ネットワークテストシステム プロジェクト
2016 年度 成果報告書

ネットワークテストシステム プロジェクトチーム

2017 年 2 月 27 日

目次

図目次

表目次

第 1 章

はじめに

1.1 本書の目的

本書は、沖縄オープンラボラトリ [1] で 2016 年度に実施された「ネットワークテストシステムプロジェクト」の成果報告書である。本書の目的は、プロジェクトの目標と実施した活動、活動の結果・成果をまとめ、報告することである。特に、最終的な成果 (物) についての解説だけでなく、そこへ至るまでの検討過程・判断や選択の基準などについても取りあげ、「考えかた」についても共有することを目的としている。

1.2 本プロジェクトの目的

情報システムを構築し、利用者に提供していくにあたって、システムを構築・運用する側は、様々な製品や技術を組み合わせてシステムを構成し、サービスを実現していく必要がある。複数の仮想化技術の導入、拡張性に対する要求とそれを実現するための自動化技術の発展などにより、システムは抽象化されて、より柔軟な制御が可能になった。その反面、大規模かつ複雑にもなっている。こうした、多数のブラックボックスを組み合せた複雑なシステムの構築や運用では、システム構成要素間の連携を把握し、システム内部の一部の変更によってシステム全体の動作やサービスにどういった影響があるかを判断することは難しくなる。一方で、利用者の要求やサービスの変化のサイクルは速くなっており、システム開発者あるいはサービス提供者は、その要求に追従していくことが求められる。

このような状況から、今後サービス提供者は、サービス全体の動作を判断し、最終的に利用者に対して価値が提供できているかどうか、サービス利用者の観点でシステムの状況を判断することがより重要になっていく。本プロジェクトは、サービス変化の迅速性を考えるうえでボトルネックになりがちなネットワークに着目する。より高速かつ網羅的にネットワークが提供するサービスの正常性を確認する (テストする) ためのシステム = ネットワークテストシステムの提案、およびその有効性を確認するための実証実験をおこなう。これによって従来のネットワーク構築・運用の迅速性の向上、リスク低減をおこない、CI/CD プロセスを実現させることを目指す。

1.3　関連研究

1.3.1　沖縄オープンラボラトリ

沖縄オープンラボラトリでは、本プロジェクト以外にネットワークテストに関する研究が複数ある。OF-Patch プロジェクト [5] は OpenFlow 制御による物理パッチパネルを実装・OSS 公開している。また、それによって実際に沖縄オープンラボのテストベッドを構築し、利用者に提供している。VNF テストシナリオ自動化 PJ [6] は複数のネットワーク仮想アプライアンスのデプロイパターンや相互接続性に着目しており、ハイパーバイザと VNF, VNF 同士の接続などさまざまなパターンで接続性や性能測定を網羅的に自動実行するシステムを構築している。OF-Patch は L1(物理トポロジ) 操作に、VNF テストシナリオ自動化は VNF の利用 (組合せ・機能・特性) に焦点をあてている。

1.3.2　ネットワークテストの自動化に関する研究・プロダクト

End-to-end の通信やネットワーク全体のふるまい、物理ネットワークのテストに着目したテストや検証の自動化に関する関連研究やプロダクト、それらの特徴について順に挙げる。

■研究　ホワイトボックススイッチと従来の (レガシーな) ネットワーク機器で構成されたテストベッドネットワークに対して、OSS を組み合わせて任意の箇所に自動で障害を発生させる検証自動化システム [7] が提案されている。レガシー機器の利用・end-to-end でのふるまいへの着目など本研究と課題感が近く、テスト環境の自動構築などテストとして求められる機能一式に対応している。本研究ではテスト対象環境の自動構築には比重を置いていないこと、テストトラフィックのディストリビューションに OpenFlow 制御を利用していること、テストシナリオ実装の考えかたなどに違いがある。

■商用製品　FW のフィルタポリシテストの自動化製品がある (NeedleWork [8])。NeedleWork は小型のアプライアンスとして実装されており、FW 単体 (あるいは FW に接続された 3 ゾーン (セグメント) 構成のネットワーク) をテスト対象としている。また、内部では netcat を使用しており [9]、L4 レベルのテストトラフィック生成をおこなうことができる。

