卒業論文 2019 年度 (令和元年)

惑星規模の分散システムのための試験環境の設計と構築

慶應義塾大学　環境情報学部
廣川昂紀

卒業論文要旨 - 2019 年度 (令和元年度)

惑星規模の分散システムのための試験環境の設計と構築

本研究では, 惑星規模の分散システムのための試験環境の構築手法を提案する. 本研究における惑星規模の分散システムとは, 分散システムの中でも世界中に地理的に分散したコンピュータによって構成されるものを指す. 惑星規模の分散システムを支える技術として,P2P があげられる. P2P システムは, 中央集権的なサーバを必要とせず, お互いに対等な関係をもつコンピュータが協調動作することによって成り立つ. ブロックチェーンはその一例である. 惑星規模の分散システムにおけるテストでは, 地理的に分散することによるネットワークでの通信の遅延を考慮する必要がある. よって, 試験環境は地理的に分散配置されたサーバで構成されるべきである.

既存の惑星規模の分散システムの試験環境としては,PlanetLab やクラウドサービスの活用,BSafe.network があげられるが, それぞれ課題があると考える. PlanetLab はネットワークサービスの開発を支援する研究ネットワークであり, 世界中の 717 地域 1353 のサーバに対して SSH を通して操作を行えるが,OS や CPU, メモリなどのサーバの環境を柔軟に変更できない. クラウドサービスでは, リージョンと呼ばれるデータセンターの地域を指定することでサーバを分散配置できるが, リージョンが限定的であることに加え, 公にサービスが実稼働する環境に比べ通信の遅延が少ないため環境に差が生じてしまう. BSafe.network は 32 の大学によって構成されるブロックチェーン技術の研究を行うためのネットワークであり, 各大学が保有するサーバを用いて開発が行えるが, 各サーバの管理権限が分かれているため複数の大学間で共同研究を行う場合はオペレータの手作業が介入する. これらの課題を解決するため, 本研究では OpenVPN と Kubernetes を組み合わせた統合的試験環境の構築手法を提案する. 本システムでは, サーバがどこに分散配置されていてもネットワークでの通信の遅延を考慮したテストが可能である. 加えて, 仮想化技術を活用することでひとつのサーバ内に複数のアプリを様々な環境下で動作させることができ, 各サーバが別々の管理下にある場合でも統合可能である.

本システムの実装後, すべてのサーバの統合管理が可能であるか, ならびに通信の遅延を考慮した上で本システムが正常に動作するかを検証した.

本研究は, 惑星規模の分散システムのテストをより公の実稼働環境に近い形で柔軟に行うことを可能とし,BSafe.network のように各サーバが別々の管理下にある場合でもオペレータの手作業を省いた統合的試験環境の構築が可能である.

キーワード:
1. 惑星規模の分散システム, 2. 試験環境, 3. OpenVPN, 4. Kubernetes

慶應義塾大学　環境情報学部
廣川昂紀

Abstract of Bachelor's Thesis - Academic Year 2019

Designing and Implementation the test environment for planetary-scale distributed system

In this study, we propose the implementation method for a test environment of a planetary-scale distributed system. The planetary-scale distributed system in this study is the distributed system consisted of geographically distributed computers. P2P is one of the generic technologies for planetary-scale distributed systems. P2P system does not require a centralized server and is run by computers that have an equal relationship with each other and cooperate. Blockchain is one of the P2P systems. The test for planetary-scale distributed systems should consider network latency. Therefore, the test environment for planetary-scale distributed systems should consist of geographically distributed servers.

There are the PlanetLab and the cloud services, the BSafe.network as the test environment for planetary-scale distributed systems, but these have problems. PlanetLab is a research network that supports the development of network services. It has 1353 servers that are able to be controlled via ssh in 717 areas around the world but can not flexibly change an environment of the server such as OS, CPU, and memory. In a cloud service, servers can be geographically distributed by specifying an area of a data center called region. However, regions are limited and the delay in network communication is less than the public production environment. BSafe.network is the network for researching the blockchain technology that consists of 32 universities around the world. In the BSafe.network, developers can research using servers owned by each university. However, the manual work of each operator is needed for the joint research between universities, because the management authority of each server is divided. To solve these problems, we propose the implementation method for the integrated test environment by combining OpenVPN and Kubernetes. In this system, it is possible to test with the delay in network communication, regardless of where the servers are geographically distributed. In addition, multiple applications can be run on a single server in various environments by utilizing virtualization technology, and can integrate even when each server is under separate management.

We verified whether integrated management of all servers is possible and whether this system operates normally considering the delay in communication.

This research makes it possible to test planetary-scale distributed systems more flexibly in a similar manner to a public production environment. Even if each server is under separate management such as BSafe.network, It is possible to implement an integrated test environment without any manual works.

Keywords :
1. Geographically Distributed System, 2. Staging Environment, 3. OpenVPN, 4. Kubernetes

Keio University Faculty of Environment and Information Studies
Koki Hirokawa

目 次

図 目 次

表 目 次

第1章　序論

本研究では, 惑星規模の分散システムのための試験環境の設計と構築を行う.

本章では, 惑星規模の分散システムを定義し, 本研究の背景である惑星規模の分散システムの発達とシステムの試験環境について概説する. 本研究の課題を明らかにした上で, 目的を明確化し, 目的を達するための仮説を示す. 最後に仮説を裏付けるための提案手法を示し, 本研究の概要を示す.

1.1　本研究の背景

本節では, 本研究の背景について述べる.

初めに, 本研究が対象とする惑星規模の分散システムと試験環境について概説する. 分散システムについて概説した上で惑星規模の分散システムを定義し, 惑星規模の分散システムの発達について述べる. 加えて, 惑星規模の分散システムに求められる試験環境について説明する.

1.1.1　惑星規模の分散システム

本節では, 本研究における惑星規模の分散システムを定義する.

分散システムは, 複数の構成要素が組み合わさって動作するシステムである. 各構成要素は独立しており, 互いに協調動作することによってシステム全体が成り立っている.

惑星規模の分散システムは, 分散システムの中でも世界中に地理的に分散したコンピュータによって構成されるものを指す. 惑星規模の分散システムの基盤技術としては,P2P があげられる.

P2P は “Peer to Peer” の略記であり, 中央集権的なサーバを必要とせず, 各コンピュータが互いに対等な関係を築き協調動作することで成り立つシステムモデルまたはその技術自体を指す. P2P は, システム内に明確な役割分担と主従関係のあるクライアントサーバモデルとは対照的なシステムモデルである. クライアントサーバモデルは中央主権的なシステムであり, 通信において常にクライアントとサーバで一対一の関係が成り立つ. クライアントはサーバに対し処理や情報を要求し, 要求を受け取ったサーバは特定の処理を行なってクライアントへ返答する. 対して P2P システムでは, システムを構成する各コンピュータの役割は状況に応じて柔軟に変化し, 通信において多対多の関係が成り立つ. クライアントとして他のコンピュータに対し要求する場合もあれば, 他のコンピュータからの

要求に対して応答する場合もある．クライアントサーバモデルに比べて, 拡張性（スケーラビリティ）ならびに耐障害性において優れているのが特徴的である．

ビットコインの中核技術であるブロックチェーンは,P2P システムの一例である．ブロックチェーンのような P2P システムでは, 地理的に分散することによるネットワークでの通信の遅延を考慮した上で協調動作可能であることを開発者は意識しなければならない．

1.1.2 惑星規模の分散システムの発達

2000 年代初頭,Winny [1] や Gnutella [2] といった惑星規模の分散システムが頭角を現した．どちらのサービスも P2P 技術を基盤としており, それまでシステムモデルとして一般的であったクライアントサーバモデルとは異なる形態を採用したことで注目が集まった．P2P 技術が研究分野で取り上げられる頻度も多くなり, サービスとしても今後一層幅を広げていくと思われたが, クライアントサーバモデルに置き換わるまでの隆盛はなく後退していった．しかし,2008 年に Satoshi Nakamoto により Bitcoin のために開発されたブロックチェーン技術が登場することによって, 再度 P2P 技術が脚光を浴びるようになり, 開発や研究の勢いが再び盛んになってきている．

1.1.3 試験環境

試験環境とは, 公の実稼働環境での運用をする前にシステム全体の試験を行うための環境である．開発者が実際に開発を行う開発環境と公の実稼働環境では, 環境の差異から動作の違いが生じ, 開発環境で正常に動作していたものが公の実稼働環境に反映した途端動作しなくなるといった事象が度々発生する．そのような事態を防ぐために開発環境と公の実稼働環境の間に試験環境を構築し, 公の実稼働環境への適応前に試験環境にてシステム全体の試験をすることで予想外の障害が発生する可能性を低減できる．惑星規模の分散システムにおいても, システムの不具合を早期に発見するために試験環境が必要である．

先に述べたように, 惑星規模の分散システムでは地理的に分散することによるネットワークでの通信の遅延が発生する．よってシステムの試験は, 通信の遅延を考慮した上で正常に協調動作することを確認する必要があるため, 試験環境は地理的に分散したサーバによって構成されたものでなければならない．

既存の惑星規模の分散システムの試験環境としては, PlanetLab やクラウドサービスの活用, BSafe.network があげられる．PlanetLab は, ネットワークサービスの開発を支援する研究ネットワークであり, 世界中の 717 地域 1353 のサーバを利用することができ, クラウドサービスでは, リージョンと呼ばれるデータセンターの地域を指定することでサーバを分散配置することが可能である．最後に,BSafe.network は 32 の大学によって構成されるブロックチェーン技術の研究を行うためのネットワークであり, 世界中の各大学が保有するサーバを用いて開発を行える．

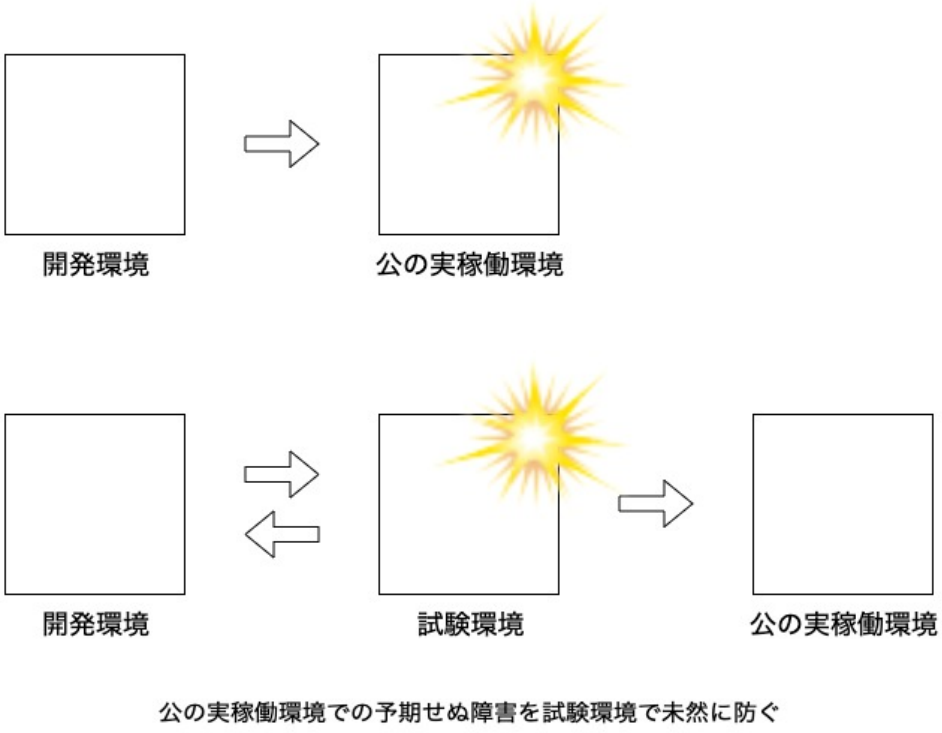

図 1.1: 試験環境

1.2 本研究の課題と目的

本節では, 本研究の課題と目的を述べる.

