修士論文

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

主指導教員 篠田 陽一

北陸先端科学技術大学院大学
先端科学技術研究科
（情報科学）

令和５年３月

Abstract

With the recent growth in a wide variety of IT services, including Web services, there is a high demand for flexible and agile systems for efficient service construction, operation, and management efficiency. In order to meet such needs, the use of container-based virtualization technology is increasing in cloud computing and distributed systems.

Container-based virtualization is an OS-level virtualization technology that enables the creation of containers, which are isolated spaces on a per-process basis, in which services and applications can be run. By using containers, it is possible to make it look as if the services inside the container are occupying OS resources. Containers are implemented using the functions provided by the kernel of the host OS. Containers do not virtualize the guest OS compared to VM, Virtual Machine, a hardware-level virtualization technology. So, containers have the advantages of fast boot-up, portability, and the ability to easily reduce environmental differences. Existing container implementations have not been able to use containers generically, so deploying services using physical servers or Virtual Machines for service provision was time-consuming and inflexible. However, with the advent of Docker, the current popular implementation of container-based virtualization, it is now possible to rapidly launch and deploy services and applications, enabling a quick response to constantly changing requirements and scalable service delivery.

The importance of service discovery that can handle dynamic changes in the endpoint that uses services in containers and fast status changes such as start and stop containers is increasing. The use cases of service provided by containers are increasing, and the number of containers is expected to increase in the future. The service providers must manage information to use many containers and provide services that allow service users to use this information. In addition, container networking builds its network inside the container host, and its namespace is hidden from the existing namespace. However, there are many access requests from outside the host to services running inside the container. We also considered that it would be important to ensure the reachability of services from outside the host by running a lot number of containers to use lightweight containers inside the computer effectively. Therefore, transparent name resolution is required to find

the service running inside the container from outside the host.

The purpose of this study is to provide a service discovery system that enables flexible external access to services running inside the container from outside the host. Additionally, we will discuss methods that can be used in general and that do not depend on managed services provided by cloud service providers. As a derivative effect, it will be possible to access services running inside containers transparently without being aware of the location of the services.

In this study, we investigated networking technologies in container environments, represented by the bridge Network, and service discovery topics in various environments. As a result, we found that the container network inside the OS hosting the container uses a technology that translates source and destination Internet and transport identifiers, making it possible to access from inside the container to outside the host. However, it was difficult to discover the services running inside the container from outside the host. To solve this problem, we design a system that satisfies the following requirements.

(1) Discoverable the services provided inside the container from outside the host

(2) Design of service name space

(3) Can be used in a general-purpose container environment

To design a system that meets the above-mentioned requirements, this study examined an optimal design method for a service discovery system that can access services running inside containers. Then, we propose a service discovery system that combines a dynamic name resolution mechanism using a distributed database on the Internet and a mechanism that exposed to the outside the host information on endpoints for reaching services running inside containers from inside the host that provides the container. The service provider stores the Internet identifier and transport identifier in the database at the beginning of service delivery using the service in the container, and the service user accesses the service by referring to this information. To take the benefit of the name resolution mechanism on the Internet, the design of the namespace and the name service were adopted to be included inside the existing namespace to ensure the transparency of the name resolution. Regarding the methodology for the method of exposing the service endpoints, we adopted a method that does not add any additional implementation to the existing

container implementation from the viewpoint of dissemination and transparency.

So as to confirm the operation of the proposed method, we conducted experiments assuming that the service is provided and that the service is used. In the experiments for service delivery, the proposed method was used to start a container and update information in a database. In the experiment for using the service, name resolution was performed using a SOCKS proxy to imitate the referring to information about the service, and Web access was performed. As a result, we confirmed that it is possible to update information for using web services provided in the container and to use the services provided inside the container. From the above, we confirmed that transparent service discovery is possible by using the method proposed in this study.

As a result of this research, we showed the possibility of transparent service discovery without being aware of the network or service location. We believe that by upgrading the methodology proposed in this study, it will be possible to achieve service discovery that does not depend on the environment of the application managing multiple groups of containers and thus managing more containers. As items for future study, the need for extended implementation of name resolver APIs that perform name resolution and the quantitative evaluation of scalability was clear.

概要

Web サービスを代表とする多種多様な IT サービスが展開される昨今、効率的にサービス構築や運用、管理を行うため、柔軟性や俊敏性の伴うシステムの構築・運用が求められている。そのような要求を満たすため、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。

コンテナ型仮想化技術は OS レベルの仮想化技術であり、これを利用することで、プロセス単位で隔離された空間であるコンテナを作成することができ、その中でサービスやアプリケーションを動かすことができる。コンテナを利用することで、コンテナ内部のサービスが OS 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。コンテナの実現には、ホスト OS のカーネルで提供される機能を利用している。コンテナはハードウェアレベルの仮想化技術である Virtual Machine と比較してゲスト OS を仮想化しない。そのため、高速に起動することが可能で、可搬性が高く環境差分をなくすことを容易に実現できるといったメリットがある。既存のコンテナ実装では、汎用的にコンテナを利用することはできていなかったため、サービス提供には実サーバや Virtual Machine を用いてのサービスのデプロイには時間がかかり、柔軟性がなかった。しかし、現在のコンテナ型仮想化の代表的な実装である Docker の登場によりサービスや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り、刻々と変化する要求に対して迅速に対応可能になり、かつスケーラビリティのあるサービス提供を実現することが可能になった。

コンテナにおけるサービス利用のエンドポイントは動的に変化し、かつコンテナの高速な起動・終了に対応できるサービスディスカバリの重要性は増している。コンテナを利用したサービス提供は増えており、今後もコンテナ数の増加が考えられる。そのため、サービス提供者は多数のコンテナにアクセスするための情報を管理する必要があり、利用者がその情報を利用できるサービスを提供する必要がある。また、コンテナの実現において、コンテナネットワーキングではコンテナホスト内部に独自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から隠蔽されている。しかし、コンテナ内部で稼働するサービスへのホスト外部からのアクセス要求は多い。また、軽量なコンテナを計算機内部で有効的に活用するためには大量のコンテナを稼働させ、ホスト外部からサービスの到達性を確保することが重要になると考えた。そのため、ホスト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。また、その実現の中で汎用的に利用可能でクラウドサービスで提供されるマネージドサービスなどに依存しない手法について検討を行う。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、サービス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。

本研究では、bridge network を代表とするコンテナ環境におけるネットワーク技術やさまざまな環境におけるサービスディスカバリに関する事項について調査した。その結果、コンテナをホストする OS 内部のコンテナネットワークでは、送信元・送信先のインターネット識別子とトランスポート識別子を変換する技術を利用しており、コンテナ内部からホスト外部へのアクセスは容易であるが、ホスト外部からコンテナ内部で稼働するサービスを発見することが困難であることがわかった。

この課題を解決するために以下の要件を満たすシステムを設計する。(1) コンテナ内部で提供されるサービスをホスト外部から発見が可能 (2) サービス名空間の設計 (3) 汎用的なコンテナ環境で利用可能

要件を満たすシステムを設計するため、本研究では、コンテナ内部で稼働するサービスに対してアクセス可能なサービスディスカバリシステムの最適な設計手法の検討を行った。そして、インターネット上の分散データベースを利用した動的な名前解決の仕組みとコンテナを提供するホスト内部からコンテナ内部で稼働するサービスに到達するためのエンドポイントの情報を外部に公開する仕組みを組み合わせたサービスディスカバリシステムを提案する。サービス提供者はコンテナにおいてサービス提供開始時にインターネット識別子とトランスポート識別子をデータベースに格納し、サービス利用者はその情報を参照してサービスに対してアクセスを行う。名前空間や名前サービスについて、インターネット上における名前解決の仕組みを利用するため、既存の名前空間の内部に包含する設計を採用し、名前解決の透過性を確保した。サービスのエンドポイントを公開する手法の実現について、普及性や透過性の観点から既存のコンテナ実装に対して追加実装を加えない手法を採用した。

提案手法の動作確認として、サービス提供時と利用時を想定した実験を行った。サービス提供時の実験では、提案手法を利用したコンテナの起動とデータベースへの情報の更新を行なった。サービス利用時の実験では、サービスレコードを参

照することを模倣するため、SOCKS プロキシを用いて名前解決を行い、Web アクセスを行なった。その結果、ホスト外部からコンテナ内で提供される web サービス利用のための情報の更新とコンテナ内部で提供されるサービスの利用が可能であることを確認した。以上から、本研究で提案する手法を利用することで、透過的なサービスディスカバリが可能になることを確認した。

本研究の結果、ネットワークやサービスの位置を意識することなく、透過的なサービスディスカバリの可能性を示した。本研究の提案手法を応用することで、複数のコンテナ群を管理するアプリケーションの環境に依存することのないサービスディスカバリの実現が可能になり、より多くのコンテナを管理することができると考えている。今後の検討項目として、名前解決を行うリゾルバ API の拡張実装やスケーラビリティ等に関する定量的評価の必要性が明らかになった。