■OSS　ネットワークのテストが可能な OSS として、InfraTester [10]、ToDD [11]、OpenVNet [13] がある。InfraTester はネットワークを経由して、特定のサービスの外側からの動作 (ふるまい) をテストすることができる。構成として、定点 (InfraTester サーバ) から見たときのサービスを観測するかたちになるため、L2/L3 のネットワーク (物理的な分散性) についてはターゲットとしていない。ToDD は、テスト対象ネットワークに Agent を配布し、ToDD サーバとメッセージキューによって情報交換をおこなう [12]。テスト自体は testlet という形で ToDD サーバから Agent へおくられ、実行される。Agent を使うことで、ネットワークの物理的な分散性 (distribution) に対応しており、拡張性のたかいアーキテクチャをもっている。OpenVNet は OpenFlow を利用したネットワーク仮想化システムだが、本プロジェクトと同様の課題感から物理ネットワークのテストへの対応をすすめている。OpenVNet で構成した仮想ネットワーク (オーバーレイ) に対して OpenFlow 対応ハードウェアを介して物理ネットワーク (ネットワーク機器) を接続する [14, 15]。仮想ネットワーク (論理構成) については terraform や cloudify などのオーケストレーションツールを併用し、論理構成を記述し、再現・再構成可能とすることで検証環境構成の自由度を確保している。

本 PoC ではターゲットとしていない (3.3 節参照) が、テスト対象となるネットワーク機器の操作をおこなうための仕組みに関する近年の動向について簡単に触れておく。従来のネットワーク機器 API(CLI) を抽象化するための OSS 開発 [16, 17, 18] は活発におこなわれている[*1]。また、Ansible [19] はバージョン 2.2 よりネットワーク機器の操作に対応 [20] しており、ネットワークの構築・運用自動化分野での利用事例が増えていくことが予想される。技術動向としては、Google 等のネットワークオペレータによってベンダ中立なネットワーク機器インタフェース (API) 策定の活動がおこなわれている (OpenConfig [21])。OpenConfig については Juniper, Cisco, Arista などの商用製品による採用 [22] もはじまっている。

1.4　前提事項

本書の読者としては、ネットワークテストシステムの有効性の判断や、利用・導入のための検討を実施したいエンジニアを想定している。本書では、以下にあげる点については基礎知識として取り上げない。

- Linux の基本的な使用方法
  - KVM による仮想マシン (VM) の操作/管理
  - 必要なツールやソフトウェアのインストール/パッケージ管理
  - Git の使用方法
  - Linux Network Namespace および iproute2 (`ip` コマンド) による Namespace の操作
- 基本的なネットワークに関する知識
  - TCP/IP および Ethernet/VLAN
  - Cisco や Juniper の L2/L3 Switch, Firewall の基本的な動作・設定
- OpenFlow
  - OpenFlow/1.0
  - Trema による OpenFlow Controller の実装
  - OpenFlow スイッチ (Open vSwitch) の基本的な動作・設定
- Ruby プログラミング
  - RubyEnv をつかった ruby 環境の管理
  - Bundler をつかったパッケージ管理
  - Cucumber によるテストシナリオ実装

また、本プロジェクトの前身となる、2015 年度に OOL で実行したプロジェクト (「L1patch 応用ネットワークテストシステム」プロジェクト [2]、通称 L1patch プロジェクト) で扱った内容については解説しない。L1patch プロジェクトについては、L1patch プロジェクト活動報告書 [3] や L1patch プロジェクト技術情報 [4] を参照すること。

[*1] 本研究では相当する機能について、よりシンプルで簡易的なツールを使用している (B.1 節参照)。

第 2 章

課題設定

2.1 情報システムの構築・運用における課題

1.2 節に示したとおり、情報システムが利用者に提供するサービスは、より速く・柔軟に変化していくことが求められるようになっている。そのため、サービス提供者側は、利用者に対して適切なサービス価値が提供できているかどうかを判断していくことが求められる。

システムを構成する各要素がサービスに対する利用者要求の変化に対して柔軟に変化していくことが求められるが、近年では特にネットワーク部分がサービス変化の上での迅速性・拡張性の面でのボトルネックになるという状況が発生している。本節ではその理由と課題点について解説する。