まず, 1.1 章で述べた本研究の背景を元に, 惑星規模の分散システムの試験環境における課題を示す. その上で本研究の目的を明確にする.

1.2.1 本研究の課題

本研究では, 既存の惑星規模の分散システムの試験環境の課題について指摘する.

惑星規模の分散システムの試験では, 地理的に分散配置されることによるネットワークでの通信の遅延を考慮した上で各コンピュータが正常に協調動作を行えるかを確認する必要があり, 試験環境では各サーバを実際に地理的に分散配置する必要性があることは 1.1 章の背景で述べた通りである. さらに既存の試験環境として, Planet Lab やクラウドサービスのリージョンの活用, Bsafe.network があげられるが, それぞれ課題があると考える. Planet Lab では世界中に分散したサーバを利用することが可能だが, OS や CPU, メモリなどのサーバの環境を柔軟に変更することができない. クラウドサービスでは, サーバの環境を自由に変更可能であり, リージョンを活用して地理的に分散した場所にサーバを設置することができるが, リージョンが限定的であり, 公の実稼働環境に比べネットワークでの通信の遅延が少ないため, 環境に差異が生じてしまう. BSafe.network では, 世界中の

32 の大学が保有するサーバを用いてシステムの試験を行えるが, 各サーバの管理権限が各大学のオペレータに委ねれられているため, 大学間での共同研究を行う場合にオペレータの手作業が介入してしまう.

本研究では, このように惑星規模の分散システムの試験環境の構築手法が整備されていないことを課題とする.

1.2.2 本研究の目的

本研究では, 惑星規模の分散システムのための試験環境の構築手法を提案することを目的とする.

1.3 本研究の仮説

1.2.1 章で述べた課題を解決するため, 本研究では地理的に分散したサーバを統合管理可能な試験環境の構築が必要であると考えた. 惑星規模の分散システムの試験環境は,

- OS や CPU, Memory といったサーバ環境を柔軟に変更可能であること
- 公の実稼働環境を想定したネットワークでの通信の遅延を考慮できること
- 異なる管理権限下にある各サーバに対し統合的管理が可能であり, 各オペレータの手作業を低減できること
- 地理的に分散した各サーバに対し, 統合的な操作が可能であること

の四点を満たさなければならない.

本研究では, 上記の必要要件を満たすことで 1.2.1 で述べた課題点を解決し, 1.2.2 で述べた惑星規模の分散システムの試験環境の構築手法の提案を達成できると考えた.

1.4 本研究の手法

本研究では, 1.3 章で述べた必要要件を満たすため,OpenVPN と Kubernetes を組み合わせた惑星規模の分散システムの統合的試験環境を提案する.

Kubernetes はコンテナオーケストレーションツールであり, コンテナ化仮想技術によってコンテナ化されたアプリケーションのデプロイやスケーリングを自動化し, 統合管理するためのシステムである. Kubernetes では複数のサーバでクラスタを構成しており, クラスタリングを行うためには各サーバが互いに IP レベルで疎通可能な状態になければならない. よって,IP レベルでの疎通が取れない別々のセグメントに配置されたサーバ間では Kubernetes クラスタを構築することはできない.

そこで地理的に分散し異なるセグメントに配置されたサーバ間を繋ぐ OpenVPN オーバーレイネットワークを構築することで, 各サーバを互いに IP レベルで疎通可能にする.

OpenVPN は,VPN ネットワークの構築をソフトウェアで実現するために開発されたオープンソースソフトウェアである.

本研究では, OpenVPN と Kubernetes を組み合わせ, 地理的に分散した拠点間で形成した OpenVPN オーバーレイネットワーク上で Kubernetes クラスタを構築した. 本システムが本研究における課題点を解決できているか推定することで, 要件を満たせることを確認した.

1.5 本論文の構成

本論文における以降の構成は次の通りである.

2 章では, 惑星規模の分散システムと試験環境ならびに本研究で使用する技術について概説し, 本研究の背景を明確化する. 3 章では, 本研究における課題を明確化し, 課題を解決するための要件, 仮説と手法について概説する. 4 章では, 本研究で提案する試験環境の構築方法について述べる. 5 章では,3 章で述べた課題に対しての評価を行い, 考察する. 6 章では, 本研究のまとめと今後の課題についてまとめる.

第2章　背景

本章では, 本研究の背景について概説する.

本研究における惑星規模の分散システムを定義し, 惑星規模の分散システムの基盤技術である P2P について概説する. 加えて, 本研究で対象とする試験環境の定義と惑星規模の分散システムに必要な試験環境について述べる. 最後に, 本研究の提案手法で用いるコンテナオーケストレーションシステムと OpenVPN について概説する.

2.1　惑星規模の分散システム

本節では, 本研究が対象とする惑星規模の分散システムを定義する. 惑星規模の分散システムについて概説する前に, 分散システムについて詳細を述べ, 違いを明らかにした上で本研究における惑星規模の分散システムを定義する.

2.1.1　分散システム

分散システムとは, 複数の構成要素が協調動作することによって成り立つシステムを指す. 各構成要素は独立して異なる役割を持っており, それらが組み合わさり互いに協調動作することによってシステム全体が動作する. 本研究で使用する Kubernetes も, コンテナオーケストレーションに必要な機能を構成要素毎に分割した分散システムである. Kubernetes に関する説明は 2.4.2 章で詳しく行う. 分散システムは複雑に思われるが, 機能が細分化され各構成要素の役割が明確化されるメリットがある. 機能同士の依存関係が希薄になるため細かい粒度での試験が可能となり, 障害時の原因特定も行いやすい. チーム開発において開発者同士で担当範囲が重複する可能性も下がるため, 開発スピードが向上するというメリットもある.

2.1.2　惑星規模の分散システム

本研究における惑星規模の分散システムとは, 分散システムの中でも世界中に地理的に分散したコンピュータが協調動作することによって成り立つシステムを指す. 近年注目を集めているブロックチェーンや 2000 年代初頭に登場した Winny といったサービスが, 惑星規模の分散システムの一例である. 惑星規模の分散システムを支える技術や例についての概説は 2.2 にて行う. 惑星規模の分散システムでは, システムを構成するコンピュータ

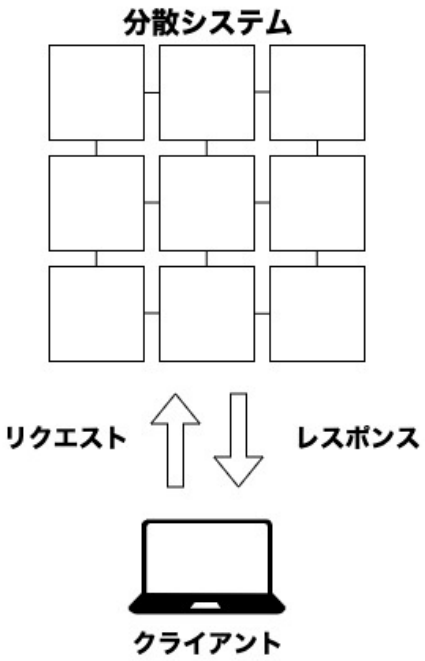

図 2.1: 分散システム

が地理的に分散していることから, 開発者はネットワークでの通信の遅延を考慮する必要がある.

2.2 惑星規模の分散システムにおける使用技術と参考例

本節では, 2.1.2 章で概説した惑星規模の分散システムを支える技術とサービス例について概説する. 惑星規模の分散システムの基盤技術として,P2P があげられる. P2P システムは中央集権的なサーバを必要としないクライアントサーバモデルとは対照的なシステムモデルである. P2P について概説したのち、P2P システムを採用するサービスとして, Winny, Gnutella, Bitcoin を紹介する.

2.2.1 P2P

P2P は “Peer to Peer” の略記である. P2P は、クライアントサーバモデルのシステムのように中央集権的な役割を担うサーバを必要とせず, コンピュータ同士が対等な関係を築く主従関係のないシステムモデルである. またはそれを実現する技術を指す。

クライアントサーバモデルでは、通信においてサーバとクライアントで常に一対一の関係性が成り立つ。また、システム内では処理や情報を要求するクライアントと要求に対し応答するサーバで明確な役割分担がある。サーバはクライアントから要求が送られてきた際は、特定の処理を行いクライアントに対し情報を返すが、それ以外では待機状態とな

る。対してクライアントは、情報を要求したり変更してもらう必要が生じた際のみサーバと通信を行う。よってクライアントサーバモデルにおける通信は、基本的にクライアントが起点となって行われる。

P2P では各コンピュータが互いに対等な関係性を築くため, クライアントサーバモデルのような明確な役割分担がシステム上ない．P2P では、各コンピュータが状況に応じてサーバとクライアントの役割を担う．各コンピュータが臨機応変にサーバとして要求に応答し, クライアントとして処理や情報を要求する動的システムが特徴としてあげられる．

クライアントサーバモデルでは, 要求する側をクライアント, 対して要求に応じる側をサーバと呼んでいる．P2P では前述した通り各コンピュータは動的に役割を変化させ、サーバとしてもクライアントとしても動作することからサーバントと呼ばれる．または単にノードと呼ばれることもある．

P2P の特徴

P2P では各コンピュータがサーバにもクライアントにも成り得るため, クライアントサーバモデルとは内部の実装も異なる．

第一に, データを保持する中央集権的なサーバが存在しないためアプリケーション上で必要になるデータは各コンピュータが保持する．アプリケーションの実装方式によっても異なるが, 各コンピュータがデータを分割して保持する場合もあれば全てのコンピュータが同じデータを保持する場合もある．例えばファイル共有システムである Winny では, 各コンピュータの保持しているデータは異なるため, データを参照する際はどのコンピュータが目的のデータを保持しているか検索し、対象となるコンピュータを決定してから通信を行う必要がある。また, ブロックチェーンでは各コンピュータが全てのデータを保持しており（全てのデータを持たない場合もある）, データを相互に検証し合うことによってデータの改竄耐性を向上させ, 堅牢性を担保している．

次に, システムのメインプログラムを各コンピュータが保持し動作させなければいけない点でもクライアントサーバモデルとは異なる．クライアントサーバモデルでは, システムのメインプログラムの実行はサーバの役割であるため, サーバのみがシステムのメインプログラムを保持しておけば良い．対して, 各コンピュータが状況に応じて役割を変える P2P ではシステムのメインプログラムを各々で保持する必要性がある．クライアントとして他のコンピュータが保持しているデータを参照したり, 情報を要求してきたコンピュータに対して応答をしなければならないからである。

P2P のメリット

本節では,P2P のメリットについて概説する．P2P システムの利点としては, 拡張性（スケーラビリティ）・耐障害性があげられる．

第一に拡張性に関しては, クライアントサーバモデルの場合, 利用者が増大するとシステムの中心であるサーバへアクセスが集中し, サーバやその周辺のネットワークへの負荷が高くなり, システム的な弱点になる．システム運用者は拡張性を高めるため, ネットワーク

機器のスペックをあげたり, 負荷が増大した際に自動でサーバの数を増やすオートスケーリングなどの対策を取らなければならない. それに対して P2P の場合, コンピュータ同士は相互に通信を行うためアクセスは分散されやすくなる. その点で P2P は拡張性に長けている.