目 次

図 目 次

表 目 次

第1章　はじめに

本章では本研究の背景と目的、本論文の構成について述べる。

1.1　背景

Web サービスを代表とする多種多様な IT サービスが展開される昨今、効率的にサービス運用・管理を行うため、柔軟性や俊敏性の伴うシステムの構築・運用が求められている。企業 IT 動向調査報告書 2022 [1] によると、IT 基盤における企業の優先課題に関する調査では、従業員規模数に関わらず、全体の 50%以上の企業が IT 基盤・システム開発において、”ビジネスに柔軟かつ迅速に対応できる IT 基盤の構築”を優先課題と認識している。

このような要求を満たすため、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（以下、コンテナ）の利用が増加している。コンテナ仮想化とは、一つのハードウェア上で複数の OS が動作しているように見せる OS レベルの仮想化技術である。コンテナの実現には、オペレーティングシステムの名前空間を利用し、分離された空間を作成して、その空間に対してプロセスやシステムリソースを制御・制限・割り当てを行う。コンテナはオーバーヘッドが小さく高速起動することが可能で、可搬性が高く環境を統一することが容易に実現できるといったメリットがある。

コンテナを利用したサービス提供の事例は増えている。コンテナを利用したサービス提供例として、ヤフー株式会社 [2] では、2020 年 6 月時点で約 13 万個のコンテナが稼働しており、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。この 6 ヶ月という短期間の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。

コンテナの実装としてさまざまなものがあるが、代表的なソフトウェアとして Docker [3] が挙げられる。既存のコンテナ実装では、汎用的にコンテナを利用する

ことはできていなかったため、サービス提供には実サーバや VM (Virtual Machine) 型仮想化を用いる必要があり、サービスのデプロイには時間を要したり、柔軟性がなかった。しかし、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り、高速なサービス提供を実現することが可能になった。

コンテナ数が増加し、その中でさまざまなサービスが提供される中、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の参照・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 [4, 5] はサービスディスカバリの一例である。

Docker コンテナ利用時に作成されるネットワークでは、ホスト内部に独自のネットワークを構築する。そのため、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検索・発見することができない。従って、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が必要である。

1.2 本研究の目的

本研究の目的は、ホスト外部から汎用的なコンテナ環境においてコンテナ環境内部で稼働するサービスをホスト外部から柔軟に検索・発見・利用可能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。また、その実現の中で多数のコンテナを生成・管理するためのツールであるコンテナオーケストレーションツールや Google や Amazon といったプラットフォーマーが提供するマネージドサービスに依存しない手法について検討を行う。

1.3 本研究の成果

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、サービス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。

1.4 本論文の構成

本論文の構成について説明する。本論文は、本章を含め、8 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、さまざまなサービスディスカバリの事項を説明する。第 4 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 5 章では、コンテナ内部で提供されるサービスを解決を行うための検討事項について説明する。第 6 章では、コンテナ内部で提供されるサービスの透過的な解決を実現するため、本研究で提案する提案手法である Transparent Containers について説明し、その設計と実装について述べる。第 7 章では、第 6 章で述べた提案手法の効果を動作確認を通して明らかにする。第 8 章では、本研究の総括し、本研究の成果と今後の展望について述べる。

第2章　コンテナ型仮想化技術

本章では、コンテナ型仮想化の概要について述べる。

2.1　コンテナ型仮想化技術の概要

コンテナ型仮想化は一つのハードウェア上で複数の OS が動作しているように見せる OS レベルの仮想化技術である [6]。コンテナ型仮想化を利用することで、プロセス単位で隔離された空間を作成し、コンテナ内部のアプリケーションが 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。図 2.1 にコンテナの概念図を示す。ホスト OS 上でコンテナエンジンが稼働し、コンテナエンジンがコンテナを動作させる。代表的なコンテナの実装である Docker では、コンテナの実現に OS のカーネルが提供する機能を利用する。コンテナ型仮想化は、VM 型仮想化と比較して、高速な起動・停止などが行えることができるなどといったメリットがある。

2.1.1　コンテナの実現

コンテナ型仮想化を実現するために、カーネルの様々な機能を利用している。

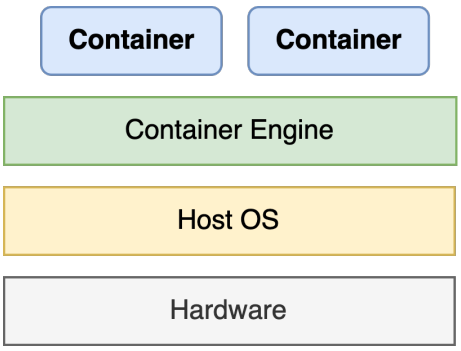

図 2.1: コンテナ型仮想化技術の概要

Namespace

Namespace [7] とは、リソースを抽象化して、新たに分離された空間を作成する機能である。名前空間内部のリソースはその他の名前空間とは分離されているため、名前空間内部のリソースは他の名前空間内部に動くリソースを見ることはできない。名前空間には、以下のような様々な名前空間が存在する。

- PID 名前空間：プロセス ID を分離
- Mount 名前空間：マウントポイントを分離
- IPC 名前空間：IPC: Inter-Process Communications プロセス間通信の分離
- Network 名前空間：ネットワークデバイス、スタック、ポートなどを分離
- User 名前空間：ユーザー ID とグループ ID を分離
- UTS 名前空間：ホスト名とドメイン名を分離
- Cgroup 名前空間：後述する cgroup を分離
- Time 名前空間：時間を分離

名前空間を制御する API(syscall) として unshare(2)、 clone(2)、 setns(2) などが挙げられる。

図 2.2 では 2 つの Namespace を作成し、PID 名前空間や Mount 名前空間がそれぞれに対してどのように見えているかを図示したものである。Namespace1 について、Namespace1 内部にあるプロセスはお互いに確認できる。しかし、Namespace1 から異なる Namespace である Namespace2 の内部に存在するプロセスを参照、利用することはできない。

Control Groups

Control groups [8]（以下、cgroups）はコンピュータに備わる CPU やメモリといった物理的なリソース・ハードウェアリソースを制限するための機能である。プロセスをグループ化してそのプロセス群に対してリソースを割当・制限を行うことができる。cgroups を利用して割り当てられたリソースよりも多くのリソースを

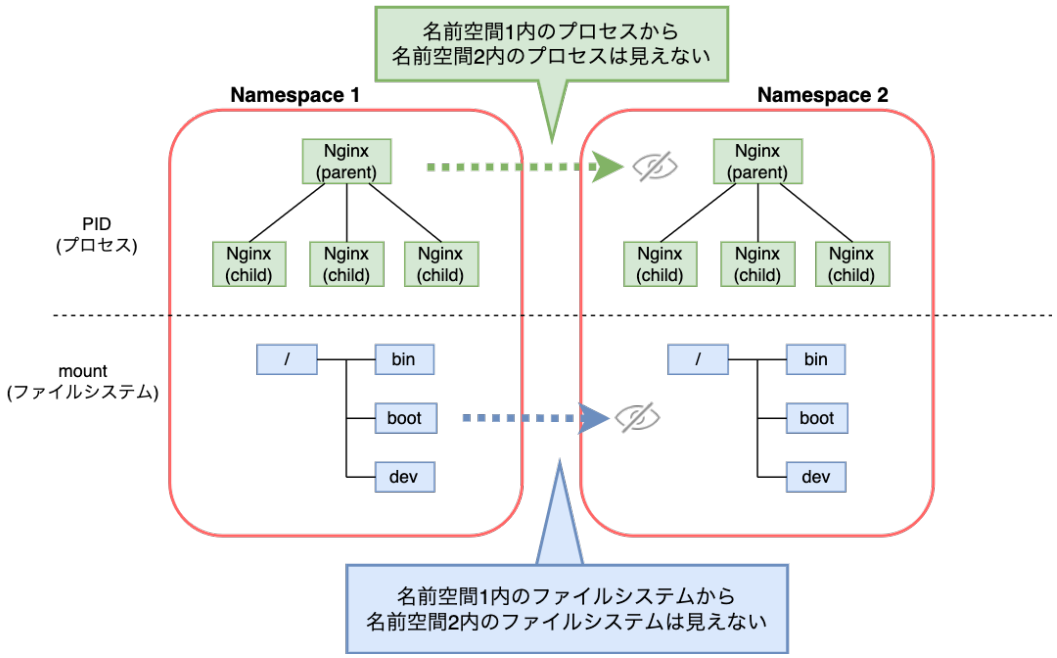

図 2.2: Namespace の概念図

使うことはできないため、コンテナではリソース制限の実現を行うために用いられる。

図 2.3 では、cgroup が二つ存在し、それらについてホスト OS が保持するリソースを提供している概念図である。それぞれの各 cgroup に割り与えられた以上のリソース以上のリソースを各コンテナは使うことができないため、有限のリソースを過剰に使わせないなどのリソース制限のために利用されている。

2.1.2 VM 型仮想化

VM 型仮想化とはハードウェアレベルの仮想化技術である。これを利用することで、物理的なマシンを複数台用意することがなくなるなどのメリットがある。VM 型仮想化には、KVM[9] や Virtual Box[10] のような実装が存在する。