2.1.1 自動化の難しさ

ネットワークで自動化がすすまない理由はいくつかあるが、ここではテストの自動化という観点から、主要な課題について解説する。

■垂直統合の歴史 歴史的に、ネットワーク機器はベンダごとに異なる OS、異なる API(コマンド) をもち、共通のインタフェースやデータモデルが存在しない。そのため、異なるベンダの機器をつかったネットワークを作ろうとした場合、設定としてはおなじ操作であっても、異なる API(コマンド) で操作する必要がある。

ネットワークに対する操作の自動化はこれまでもおこなわれているが、機器 (OS) ごと、機器の設計上の役割や運用上のオペレーションごとに多数の自動化スクリプトを用意する必要があり、複雑かつ汎用性が低い状態になっている。また、複数のデバイスを操作するうえでは、設定が反映され動作がきりかわるタイミングなどをふまえたうえで、全体のワークフローを考えなければならないという課題もある。そのため自動化されるのは、シンプルで定型的な処理にとどまることが多い。[*1]

■物理的な位置の操作 ネットワークは、情報システムの構成要素 (計算機リソースなど) の物理的な配置を抽象化する機能をもつため、ネットワークそれ自体のテストについては、物理的な配置 (場所) を考慮する必要がある。テストされていないネットワークでは、何らかの問題によりネットワークが通信不可能になる (あるいは一部のネットワークで正しく通信ができない) おそれがある。保証されたネットワークの存在を前提と

[*1] NW 機器を抽象化し統一した方法で異なる API の機器を操作可能にする製品や OSS も存在する (1.3 節)。しかし、対応していない製品の利用にあたっては、「ドライバ」と呼ばれる操作対象機器の API や取得情報などを機器ごとに別途開発するコストがかかる。API については、Netconf/YANG などをベースにしたインタフェースやデータモデル標準化の動きはあるものの、現時点では実装されている機器はまだ少数であり、ベンダ/OS ごとに個別に取り扱う必要がある、という状況である。

してよい機能やサービスでは、コネクティビティを前提として作業ができるため、ひとつの要素に着目して作業実施・自動化可能なことが多い。しかし、ネットワークのテストはコネクティビティ自体を保証するための作業であり、コネクティビティがあることを前提にできない (図 2.1)。

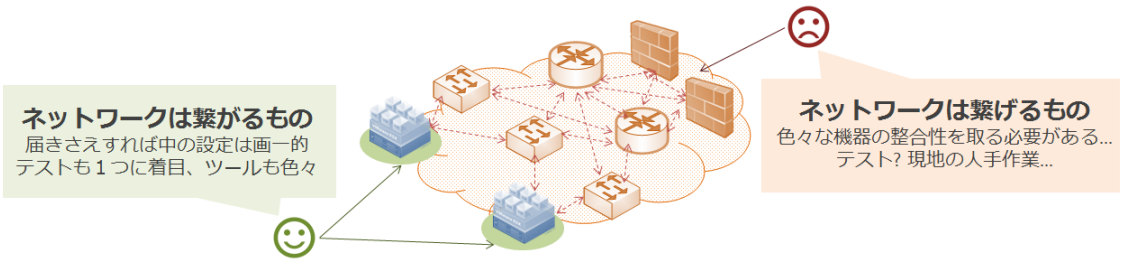

図 2.1　ネットワークとその他のテストの違い

ネットワークの不備あるいはトラブル等により、リモートでのネットワーク機器へのアクセスが不可能になってしまった場合、機器の現物を直接操作して復旧させなければならない[*2]。こうした物理構成上の要求が発生するテスト[*3] は、その「実体を直接操作したい」という要求の性質上、自動化することが難しい。

■テストケースの組み合わせ爆発　ネットワークは自律分散制御され、隣接する機器が相互に通信規約の整合性をとることで、end-to-end の通信が実現される。ネットワークが狙ったとおりに動作しているかどうかというテストでは、物理構成・論理構成を加味した多数の組み合わせを考慮する必要がある。ネットワークのテストパターンは、ネットワークを構成している機能要素の組み合わせによって決まるため、構成要素の数に対して爆発的に増加してゆく。特に近年では仮想化技術の導入がすすみ、テストパターンもより多くなる傾向がある。