次に耐障害性である. クライアントサーバモデルの場合, 何らかの原因でサーバが落ちるとサービス自体が停止してしまいサーバが構造上の単一障害点となる. しかし、P2P ではあるコンピュータが停止した場合でも, 正常なコンピュータ同士で新たなネットワークを形成することで滞りなくシステムの動作を継続することができる. 構造上の単一障害点が存在しないため, 障害性に長けている.

P2P のデメリット

本節では,P2P のデメリットについて概説する.

第一に情報伝達における遅延があげられる. P2P では接続先のコンピュータが常に決まっていないため, 状況に応じて接続先を変更する必要がある. すなわち, 目的の情報を保持しているコンピュータを探し出したり, そもそもネットワーク上で近い距離に他のコンピュータが存在しない場合, 情報の取得や送信に遅延が生じてしまう. 全てのコンピュータで同じデータを保持するブロックチェーンのようなシステムにおいては, コンピュータ同士がバケツリレーのようにデータを受け渡さなければならず, 端から端までデータを伝えるまでに時間が掛かってしまう問題点がある.

次にシステム全体での管理のしにくさがあげられる. P2P システムでは各コンピュータでアプリケーションを動作させるため, 中央集権的なサーバと異なり, 管理は各コンピュータ管理者に委ねられることになる. よって, システムに問題点が見つかり開発者が修正を含んだ更新版を配布した場合でも, 実際に動作しているアプリケーションが更新されるかどうかは保証されない. 同様にシステム全体の監視を行うことも困難である.

2.2.2 Winny

Winny [1] はソフトウェアエンジニア金子勇氏が開発し,2002 年に発表されたファイル共有ソフトである. システム上で中央集権的なサーバを保持せず, コンピュータ同士が相互に接続することで実現される P2P アプリケーションとして注目を浴びた. ユーザはコンピュータ内に保持されたファイルを他のコンピュータと共有することができるため, 任意のファイルをアップロードしたり, 逆に他のコンピュータが保持しているファイルをダウンロードすることができる. Winny では, 受信ファイルの送信元や送信ファイルの宛先をユーザが確認することはできず, バックグラウンドでの処理はユーザに見せないよう高い秘匿性が担保されていた. クライアントサーバモデルのシステムアーキテクチャとは打って変わって出た新しい形のアプリケーションであったが, 高い匿名性も起因して, 一部のユーザが違法な音楽ファイルや動画ファイル, コンピュータウイルスを Winny にアップロードしたことで著作権法違反が問われた. 開発者である金子氏にも疑いがかけられ 2004

年に逮捕, その後画期的な発明であった Winny も衰退していった. なお, 金子氏は裁判を経て 2011 年に無罪となった.

2.2.3 Gnutella

Winny に同じく Gnutella [2] も中央集権型サーバに依存せず, コンピュータ間の通信のみでファイルの送受信を行うファイル共有アプリケーションである. ファイルと言えど,Gnutella では主に音楽ファイルが共有されていた. Gnutella は AOL（アメリカ・オンライン）社のチームが開発したものである. 著作権保護の観点から, 法的に違法性を問われ公開も開発もすぐに中止されてしまった. Gnutella では, 最初のプログラムの起動時には, 接続先を自動で認識できないためファイル交換や検索機能は使用できない. Gnutella のシステムへ参加したい場合は, メールや掲示板を通して他の Gnutella サーバの IP アドレスとポート番号を教えてもらい, 自分の Gnutella サーバへ設定することで他のサーバとの通信を確立できる. 通信確立後は, 最初に接続した Gnutella サーバを通して他のサーバとも連携を取ることが可能となり, 音楽ファイルの交換や検索を行うことができる.

2.2.4 Bitcoin

Bitcoin [3] は 2008 年に Satoshi Nakamoto と名乗る人物によって論文にて提唱されたものである. 2009 年にはソフトウェアとして公開されており, 今では多くのユーザに使用されている上, 仮想通貨の先駆けとして他の仮想通貨を生む大きな起点となった. 同時に,2000 年代後半に勢いを失っていた P2P システムの存在を再度世に知らしめ, 開発の促進を促す起爆剤の役割を果たしたと考えられる. Bitcoin は基盤技術のひとつとして Winny や Gnutella と共通する P2P ネットワークを採用している. 参加するコンピュータはそれぞれがシステム上のデータを保持し相互にデータを検証しあうことで, 第三者的監視機関を必要とせずにデータの堅牢性を担保することが可能である.

2.3 試験環境

本節では, 本研究で着目する試験環境について概説する.

試験環境とは, システムの試験を行うための環境である. サービスを本番運用する環境と同じものを試験環境として構築し, 本番環境へデプロイする前に, 開発したシステムが期待する動作を行うか確認する. 試験環境で不備を発見した場合, 本番環境への適応はせずに開発環境にて修正を行う. 対して試験環境でのシステムの正常な動作を確認できた場合は, 本番環境へのデプロイ作業へ移行する. 開発環境と本番環境の間に試験環境を挟むことで, サービスの予期せぬ不具合や軽微なバグ等を早期に発見できる.

以下, 2.1 章で概説したシステムの試験方法ならびに必要となる試験環境について, それぞれ説明を行う.

2.3.1 モノリスの場合

モノリスは, 単一のコンポーネントで構成されるシステムである. 試験方法は, システムに対して任意の入力を与えた際に, 入力に対して期待する出力が行われるかどうかを確認すればよい. 具体的には, 特定の URL に対してリクエストを送信した場合に期待する Web ページが出力されるか, または Web ページのボタンを押した際に DB に対して期待する値が書き込まれるかなどである. システム内のコンポーネントはひとつであるため, 試験対象もひとつに限られる. 試験環境の構築時には, モノリスなコンポーネントを用意すればよい. サーバを用意する場所については本番環境に合わせればよいため, 自社サーバを用いる場合はオンプレ環境に, クラウド環境を用いる場合は任意のクラウドサービスを使用してサーバを準備すればよい.

2.3.2 分散システムの場合

分散システムは, モノリスとは異なり, 複数のコンポーネントから成り立つシステムである. 分散システムの試験を行う場合, 各コンポーネントに関してはモノリスと同じく, 任意の入力に対して期待する出力が返されるかを確認すればよい. しかしモノシリックなシステムとは違う点として, コンポーネント同士の協調動作が正常に働いているかどうかを確認する必要がある. 各コンポーネントが正常に動いた場合でも, システム全体が正常な状態にあるとは限らないからである. 例えば, システム内の特定のコンポーネントに障害が発生した場合, システム全体としてはエラーを返して障害が発生したことを明らかにしなければならない. 処理の巻き戻しや停止を行う必要がある場合もある. このような複数コンポーネントを跨いだ処理は, 一連の流れを通した上で確認する必要があり, 各コンポーネントの試験では不十分である. 一方, 試験環境の構築においてはモノリスと大きな違いはない. 本番環境を想定した場所にサーバを用意し, システムの動作に必要なすべてのコンポーネントを準備すればよい.

2.3.3 惑星規模の分散システム

惑星規模の分散システムは, 分散システム同様複数のコンポーネントで構成されるシステムである. 独立したコンポーネントがお互いに協調動作することによってシステム全体が動いている. 2.3.2 の分散システム同様, 各コンポーネントでの試験とシステム全体での試験を行えば, システムが正常に動作することを保証できる. しかし, 惑星規模の分散システムの試験において各コンポーネントの試験は他のシステム同様に行えるが, システム全体の試験は容易ではない. 何故なら, 惑星規模の分散システムでは各コンピュータ（コンポーネント）は地理的に分散しており, 試験環境の構築では地理的な場所を指定し分散させてサーバを配置したいからである. モノリスや分散システムの場合, サーバの配置は開発者が任意に設定できるため, 本番環境と同じ環境を容易に再現できたが, システムの構成要素の配置と数を固定できない惑星規模の分散システムにおいては再現が難しい. とはいえ無限にスケーリングする可能性のある惑星規模の分散システムを完璧に再現すること

は不可能であるため, 地理的に分散させた幾つかのサーバによって構成される試験環境を構築する必要があると考える. 実際に地理的に分散させたサーバによって成り立つ試験環境上で, コンポーネント同士の協調動作が正しく働いていることを確認できれば, 対象の場所においてのシステムの正常性が担保される.

2.4 コンテナオーケストレーションシステム

コンテナオーケストレーションシステムは, コンテナ型仮想環境を統合管理するためのプラットフォームおよびツールを指す. 2010 年代半ばから脚光を浴びるようになり, 今では世界的に数々のプロジェクトで本番環境に適用されている. サービスの立ち上げや運用過程において必要となる機能が数多く搭載されており, 開発者は素早くかつ効率的に開発を進められる. コンテナは Virtual Machine（以下,VM）のデメリットを考慮して作られており, 今後 VM の代わりを担う次世代の技術としてより一層注目されていく技術であると考える.

本研究では, コンテナオーケストレーションシステムとして Kubernetes を,CRI（コンテナ・ランタイム・インターフェース）として Docker を使用した. 本節では, コンテナおよびコンテナオーケストレーションの概説と実際に使用した Kubernetes や Docker といったツールについて紹介する.

2.4.1 コンテナ

本節では, コンテナおよび Docker について概説する.

コンテナ型仮想化は, ひとつのコンピュータ上で仮想的に別のコンピュータを動作させる技術である. ホスト OS の上で動いている別のコンピュータをひとつひとつをコンテナと呼ぶ.

コンテナについて説明するにあたり,VM や物理マシンと比較しながら特徴を示していく. コンテナは, 挙動としては VM と似ており, どちらも同じ課題を解決している. VM が登場する前, 開発者はひとつのサーバ上で複数のアプリケーションを動作させることに頭を悩ませていた. 何故なら, アプリケーションのうちのひとつがサーバのリソースを大幅に占有した場合, 他のアプリケーションのパフォーマンスが低下してしまうからである.

解決策のひとつとして, アプリケーション毎に別々のサーバ上で動作させるものがあったが, デメリットとして維持費が嵩むことと使用されない無駄なリソースが生まれてしまうことがあった.