2.1.3 コンテナ型仮想化と VM 型仮想化の比較

図 2.4 はコンテナ型仮想化と VM 型仮想化の比較概念図である。コンテナ型仮想化は、単一のホスト OS 上でコンテナエンジンを稼働させ、それを利用してコン

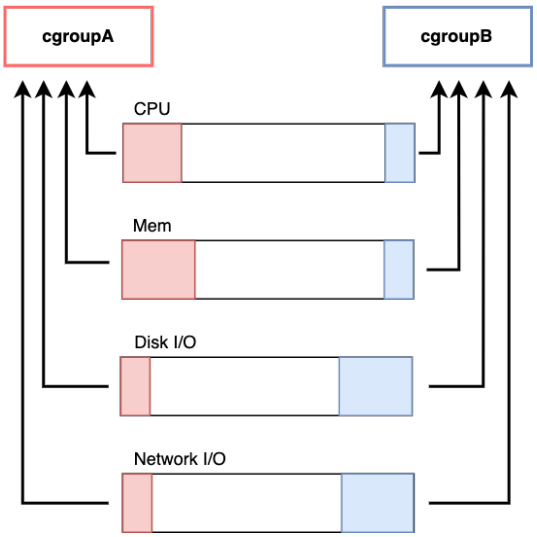

図 2.3: Control Groups の概念図

テナを動かす。それに対して、VM 型仮想化では、それぞれの VM で OS を仮想化し、その中でアプリケーションを動かす。

メリット

コンテナ型仮想化は VM 型仮想化と比較して、OS を仮想しない。そのため、コンテナは、VM 型仮想化と比較して実行環境が非常に軽量であり、起動・終了をはじめとする動作が高速である。また、オーバーヘッドが小さく計算資源の集約率・収容率の向上につながる。

コンテナ環境の利用により、他の環境への環境の統一が容易に可能になった。そのため、環境差分を意識することなくアプリケーションの実行が可能になる。具体的には、開発環境と本番環境の差分をなくし、開発や運用コストを削減することが可能になる。

デメリット

コンテナ型仮想化では、基本的にコンテナをホスティングする環境と同じ OS に限定される。なぜなら、コンテナはホスト OS を共有して利用しているからである。そのため、コンテナをホストする OS と異なった OS をコンテナで利用してサービス提供を行うことは出来ない。

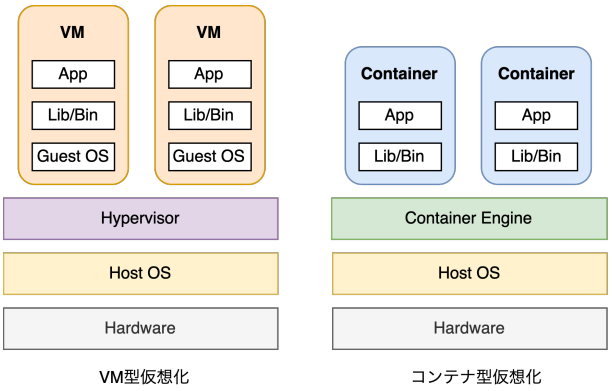

図 2.4: コンテナ型仮想化と VM 型仮想化の比較概念図

また、コンテナ型仮想化はホストマシンの OS を共有して使っているため、VM 型と比較して、仮想化が弱い。一例として、コンテナをホストするマシンの OS に脆弱性などが発見されると、そのホストで稼働している全てのコンテナに影響を及ぼす場合がある。そのため、コンテナを実行するコンテナランタイムの隔離性を高めるために gVisor[11] や Kata Containers[12] などを利用することで、セキュリティの向上につながる。

2.2 Docker

Docker とは Linux 上で動作するコンテナ型仮想化ソフトウェアである [3]。また、Docker を取り巻くプラットフォーム全体を指すこともある。

Docker は、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利用可能にしている。コンテナの作成では、Docker イメージを利用してプラットフォームに依存しないさまざまな環境下で効率的な開発を可能にしている。コンテナの共有では Docker 公式のイメージや Docker Hub[13] という他の開発者が作成したイメージを共有することを可能である。コンテナの実行では、多種多様なアプリケーションをテスト環境・ステージング環境・本番環境で同様に実行することができる。

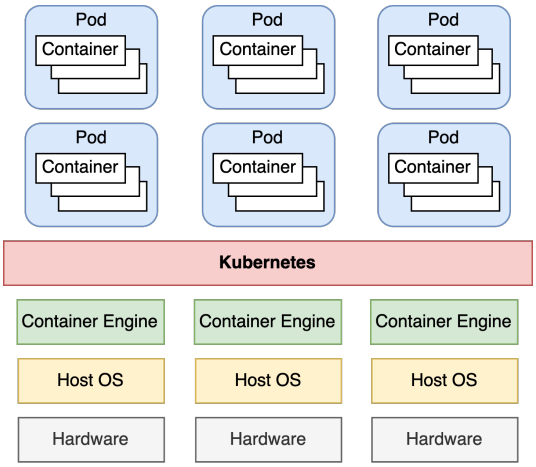

図 2.5: Kubernetes の概念図

2.3 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 [14] が挙げられる。

2.3.1 Kubernetes

コンテナオーケストレーションの代表的な実装の一つに Kubernetes が挙げられる。Kubernetes は Google 独自のクラスタリングシステムである Borg[15] を基に OSS 化されたオーケストレータである。Kubernetes では、構造化されたファイル上にシステム構成を記述することで、デプロイするコンテナやリソースを宣言的に管理することができる。

Kubernetes の概念図を図 2.5 に示す。複数のノードを連携して、Kubernetes を構成し、コンテナの管理・実行を行う。Kubernetes では、コンテナを管理する最小単位を複数のコンテナ群である Pod として扱う。

第3章　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。

3.1　インターネット上におけるサービスディスカバリ

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サービスディスカバリシステムとして Domain Name System（以下、DNS）が挙げられる。

DNS が生まれる以前は、インターネットの前身である Advanced Research Project Agency NETwork（ARPANET）では HOSTS.TXT を利用してホスト名と IP アドレスの対応関係を管理していた (図 3.1)。このファイルの管理については Stanford Research Institutes Network Information Center（以下、SRI-NIC）が行っていた。対応関係の変更や更新時にはその都度 SRI-NIC に連絡を行う必要があった。各利用者は FTP を利用して HOSTS.TXT を入手し、それを参照して名前解決を行う必要があった。

ホスト数が増大するに従い、以下のような問題が明らかになった。

- 利用者
  - HOSTS.TXT の更新頻度の増加

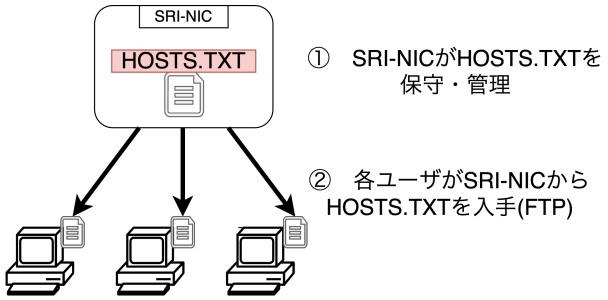

図 3.1: HOSTS.TXT を利用した名前解決の運用方法

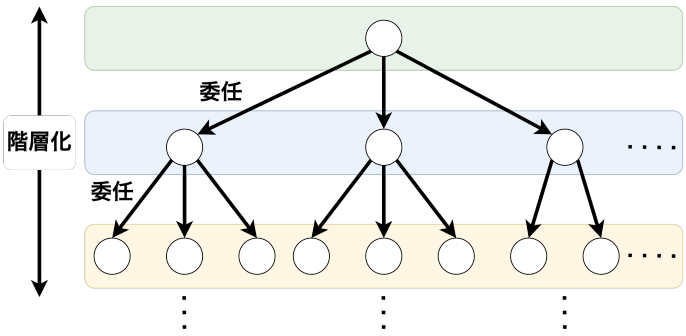

図 3.2: DNS の階層構造

  - 設定頻度の増加

- 管理者（SRI-NIC）
  - 更新頻度の増加
  - HOSTS.TXT の肥大化
- 計算機資源
  - ネットワーク帯域の消費
  - サーバの負荷

以上のような課題を解決するために DNS が開発された。

DNS はドメイン名空間をツリー構造に構成している分散データベースである。DNS は、名前空間を階層化し、階層の上位から下位へ管理を任せる委任を利用することで分散管理を可能にしている (図 3.3)。DNS における木構造の頂点となるサーバは Root DNS と呼ばれている。これらは世界各国 13 ヶ所で運用されており、日本では WIDE Project によって M Root[16] が運用されている。

DNS における名前解決の概略を図 3.3 に示す。DNS を利用した名前解決時には権威サーバ、フルサービスリゾルバ、スタブリゾルバを利用する。クライアントやアプリケーションからスタブリゾルバが呼び出される。スタブリゾルバがフルサービスリゾルバに対して再帰的問合せを行う。フルサービスリゾルバはスタブリゾルバから要求されたドメイン名の解決を行うため、権威サーバ群に対して非再帰的問合せを行う。権威サーバは、ゾーンの内部にあるドメイン情報を応答や委任情報を保持しており、フルサービスリゾルバから受信したクエリに対して応答する。