2.1.2　ネットワーク全体の動作確認の難しさ

自律分散制御によって構築されるネットワークでは、ネットワークを構成する機器 (ノード) それぞれが周囲の機器と情報を交換しながら独自に通信制御をおこなうことで、ネットワーク全体としての機能や動作が決められる。

したがって、ある目的に対して、どの機器にどのような制御をさせるかは、最終的に実現したいネットワークの動作 (目的) をもとに機器ごとに決定される。個々のネットワーク機器が問題なく動作していても、ネットワーク全体の動作として個々の動作が連結された結果が、期待していた要求を実現できているかどうかを判断するのは難しい (図 2.2)。

ネットワーク全体としての動作保証の難しさは以下のような要素に起因すると考えられる。

- ネットワークでは機器それぞれが様々なポリシをもとにトラフィックを制御する。それらは周囲の機器と交換される制御情報、実際にながれているその時々のトラフィックなどにも影響される。

[*2] こうしたリスクを回避するために、リモートアクセス用のネットワークとサービス用のネットワークを物理的に分離して設計したり (out-of-band management network)、機器コンソールをリモートで利用可能にするための機器 (シリアルコンソールサーバ) を導入したりすることがある。しかし、デバイスの物理的な故障などについてはやはり何らかのかたちで現地・現物での作業が発生する。

[*3] 例えば、リンクダウンなどの物理障害を発生させるケース、ネットワーク機器の追加 (拡張)・削除といったネットワークの物理構成 (トポロジ) を変更するケースなど。

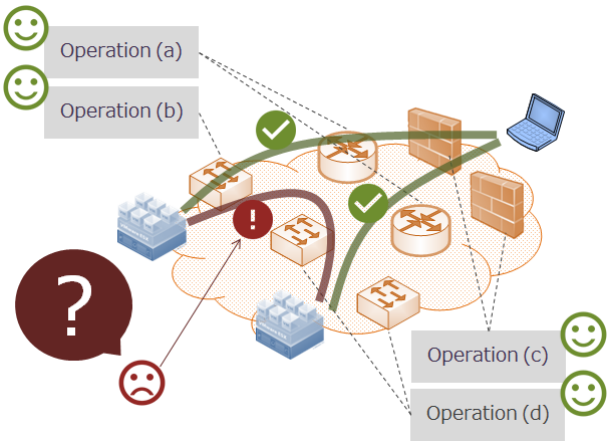

図 2.2　ネットワーク全体の動作確認の難しさ

- ネットワーク全体の動作は隣接する機器間で相互に設定情報の整合性をとる必要があり、使用している機器や技術の数が増加するほど検討すべき組み合わせのパターンが増大する。
- ネットワークは、そのリソースを通常複数のシステムやユーザで共有する。システムの上流ネットワークほど共有度が高くなり、機能変更や障害時の影響範囲が広くなる。
- ネットワークは状態に応じて動作が変化する。ネットワークの状態としては、機器内部でキャッシュされる転送制御情報・発生した通信のセッションやコネクションの情報といった機器単体の状態、物理リンク増減・トポロジのようなネットワーク全体の状態がある。ネットワーク全体としての動作は、こうしたさまざまな状態に応じて変化するため、同一のイベントに対して常に一定の動作がおこなわれるとは限らない。

こうした理由から、大規模なネットワークや、仮想化技術などをつかった複雑性・共有度が高くかつ動的な構成変更がおこなわれるネットワークでは、ネットワークに対する設定変更・構成変更の影響範囲を正確に判断することは非常に難しくなっている。

2.2　従来のネットワークテストとネットワークテストに着目する理由

ネットワークの構築や運用では、自動化の難しさや設定変更による影響範囲の予測のしにくさなどの課題がある (2.1 節)。こうした課題により、従来のネットワーク (特に高いサービスレベルが要求されるシステムのネットワーク) の構築・運用では以下のような状況がみられる。

- ネットワークの構成変更を実施する際は、その物理構成操作の要求やトラブル発生時の現地対応リスクなどを加味して、特定の場所に人や端末を配置しながら人手でテストを実行していくという、人海戦術的オペレーションをとる。
- 特定の設定変更によってどの程度のサービス影響があるのか判断することが難しいため、機器やサービスの知見者を集めて、変更した場合の影響検討をくりかえしおこなう。(複数回のレビューを実施する。)

