卒業論文 2019 年度 (令和元年)

地理的に分散したシステムのためのステージング環境の設計と構築

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)

地理的に分散したシステムのためのステージング環境の設計と構築

本研究では，地理的に分散したシステムの検証実験における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を解決するためのシステムを提案する．ブロックチェーンの基盤技術として知られる P2P ネットワークなどは，地理的に分散したノードによって形成されており，これらはクライアント・サーバ方式のソフトウェアに比べテストが行いづらい．何故なら，実際に離れた地点に設置されたノードを用いてテスト用のステージング環境を構築するためには，多くの人手とその間でのコミュニケーションが必要になるからである．

そこで，OpenVPN と Kubernetes を利用することで地理的かつネットワーク上で論理的に離れたノードを統合管理できるステージング環境を構築する．特定のポイントから一斉にすべてのノードに対しての操作を行うことで，デプロイ作業やアップデート作業におけるコミュニケーションコストとヒューマンリソースを解決することが可能だと考えた．

本システムの実装後，実際にネットワーク上で別セグメントに点在するノードに対して特定のポイントから一斉にソフトウェアを起動，更新，停止できることを確認して，本研究での提案が実現可能であることを証明する．

評価として，BSafe.network にて本システムを使用した場合と使用しなかった場合で，作業工程数にどれほどの差が生じるかを検証した．結果，本システムを使用した場合には大きく工程数を削減することができ，本研究で課題とした地理的に分散したシステムの検証実験における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を解消可能にした．

よって本研究は，地理的に分散したシステムの開発における検証作業の効率性を促進し，システムの堅牢性の向上に役立つと考える．

キーワード:
1. 地理的に分散したシステム, 2. ステージング環境, 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 staging environment for geographically distributed system

The porpose of this study is proposing the system to solve an overhead of communication costs and human resources in a staging environment of geographically distributed systems. The test for geographically distributed system such as P2P network, which is known as the basic technology of blockchain, is harder than client-server software. This is because it needs many communications and human resources to build a test staging environment consisted by distributed nodes that are located at remote points.

Therefore, we propose the integrated staging environment using OpenVPN and Kubernetes. The system can oprate all distributed nodes from a specific point at the same time, it is possible to solve the overhead of communication costs and human resources.

After we implemented this system, confirm that it can deploy and update, stop the application on all distributed nodes. It proves that the proposal in this study is feasible.

As an evaluation, we verified how much difference in the number of work processes occurred when BSafe.network did not use this system and when it was used. As a result, when this system is used, the number of processes can be greatly reduced. Communication costs and human resource overhead can be eliminated.

Therefore, this study will promote the efficiency of test work in the development of geographically distributed systems, and will help to improve the robustness of the system.

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

本章では本研究の背景，解決すべき課題および手法を提示し，本研究の概要を示す．

1.1　地理的に分散したシステムの発達

本節では，本研究で着目する地理的に分散したシステムの定義と概要，またその発展について説明する．

1.1.1　分散システム

本研究における地理的に分散したシステムとは，分散システムのうちブロックチェーン [1] のような世界中に地理的に分散したノードがお互いに協調動作することによって成り立つシステムを指す．言い換えれば，地理的に分散したノードによって形成される分散システムである．

クライアント・サーバ方式では，システムに参加する計算機の役割が明確に分かれており，通信において常にクライアント対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てリクエストを送り，リクエストを受け取ったサーバは特定の処理に基づいてクライアントに対しレスポンスを返す．

対してブロックチェーンのような P2P 方式のシステムでは，参加する計算機の役割は状況に応じて柔軟に変化する．時にクライアントとして他のノードに対して要求し，時に他のノードからの要求に対して応答する場合もある．クライアント・サーバ方式に比べて，耐障害性・冗長性・可用性において優れているのが特徴的である．

1.1.2　分散システムの発達

2000 年代初頭，Winny [2] や Gnutella といった P2P アプリケーションが頭角を現した．それまでサービスの構成として一般的であったクライアント・サーバ方式とは異なり，それぞれのノードが対等な関係をもつ P2P アプリケーションに対し注目が集まったが，クライアント・サーバ方式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 アプリケーションが脚光を浴びるようになり，開発や研究の勢いが盛んになってきている．