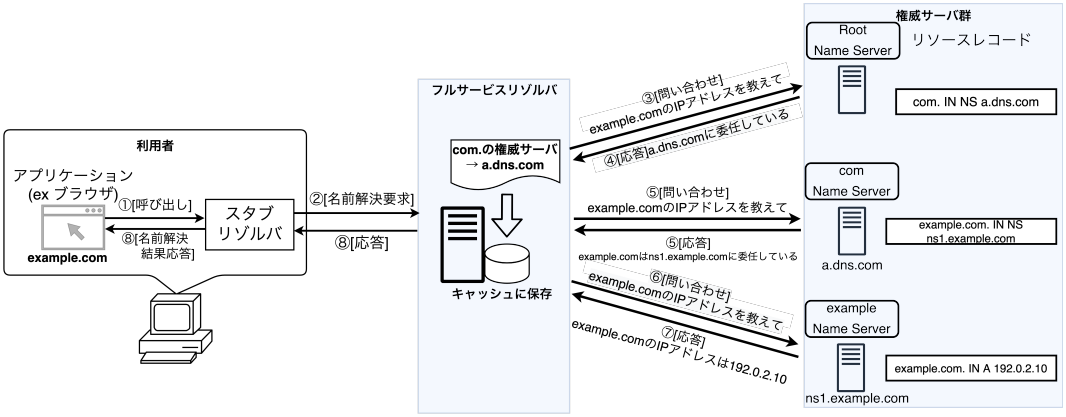

図 3.3: 名前解決の動作例（example.com）

名前解決毎に名前解決の実行や、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。そこでさまざまな高速化手法が利用される。代表的な高速化手法として用いられるキャッシュについて説明する。

キャッシュとは、DNS を利用した名前解決時に名前解決結果を一定時間だけ保存する仕組みである。フルサービスリゾルバはキャッシュされた内容を使用して名前解決を行うことが可能になる。このキャッシュを利用することで名前解決を行うフルサービスリゾルバの負荷軽減や、名前解決時に要する時間の削減ができる。キャッシュは DNS レコードに記述される TTL（Time to Live）に記載された期間のみ保持される。該当する DNS レコードが存在しないことを保存するネガティブキャッシュも存在する。

3.1.1 SRV レコード

サービスの位置を指定するための DNS レコードの一つとして SRV レコードが定義されている [17]。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。

3.1.2 HTTPS/SVCB レコード

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を扱うことを可能にした SVCB レコードが提案されている [18]。これらのレコードには、ポート番号の情報を記述することができるため、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。

3.2 クラウド環境におけるサービスディスカバリ

3.2.1 Amazon Resource Name

Amazon Web Service（以下、AWS）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 (以下、ARN)[19] 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 3.1 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*

ARN を利用したサービスディスカバリは AWS 内部のリソースを一意に識別することを可能にしたが、AWS 外部を含む汎用的なサービス識別のためには利用することはできない。

3.2.2 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、様々なリソースを提供している。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

表 3.1: 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

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[20] 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。

Kubernetes において、クラスタ内部の Pod に外部疎通性を提供する Service リソースとして、NodePort や LoadBalancer などが存在する。また、特定のプロトコル限定で外部疎通性を提供する Ingress リソースが存在する。

NodePort は、クラスタ内部に存在する Pod に対して、クラスタノードの IP アドレスとポート番号を指定することでクラスタ外部との疎通性を提供する。

LoadBalancer では、クラスタ外部に LoadBalancer Service を作成して、IP アドレスを払い出すことによって、外部からの疎通性を提供する。このサービスでは、クラスタ内部に NodePort を作成し、LoadBalancer への通信は NodePort を経由して Pod に転送される。この LoadBalancer はオンプレミスの場合には Kubernetes 標準の実装が提供されていないため、MetalLB [21] のような LoadBalancer の実装を利用する必要がある。また、Google Cloud Platform や Amazon Web Service のようなパブリッククラウド上で LoadBalancer を利用する場合には、各パブリッククラウド提供者が提供する LoadBalancer を利用することが出来る。

表 3.2: 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

Kubernetes において、クラスタ管理のためのリソースとして、Service リソースの他に Ingress リソースが存在する。Ingress とは、HTTP/HTTPS という特定のプロトコルに対して外部疎通性を提供するリソースである。Ingress を利用することで、URL を利用したパスベースルーティングや負荷分散の機能を提供している。

3.3 Docker におけるサービスディスカバリ

Docker が利用可能なネットワークは主に三種類あり、表 3.2 に示す。

デフォルトで bridge が適用され、Docker engine において DNS や DHCP のサービスが提供される。図 3.4 に bridge network を利用した Docker コンテナのネットワーク図を示す。Docker のようなコンテナエンジンでは、ホスト外部との接続性を提供するために NAT を利用する。

Docker では、同一ホスト内部のコンテナ同士は bridge ネットワークを利用することで、IP アドレスではなく、コンテナ名を指定してアクセスすることが可能になる。内部のネットワークは NAT を利用するため、外部からはアクセスできない。そのため、コンテナ外部から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、ホスト外部からコンテナ内部で稼働するサービスを利用するには、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。

3.4 NAT

Network Address Translation(以下、NAT) とは、IPv4 アドレス枯渇に対応するため、グローバル IP アドレスとプライベート IP アドレスの対応付けを行う技術である。この NAT を利用することでグローバル IP アドレスを大量に使うことなく、インターネットに接続することができる。

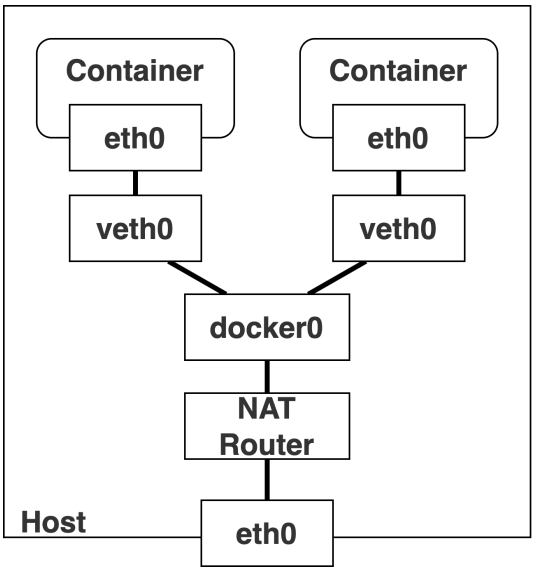

図 3.4: Docker の bridge 利用時のコンテナネットワークの概念図

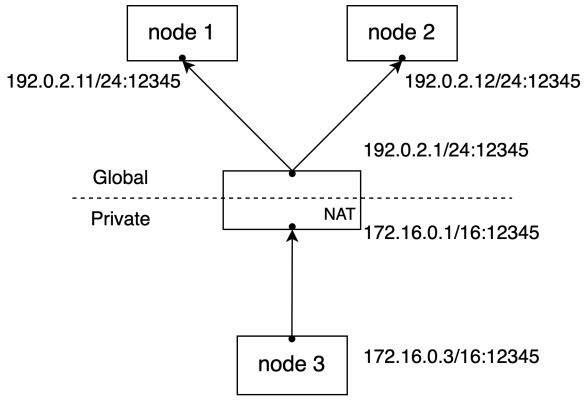

図 3.5: Cone NAT の動作

3.4.1 NAT の種別

NAT の種別には大きく、Cone NAT と Symmetric NAT が存在する [22]。

Cone NAT

Cone NAT とは、宛先が異なっても同じ送信元に変換を行う NAT のことである。図 3.5 に Cone NAT の動作を示す。Cone NAT 利用すると、宛先が異なっても NAT 上で送信先アドレスが特定の送信先アドレスに変換される。

さらに、Cone NAT には受信時の挙動によってさらに細分化することができる。

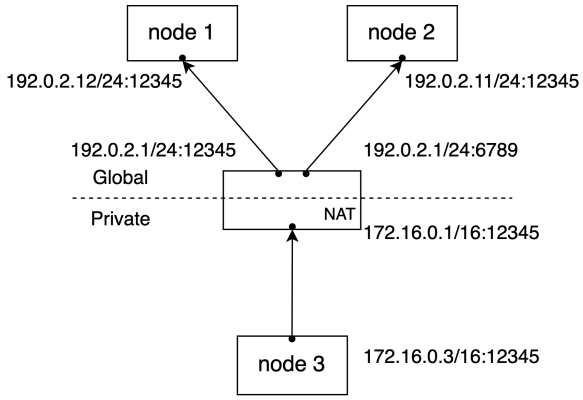

図 3.6: Symmetric NAT の動作

- Full Cone NAT
- Address-Restricted Cone NAT
- Port-Restricted Cone NAT

Full Cone NAT とは、全ての送信元から受信をプライベート空間に転送を行う。Address-Restricted Cone NAT とは、送信したことのある IP アドレスからの受信に対してのみ NAT 上で変換を行い、プライベート空間に転送を行う。Port-Restricted Cone NAT とは、送信したことのある IP アドレスとポート番号のペアの受信に対してのみ NAT 上で変換を行い、プライベート空間に転送を行う。

Symmetric NAT

Symmetric NAT は送信先が異なる場合には異なる送信元に変換を行う。図 3.6 に Symmetric NAT の動作を示す。送信先アドレスと送信先ポート番号のペアと送信先アドレスと送信先ポート番号のペアが一対一対応する。