これを解決するために開発されたのが VM である. VM はソフトウェアによって仮想的に物理マシンを実現する技術であり, ひとつの物理マシン CPU 上で複数の VM を動作させることが可能である. アプリケーションはそれぞれ独立しておりお互いに不可侵な関係性であるため, ひとつのアプリケーションがリソースを占有することはなく, よりリソースを効率的に使用できる. スケーラビリティにも長けており, 開発者はいつでもアプリを追

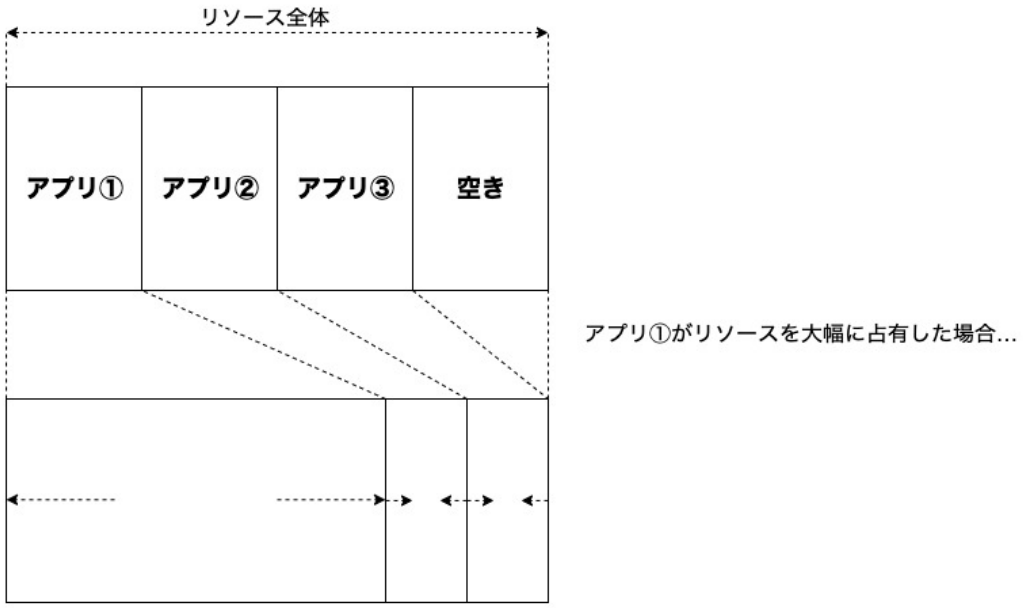

図 2.2: アプリケーションのひとつがリソースを大幅に占有した場合

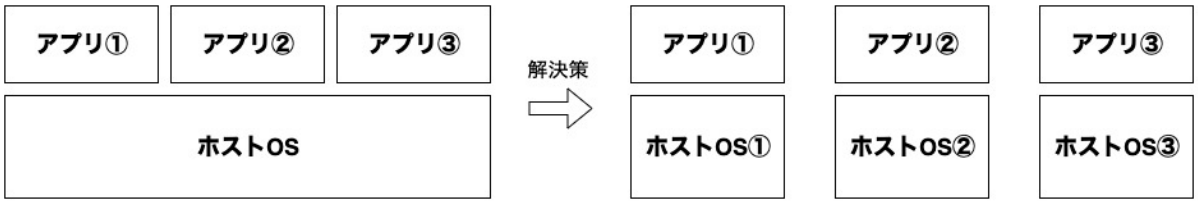

図 2.3: ひとつの物理サーバでひとつのアプリケーションを動作させる解決策

加・削除でき, ハードウェアコストの削減にも貢献している. しかし,VM は処理におけるオーバーヘッドが大きく起動時間が長いなどデメリットも存在する.

VM の後に登場した技術がコンテナ型仮想化である. コンテナ型仮想化では, 各アプリケーションはひとつのホスト OS を共有するため,VM より軽量で起動時間も短い. コンテナはコンテナイメージから作成され, イメージは宣言的なファイルに基づいて生成される. これによって開発者はより簡単かつスピーディに開発を進めることが可能である. ”Build Once, Run Anywhere”というコンセプトが掲げられており, 一度生成されたイメージはどの環境でも動作し冪等性が担保される. 一方, ホスト OS を共有するためセキュリティ面では課題が見られる.

コンテナ仮想環境を構築するためのランタイムである CRI には,dockershim（Docker）, containerd, cri-o, Frakti, rktlet（rkt）などが挙げられる. 本研究では,CRI のデファクトスタンダードである Docker を採用している.

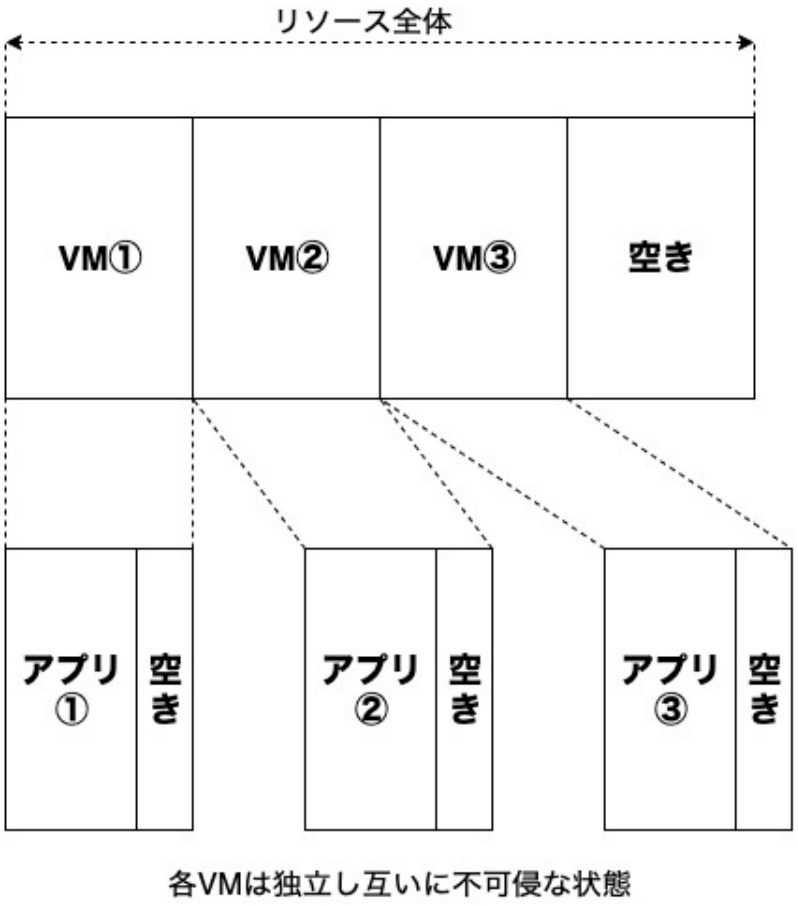

図 2.4: VM の相互独立性

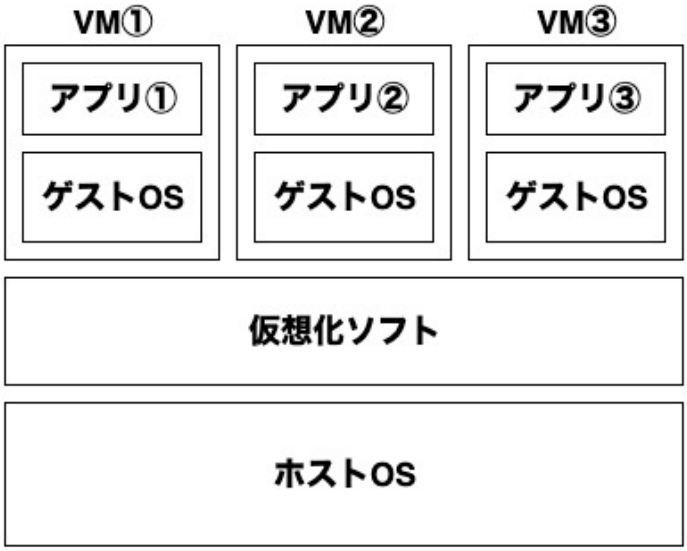

図 2.5: VM の構成

図 2.6: コンテナ型仮想化

Docker

Docker [4] はコンテナ型仮想環境を実現するためのプラットフォームおよびツールである．前述したように Docker では, 宣言的なファイルから生成したコンテナイメージを元にコンテナを起動する．設計書となる宣言的なファイルは Docker ファイルと呼ばれる．Docker ファイルでは, ベースとなるイメージをインポートしたり, 特定のコマンドの実行やファイルのコピーを行うためのコマンドが提供されている．ミドルウェアや各種環境設定をコード化して管理することができ（Infrastructure as Code）, 別の環境で何度実行しても同じ結果が保証される．Docker イメージをバージョン毎に管理するための Docker Hub というサービスがあり, 開発者は自身のレポジトリにイメージをプッシュしたり, 他のレポジトリからイメージを取得することも可能である．

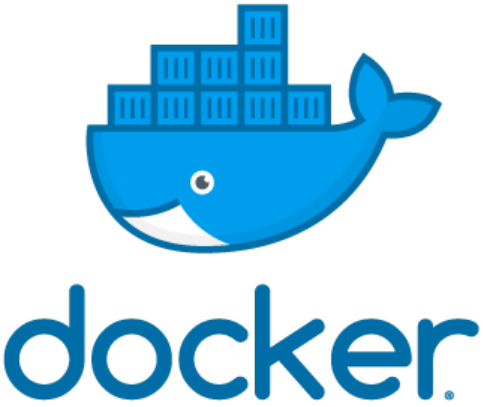

図 2.7: Docker のロゴ

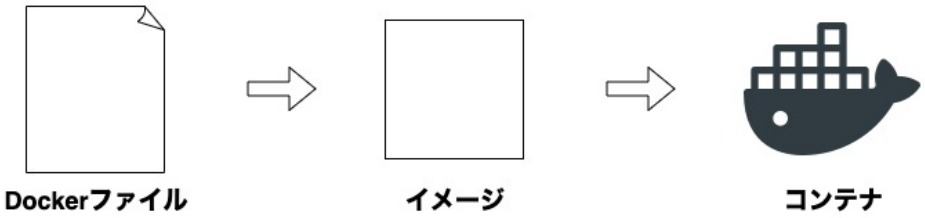

図 2.8: Docker でのコンテナ作成手順

2.4.2 Kubernetes

Kubernetes [5] はコンテナオーケストレーションエンジンであり, コンテナ化されたアプリケーションのデプロイやスケーリングなどの管理を自動化するためのプラットフォームである.

図 2.9: Kubernetes のロゴ

もともと Google 社内で利用されていたコンテナクラスタマネージャの「Borg」を基盤にして作られたオープンソースソフトウェアであるため信頼性が高く, 現時点でコンテナオーケストレーションシステムのデファクトスタンダードとなっている. Kubernetes では, 複数の Kubernetes Node の管理やコンテナのローリングアップデート, オートスケーリング, 死活監視, ログ管理などサービスを本番環境で動かす上で必要不可欠となる機能を備えている. Docker 同様, デプロイするコンテナとその周辺のリソースは YAML 形式や JSON 形式で記述した宣言的なコードによって管理する. Infrastructure as Code に則っているため, 実行環境に左右されず毎回常に同じコンテナが起動される.

GCP を筆頭にクラウド環境でもサポートされるようになり, 現時点で AWS と Azure においても提供されている. そのため Kubernetes は徐々に注目を集めるようになり, 今では多くの企業の本番環境で取り入れられている.

