されるため、ホスト外部からコンテナに対して直接アクセスできない。そこで、ホスト外部からコンテナへの通信にはポートフォワーディング機能を用いる。そのため、コンテナ内部で稼働するサービスへアクセスするためにはホストの IP アドレスとポートフォワーディングで利用されるポート番号の情報を事前に把握する必要がある。

2.4　コンテナオーケストレーション

コンテナ技術の利用の増加に伴い、実プロダクトでもコンテナを利用するケースが増加している。そのような状況の中、大量のコンテナを人手で管理・運用するのは困難である。

そのような背景でコンテナオーケストレーションが登場した。コンテナオーケストレーションツールではコンテナ化されたアプリケーションのデプロイ・スケジューリングなどを一元管理することができる。コンテナ間の通信を行うためのネットワークやサービスディスカバリについてもコンテナオーケストレーションツールが提供している。代表的なコンテナオーケストレーションツールとして Kubernetes [6] が挙げられる。

© 2022 Information Processing Society of Japan

3. コンテナ環境におけるサービスディスカバリ

前章では、コンテナ環境に関する基礎事項を述べた。本章ではコンテナ型環境におけるサービスディスカバリにおいて本論文で解くべき課題を定式化する。

3.1 課題

コンテナが稼働しているホスト外部からコンテナ環境内部で稼働しているサービスに対してアクセスするには IP アドレスとポート番号の情報が必要となるが、グローバルなスコープでは名前解決することはできない。以上からグローバルなスコープでコンテナ内部で稼働するサービスにアクセスするための透過的なサービス名解決を行うサービスディスカバリ機構が求められる。

3.2 要求事項

以上のような課題から本研究で求められる要求事項をまとめる。

(1) サービスディスカバリ機構の設計
(2) サービス空間の設計
(3) 汎用的なコンテナ環境で利用可能

第一項目であるサービスディスカバリ機構の設計では、コンテナ内部で稼働するサービスへのエンドポイントをどのように共有して利用者にサービス名を解決させるか議論する。第二項目であるサービス空間の設計では、サービスの空間をどのように設計するかという議論を行う。第三項目は、特定の環境下ではなく、汎用的なコンテナ環境で利用可能であるためにどのような設計・実装を行うべきか議論する。本研究では、特に第一項目であるサービスディスカバリの観点に着目する。

4. サービスディスカバリ

本章ではサービスディスカバリに関連する事項をまとめる。

4.1 DNS

DNS はインターネット上における IP 通信実体の識別子である IP アドレスと人間が容易に識別できるよう決められたドメイン名の対応付けを行う名前解決機構である。

DNS はドメイン名空間をツリー構造に設計している。名前解決時には世界中の 13 の組織によって管理されているルートサーバから権限委譲されている DNS サーバに問い合わせを行うことでドメイン名の分散管理を実現している。

名前解決毎にルートサーバからの問い合わせを行うと時間がかかるという問題点がある。そのため、キャッシュを用いて同様の内容の問い合わせを一時的に保存して再利用する方法が用いられる。そのため、DNS には TTL: Time to Live がレコード内で定義されていて、その期間内は保持されたレコード情報が利用される。

4.1.1 SRV レコード

サービスロケーションを指定するための DNS レコードの一つとして SRV レコードが定義されている [7]。SRV レコードを用いることでポート番号を DNS 上で解決することが可能になる。

しかし、SRV レコードはドメイン名と IPv4/IPv6 の名前解決を行う A レコードや AAAA レコードと異なり、一般的に名前解決で利用される getaddrinfo() 関数などでは SRV レコードは参照されるように実装されていない。そのため、SRV レコードを透過的に利用するためには拡張実装や特殊な機構を準備する必要がある。

4.1.2 SVCB レコード

近年、HTTPS に関する接続情報を DNS 上で取得するために HTTPS レコードや HTTPS 以外の汎用的なプロトコルでも扱うことができるようにした SVCB レコードが提案されている [8]。これらのレコードにはポート番号を載せることもできるため、SRV レコードと同様にサービスを提供しているロケーションを示すことができる。

4.2 クラウド環境におけるサービスディスカバリ

4.2.1 Amazon Resource Name

Amazon Web Service では仮想サーバを扱う Elastic Compute Cloud(EC2) やストレージを扱う Simple Storage Service(S3) など様々なサービスが提供されている。そのような中、広大なサービス空間の中から特定のサービスを一意に発見する方法として Amazon Resource Name [9] が存在する。

Amazon Resource Name は AWS で提供されているリソースを識別することができる文字列である。ARN を用いることで、AWS 全体で提供されているリソースを一意に特定することを可能にしている。ARN の形式は以下でそれぞれの項目の意味については表 2 で示す。