3.4.2 NAT 越え技術

NAT を利用された環境では、互いのサービス提供場所が双方向にわからなくなってしまうことがある。そのため、STUN や TURN といった NAT 越え技術が利用される。それらについて説明する。

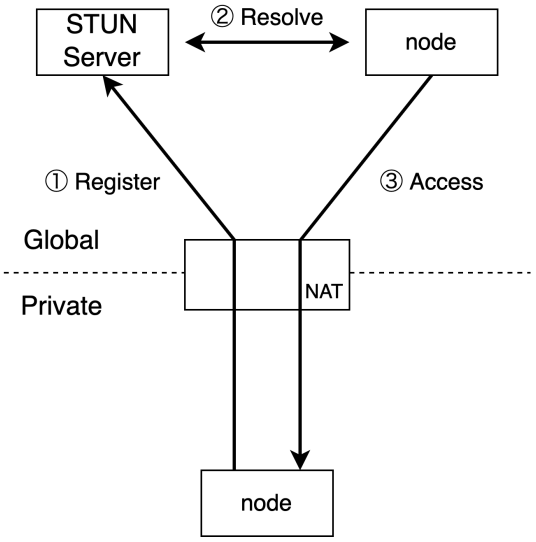

図 3.7: STUN の動作

STUN: Session Traversal Utilities for NATs

STUN[22] はグローバル空間に STUN サーバを用意して、NAT 外のグローバル空間において使われている IP アドレスを確認する手法である。これを利用することで、Cone NAT が利用される環境において NAT 越えを行うことができる。

図 3.7 は STUN の動作例である。初めに NAT 配下のノードがプライベート IP アドレスを利用して、グローバルアドレス空間に存在する STUN サーバに対して送信をする。すると、Cone NAT は送信先が異なる場合でも同じアドレスに変換されるため、グローバルアドレス空間を利用したクライアントも STUN サーバを参照して NAT 変換された後の情報を確認することで、プライベート空間のノードにアクセスするための情報を確認することができるためアクセスを行うことができる。

TURN: Traversal Using Relay around NAT

Cone NAT は STUN を用いることで、グローバル IP アドレスを確認することができるが、Symmetric NAT は通信先と一対一で変換を行うので、STUN ではグローバル IP アドレスを確認することができない。Symmetric NAT を越える手法として TURN[23] が挙げられる。TURN はノード間に中継サーバを用意して、リレー形式で通信を行う手法である。

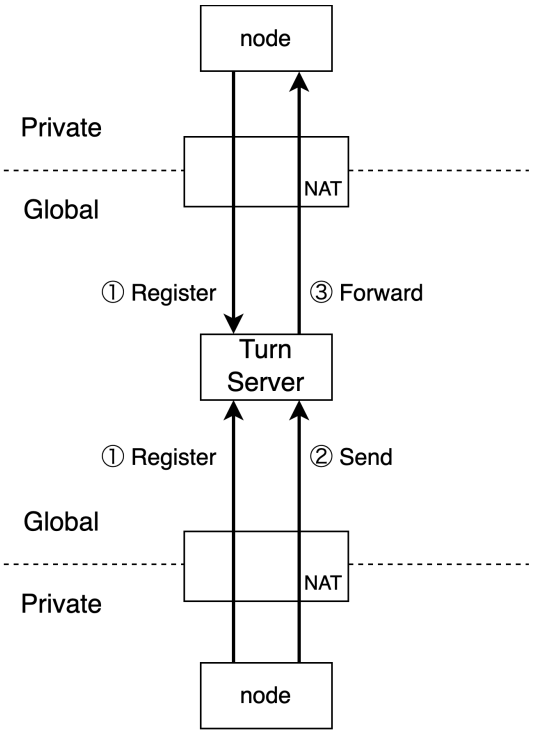

図 3.8: TURN の動作

図 3.8 は TURN の動作例である。TURN を利用した通信には、通信前に TURN サーバに認証や許可確認を行う。認証・許可を得られた場合には、TURN サーバを経由したデータの送受信を開始する。

第4章　コンテナ環境におけるサービスディスカバリの要件

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。

4.1　ホスト外部からのサービス到達性の重要性

コンテナを利用が増えている中、ホスト外部からのサービス到達性が重要になると考える。

高性能な汎用サーバに対して、単一ホストで稼働するサービスはそこまで大きくならないのではないかと考えている。そのため、大量のコンテナを単一ホストの中で提供して、外部からアクセス可能にすることで効率が上がると考えている。以上から、ホスト外部からコンテナ環境で提供されるサービスの到達性を確保することが重要になると考えた。

4.2　課題

コンテナ環境の利用が増加していく中、コンテナを利用する上でホスト外部からのサービス到達性が重要になると前節で述べた。

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。しかし、ホスト外部の名前空間と互換性が確保されていないため、ホスト外部からコンテナ内部で稼働するサービスを発見することはできない。

以上から、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システムが求められる。

4.3 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. コンテナ環境におけるサービスディスカバリシステム
2. サービス名空間の設計
3. 汎用的なコンテナ環境で利用可能

4.3.1 コンテナ環境におけるサービスディスカバリシステム

第一項目であるサービスディスカバリシステムの設計では、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。サービスディスカバリシステムの設計時には、データを保持するデータベースには、大量のコンテナを動作可能であり、アクセスを処理できる負荷分散の観点からも議論を行う。

4.3.2 サービス名空間の設計

第二項目であるサービス名空間の設計では、サービス名の空間をどのように設計するかという議論を行う。インターネット上における既存の名前空間であるドメイン名空間との互換性を保つためにどのような設計が最適か検討を行う。

4.3.3 汎用的なコンテナ環境で利用可能

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。既存の実装に可能な限り変更を加えることなく、透過的なサービス解決の実現を目指す。

第5章　コンテナ環境におけるサービスディスカバリに関する検討事項

5.1　名前空間の設計

既存のアプリケーションに対して変更を加えることなく、透過的な名前解決を実現するために名前空間の設計を行う。

名前空間の設計方針として、大きく 1. DNS における名前空間の内部にコンテナ環境における名前空間を包含する方法と 2. DNS における名前空間とコンテナ環境における名前空間を分離する方法がある (図 5.1)。

5.1.1　既存の名前空間の内部に包含する方法

DNS における名前空間の内部に置く場合には、DNS が提供する名前空間の管理や委任といった DNS が提供する機能を享受することができる。デメリットとして、DNS には一定期間クエリの結果を保存して、再利用するキャッシュ機構があ

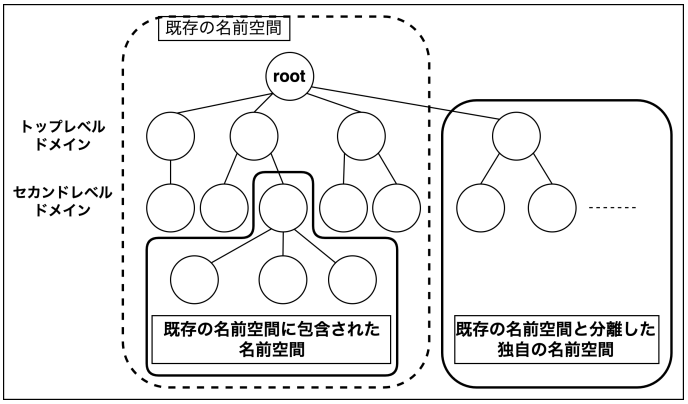

図 5.1: 名前空間の設計概念図

るため、動的な解決ができないという課題がある。そのような課題を解決するためにはDynamic DNS [24] を利用するという代替案も存在する。

5.1.2 既存の名前空間と分離する方法

既存の名前空間とは分離した独自の名前空間を作成する方法を検討する。具体的な手法として、DNSで管理されているトップレベルドメイン以外を利用することである。メリットとして、自由なシステムを設計・実装することでDNSを利用しない名前解決を実現することができる。しかし、デメリットとしてシステムを一から設計・実装する必要がある。

独自の名前空間を利用してサービスを提供している具体例としてmulticast DNS（以下、mDNS）[25] が挙げられる。mDNSはマルチキャストアドレスを利用することで、DNSと同様の名前解決を行うことを可能にしている。mDNSはDNSにおける権威サーバのような中央集権的なサーバを必要とすることなく名前解決が可能になる。mDNSの命名規則では、一般的にTLDが *.local* を利用する。そのため、既存のDNSと分離された名前空間を形成し、名前解決を可能にしている。

5.2 名前サービスの検討

名前サービスとして、以下の二種類のサービスを比較した。

- DNS
- DHT

DNSは木構造を利用した階層型になっている分散データベースである。一方、分散ハッシュテーブル（Distributed Hash Table、以下DHT）はPeer to Peerネットワーク上でハッシュテーブルを形成し、key-valueのデータベースを形成するアルゴリズムである。DHTのアルゴリズムとして、以下のようなアルゴリズムが挙げられる。

- Chord [26]
- Kademlia [27]

- Pastely [28]

DHT を利用するメリットとしてスケーラビリティが優れている。また、中央集権的なサーバが必要ない。しかし、P2P の特性上、サービスを利用・提供するユーザが存在する必要がある。また、負荷分散や空間の管理などを一から設計する必要がある。さらに、名前解決時に既存の名前解決機構を利用しない手法を採用すると、名前解決に利用されるリゾルバに対して追加実装を行う必要がある。