Kubernetes は, 複数のサーバを束ねたクラスタの上で動作する. サーバの役割は二つに分かれており, システム全体を統合管理するサーバをマスターノード（コントロールプレーン）, 実際にコンテナを起動させるサーバをワーカーノードと呼ぶ. マスターノードはシングルでも動作するが, 基本的には冗長性や耐障害性を考慮して複数のマスターノードをクラスタリングすることが多い. クラウド環境を用いた場合, クリックひとつで Kubernetes クラスタを用意することができる. 状況に応じてワーカーノードを追加・削除でき, 自由にスケーリング出来る点も強みである. クラウドの種類によっては特定の条件に合わせて自動でノードのオートスケーリングを行うこともできるが, オンプレ環境では自前で実装する必要がある.

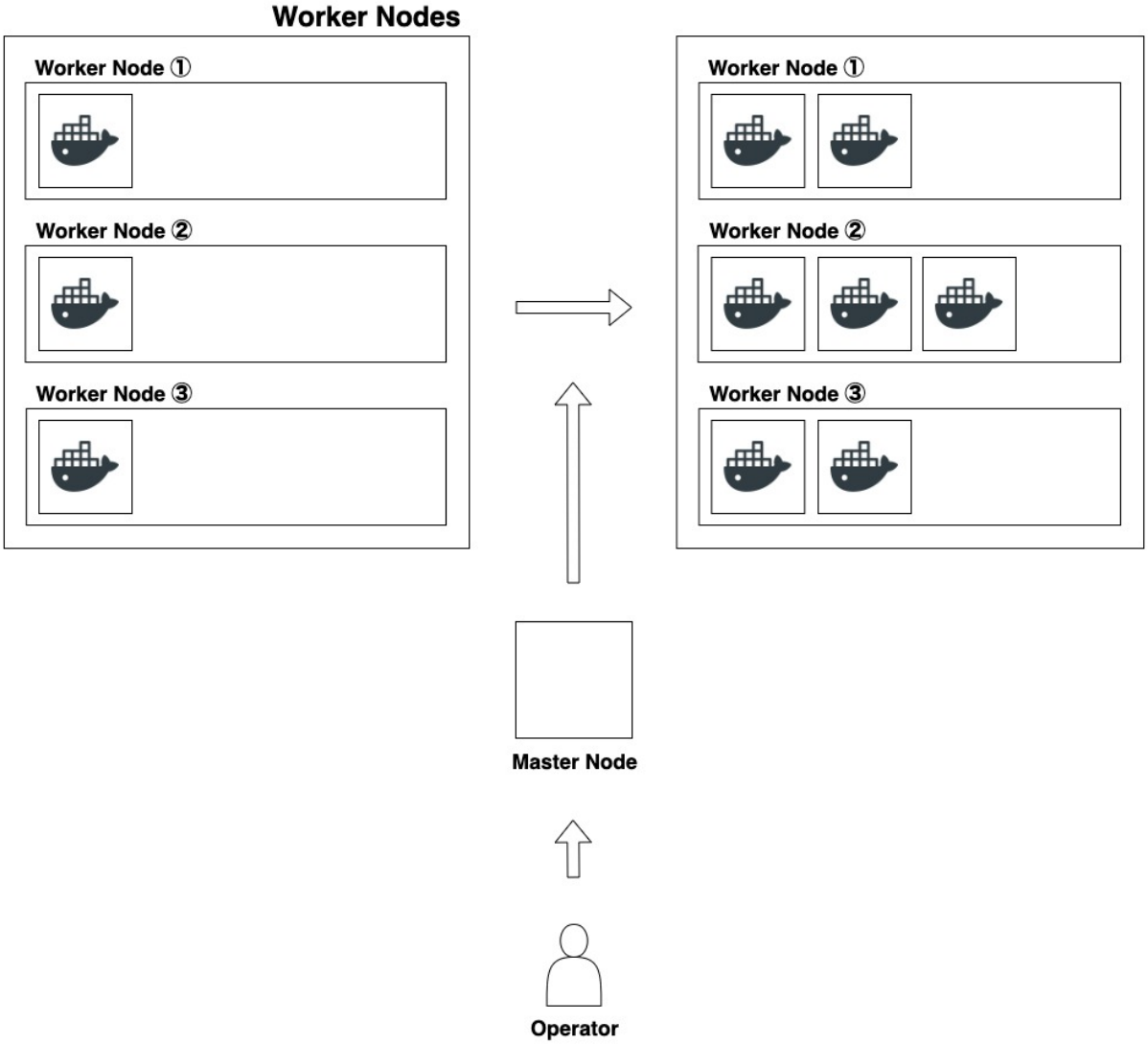

図 2.10: Kubernetes でのコンテナデプロイ

Kubernetes 自体は, 多数のコンポーネントによって構成されるマイクロサービスアーキテクチャを採用している. すべてのコンポーネントが kube-apiserver と呼ばれる Kubernetes

内の API サーバを中心として動作しており, 殆ど全ての処理は kube-apiserver を通して実行される. kube-apiserver はマスターノードに含まれる. 他にもマスターノード内で動作するコンポーネントとしては, Kubernetes クラスタのすべての情報を保持する etcd, コンテナを起動させるノードをスケジューリングする kube-scheduler, ノード上で動作するコンテナを監視し必要に応じてコンテナを追加・削除するよう指示する kube-controller-manager などが挙げられる. 対してワーカーノードで動作する主なコンポーネントには,kubelet などがある. kubelet を含め, 本研究の実装で用いた kubeadm,kubectl に関しては以下で詳細に説明する.

Kubeadm

Kubeadm [6] は,Kubenetes クラスタを構築するためのベストプラクティスを提供するツールである. Kubeadm が提供するコマンドをいくつか以下に示す.

kubeadm init
クラスタの最初のコントロールプレーンとなるノードを起動する.

kubeadm join
クラスタに追加のコントロールプレーンまたはワーカーノードを参加させる.

kubeadm upgrade
クラスタのバージョンを最新へアップグレードする.

kubeadm reset
kubeadm init や kubeadm join によって生じた変更を取り消す.

kubelet

kubelet [7] は,Kubernetes クラスタ内の各ワーカーノードで動作するコンポーネントである. kubectl は,Docker などの CRI と連携して実際にコンテナを起動・停止する役割をもつ. 具体的には etcd の情報を監視して, 自身のノードに割り当てられてまだ起動していないコンテナがあれば起動する. etcd に格納された情報は,kube-apiserver や kube-controller-manager によって kube-apiserver を通して書き換えられ, 実際のコンテナの操作に関しては kubelet が担うといった役割分担がされている. 2.4.2 章の kubeadm, ならびに 2.4.2 章の kubectl は,Kubernetes クラスタ構築時や操作時に用いるコマンドツールであるのに対して, kubelet はコンテナの管理を行うデーモンとして動作する.

kubectl

kubectl [8] は,Kubernetes クラスタをコントロールするためのツールである. 新規コンテナのデプロイや削除, アップデートから, 動作中のコンテナやクラスタを構成するノードの情報の取得など, サービスの運用を支援する API が提供されている. kubectl が提供するコマンドをいくつか以下に示す.

kubectl get nodes
クラスタに参加するノードのステータスやロール（役割）,IP アドレス等を取得する.

kubectl get pods
ポッドの名前やステータス, 再起動の回数等を取得する.

kubectl apply
ポッドに新たな設定を反映させる.

2.5 OpenVPN

本節では, 本研究で使用した VPN 技術ならびにソフトウェア VPN である OpenVPN について概説する.

VPN とは,“Virtual Private Network” の略で, 日本語では “仮想専用線” と呼ばれる. VPN は, インターネット上に擬似的なプライベートネットワークを実現する技術, またはそのネットワーク自体を指す. VPN を使用することで, インターネット上の異なるセグメント同士であっても, あたかも専用線で接続されているかのように通信することが可能である. VPN には L2VPN と L3VPN があり,L2VPN は異なるセグメント同士を接続しひとつの擬似的な LAN を構築するものであり,L3VPN では IP プロトコルでの通信が可能となる. セキュリティ面においては, 通信内容をカプセル化することでパケットの中身の覗き見や改竄のリスクを低減することができる.

2.5.1 OpenVPN

OpenVPN [9] は,OpenVPN Technologies, inc. が中心になって開発しているオープンソースの VPN ソフトウェアである.

OpenVPN は幅広い OS でサポートされており,Window, Linux, Mac OS, iOS, Android で利用可能である. 異なる OS 間でも利用可能であるため, ひとつの VPN ネットワーク内に異なる OS が混在していても正常に動作する. OpenVPN では, 対応した OS のサーバがひとつでもあれば簡単に VPN サーバを構築することが可能である. VPN ネットワークに参加するためには認証が必要であり,OpenVPN では静的鍵による認証や証明書認証,ID/パスワード認証, 二要素認証といった複数の認証方法から任意のものを選択できる. 接続

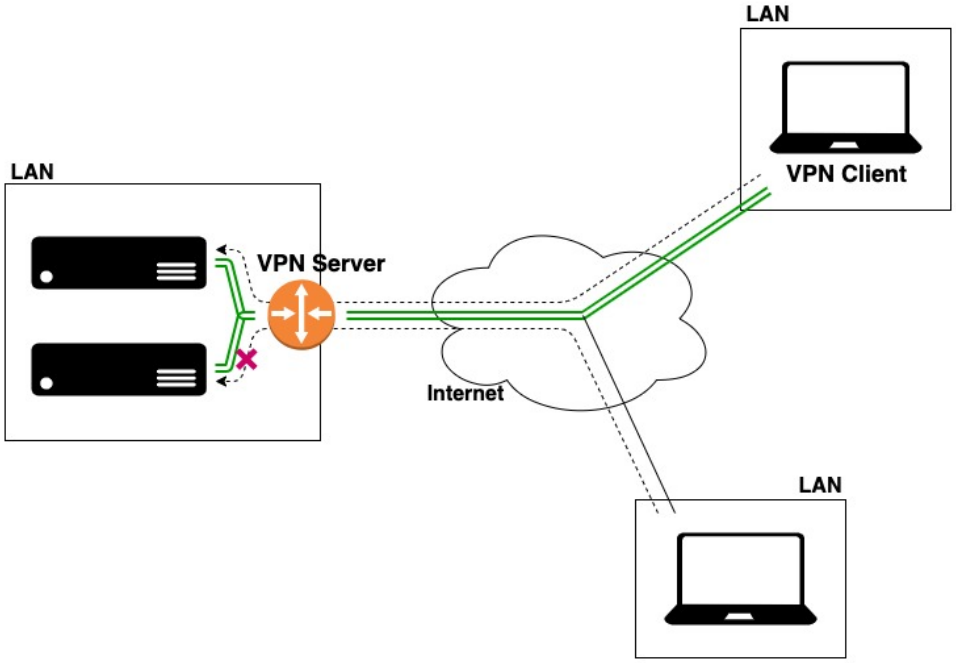

図 2.11: VPN

図 2.12: OpenVPN のロゴ

方法としては, ルーティングとブリッジが提供され, ルーティングは L3VPN, ブリッジは L2VPN に対応する. VPN ネットワーク内でブロードキャストを行いたい場合など, 擬似的な LAN を構築したい場合以外は基本的に L3VPN を用いる. L3VPN にあたるルーティングでは, クライアントサーバ接続とサイト間接続が提供されている. クライアントサーバ接続では, 各クライアントに認証設定が必要となり, 接続の準備としてサーバ側での認証情報の生成が必要となる. 認証情報を共有後, クライアント側では接続のために設定ファイルを用意し, コマンドやアプリケーションを用いて接続を行う. 対するサイト間接続では,VPN の設定は各セグメントの VPN サーバで完結する. VPN サーバ間での接続が確立

できれば, 各拠点に配置されたサーバはお互いに疎通が可能となる. 本研究の提案手法においては, ルーティング形式のサイト間接続を活用した.