- *arn*:*partition*:*service*:*region*:*account-id*:*resource-id*
- *arn*:*partition*:*service*:*region*:*account-id*:*resource-type*/*resource-id*
- *arn*:*partition*:*service*:*region*:*account-id*:*resource-type*:*resource-id*

表 2 ARN 構成文字列の意味

項目	概要
partition	リソースが置かれているパーティション
service	AWS サービス名前空間
region	リージョンコード
account-id	AWS アカウントの ID
resource-type	リソースの種類
resource-id	リソース識別子

© 2022 Information Processing Society of Japan

4.2.2 Kubernetes におけるサービスディスカバリ

Kubernetes ではクラスタを管理するため、様々なオブジェクトを提供している。その一つに Service リソースがある。Service リソースでは、クラスタ上で実行されるコンテナに対するエンドポイントの提供やサービスディスカバリを提供している。Kubernetes におけるサービスディスカバリの方法として大きく二種類ある。

- クラスタ内部 DNS
- 環境変数

クラスタ内部 DNS をサービスディスカバリに利用する場合には CoreDNS [10] のような機構を用いて Service リソースの情報を基にサービスディスカバリに利用される。しかし、他のオーケストレーションツールなどを含めて一意にサービスディスカバリを行うことができず、単一の環境下でしか利用できないという課題がある。

5. 提案手法

サービスを利用する際にはそのサービスが提供をおこなっているネットワーク上の位置や名前を利用の前に把握する必要がある。そのため、サービスの位置や名前を解決するサービスディスカバリを行う必要がある。

従来の物理マシンや仮想マシンは比較的静的なため、特定の値を埋め込む手法 (ハードコーディング) を用いてもサービスを利用することは可能である。しかし、コンテナのようなサービス利用のためのエンドポイントが動的に変化したり、状態が変化する環境ではサービスディスカバリは必要不可欠である。そのため、サービスディスカバリの重要性は増している。文献 [11] ではサービスディスカバリのデザインパターンについて記述されている。

5.1 サービスディスカバリにおけるデザインパターン

サービスディスカバリのデザインパターンについて大きく二つ存在する。

5.1.1 サービスレジストリ

サービスレジストリはサービスのエンドポイント情報を保持するデータベースである。クライアントはサービスレジストリを参照してサービスのエンドポイントを把握してサービスの実態にアクセスすることが可能となる。そのため、サービスディスカバリを行う環境では必要な不可欠なコンポーネントになるため、高可用性・高信頼性を有する必要がある。

5.1.2 クライアントサイドサービスディスカバリ

クライアントサイドサービスディスカバリではクライアントがサービスレジストリに対してクエリを送信して、その結果を基にクライアントがサービスに直接アクセスしに行く形態である (図 3)。

データベースを役割を果たすサービスレジストリのみを準備すれば良いので、システム的に非常に単純である。しかし、クライアントがサービスレジストリとサービスの実態にアクセスする必要がある。そのため、複数回のアクセス処理をを行う必要があるため、レイテンシーの面で課題がある。また、サービスへの負荷分散の機能を持たせることはできない。セキュリティの面では、サービスレジストリに対して不特定多数のユーザがアクセスすることを許容する必要がある。そのため、アクセス権限などを適切に付与しないと内容の書き換えなどの問題が発生してしまう。

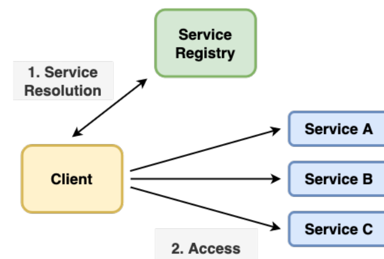


図 3 クライアントサイドサービスディスカバリの概念図

5.1.3 サーバサイドサービスディスカバリ

クライアントがサービスプロキシにアクセスし、サービスプロキシがサービスレジストリにクエリを送信して、その結果を基にサービスプロキシがサービスにアクセスしに行く形態である (図 4)。

この形態では、クライアントはサービスプロキシにアクセスするのみで、サービスの実態へのアクセスやサービスレジストリへのエンドポイント参照はサービスプロキシで行われるため、クライアントは一回のリクエストのみで良い。また、サービスプロキシにおいて負荷分散の機能を持たせることも可能である。サービスレジストリはサービスプロキシからのアクセスを想定するので不特定多数のアクセスは設計上考えられていない。しかし、サービスレジストリ以外にサービスプロキシが必要となるため、コンポーネントが増えて複雑になる。また、サービスプロキシが単一障害点となるため、サービスレジストリと同様に高可用性が求められる。サービスアクセスのリクエストがサービスプロキシの処理能力を超えてしまうボトルネックになる可能性がある。