DNS と DHT のそれぞれの名前サービスは、以下の 4 項目について比較・検討を行った。

- 運用安定性: サービスを安定的に運用可能かどうか
- 負荷分散: 名前サービスに大量の名前解決要求が行われた際に適切に負荷分散されるかどうか
- 導入容易性:　名前サービスを使用開始したい時に容易に導入可能か
- 設計自由度: 名前サービスにおいてサービス名空間をどのくらい自由に設計可能か

5.2.1　運用安定性

DNS は階層構造と委任を利用して、名前の空間を管理している。しかし、ルートサーバを代表する権威 DNS サーバは単一障害点（SPOF: Single Point of Failure）である性質を持っている。そのため、セカンダリサーバを利用することで冗長性の確保を行っている。

DHT は、P2P ネットワークを利用したアルゴリズムである。そのため、ノードが常に出入りするため安定した状態がない。その状態を churn と呼び、DHT で提供しているサービスに対して、安定的にアクセスすることができなくなる場合がある。

5.2.2　負荷分散

DNS は階層構造と委任を利用して、各管理者が管理する名前空間を分割して、負荷分散を行っている。

DHT は P2P ネットワークに参加するノードが自立分散的に分散ハッシュテーブルを構成することで、負荷分散を実現している。

5.2.3　導入容易性

DNS は木構造の末端に DNS サーバを用意する必要がある。DNS サーバを用意して、上位のレベルのドメインから委任を受けると、自分の管理する名前空間を管理・運用することができる。

DHT は、中央集権的なサーバが必要ない。しかし、P2P の特性上、サービスを利用・提供するノードが自立分散的に存在する必要がある。そのため、ユーザが P2P システムを利用するモデルなどを検討する必要がある。

5.2.4　設計自由度

DNS を利用した名前サービスを提供する場合には、DNS の枠組みに乗っ取って設計する必要がある。そのため、設計自由度については DNS で提供されるサービスに依存する。

DHT を利用した名前サービスを提供する場合には、自由に設計することができる。一方、名前付けの規則や空間の設計・管理を一から検討する必要がある。

5.3　サービスディスカバリにおけるデザイン空間

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、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 [29] ではサービスディスカバリのデザイン空間について記述されている。

サービスディスカバリのデザイン空間について大きく二つ存在する。

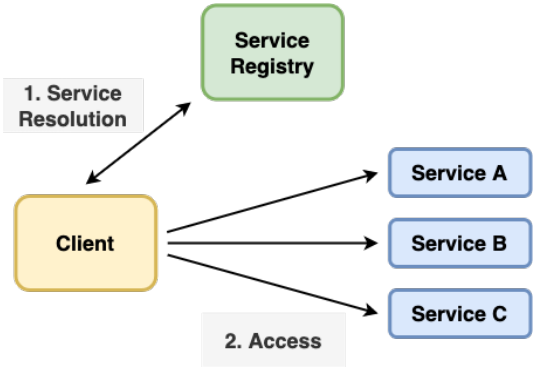

図 5.2: クライアントサイドサービスディスカバリの動作概念図

5.3.1 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。

5.3.2 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を行う形態である（図 5.2）。

5.3.3 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を行う形態である（図 5.3）。

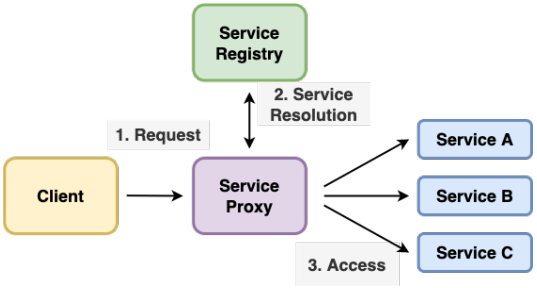

図 5.3: サーバサイドサービスディスカバリの動作概念図

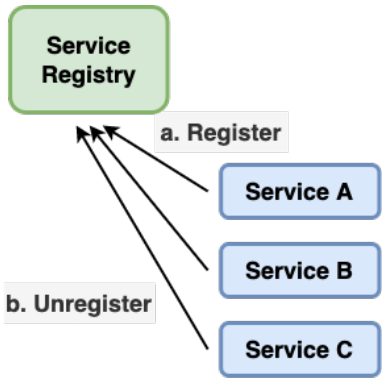

図 5.4: self registration 方式の動作概念図

5.3.4 サービスの登録方法

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方式である（図 5.4）。

構成要素がサービスの他に、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。

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方式である（図 5.5）。

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。

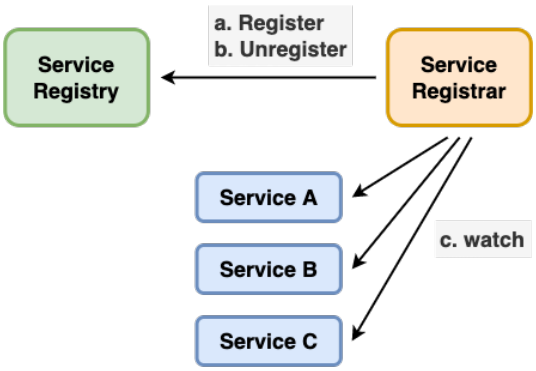

図 5.5: third-party registration 方式の動作概念図

第6章　提案手法: Transparent Containers

本章では、提案手法である Transparent Containers について述べる。

6.1　概念実証の設計

概念実証の全体システム構成を図 6.1 に示す。

名前空間について、既存の名前空間の内部に包含する手法を採用する。名前サービスは、DNS を採用する。そして、A レコードと SRV レコードを利用してドメイン名とコンテナ内部で稼働するサービスをマッピングする。サービスディスカバリのデザイン空間について、クライアントサイドサービスディスカバリを用いる。クライアントサイドサービスディスカバリのサービスレジストリとして、Dynamic DNS を利用する。

本設計の妥当性について後述する。

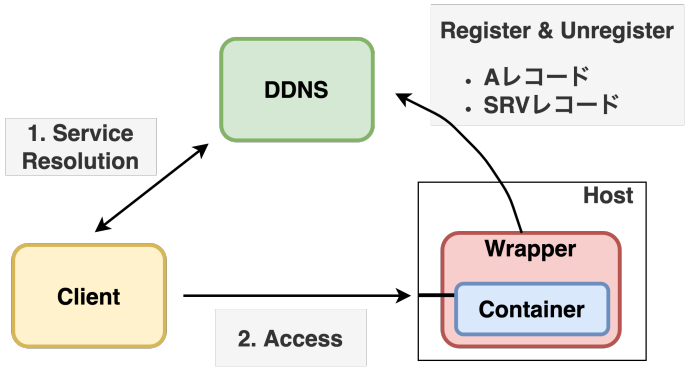

図 6.1: 概念実装の設計

6.1.1　名前空間の設計

既存の名前空間に包含する手法を採用することで、既存の名前解決手法であるDNS における名前空間の管理や委任を享受をすることができる。また、既存の名前空間と分離した名前空間を利用すると、既存の名前解決手法との透過性を確保するために追加実装が必要になり、透過性がなくなると判断した。以上から、既存の名前空間に包含する手法を採用した。

コンテナ環境内部で稼働するサービスを提供するには、ポートフォワーディングされるポート番号を共有するためのレコードを必要である。そのため、SRV レコードを用いてポート番号を共有する手法を採用した。その結果、ドメイン名に対して IP アドレス空間とポート番号空間をマッピングする設計となった。

6.1.2　名前サービス

名前サービスについては、DNS、DHT 共に実現可能であることがわかった。設計自由度について、DNS を採用する場合には、DNS で定義されているレコードを利用するという制約がある。DHT について、ノードが自立分散的に存在する必要があり、導入容易性が低い。そこで、本研究では、設計自由度について制約があり、導入容易性が高い DNS を名前サービスとして採用する。

DNS はインターネットのような大規模かつ静的な環境で用いられている。しかし、コンテナのような動的な環境では向かない。そのため、DDNS を用いることで動的な変化に対応する。

6.1.3　サービスディスカバリにおけるデザイン空間の比較・検討

クライアントサイドサービスディスカバリは、クライアントとサービスの実態に加えてサービスレジストリのみを準備すれば良いのに対して、サーバサイドサービスディスカバリでは、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しまう。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。

クライアントサイドサービスディスカバリでは、クライアントはサービス解決とサービス実態へアクセスの複数回のアクセスが必要となり、レイテンシーの面で課題がある。サーバサイドサービスディスカバリでは、クライアントはサービ

表 6.1: デザイン空間の比較

形態	コンポーネント数	アクセス要求数	アクセス制約
Client-Side Service Discovery	◯	△	△
Server-Side Service Discovery	△	◯	△

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。

クライアントサイドサービスディスカバリにおいて負荷分散の機能を持たせることはできないのに対して、サーバサイドサービスディスカバリでは、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が可能である。しかし、サービスアクセスの要求がサービスプロキシの処理能力や帯域を超えてしまうとサービスプロキシがボトルネックになってしまう可能性がある。

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。

本研究では、サーバサイドサービスディスカバリにおいてコンポーネント数の増加、サービスプロキシに対して高可用性を求められることを考慮して、クライアントサイドサービスディスカバリを採用する。