いずれにせよ、これらは現地・現物・人海戦術的なやりかたとなる傾向があり、システム (サービス) にもとめられる迅速性 (agility) を落とすボトルネックになりがちである。

影響範囲予測の難しさへの対策として、検証環境を用意して、そこで実際に想定されるオペレーションを実行することもおこなわれる。しかし、ネットワーク規模や構成の大規模化・複雑化とそれによるテストパターン数の増大にともない、テストパターンをすべて人手で網羅することは非常に難しい。そのため次のような状況 (リスク) を受け入れざるをえない。

テスト作業用リソースの確保　通常、テストをおこなうための人・機器の準備には制約がある。人手による作業の場合、作業コスト・時間や規模がどうしてもスケールさせられないため、小規模なオペレーションで は十分なテストができないまま本番環境での作業になる傾向がある。

本番環境と検証環境の差分　リソース確保の都合、多くの場合では検証環境と本番環境では使用する機材や構成などに違いがある[*4]。また、実際にネットワーク内部を流れるトラフィックなどを再現するのは非常に難しい[*5]。そのため、検証環境では発生しなかった問題やトラブル、影響の見落としが本番環境の作業で初めて現れることがある。

代表パターンのみのテスト　代表例のピックアップ (サンプリング) をしたテストケースでは、どうしてもサンプリングからもれた一部の設定ミスや不整合などを見落とすリスクがある。

テスト結果判断のばらつき　手順書の解釈、操作の実行や結果の取得・判断などがテスト実行者に依存するため、本来問題となる事象を見落としてしまうリスクがある。これに加えて、複数人でテストを実行している場合は、個別に取得した情報を統合してネットワーク全体としての動作を判断しなければならないというオーバーヘッドが発生する。

[*4] 同一製品ラインだがスペック・ライセンスの異なるものを使う、あるいは仮想アプライアンス版で検証をおこなう、などが検証環境での選択肢としてあるが、特定の機器や機能の組み合わせ、特定のハードウェアやソフトウェアでのみ発生するトラブルなどもある。

[*5] 特に非機能要件、性能や可用性要件などを検証環境で保証することは難しい。環境を段階的に拡張した結果として性能問題がおきる、特定の利用者による過負荷によりトラブルがおきるなど、事前・別環境での再現が難しい問題がある。

第 3 章

課題に対するアプローチ

大規模化・複雑化するネットワークでは、検証環境での手作業によるテストや人による多重レビューによってサービス影響や問題点を完全に洗い出すのは、時間的・リソース上の制約によって非常に困難である (2 章)。こうした課題に対して、本プロジェクトは以下のようなアプローチで対応を試みる。

テストを自動化する　自動化することにより、人力ではできない速度で、人力では見きれない範囲 (パターン) をテストする。

実機・実際のシステムをもとにテストをする　本番環境をどこまで検証環境で再現するかというリソース問題は残るが、テスト対象ネットワークとしては特定のソフトウェア/ハードウェアを区別しない。一般的かつ実際に使用しているネットワークを自動テスト可能にする。

3.1　自動テストの基礎知識

3.1.1　自動テストの一般的な動向

ソフトウェア開発においては、システム (アプリケーション) の自動ビルド・自動デプロイ・自動テストなどがおこなわれ、特に近年では CI/CD や DevOps といった形でノウハウやベストプラクティスの蓄積・共有、ツールや環境の整備といった取り組みをすることが一般的になった。

また、クラウドサービスへのシフトを背景に、アプリケーション (ソフトウェア) だけでなくシステム基盤についても、ソフトウェアによる構築や制御が可能になってきた。ソフトウェアによってシステム基盤全体を制御するという考え方は、IaC (Infrastrucure as Code) や SDx(Software Defined Anything) / SDI(Infrastructure) / SDDC(DataCenter) など様々なコンセプトで実現されるようになってきている。