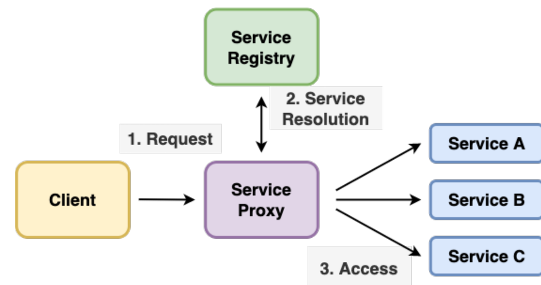


図 4 サーバサイドサービスディスカバリの概念図

© 2022 Information Processing Society of Japan

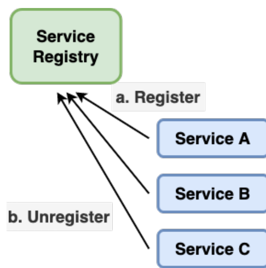


図 5　self registration 方式の概念図

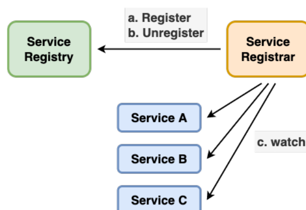


図 6　third-party registration 方式の概念図

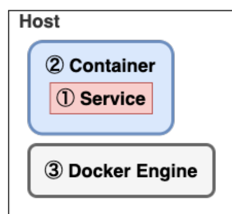


図 7　実行主体の検討

5.2　サービスの登録方法

5.2.1　self registration 方式

self registration 方式はサービス自身がサービスレジストリに対して登録や削除を行う手法である (図 5)。

構成要素がサービスレジストリのみなのでシンプルな構成である。しかし、サービスのインスタンスごとに登録する機構が必要である。

5.2.2　third-party registration 方式

third-party registration 方式はサービスの状態を監視するサービスレジスタラがサービスレジストリに対して登録や削除を行う手法である (図 6)。

サービスインスタンスの状態変更を検知したレジスタラがサービスレジストリに登録・削除を行う。サービスインスタンスに状態変化を通知する機構を組み込む必要がない。

5.2.3　サービスの状態変更情報共有の実行主体について

コンテナ内部で稼働するサービスにアクセスするために必要な情報をどの実行主体が登録・削除のような更新を行うかについて検討する。大きく実行主体として挙げられるのは以下の 3 候補である。

(1) サービス自体
(2) コンテナ
(3) Docker Engine

一つ目はコンテナ内部で稼働するサービスがポートなどの情報をサービスレジストリに登録する方法である。サービスの実態はコンテナの内部で稼働しているため、外部のエンドポイントの情報を保持しない。そのため、コンテナ内部からコンテナ外部に対して情報をリレーするためにプロキシのような機構が必要である。

二つ目はコンテナがサービスレジストリに登録する方法である。コンテナはコンテナ内部で稼働するサービスの情報やポートフォワーディング情報は所持しているため、コンテナ内部で稼働するサービスにアクセスするための情報を確認することができる。

三つ目はコンテナランタイムがサービスレジストリに登録する方法である。コンテナランタイムはコンテナに関する情報を全て有しているため、２つ目の方法と同様にコンテナ内部で稼働するサービスにアクセスするための情報を確認することができる。

3 種類のうち、コンテナ内部で稼働するサービスが登録などを行う方法の場合、サービス毎に新たにリレー機構を実装する必要があり、困難である。3 つ目のコンテナランタイムに実装する方法はコンテナランタイムがコンテナに関する情報を全て保持しているため可能である。しかし、ランタイムに対して実装を行い、それを普及させることは非常に難しい。そのため、コンテナに対してラッパー機構を提供する 2 番目の手法を採用する。この手法を用いることで、コンテナやコンテナエンジンに対して追加実装を行う必要がなく透過的に機能を提供することができる。

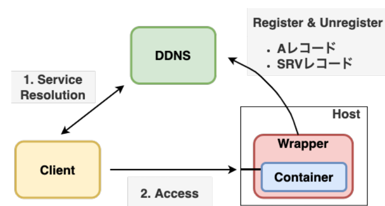


図 8　概念実装の設計

5.3　概念実証の設計と実装

概念実証の全体システム構成を図 5.3 に示す。

サービスディスカバリのデザインパターンについてサーバサイドサービスディスカバリではコンポーネントが増加することやアクセスの制約の観点からクライアントサービスディスカバリを用いる。サービスディスカバリのサービスレジストリとして DDNS [12] を用いる。