6.1.4 ラッパー機能の提供位置について

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データベースに対して共有するラッパー機能の提供位置について議論する。ラッパー機能の提供位置について、以下の 3 候補について比較・検討を行う (図 6.2)。

1. サービス自体
2. コンテナ
3. コンテナエンジン

コンテナ内部で稼働するサービス

1 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

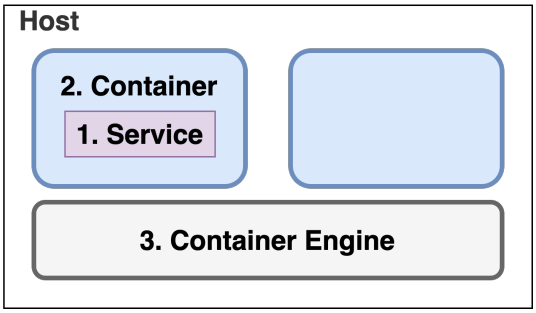

図 6.2: ラッパー機能の提供位置

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。

コンテナ

2 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。

コンテナエンジン

3 つ目はコンテナエンジンがサービスレジストリに登録する方法である。コンテナエンジン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 候補のうち、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 つ目のコンテナエンジンに実装する方法はコンテナエンジン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、それに対して実装を行い、普及させることは非常に難しいと考えた。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。

表 6.2: ラップ対象の比較

ラップ対象	機能追加が不要	普及性
1. サービス	△	◯
2. コンテナ	◯	◯
3. コンテナエンジン	◯	△

6.2 PoC の実装

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 [30] を実装に用いた。設計に基づき、CLI で操作可能な Transparent Containers を実装した。

第7章　動作確認

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

7.1　実験環境

本章では実験環境について説明する。

システム構成については図 7.1 にまとめた。この実験では、篠田研究室で運用する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 7.1、 7.2 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表 7.3 に示す。

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 を利用している。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 Nginx（HTTP）を使用する。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 DDNS を稼働させる。利用するソフトウェアとして Knot DNS[31] を採用した。DDNS では、コンテナ環境にアクセスするための IP アドレスの情報とポート情報を格納する。そのため、IPv4 アドレスの情報を記述する A レコードとポート番号の情報を記述する SRV レコードの二種類を利用する。

表 7.1: 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

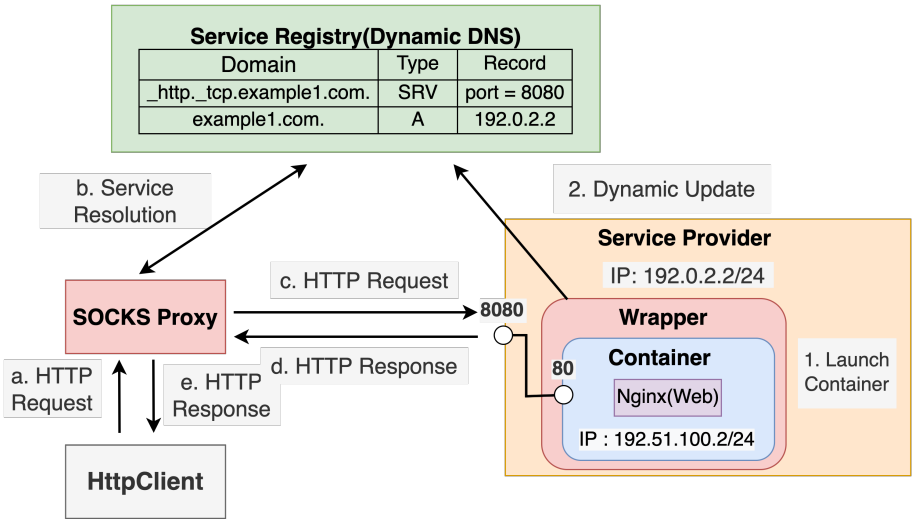

図 7.1: 実験環境

表 7.2: 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

最後に、SOCKS Proxy では SOCKS5[32] を稼働させる。今回の実験では C 言語で実装された SOCKS5 Proxy を利用する。

7.2 実験シナリオ

動作確認ではコンテナで稼働するサービス提供とサービス利用を想定した二種類の実験を行う (図 7.1)。

表 7.3: 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

7.2.1 サービス提供開始時

初めにサービス提供時を想定した実験には、提案手法である Transparent Containers を利用したコンテナの起動と DDNS のアップデートができるか確認する。

コンテナの起動については、実行時にコンテナイメージ名を指定することが可能になっている。また、起動時に Service Provier の IP アドレスに対応する A レコードと、コンテナがポートフォワーディングされるポート番号に対応する SRV レコードのアップデートを DDNS に対して行う。

7.2.2 サービス利用時

サービス利用時を想定した実験には curl コマンドを利用して Proxy を経由した HTTP 接続ができるか試みる。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 SOCKS 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可能になる [33]。その応答を用いて Service Provider 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。

7.3 実験結果

7.3.1 サービス提供開始時

実験結果 7.1 に Service Provider において、Transparent Containers を利用したサービス提供開始時のシナリオを実行した結果を示す。

実験結果 7.1: サービス提供開始時の挙動

```
1 $ ./tc -domainname=test.test.tc -extport=8080 -intport=80 -
      containerhost=150.65.117.107 run tc-test-image:1.0
2 container starting
3 container started
4 containerRunning: 6
      aa91ae68df658ae8f0bfff92e6514f68998c4ac6e8fbb07d5ee3590a99f636a
5
```

```
6 dns registering
7 sending to DNS(150.65.117.87) - test.test.tc.research.g1bbs.
      ninja:8080
8 dns register finished
```

正しく実行ができたかコンテナの確認と DNS のレコードの確認を行った結果を示す。

まず、実験結果 7.2 に dig を利用して DNS のアップデートを確認した結果を示す。その結果、正常にアップデートされていることが確認できた。

実験結果 7.2: DDNS のレコード確認

```
1  $ dig +norec SRV @150.65.117.87 _http._tcp.test.test.tc.research
       .g1bbs.ninja.
2
3  ; <<>> DiG 9.10.6 <<>> +norec SRV @150.65.117.87 _http._tcp.test
       .test.tc.research.g1bbs.ninja.
4  ; (1 server found)
5  ;; global options: +cmd
6  ;; Got answer:
7  ;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 12939
8  ;; flags: qr aa; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL:
       2
9
10 ;; OPT PSEUDOSECTION:
11 ; EDNS: version: 0, flags:; udp: 1232
12 ;; QUESTION SECTION:
13 ;_http._tcp.test.test.tc.research.g1bbs.ninja. IN SRV
14
15 ;; ANSWER SECTION:
16 _http._tcp.test.test.tc.research.g1bbs.ninja. 300 IN SRV 100 100
       8080 test.test.tc.research.g1bbs.ninja.
17
18 ;; ADDITIONAL SECTION:
19 test.test.tc.research.g1bbs.ninja. 300 IN A 150.65.117.107
20
21 ;; Query time: 2 msec
22 ;; SERVER: 150.65.117.87#53(150.65.117.87)
23 ;; WHEN: Wed Jan 18 18:36:13 JST 2023
24 ;; MSG SIZE rcvd: 142
```

次に、実験結果 7.3 に Docker コマンドを利用して、コンテナが正常に起動しているか確認した結果を示す。その結果、正常にコンテナが起動していることが確認できた。

実験結果 7.3: Docker コンテナの起動確認

```
1 $ docker container ls
2 CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
3 6aa91ae68df6 tc-test-image:1.0 "/docker-entrypoint···
    ." 17 minutes ago Up 17 minutes 0.0.0.0:8080->80/tcp,
    :::8080->80/tcp vibrant_goldwasser
```

以上から、Transparent Containers を利用して、コンテナの動作と DNS レコードの登録が完了したことを確認した。

7.3.2 サービス利用時

コンテナ内部で稼働するサービスを利用可能か検証を行う。httpClient から、SOCKS5 を経由したWebアクセスを想定して検証を行った。実験結果7.4にSOCKS Proxy をプロキシとして指定したを curl コマンドの実行結果を示す。

実験結果 7.4: curl の実行結果

```
1 $ curl http://test.test.tc.research.g1bbs.ninja -x socks5h
    ://10.0.7.234
2 Hello from Transparent Containers
```

以上から、SOCKS Proxy を利用してコンテナ内部で稼働しているサービスにアクセスすることが確認できた。

7.4 考察

本実験では、提案手法である Transparent Containers 利用して、コンテナ環境で提供されるサービスを外部から透過的に利用可能であることを確認した。解決すべき課題となっていたポート番号の情報をユーザが認知する必要することなくサービスに対してアクセスすることが確認することができた。

第8章　おわりに

本章では、本研究における総括と今後の展望・課題について述べる。

8.1　本研究の総括

本研究ではコンテナ環境における汎用的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。概念実証では、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の形態を採用した。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。動作確認により、ネットワークやサービスの位置を意識せずコンテナ内部で稼働するサービスに対して透過的な外部アクセスが可能なことを示した。