第3章　本研究における課題定義と仮説

本章では, 2 章で述べた背景より, 本研究における課題とその要件について議論し, 先行研究および提案システムを概説することで本研究で用いるアプローチについて述べる.

3.1　本研究における課題定義

本研究では, 惑星規模の分散システムのテストのためのステージング環境の構築が困難であり, 未だ整っていないことを課題とする. 2 で述べた背景より, 順を追って説明する.

モノリスや分散システムにおいて, 開発者はサーバの配置や数を任意で決定することができる. そのため, 本番での実運用をする場合にオンプレ環境を用いる場合はステージング環境もオンプレ環境に, クラウド環境を活用する場合は AWS [10] や GCP [11],Azure [12] といったクラウドサービスを利用することで, 比較的簡単にステージング環境の構築を行える. 最近では,AWS の EKS や GCP の GKE, Azure の AKS などクラウドサービス上でフルマネージドな Kubernetes クラスタをクリックひとつで用意することが可能となっている.

対して, 冒頭で述べたように惑星規模の分散システムにおいてはステージング環境の構築が困難である. 惑星規模の分散システムとは, 分散システムの中でも地理的に分散配置されたコンピュータが協調動作することによってシステム全体が動作するものである. さらに, 開発者はシステムに参加するコンピュータの配置を固定化することはできず, システム全体もスケーリングする可能性がある. よって, 惑星規模の分散システムのステージング環境においては, 実際に地理的な場所を指定し分散配置させたサーバが必要となる. サーバを分散配置させた場合, システム全体を統合管理することが困難になる. すべてのサーバに対し一斉にオペレーションを送ることが出来ないため, 各地点のオペレータがアプリのインストールやアップデートといった作業を手作業で行う必要がある. オペレータ同士での作業の確認や手作業を考慮すると, ステージング環境でのテストの開始までに多くの時間が掛かり, 何らかの修正を加える度にこれらの作業を繰り返さなければならない.

そこで本研究では OpenVPN と Kubernetes を活用し, 地理的かつネットワークにおいて論理的に離れたサーバをオーケストレーションすることにより惑星規模の分散システムのためのステージング環境を提案した.

3.2　課題解決における要件

本節では, 課題解決における要件を定義する.

惑星規模の分散システムをテストするためのステージング環境の構築には, サーバの配置に地理的な場所を指定し分散させる必要がある. さらに, テストを円滑に進めるため, 物理的に離れた場所に置かれたサーバに対して統括的な指示を出せることが求められる. そこで, 本研究の課題解決のための要件として, 実際性・統合性・拡張性の三つを挙げた.

3.2.1 実際性

P2P システムの検証は, 実際のネットワーク上で行う必要がある. テスト等の論理的検証では不十分である. P2P システムでは, 状況に応じてノード同士の関係性・役割が変化し, 条件が固定的でないからである. 複雑な条件下での運用が必要であるから, ステージング環境においても, 実際に地理的に分散したノードによるネットワークが求められる.

3.2.2 統合性

ステージング環境においては, ある地点から全てのノードを統合的に操作できる必要がある. 現状, アプリケーションの配布・実行・停止等において多大なコミュニケーションコストとヒューマンリソースのオーバーヘッドが課題となっており, システム内のノードの管理に統合性を持たせることによってこれらのオーバーヘッドを削減する必要がある.

3.2.3 拡張性

ステージング環境では, アプリケーションの修正に伴うアップデートならびにノード数の増加・減少といった変化への柔軟性が必要である. P2P システムは刻一刻と変化するシステムであること. ノードの数によって関係性が変化する. また, ステージング環境では頻繁なアップデートが予想され, その際に生じるオーバーヘッドの削減が必要である.

3.3 先行研究

P2P システムのためのステージング環境の構築手法としては, すでにいくつかの先行研究が存在する.

3.3.1 独自実装のデバッグエージェントによるテスト

既存の提案として, 地理的に分散したノードを統合管理・操作するために別アプリケーションを独自で開発する手法がある. 別アプリケーションとは, 対象アプリケーションに対して命令を送信したり通信内容をログとして抽出するなどのデバッグエージェントして動作する. ノードを統合管理出来る点では要件を満たしており, コミュニケーションならびに工数の削減に繋がると考えられる. しかし対象アプリケーションにパッチを適用した

い場合, 同様にそれを操作するデバッグエージェントにも変更を加える必要があり, 変更への弱さが窺える. アップデートへの柔軟性が不足している限り, それによって生じるオーバーヘッドを削減することが出来ず根本的な解決に繋がらないと思われる. 分散したノードを一斉にコントロールだけでなく, アプリケーションの停止や更新といった変更においてもより少ない手間で抑えられることが求められ, それを満たした際に惑星規模の分散システムの十分なステージング環境が成り立つと考えられる.

3.3.2 PlanetLab

惑星規模のサービスを開発するためのオープンなプラットフォームとして,PlanetLab [13] が挙げられる. PlanetLab は, 新規サービスの開発をサポートするグローバルな研究ネットワークであり,900 以上のノードから構成される. 2003 年から始動し,1000 人以上の研究者が分散ストレージ, ネットワークマッピング,P2P システム, 分散ハッシュテーブルなどの新たな技術の開発のために PlanetLab を利用している. それぞれのノードは仮想マシンを提供しており, ユーザに割り当てられた仮想マシンのセットは Slice と呼ばれている. ユーザは socket API を通じて個別の開発環境を構築することが可能であり,ssh を通して仮想マシンにアクセスしアプリケーションをデプロイできる.

3.3.3 Emulab

分散システムや分散ネットワークを, ネットワークエミュレータによって構築された仮想的なネットワーク上で研究や開発する取り組みとしては, Emulab [14] が挙げられる. Emulab は大規模なソフトウェアシステムであり, 仮想ネットワーク内に点在するマシン同士の接続環境を自由に設定することが可能である. ネットワークエミュレータを活用し, 開発環境で大規模な分散システムのための開発環境を構築する手法 [15] も提案されている. 数台のコンピュター上に数千台の仮想環境をプロセスレベルで構築し, それらをネットワークシミュレータにより相互接続することによって, 擬似的なネットワーク環境においての動作検証を可能にするものである.

3.4 本研究における仮説

本研究では 3.2 章で述べた実際性, 統合性, 拡張性を担保しながら地理的に分散したシステムのためのステージング環境を構築したい. そこで,OpenVPN と Kubernetes を活用することで, それらの要件を満たしたシステムが構築できるのではないだろうかと考えた. それぞれの要件に対して, 本研究で提案するシステムによる実現が可能であると考えられる点を本節では述べる.

3.4.1 実際性

OpenVPN を活用することで, ネットワーク上で論理的に異なるセグメントに位置するノード同士で疎通が可能なオーバーレイネットワークを構築することができる. さらに,IP Reachable な条件下であれば Kubernetes によるクラスタリングが可能である. よって, 実際のネットワーク上にステージング環境を構築することが可能となり,P2P システムの検証における実際性が担保されると考えられる.

3.4.2 統合性

Kubernetes 自体がオーケストレーションシステムであり,Kubernetes クラスタに参加するワーカーノードはマスターノードからの統合管理が可能である. そのため本研究では,P2P システムに参加するノードを Kubernetes クラスタのワーカーノードとして運用することで, マスターノードを経由したアプリケーションの配布や実行が可能となり, 統合性が担保されると考えられる.

3.4.3 拡張性

Kubernetes ではアプリケーションをコンテナとして動かすため, ワーカーノード内でコンテナ数を増減したり, コンテナのアップデートを行える. また, 本研究では Kubernetes クラスタ構築時に kubeadm を使用しており, これを用いることで新たなノードをクラスタに参加させることも可能となる. これによって, 修正が重なる可能性のあるステージング環境に必要な拡張性が担保されると考えられる.

3.5 提案システム概要

提案システムの概要を述べる. ステージング環境において P2P システムに参加するノード同士を,OpenVPN を利用することで相互に疎通可能な状態にする. OpenVPN オーバーレイネットワーク上で Kubernetes クラスタを構築し, 全てのノードをクラスタに参加させる.Kubernetes クラスタ内のマスターノードを介して, 全てのノードに対して操作を行うことができる.

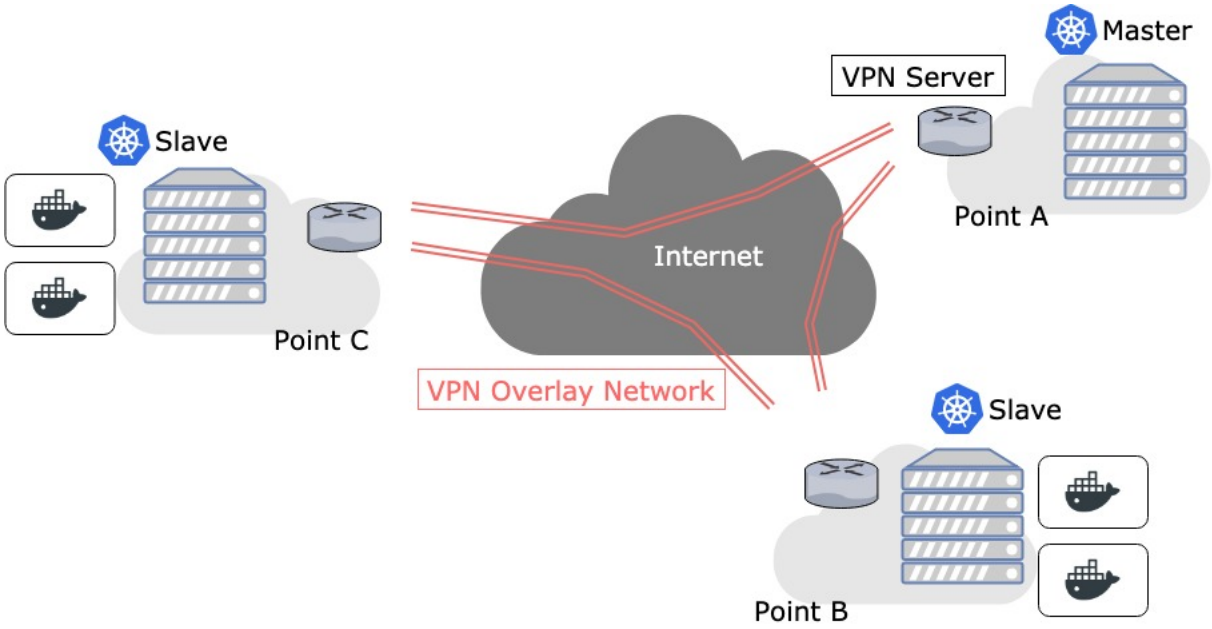
Master
VPN Server
Slave
Point A
Internet
Point C
VPN Overlay Network
Slave
Point B

図 3.1: システム概要図

第4章　実装

本章では提案手法の実装について述べる.