1.2 本研究の着目する課題と目的

本節では，本研究で着目しているステージング環境の説明とその必要性，ならびに地理的に分散したシステムのステージング環境を構築する際の課題について説明し，最後に目的を明確化する．

1.2.1 ステージング環境

ステージング環境とは，本番環境での運用をする前に実際の環境を想定してシステムの動作確認を行うための環境である．開発者が実際に開発を行うローカル環境と実運用する本番環境では，環境の差異から動作の違いが生じ，手元で正常に動作していたものが本番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ローカル環境と本番環境の間に，本番環境を想定したステージング環境を構築し，本番環境へのデプロイ前にステージング環境にて動作を確認することで予想外の障害が発生するリスクを抑えることができる．

1.2.2 ステージング環境の必要性

地理的に分散したシステムにも本番環境での予想外の障害に備えたステージング環境が必要である．加えて，地理的に分散した P2P アプリケーションでは，インターネット上で動作させた際のシステムへの影響や，参加するノードが増加した際の協調動作の正常性を検証する必要があると考えられる．

1.2.3 課題

本研究では，地理的に分散したシステムのためのステージング環境の構築と運用において解決すべき課題があると考えた．

通常，ステージング環境では必要となるサーバの設置，アプリケーションのデプロイ，修正を含んだパッチの適用などの多くの変更が行われる．AWS [3] や GCP [4],Azure [5] と行ったクラウドサービスや開発支援ツールが整った昨今，これらの殆どが自動化され開発者はサービス開発のみに集中できるようになった．

しかし，地理的に分散したシステムにおいてはこれらの恩恵を得られていない．クラウドサービスでは固定のゾーンが存在するためサーバの地理的な位置を自由に決定することができず，遠距離間でサーバを一斉にコントロールすることも難しい．結果的に，地理的に分散したシステムのためのステージング環境の構築と運用では，各地点のサーバ管理者による情報共有と個別の作業が必要になる．

つまり，各地点でのサーバの設置から不定期で頻繁に発生するアップデート作業まで全てにおいて，コミュニケーションコストの増大，作業時間と労力の消費，人為的ミス等の不安要素の発生といった課題点が予想される．

1.2.4 目的

本研究では，地理的に分散したシステムのためのステージング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

1.2.3 で述べた課題を解決するためには，地理的に分散したサーバを統合管理することで，ステージング環境での変更に対し柔軟かつ迅速に対応する必要があると考えた．

地理的に分散したサーバを統合管理するためには，

- ステージング環境に含まれるサーバ同士が，お互いに通信可能な状態であること
- ある特定の地点から全てのサーバに対して操作が可能であること

の二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 1.2.3 で述べた課題点を解決し， 1.2.4 で述べた地理的に分散したシステムのためのステージング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 で述べた必要要件を満たすため，OpenVPN と Kubernetes を用いる．

まず，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，クラスタ化には各サーバがお互いに IP レベルで疎通可能な状態になければならない．よって，Kubernetes は単一のデータセンター内での使用に適している一方，複数のデータセンターを跨がった構成での使用には適していない．

そこで，地理的に分散したサーバ間を結ぶ OpenVPN オーバーレイネットワークを構築することで，各サーバがお互いに IP Reachable な状態にする．OpenVPN とは，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，地理的に分散したノード間で形成した 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．別々のセグメントに位置するノードを用いて Kubernetes クラスタを構築し，コンテナ化したアプリケーションをクラスタ上にデプロイできていることを確認し，本システムの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，地理的に分散したシステムとその実験的運用方法およびそれに伴う課題点について議論し，本研究の背景を明確化する．3 章では，本研究で着目する問題を解決するための要件，仮説と手法について説明する．4 章では，**??**章で述べた提案手法について述べる．5 章では，2 章で求められた課題に対しての評価を行い，考察する．**??**章では，5 章で導き出された結果と関連研究から本研究の妥当性を考察する．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と関連する技術について概説する.