コンテナのような動的な環境に対応するため、DDNS を用いることで動的な変化に対応する。ラッパー機構を提供してコンテナ起動や終了時といった状態変更時にはコンテナのホストに付与される IP アドレスに対応する A レコードとサービスの位置を示すポート番号に対応する SRV レコードを DDNS に対して更新を行う。

本研究ではコンテナを用いるためのライブラリが豊富な Go 言語 [13] を実装に用いた。設計に基づき、CLI で操作

© 2022 Information Processing Society of Japan

表 3　実験環境 (Service Provider, Service Registry)

項目	内容
OS	Ubuntu 20.10 (Groovy Gorilla)
CPU	QEMU vCPU version 2.5+ (2GHz) x 4
メモリ	8GB

表 4　実験環境 (Proxy)

項目	内容
OS	NetBSD 9.2
CPU	QEMU vCPU version 2.5+ (2GHz) x 1
メモリ	512MB

表 5　使用ソフトウェア

項目	内容
Service Provider	Docker (Version20.10.8)
Service Registry	Knot DNS, version 3.2.dev
Proxy	SOCKS5 Proxy

可能な docker-gport.go を実装した。

6. 実証実験

前章までに、概念実証の設計と実装を行った。本章では、実証実験によって提案手法の効果を確認する。

6.1 実験環境

この章では実験環境について説明する。この実験ではプライベートクラウドを利用し、仮想マシンを 3 台使用した。使用した計算機環境については表 3, 4 に、それぞれの計算機で主として利用しているソフトウェアについて 5 に示す。また、システム構成については図 6.2 にまとめた。

まず、Service Provider ではサービスを提供するためのコンテナ環境を稼働させる。コンテナ環境として Docker を利用した。今回の実験では提供するサービスとして Web を想定する。Web アプリケーションとしてコンテナ内部で Nginx を稼働させる。コンテナ内部で稼働するサービスにホスト外部からアクセスするためポートフォワーディング機能を用いる。

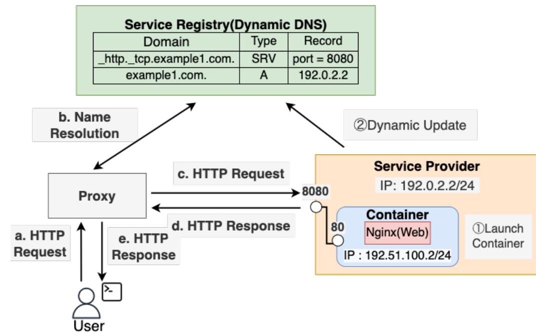
次に、Service Registry ではサービスのアクセス情報を格納するため、DDNS を稼働させる。利用するソフトウェアとして Knot DNS [15] を採用した。Service Registry で稼働する DDNS には Service Provider からアップデートから受信した内容を基にコンテナ環境にアクセスするための IP アドレス情報である A レコードとポート情報である SRV レコードを追加・削除する。

最後に Proxy では SOCKS Proxy を稼働させる。今回の実験では C 言語で実装された SOCKS5 [14] を利用する。

6.2 シナリオ

実証実験ではコンテナで稼働するサービスの提供とサービス利用を想定した実験を行う。

図 9　実証実験における環境



初めにサービス稼働時について説明する。コンテナ内部でサービスを提供し始める時に本研究で実装を行った docker-gport を利用する。docker-gport を利用すると DDNS を用いたレコードの登録とコンテナの起動が行われる。この時、起動する docker イメージやポート番号やドメイン名を付与することができる。

サービスを利用する際には DDNS を参照することでサービスのエンドポイント情報を取得してサービスの実態にアクセスを行う。クライアントから受信した HTTP リクエストを Proxy が受信する。そのリクエストにおけるドメイン名について Socks Proxy が名前解決を行う。この時、名前解決を行う際に実装内部で用いられる getaddrinfo() 関数内部で AI_SRV フラグを立てることによって、DNS の SRV レコードを先に参照することができる [16]。その結果を用いて他ホストのコンテナ内部で稼働するサービスに透過的にアクセスすることができる。

サービス提供時を想定した実験には docker-gport を利用してコンテナの起動と DDNS の追加ができるか試みる。サービス利用時を想定した実験には curl コマンドを利用し Proxy を経由した HTTP 接続ができるか試みる。