4.1　実装環境

本節では, 本研究で構築した実装環境について概説する.

4.1.1　ハードウェアおよびソフトウェア

本研究で使用したハードウェアおよびソフトウェアとそのバージョンを以下に示す.

表 4.1: 使用したハードウェアおよびソフトウェア

ハードウェア/ソフトウェア	機種/バージョン
Server	FUJITSU PRIMERGY S6（12 CPUs, Memory 48GB）
VMWare ESXi	6.5
VyOS	1.2.1
OpenVPN	2.3.4
Ubuntu	18.04
kubeadm	1.16.3
kubelet	1.16.3
kubectl	1.16.3
HA-Proxy	1.8.8

4.1.2　物理サーバの準備

本研究では, 実装において複数のセグメントおよび Kubernetes クラスタの構築に複数のサーバが必要であったため, それらを仮想的に作成できる VMWare ESXi（以下,ESXi）を導入した. 使用したのは,ESXi6.5 である. ESXi はホスト OS を必要とせず, 直接ハードウェアにインストールさせて動作させるハイパーバイザー型であるため, まず初めに ESXi インストーラのブータブルイメージを USB メモリに書き込み,FUJITSU サーバにインス

トールした. 計二台の FUJITSU サーバに ESXi をインストールし, それぞれ以下の IP アドレスを設定した.

表 4.2: ESXi の IP アドレス

名前	IP アドレス
1 台目	10.4.0.13
2 台目	10.4.0.14

4.1.3 ネットワーク構成

本研究で構築したネットワーク構成について説明する.

まず初めに,ESXi の仮想スイッチと VLAN を用いて二つの ESXi サーバ上に新たに計三つの論理セグメントを構築した. Vlan によって論理的にセグメントを分割することで, お互いに通信不可能な環境とした. 以下に,Vlan ID と対応するアドレスプレフィックスを示す. なお,10.4.0.0/16 のアドレスプレフィックスは Vlan ID 0 に対応している.

表 4.3: Vlan ID と対応するアドレスプレフィックス

Vlan ID	アドレスプレフィックス
0	10.4.0.0/16
10	192.168.10.0/24
20	192.168.20.0/24
30	192.168.30.0/24

4.1.4 VM の配置

ネットワーク構築後,Kubernetes クラスタの構築に必要なサーバを VM として立ち上げた. それぞれの VM の OS には Ubuntu18.04 を採用した. 以下に構築したサーバの詳細を示す.

表 4.4: 設置した VM の詳細

名前	ネットワークインターフェース名	Vlan ID	IP アドレス	役割
lb	ens160	10	192.168.10.253	マスターノードのロードバランサー
master01	ens160	10	192.168.10.101	マスターノード
master02	ens160	10	192.168.10.102	マスターノード
master03	ens160	10	192.168.10.103	マスターノード
node01	ens160	20	192.168.20.101	ワーカーノード
node02	ens160	20	192.168.20.102	ワーカーノード
node03	ens160	30	192.168.30.101	ワーカーノード
node04	ens160	30	192.168.30.102	ワーカーノード

4.1.5 ルーターの配置

次に各拠点に OpenVPN の設定をするルーターを設置した．ルーターの OS には VyOS 1.2.1,OpenVPN はバージョン 2.3.4 を採用した．以下にルーターのネットワーク情報を示す．

表 4.5: 設置したルーターの詳細

名前	ネットワークインターフェース名	Vlan ID	IP アドレス
vyos01	eth0	0	10.4.0.90
vyos01	eth1	10	192.168.10.1
vyos02	eth0	0	10.4.0.91
vyos02	eth1	20	192.168.20.1
vyos03	eth0	0	10.4.0.92
vyos03	eth1	30	192.168.30.1

全てのルーターはお互いに疎通可能である．さらに, 各拠点に設置されたサーバと疎通できるよう eth1 のネットワークインターフェースには別の IP アドレスを設定した．この時点での各サーバの疎通性は以下の通りである．

表 4.6: OpenVPN 設定前の各サーバの疎通性

	lb	master01	master02	master03	node01	node02	node03	node04
lb		○	○	○	×	×	×	×
master01	○		○	○	×	×	×	×
master02	○	○		○	×	×	×	×
master03	○	○	○		×	×	×	×
node01	×	×	×	×		○	×	×
node02	×	×	×	×	○		×	×
node03	×	×	×	×	×	×		○
node04	×	×	×	×	×	×	○	

4.1.6 OpenVPN の設定

4.1.5 で示したように,OpenVPN の設定をする前ではすべてのサーバはお互いに疎通可能な状態にはない．Kubernetes は, クラスタに参加するサーバのすべてが疎通可能, 厳密には IP reachable な環境下にある必要がある．そこで OpenVPN を用いて, 複数の分離した LAN を仮想的に接続し Kubernetes の要件を満たそうと試みた．本実装では,OpenVPN の site-to-site モードを採用した．client-server モードを採用しなかった理由としては以下の二点が挙げられる．

1. Kubernetes は通信時にデフォルトゲートウェイに設定したネットワークインターフェースを使用するため, サーバ毎に OpenVPN を設定する client-server モードではトンネルインターフェースを通して通信ができない.

2. サーバ毎に証明書と鍵の管理が必要なため扱いづらい.

対して,site-to-site モードでは以下の利点が挙げられる.

1. ルーティングはルーターに任せられるため, サーバは通信時にデフォルトゲートウェイに設定されたネットワークインターフェースを使用できる.

2. OpenVPN の設定は LAN 内のルーターのみ.

以下に,OpenVPN 設定後の各サーバの疎通性を示す.

表 4.7: OpenVPN 設定前の各サーバの疎通性

	lb	master01	master02	master03	node01	node02	node03	node04
lb		○	○	○	○	○	○	○
master01	○		○	○	○	○	○	○
master02	○	○		○	○	○	○	○
master03	○	○	○		○	○	○	○
node01	○	○	○	○		○	○	○
node02	○	○	○	○	○		○	○
node03	○	○	○	○	○	○		○
node04	○	○	○	○	○	○	○	

4.1.7 Kubernetes クラスタの構築

OpenVPN による拠点間の接続を行った後,Kubernetes クラスタを構築した. 本研究の実装では,Kubeadm を使用した高可用性 Kubernetes クラスタを構築するため, まず初めに複数マスターへのリクエストを振り分けるロードバランサーを設置した. ロードバランサーの構築には,HA-Proxy 1.8.8 を採用した.

```
1   frontend kubernetes
2       bind *:6443
3       option tcplog
4       mode tcp
5       default_backend kubernetes-backend
6
7   frontend etcd
8       bind *:2379
```

```
9         option tcplog
10        mode tcp
11        default_backend etcd-backend
12
13    backend kubernetes-backend
14        mode tcp
15        balance roundrobin
16        option tcp-check
17        server master01 192.168.10.101:6443 check
18        server master02 192.168.10.102:6443 check
19        server master02 192.168.10.103:6443 check
20
21    backend etcd-backend
22        mode tcp
23        balance roundrobin
24        server master01  192.168.10.101:2379 check
25        server master02  192.168.10.102:2379 check
26        server master03  192.168.10.103:2379 check
```

上記の設定で, ロードバランサーのポート 6443 番とポート 2379 番へのリクエスを三台のマスターノードへと振り分けている.

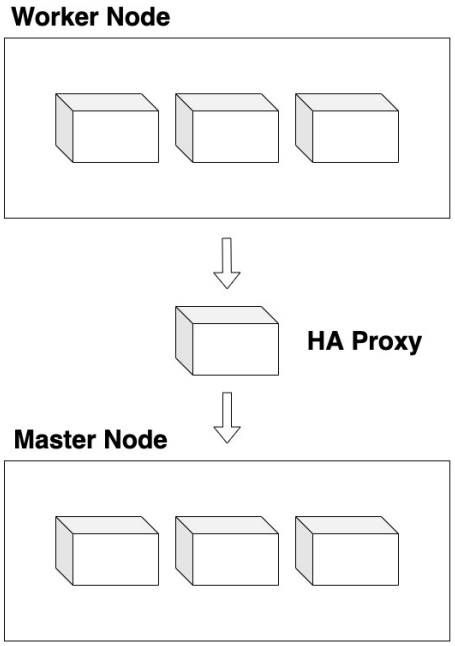

図 4.1: マスターノードとワーカーノードの関係性

次に, マスターノードとワーカーノードを立ち上げるにあたり必要なパッケージをインストールした. Kuberntes のランタイムとして使用する Docker に加え, クラスタ構築時に用いる kubeadm と kubelet, クラスタ操作時に必要な kubectl を apt によって取得した. パッケージの用意が完了したのち, マスターノードからクラスタ構築作業を行った. kubeadm ではクラスタの初期化用に init コマンドが用意されており, 初めのマスターノードにて実行することでクラスタの基盤を作成可能である. 初期化に成功した場合, 以下のようなテ

キストが出力される.

```
1  You can now join any number of control-plane nodes by copying
       certificate authorities
2  and service account keys on each node and then running the
     following as root:
3
4    kubeadm join 192.168.10.253:6443 --token { token } \
5      --discovery-token-ca-cert-hash sha256:{ hash }} \
6      --control-plane
7
8  Then you can join any number of worker nodes by running the
     following on each as root:
9
10 kubeadm join 192.168.10.253:6443 --token { token } \
11     --discovery-token-ca-cert-hash sha256:{ hash }}
```

出力にある通り, 与えられたコマンドを他のマスターノードとワーカーノードから実行することでクラスタへの参加が行える. 各サーバにて上記のコマンドを実行した結果, マスターノードからクラスタが構築できていることを確認できた.

```
1  $ kubectl get nodes
2  NAME       STATUS     ROLES    AGE     VERSION
3  master01   Ready      master   58d     v1.16.3
4  master02   Ready      master   58d     v1.16.3
5  master03   Ready      master   58d     v1.16.3
6  node01     Ready      <none>   8d      v1.16.3
7  node02     Ready      <none>   4d22h   v1.16.3
8  node03     Ready      <none>   6d16h   v1.16.3
9  node04     Ready      <none>   8d      v1.16.3
```

4.2 システム全体

本研究で構築した実装環境の図を以下に示す.