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の定義を明らかにする。定義を分かりやすくするため、先にモノリスと分散システムについて概説し、違いを明らかにした上で惑星規模の分散システムを定義する。

2.1.1　モノリス

モノリスとは、モノシリックなシステムを指す。モノリスは英語で一枚岩を意味し、システムにおいては、大きな単一のプログラムによって特定の処理を実行するアーキテクチャといえる。このように、本来では大量のコードによって成り立つという意味を含むが、比較化のため本研究では、プログラムの大きさに関わらず単一のコンポーネントで構成されたシステムをモノリスと呼ぶことにした。単一のコンポーネントで構成されるシステムはアーキテクチャがシンプルなため、システム自体が小さい場合は取り扱いやすい。しかし、システムが大きくなるにつれて機能同士の依存関係が密な状態になることで細かい粒度でテストを行えなかったり、チーム開発における並行作業が困難になる傾向にある。

2.1.2　分散システム

分散システムとは、マイクロサービスアーキテクチャなシステムを指す。システム全体が複数の独立したコンポーネントを組み合わせて成り立っている点において、モノリスとは対照的である。各コンポーネントは異なる役割を持っており、お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コンポーネント毎に分割した分散システムである。Kubernetes に関する説明は 2.5.2 章で詳しく行う。分散システムは複雑で取り扱いづらいように思われるが、実際には機能が細分化され各コンポーネントの役割が明確になる。機能同士の依存関係が希薄になるため細かい粒度でのテストが可能となり、障害時の原因特定も容易になる。チーム開発において開発者同士で同じ箇所を担当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。

2.1.3 惑星規模の分散システム

本研究で対象とする惑星規模の分散システムとは、 **??**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にあたる。これらのシステムに使用されている技術や、サービスについての概説は 2.2 にて行う。惑星規模の分散システムは、システムを構成するコンピュータの場所を開発者が固定できないことと、システム全体がスケーリングしていく点で 2.1.2 章の分散システムとは異なる。開発者は、コンピュータの位置やスケーリングした際の挙動を考慮した上で開発を行わなければならない。よって、システムの挙動を左右する条件が 2.1.1 章のモノリスや 2.1.2 章の分散システムに比べて多くテストが難しい。

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、 2.1.3 章で概説した惑星規模の分散システムの主な使用技術とサービスの参考例について概説する。使用技術としては、P2P が一番にあげられる。クライアントサーバモデルとは異なるシステムモデルであり、最近ではブロックチェーン技術にも取り入れられている。サービスの参考例では、Winny や Gnutella、Bitcoin について触れる。Bitcoin は、先述したブロックチェーン技術を用いて動作するシステムであり、仮想通貨として広く世間に認知されている。

2.2.1 P2P

P2P は “Peer to Peer” の略である。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るシステムモデルである。

クライアントサーバモデルでは，クライアントがリクエストを投げサーバがレスポンスを返すという明確な役割分担がある．サーバはクライアントからのリクエストを待ち，リクエストが来たときのみ必要な処理を行ってクライアントへレスポンスを返す．対してクライアントは、サーバに問い合わせる必要がないときは何もせず，データを要求したり変更する必要が生じたときのみサーバとの通信を行う．よって通信は常にクライアントが起点となり，基本的にサーバ起点の通信は行われない．

対象的に，P2P では各コンピュータが対等な関係性を持つため，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．各コンピュータは臨機応変にサーバとしてレスポンスを返し，クライアントとしてリクエストを投げ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，リクエストを発信する側をクライアント，それに対してレスポンスを返す側をサーバと呼んでいる。対して 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くことからサーバントと呼ばれる．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．

まず第一に，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、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タイプとしてシステムに参加することも可能），データを相互で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．また，ファイル共有システムである Winny では，各コンピュータが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サーバを決定してから通信を行うといった処理が必要となる．

次に，システムを動かす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，データ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。

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．

まず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 ではどこかのコンピュータが停止したとしても，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問題なくサービス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固定ではないため，状況に応じて接続先を変更する必要がある．すなわち，目的のデータ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かすため，中央集権的なサーバと異なり，管理は各々のコンピュータ保持者に委ねられることになる．よって，たとえシステムに問題点が見つかりアプリケーション開発者がパッチを含んだアップデートバージョンを配布した場合でも，実際に動かしているアプリケーションがアップデート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 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，ノード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．ユーザはノード内に保持されたファイルを他のノード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，逆に他のノード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．

2.2.3 Gnutella

Winny に同じく Gnutella も中央集権型サーバに依存せず，P2P ネットワーク上のノード間の通信のみでファイルを送受信を行うファイル共有アプリケーションである．