6.3 実験結果

サービス提供時を想定した実験について、docker-gport を利用して DDNS へのレコードの登録とコンテナの起動を確認する。コンテナ起動については docker コマンドを実行し、起動できていることを確認した。また、DDNS へのレコード登録については dig コマンドを用いて確認を行い、登録ができていることを確認した。

サービス利用時を想定した実験について、curl コマンドを用いてコンテナ内部で稼働する Web コンテンツを確認することができた。

7. まとめ

本章では、本研究における今後の総括・展望についてまとめる。

© 2022 Information Processing Society of Japan

7.1 総括

本研究ではコンテナ環境を用いたグローバルな環境においてサービスディスカバリについて検討を行った。本研究では DDNS をサービスレジストリとして利用してクライアントサイドサービスディスカバリを採用した。空間の設計についてはドメイン名に対して、IP アドレスとポート番号をマッピングすることでサービス名空間を設計した。実証実験により、ネットワークやサービスの位置を意識せず透過的なアクセスが可能なことを確認した。

7.2 展望

本研究での実証実験は最小の規模で行った。実際の運用の場合には多くのコンテナ群を扱うことが想定されるため、大量のコンテナを利用時の挙動を調査する必要がある。

また、DNS におけるポート番号に関するサービスディスカバリを容易にするため、SRV レコードを利用した。現状のレゾルバでは SRV レコードを解決する実装がほとんどなく、拡張実装が必要となる。そのため、サービス名を参照する SRV レコードや SVCB レコードを普及していくことでよりポート番号を透過的に解決できる汎用的なサービス名解決が行えると考えられる。

参考文献

[1] 川口 直也: コンテナ型仮想化概論, pp.7-10, カットシステム (2020).

[2] 坂下幸徳: マルチコンテナオーケストレーションを用いた大規模コンテナ環境の設計と運用, 情報処理, Vol. 62 No. 8, pp.d33-d56, (2021).

[3] P. Mockapetris: RFC1034, DOMAIN NAMES - CONCEPTS AND FACILITIES(オンライン), 入手先 ⟨https://datatracker.ietf.org/doc/html/rfc1034⟩ (参照 2022-05-21).

[4] P. Mockapetris: RFC1035, DOMAIN NAMES - IMPLEMENTATION AND SPECIFICATION(オンライン), 入手先 ⟨https://datatracker.ietf.org/doc/html/rfc1035⟩ (参照 2022-05-21).

[5] Docker, Docker(オンライン), 入手先 ⟨https://www.docker.com/⟩ (参照 2022-05-21).

[6] Kubernetes, Kubernetes(オンライン), 入手先 ⟨https://kubernetes.io/⟩ (参照 2022-05-21).

[7] A. Gulbrandsen: RFC2782, A DNS RR for specifying the location of services (DNS SRV)(オンライン), 入手先 ⟨https://datatracker.ietf.org/doc/html/rfc2782⟩ (参照 2022-05-21).

[8] IETF: Service binding and parameter specification via the DNS (DNS SVCB and HTTPS RRs)(オンライン), 入手先 ⟨https://datatracker.ietf.org/doc/draft-ietf-dnsop-svcb-https/⟩ (参照 2022-05-22).

[9] Amazon Web Services: Amazon リソースネーム (ARN)(オンライン), 入手先 ⟨https://docs.aws.amazon.com/ja_jp/general/latest/gr/aws-arns-and-namespaces.html⟩ (参照 2022-05-17).

[10] CoreDNS, CoreDNS(オンライン), 入手先 ⟨https://coredns.io/plugins/kubernetes/⟩ (参照 2022-05-21).

[11] Chris Richardson: Service Discovery in a Microservices Architecture(オンライン), 入手先 ⟨https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/⟩ (参照 2022-05-21).

[12] P. Vixie: RFC2136, Dynamic Updates in the Domain Name System (DNS UPDATE)(オンライン), 入手先 ⟨https://datatracker.ietf.org/doc/html/rfc2136⟩ (参照 2022-05-23).

[13] Google, The Go Programming Language(オンライン), 入手先 ⟨https://go.dev/⟩ (参照 2022-05-22).

[14] M. Leech: RFC1928, SOCKS Protocol Version 5(オンライン), 入手先 ⟨https://datatracker.ietf.org/doc/html/rfc1928⟩ (参照 2022-05-22).

[15] Knot DNS(オンライン), 入手先 ⟨https://www.knot-dns.cz/⟩ (参照 2022-05-22).

[16] getaddrinfo(3) - NetBSD Manual Pages(オンライン), 入手先 ⟨https://man.netbsd.org/NetBSD-8.0/getaddrinfo.3⟩ (参照 2022-05-22).

© 2022 Information Processing Society of Japan