8.2　今後の展望

8.2.1　実験規模

本研究での動作確認では、サービス提供側は 1 ホスト 1 コンテナという本研究で提案する手法が取りうる最小の規模で行った。しかしながら、実際のサービス提供・運用時には非常に多くのコンテナ群を扱うことが容易に想像できる。そのため、大規模なサービス提供を想定したマルチホスト環境でサービスが提供可能かどうか、大量のコンテナを利用時の挙動の調査などについて実証実験を行う必要がある。

8.2.2　定量的評価

本研究での動作確認では、定量的な評価については行っていない。本研究における提案手法では、既存のサービスディスカバリシステムである DDNS をサービ

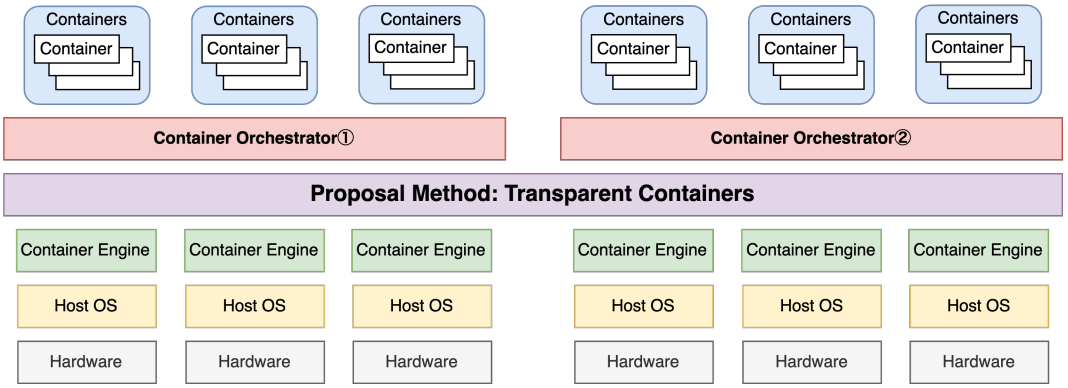

図 8.1: オーケストレータ非依存のサービスディスカバリ

スレジストリとして利用している。そのため、本研究の提案手法の定量的評価については DDNS に依存し、DDNS のアップデートの時に最大負荷がかかると考えられる。また、DNS にはキャッシュ機構があるため、その挙動を考慮する必要がある。

8.2.3 DNS リゾルバ実装における課題

DNS を利用したサービスディスカバリを可能にするため、本研究の動作実験では、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 レコードを解決することができる実装が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。

8.2.4 特定のオーケストレータに依存しないサービスディスカバリ

本研究では、汎用的なコンテナサービス解決の手法を提案した。この手法を利用することで特定のオーケストレータに依存することなく、大量のコンテナ発見可能になると考えている（図 8.1）。これが可能になることによって、多種多様なオーケストレータの差分を意識することなく透過的なコンテナサービス発見に寄与すると考えている。

参考文献

[1] 一般社団法人日本情報システム・ユーザー協会（JUAS）. 企業 IT 動向調査報告書 2022. https://juas.or.jp/cms/media/2022/04/JUAS_IT2022.pdf (参照 2022/09/30).

[2] 幸徳坂下. 快適な運用管理を支えるインターネットと運用技術：招待論文：2. 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, jul 2021.

[3] Docker. Docker. https://www.docker.com/ (参照 2019-01-01).

[4] Domain names - concepts and facilities. RFC 1034, November 1987.

[5] Domain names - implementation and specification. RFC 1035, November 1987.

[6] 川口直也. コンテナ型仮想化概論. カットシステム (2020).

[7] namespaces(7) — Linux manual page. https://man7.org/linux/man-pages/man7/namespaces.7.html (参照 2023/01/26).

[8] cgroups(7) — Linux manual page. https://man7.org/linux/man-pages/man7/cgroups.7.html (参照 2023/01/26).

[9] Avi Qumranet, Yaniv Qumranet, Dor Qumranet, Uri Qumranet, and Anthony Liguori. Kvm: The linux virtual machine monitor. *Proceedings Linux Symposium*, Vol. 15, , 01 2007.

[10] Oracle. Oracle vm virtualbox. https://www.virtualbox.org/ (参照 2022/12/26).

[11] gVisor. The container security platform. https://gvisor.dev/ (参照 2022/12/26).

[12] OpenInfra Foundation. The speed of containers, the security of vms. https://katacontainers.io/ (参照 2022/12/26).

[13] Docker Hub Container Image Library — App Containerization. https://hub.docker.com/ (参照 2023-01-29).

[14] Production-Grade Container Orchestration. https://kubernetes.io/ (参照 2019-01-01).

[15] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery.

[16] M-Root DNS Server. https://m.root-servers.org/ (参照 2022/09/29).

[17] Arnt Gulbrandsen and Dr. Levon Esibov. A DNS RR for specifying the location of services (DNS SRV). RFC 2782, February 2000.

[18] Benjamin M. Schwartz, Mike Bishop, and Erik Nygren. Service binding and parameter specification via the DNS (DNS SVCB and HTTPS RRs). Internet-Draft draft-ietf-dnsop-svcb-https-11, Internet Engineering Task Force, October 2022. Work in Progress.

[19] Amazon リソースネーム (ARN) - AWS 全般のリファレンス. https://docs.aws.amazon.com/ja_jp/general/latest/gr/aws-arns-and-namespaces.html (参照 2022/07/30).

[20] CoreDNS: DNS and Service Discovery. https://coredns.io/ (参照 2022/09/29).

[21] MetalLB, bare metal load-balancer for Kubernetes. https://metallb.org/ (参照 2023/01/26).

[22] Jonathan Rosenberg, Christian Huitema, Rohan Mahy, and Joel Weinberger. STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs). RFC 3489, March 2003.

[23] Tirumaleswar Reddy.K, Alan Johnston, Philip Matthews, and Jonathan Rosenberg. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). RFC 8656, February 2020.

[24] Paul A. Vixie, Dr. Susan Thomson, Yakov Rekhter, and Jim Bound. Dynamic Updates in the Domain Name System (DNS UPDATE). RFC 2136, April 1997.

[25] Stuart Cheshire and Marc Krochmal. Multicast DNS. RFC 6762, February 2013.

[26] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM'01*, pp. 149–160, 2001.

[27] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In Peter Druschel, Frans Kaashoek, and Antony Rowstron, editors, *Peer-to-Peer Systems*, pp. 53–65, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[28] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In Rachid Guerraoui, editor, *Middleware 2001*, pp. 329–350, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[29] Chris Richardson. Service Discovery in a Microservices Architecture. https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/ (参照 2022/05/21).

[30] The Go Programming Language. https://go.dev/ (参照 2022/09/30).

[31] High-performance authoritative dns server. https://www.knot-dns.cz/ (参照 2022/08/02).

[32] Marcus D. Leech. SOCKS Protocol Version 5. RFC 1928, March 1996.

[33] getaddrinfo(3) - NetBSD Manual Pages. https://man.netbsd.org/NetBSD-8.0/getaddrinfo.3 (参照 2022/08/04).

謝辞

本研究を進めるにあたり、多くの方に多大なご助言・ご助力をいただきました。ここに深く感謝し、心からお礼申し上げます。

本研究を進めるにあたり、主指導教員である篠田陽一卓越教授には多くのご助言とご指導を賜わりました。また、日々のご指導のみならず、物事の考えかたや知識の運用方法など研究のみに限らない物の考え方を教えていただきました。深く感謝いたします。宇多仁准教授には、ゼミなどの日々の研究に関する議論等におきまして多大な助言やご指導を賜りました。深く感謝いたします。丹康雄教授には、インターンシップ指導教員をお引き受け頂けただけではなく、研究に関しても様々な助言を賜りました。深く感謝いたします。リム勇仁准教授には、副指導教員をお引き受け頂いただき、研究に関する様々な助言を賜わりました。深く感謝いたします。前副指導教員である知念賢一教授には、研究に関する様々なご助言をいただきました。深く感謝いたします。

本研究室修了生の渡邊司揮氏、吉原昴司氏、油布翔平氏、馬越紘氏、本間可楠氏、梅内翼氏、岡田真一氏、瀧島和則氏、旧知念研究室修了生の門脇真之佑氏、古寺雄馬氏には研究に関する様々な議論や研究生活を送る上での多大なご助力を頂きました。本研究室の前期博士課程の髙橋亮真氏、中川颯馬氏には研究に関する活発な議論に加えて研究生活を送る上でも多大なご支援を頂きました。深く感謝いたします。

また、WIDE プロジェクトに所属する方々には、専門的な立場からのご意見やご指摘を頂けたこと、深く感謝いたします。

本論文の執筆にあたり、本研究室修了生の小比賀亮仁博士、三島航氏、ARIA プロジェクトの京都大学 廣井慧准教授には論文に関する助言や添削など多くの相談に乗っていただきました。深く感謝いたします。

最後に、研究生活を支えて頂いた家族、父 孝、母 マリ子、弟 雄飛、犬 ジョンに深く感謝いたします。本当にありがとうございました。

本研究に関する対外発表

[1] 片岡 拓海, 篠田 陽一: 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, マルチメディア、分散、協調とモバイル （DICOMO2022）シンポジウム論文集，vol.2022, pp.1117–1123 (2022).

[2] T. Kataoka: Design of Namespaces in Container Environments and Proposal of a Distributed Name Resolution Mechanism, WIDE 2021 May Meeting (2021).