2.2.4 Bitcoin

Bitcoin [1] 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．参加するノード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 ステージング環境

本節では、本研究で着目するステージング環境について概説する。

ステージング環境とは、システムのテスト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ステージング環境として構築し、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。ステージング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。対してステージング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ステージング環境を挟むことで、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テスト方法ならびに必要となるステージング環境について、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。テスト方法は、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、テスト対象もひとつに限られる。ステージング環境の構築時には、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、 2.3.1 章の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。分散システムのテストを行う場合、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。しかしモノリス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テストでは不十分である。一方、ステージング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

2.4 地理的に分散したシステムとステージング環境での動作確認

本節では，地理的に分散したシステムのステージング環境と動作確認について概説する．

2.4.1 最小限の動作確認

最も簡単に行える動作確認は，ふたつのノード間で行うテストである．ネットワーク上の二点でそれぞれノードを立ち上げ，システムの機能が正しく動作するかを確認する．クライアントサーバモデルでは，最低限ではあるが機能の保証ができる．クライアントサーバモデルでは，中央集権的サーバとクライアントが一対一の関係で繋がっており，開発者はクライアントとの通信ただひとつに注力すればいいからだ．ユーザが増加した場合の障害対策やレスポンスタイムの向上は確かに必要であるが，サーバとクライアントの一対一の関係性は不変であるため，ネットワーク自体が正常で有る限り問題は二点間に閉ざされておりテストがしやすい．

一方，中央集権的サーバがなくノード同士がサーバにもクライアントにもなり得る地理的に分散したシステムでは，この方法は十分ではない．ネットワークに参加するノードが増加すれば個々のノード同士の関係性は変化し，関係性が固定されないためである．もうひとつの理由として，ノード周辺のネットワーク環境によって動作に影響が出る可能性が考えられる．地理的に分散したシステムの具体例として挙げた Bitcoin では，参加するノードは全て同じデータを保持する．データの送信や受信において遅延が発生すれば何らかの影響が出ることは簡単に予想可能である．例えシステム上でデータの不整合を防ぐロジックが組まれていたとしても，ロジックを表現したコードが実際の環境で正常な動作をすることを動作確認無しで担保することは難しい．以上の理由から，地理的に分散したシステムの動作確認をするにあたって二点間でのステージング環境は不十分であり，より多

くのノードを実際の世界規模のネットワーク上で動かしたステージング環境が必要であると考えられる．

2.4.2 地理的に分散したノードによる動作確認

上記で述べた通り，地理的に分散したシステムのステージング環境は世界規模のネットワーク上で構築する必要性がある．しかしこの方法は，ステージング環境の構築ならびに動作確認の進行において多大なるコミュニケーションコストとヒューマンリソースが予想される．まず環境構築において地理的に離れた地点にノードを設置する必要性がある．地点ごとにノードを設置する人に加え，ノードのスペックやネットワークの構成等について共有するためのコミュニケーションが必要となる．必要な物理筐体が揃ったのち，地理的に分散したシステム上で走らせるアプリケーションを各ポイントに配布し，各開発者は受け取ったアプリケーションファイルを設置したノードの上で走らせる必要がある．ステージング環境でシステムを走らせた後に関しては，機能面や性能面での動作確認を行い，修正箇所があれば開発者がパッチを適応した後，修正後のアプリケーションファイルを各ポイントに配布するところから再度やり直さなければならない．修正箇所が増加するに比例して，コミュニケーションコストと必要なヒューマンリソースは膨れ上がることが予想される．さらにコミュニケーションの不足や伝達ミス等の人的ミスにより理想的な動作確認が行えないケースも考えられる．以上の点から，地理的に分散したシステムのステージング環境においてコミュニケーションベースの動作確認には多くの課題があり現実的に困難である．それ故，地理的に分散したノードを任意のポイントから統合的に管理することによって各地点での作業や地点間でのコミュニケーションを削減する必要性があると考えられる．

2.4.3 独自実装のデバッグエージェントによる動作確認

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地理的に分散した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．

2.5 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。

2.5.1 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。

Docker

Docker [6] 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。

2.5.2 Kubernetes

Kubernetes [7] 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．

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。

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 [8] 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 [9] 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。 2.5.2 章の kubeadm、なら