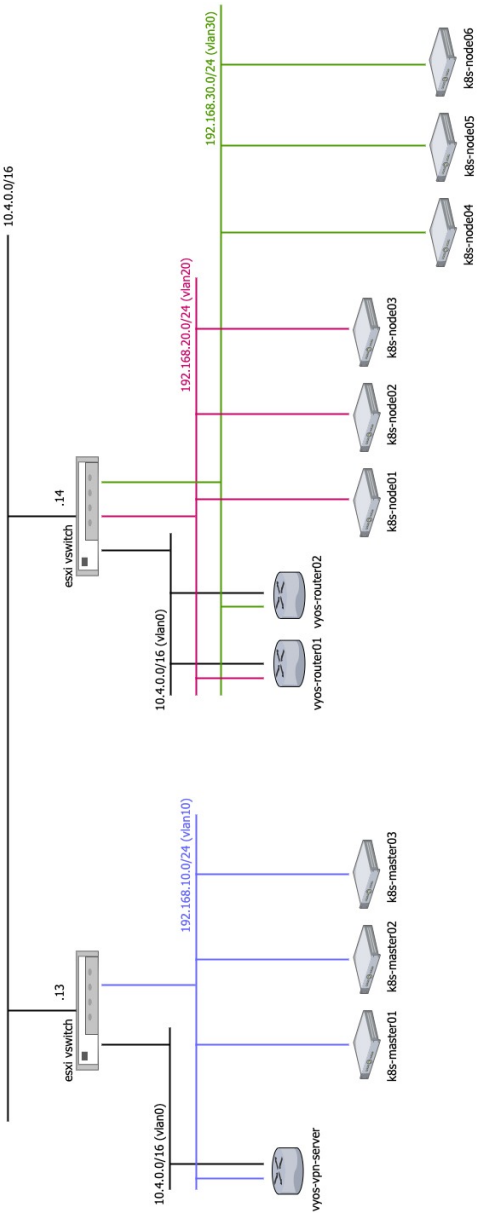

図 4.2: ネットワーク構成図

第5章　評価

本章では, 本研究の提案が 3.2 で述べた問題解決における要件を満たしているか評価を行う.

5.1　実際性

実際性の評価をするため以下二点が実現されているか確認した.

1. 異なる LAN 内に配置されたサーバ同士がネットワーク上で疎通できているか
2. 複数の論理セグメントに跨って Kubernetes クラスタを構築できているか

一点目は, あるサーバから異なるサーバに対する ping コマンドを用いて疎通性を確認した. 以下は,node01（192.168.20.101）から master01（192.168.10.101）に対して ping コマンドを使用した際の出力である.

```
$ ping 192.168.10.101
PING 192.168.10.101 (192.168.10.101) 56(84) bytes of data.
64 bytes from 192.168.10.101: icmp_seq=1 ttl=63 time=1.13 ms
64 bytes from 192.168.10.101: icmp_seq=2 ttl=63 time=1.50 ms
64 bytes from 192.168.10.101: icmp_seq=3 ttl=63 time=1.33 ms
64 bytes from 192.168.10.101: icmp_seq=4 ttl=63 time=1.03 ms
64 bytes from 192.168.10.101: icmp_seq=5 ttl=63 time=1.58 ms

--- 192.168.10.101 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4006
   ms
rtt min/avg/max/mdev = 1.037/1.319/1.584/0.211 ms
```

二点目は,kubectl コマンドにてクラスタを構成するノードの IP アドレスを確認し, それらが別々のセグメントに位置することを確認した.

```
$ kubectl get nodes -owide
NAME       STATUS     ROLES    AGE     VERSION   INTERNAL-IP
        EXTERNAL-IP   OS-IMAGE             KERNEL-VERSION
        CONTAINER-RUNTIME
master01   Ready      master   58d     v1.16.3
   192.168.10.101   <none>        Ubuntu 18.04.3 LTS
   4.15.0-70-generic   docker://18.9.7
```

```
4   master02   Ready      master   58d     v1.16.3
       192.168.10.102   <none>        Ubuntu 18.04.3 LTS
       4.15.0-70-generic   docker://18.9.7
5   master03   Ready      master   58d     v1.16.3
       192.168.10.103   <none>        Ubuntu 18.04.3 LTS
       4.15.0-70-generic   docker://18.9.7
6   node01     Ready      <none>   8d      v1.16.3
       192.168.20.101   <none>        Ubuntu 18.04.3 LTS
       4.15.0-74-generic   docker://18.9.7
7   node02     Ready      <none>   4d22h   v1.16.3
       192.168.20.102   <none>        Ubuntu 18.04.3 LTS
       4.15.0-74-generic   docker://18.9.7
8   node03     Ready      <none>   6d16h   v1.16.3
       192.168.30.101   <none>        Ubuntu 18.04.3 LTS
       4.15.0-74-generic   docker://18.9.7
9   node04     Ready      <none>   8d      v1.16.3
       192.168.30.102   <none>        Ubuntu 18.04.3 LTS
       4.15.0-74-generic   docker://18.9.7
```

以上の結果より, 論理的に隔離された LAN に跨って Kubernetes クラスタが構築可能であることを示した. よって, 惑星規模の分散システムのためのステージング環境を実際のインターネット上に構築することが可能であることが言える.

5.2 統合性

以下二点を明らかにすることで, 統合性の評価を行う.

1. 特定のノードからステージング環境に属する全てのノードに対して一斉に指示を送ることができるか
2. 本研究の提案手法を用いず従来の手作業を含む手法を選んだ場合, 工数にどのような差が生じるか

一点目は,kubectl コマンドを用いてステージング環境のワーカーノードに対して同時にアプリケーションをデプロイすることができたかを確認した.

```
1   $ kubectl create deployment --image nginx hello-world
2   $ kubectl get pods -owide
3   NAME                           READY   STATUS    RESTARTS
       AGE     IP           NODE       NOMINATED NODE   READINESS
       GATES
4   hello-world-c6c6778b4-5n74d   1/1     Running   0          4
       d22h   10.44.0.1   node01     <none>           <none>
5   hello-world-c6c6778b4-6mrj4   1/1     Running   0          4
       d22h   10.42.0.1   node03     <none>           <none>
6   hello-world-c6c6778b4-fmnxt   1/1     Running   0          4
       d22h   10.47.0.1   node02     <none>           <none>
7   hello-world-c6c6778b4-r8b5w   1/1     Running   0          4
       d22h   10.44.0.2   node04    <none>           <none>
```

二点目は, まず従来の手法を用いた場合の作業工程を列挙し, 提案手法との違いを定性的に評価した. ここでは複数の大学によって構成される研究ネットワークにて, 新たに開発した惑星規模の分散システムをデプロイするケースを考える.

1. 各大学のリソース（OS, CPU, Memory）の共有
2. 各大学における作業内容の確定・共有
3. それぞれの大学のサーバ管理者とのスケジューリングの調整
4. 各大学でのデプロイ作業
5. 各大学からの作業完了の連絡
6. 全大学での作業完了の共有
7. ステージング環境の使用開始

上に列挙したように本研究の提案手法を用いない場合, ステージング環境で何かしらの変更を行う度に開発者間での多くのコミュニケーションと手作業が生じる. すべての作業が単独で並行に行われれば, 作業中のミスによる中断やコミュニケーション不足による手戻りが発生する可能性もある. 対して本研究の提案手法では, ステージング環境が統合的に管理されており, 一括ですべてのワーカーノードに対して処理を実行できる. 例えば, パッチを当てた修正版のアプリケーションを新たにデプロイする場合, 作業は一コマンドで完結し. すべての処理は自動化されているため冪等性が担保される.

5.3 拡張性

拡張性の評価において, 新規ノード追加時の必要時間を計測した. 計測では, 必要なパッケージのインストールに掛かる時間とクラスタへの参加時間の二つを対象とした. パッケージのインストールは Ansible を用いて自動化し,apt update から docker, kubeadm 等のインストール, リブートまでを含んでいる. クラスタへの参加時間は,kubeadm join に要した時間と, マスターノードでノードの参加を確認しステータスが Ready になるまでの時間を加算したものである.

表 5.1: 新規ノード追加時の必要時間

内容	経過時間
パッケージのインストール	509.50s
クラスタへの参加	105.86s

以上の結果から, 新規ノードの追加に要する時間はパッケージのインストールからクラスタへの追加まで 10 分程度で行えることが確認できた. 短い時間でステージング環境のスケーリングを行えたことを持って, 拡張性を担保できていると考えた.

第6章　結論

本章では, 本研究のまとめと今後の課題を示す.

6.1　本研究のまとめ

本研究では, 惑星規模の分散システムのテストを行うためのステージング環境における各地点のオペレータ間のコミュニケーションや手作業によって生じるオーバーヘッドを解決するため, OpenVPN と Kubernetes を利用したステージング環境の提案をした. 着目した課題に対する解決策として実際性, 統合性, 拡張性の三つの要件が求められると考えた. 第一に,OpenVPN を用いることでネットワーク上で論理的に離れたノード間での疎通性を獲得した. これによって対象のノードを開発環境から実際のインターネット上に拡張することができ, 惑星規模の分散システムのテストに必要である実際性を満たすことが出来たと考える. 第二に拡張性については,OpenVPN によるオーバーレイネットワーク上で Kubernetes クラスタを構築することで, 分散したコンピュータに対し統合的な操作を可能にすることで解決した. 第三に拡張性であるが,Kubernetes クラスタ上ではアプリケーションをコンテナ型仮想マシンとして動作させるため, 容易にコンテナの追加や削除を行うことが可能である. 加えて,kubeadm によりクラスタへ新規にノードを追加することも可能であるため, ステージング環境を自由に拡張することが可能である. よって拡張性も満たしていると考えられる.

6.2　本研究の課題と展望

本研究では,OpenVPN を用いたオーバーレイネットワーク上に Kubernetes クラスタを構築した. すべてのコンピュータは VPN サーバと接続し,Kubernetes クラスタ上での通信はすべて VPN サーバを通して行う. 本研究では VPN サーバの負荷とそれに伴う Kubernetes クラスタへの影響までを測定することができなかった. 実用に向けた次のステップとしては,VPN サーバの負荷とレイテンシについての詳細な実験をする必要性があると考えた.

謝辞

本論文の執筆にあたり, 常に優しく, 最後まで見捨てずにご指導してくださった慶應義塾大学政策・メディア研究科特任准教授鈴木茂哉博士, 同大学政策・メディア研究科博士課程阿部涼介氏に感謝致します. お忙しいにも関わらず, 毎週のようにミーティングを設けてくださったこと, 研究について一から教えてくださったこと, 行き詰まっている際に親身に相談に乗ってくださったことには本当に感謝しております.

参考文献

[1] 金子 勇. Winny の技術, 2005.

[2] Gtk-gnutella. http://gtk-gnutella.sourceforge.net/en/?page=news.

[3] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. http://www.cryptovest.co.uk/resources/Bitcoin%20paper%20Original.pdf, 2008.

[4] Docker. https://www.docker.com/.

[5] Kubernetes. https://kubernetes.io/ja/.

[6] Kubeadm. https://github.com/kubernetes/kubeadm.

[7] kubelet. https://github.com/kubernetes/kubelet.

[8] kubectl. https://github.com/kubernetes/kubectl.

[9] Openvpn. https://openvpn.net/.

[10] Amazon web services. https://aws.amazon.com/jp/.

[11] Google cloud platform. https://cloud.google.com/?hl=ja.

[12] Microsoft azure. https://azure.microsoft.com/ja-jp/.

[13] Planetlab. https://www.planet-lab.org/.

[14] Emulab. https://www.emulab.net/portal/frontpage.php.

[15] 米澤 明憲 西川 賀樹, 大山 恵弘. プロセスレベルの仮想化を用いた大規模分散システムテストベッド. https://ipsj.ixsq.nii.ac.jp/ej/?action=repository_action_common_download&item_id=18170&item_no=1&attribute_id=1&file_no=1, 2008.