びに 2.5.2 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 [10] 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．

2.6 OpenVPN

2.6.1 VPN

VPN は “Virtual Private Network” の略で，日本語では “仮想専用線” と呼ばれる．VPN は，パブリックネットワーク上で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を指す．トンネリング技術によって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提言することも可能である．

2.6.2 OpenVPN

OpenVPN [11] は OpenVPN Technologies, inc. を中心に開発が行われているオープンソースの Virtual Private Network ソフトウェアである．OpenVPN は Windows や Linux，Mac OS，iOS, Android でも利用でき，幅広い OS 上で動作可能だ．認証方法も豊富であり，静的鍵による認証や証明書認証，ID/パスワード認証，二要素認証をサポートしている．VPN に関しても，マルチクライアント VPN に加えサイト間 VPN の設定が可能であり，用途によって使い分けることができる．

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，地理的に分散したシステムのステージング環境のための統合環境が未だ整っていないことを課題とする．

中央システムの形式をとるサービスでは，AWS [3] や GCP [4],Azure [5] といったクラウドサービス等を活用することで，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．

対して地理的に分散したシステムでは，ステージング環境の構築は困難である．P2P システムでは個々のノードが地理的に分散し，かつシステムに参加するノード数やノード周辺のネットワーク環境によりシステムの動きが柔軟に変化するため，ステージング環境はこれらを考慮して構築する必要がある．地理的かつネットワーク上で論理的に分散したノードを統合的に管理することが困難であるため，そもそも検証作業を行うステージング環境を構築すること自体が困難である．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ノードをオーケストレーションすることにより，P2P システム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，P2P システムのステージング環境に必要な要件を述べる．

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 デバッグエージェントを用いた統合開発環境の提案

検証対象のアプリケーションを操作するデバッグエージェントを，あらかじめノード内で起動しておくことにより，アプリケーションの配布や実行，終了をデバッグクライアントからデバッグエージェントを介して一斉に操作する手法も提案されている．デバッグクライアントからは GUI の操作，ノード間の接続関係の可視化，ノード間で行われる処理のログ収集などが可能である．

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 [12] 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 [13] 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，ローカル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 [14] 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．

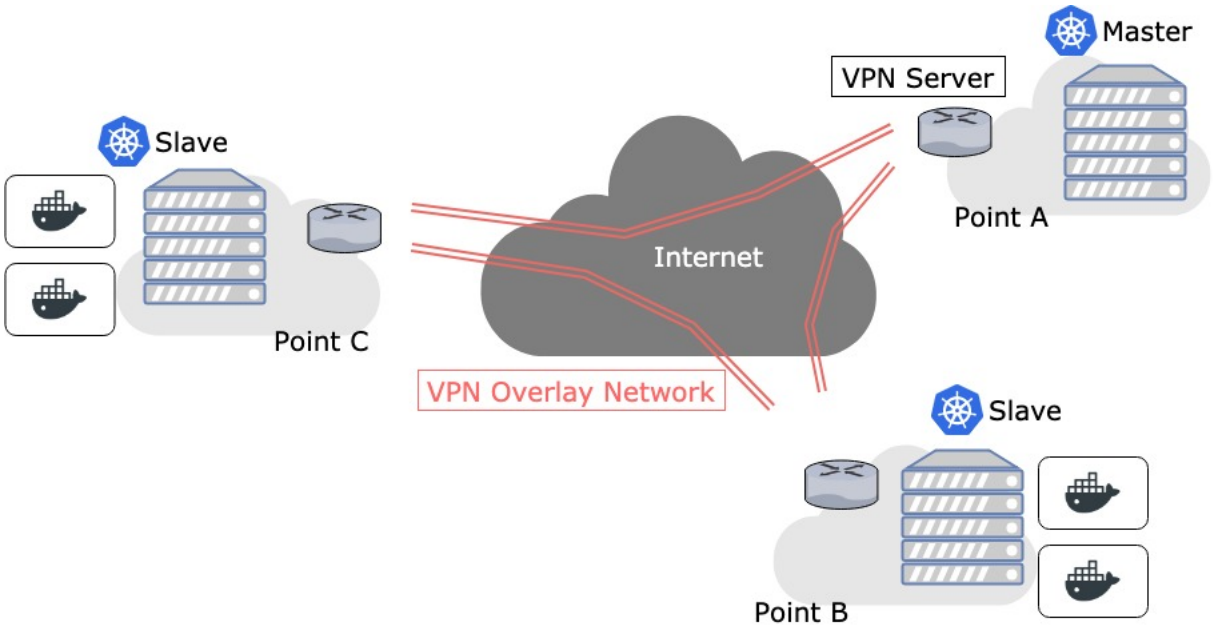

図 3.1: システム概要図

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
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．

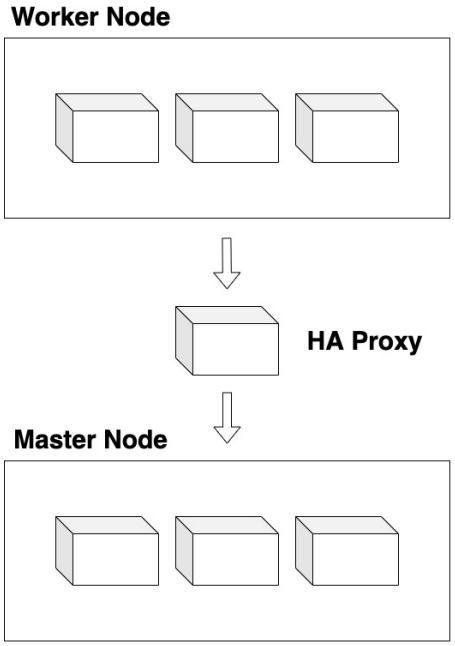

図 4.1: マスターノードとワーカーノードの関係性

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．

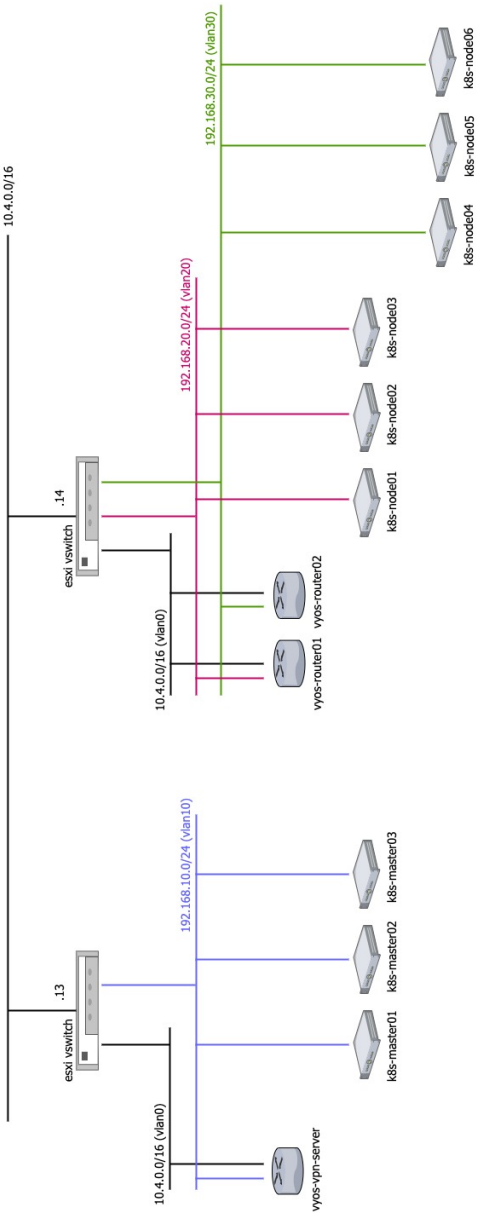

図 4.2: ネットワーク構成図

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，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，新たに開発した地理的に分散した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，地理的に分散したシステムの本番適応前の動作検証における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．着目した問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ローカル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，地理的に分散したシステムの動作検証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，分散したノード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．すべてのノードは 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が，本研究では参加ノードに対する VPN サーバの負荷と 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] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. http://www.cryptovest.co.uk/resources/Bitcoin%20paper%20Original.pdf, 2008.

[2] 金子 勇. Winny の技術, 2005.

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

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

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

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

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

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

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

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

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

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

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

[14] 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.