CI/CD や DevOps などの取り組みは、単なるソフトウェア開発の範囲だけでなく、ソフトウェアによって制御可能な (クラウドサービスベースの) システム基盤を含むシステム (あるいはサービス) 全体で取り組まれるようになっている。こうした取り組みでは、アプリケーションとシステム基盤全体の構築・運用を最適化し、システムの利用者への価値提供 = サービス価値を最大化することをねらっている。

3.1.2　「ふるまい」のテスト

3.1.1 節で取り上げたように、CI/CD といった開発・運用プロセスの取り組みがソフトウェア開発分野を中心におこなわれている。本プロジェクトではそうした考えかたをシステム基盤 (ネットワーク) の構築・運用

に適用していく。ネットワークのテストを自動化するにあたって、独自の手法や考えかたを考案するのではなく、すでにあるソフトウェアの自動テストなどで確立されたベストプラクティスやツールなどを応用する。再発明を回避し、既存の考えかたやノウハウを活用しながら、ソフトウェア開発とシステム基盤 (ネットワーク) が同じ手法でシームレスに開発・運用できるような状況を作りだすことが効果的である。

本プロジェクトでは、ネットワークを BDD(Behavior Driven Development) の考えかたをもとにテストすることを考えた。BDD は TDD から派生した開発手法で、開発するソフトウェアに期待される「ふるまい」に対するテストである [27]。BDD ではソフトウェアに期待される「ふるまい (Behavior)」、つまり、ソフトウェアの複数の機能を連結した end-to-end でのインテグレーションテストを行なう (図 3.1)。

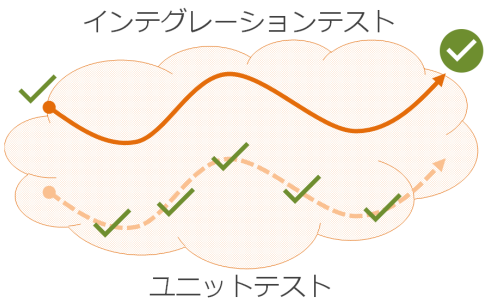

図 3.1　インテグレーションテストとユニットテスト

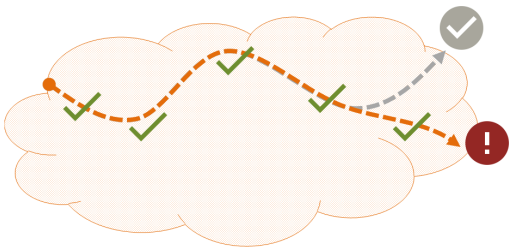

図 3.2　ユニットテストの課題

仕様 (spec) に基づいたインテグレーションテストを最初におこなうようにすることで、以下のようなメリットがある。

- テストの目的を明確にする
- 実際的なテストができる
- 無駄なテストをおこなわない

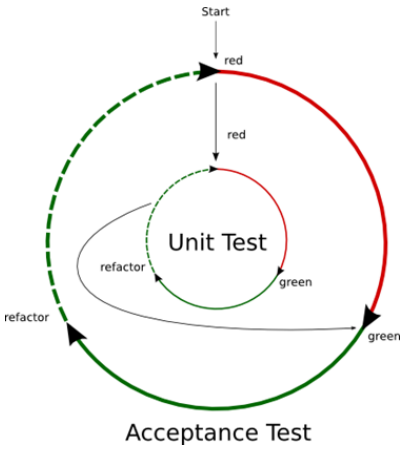

図 3.3　BDD サイクル [23]

BDD では、まずインテグレーションテスト (仕様上期待されるふるまい) についてのテストをおこない、問題があった場合に内部を詳細化したユニットテストに分割して原因を切り分ける (図 3.3)。

インテグレーションテストがパスすれば詳細なテストを省略し、インテグレーションテストで失敗した場合にユニットテストを実施して原因の切り分け・修正をおこなう。こうすることによって、不要なユニットテストの実装・実行にともなう開発効率の低下を回避し、最終的にシステム (ソフトウェア) の利用者に対して価値を提供しているか (期待される仕様を満たすか) どうかを基準にテストをおこなう。インテグレーションテストでは、テスト対象とするシステム (ソフトウェア) の最終的なふるまいをテストするため、仕様とテストシナリオがほぼ直接対応する。したがって、BDD のテストシナリオを記述する作業は製品が満たすべき仕様を具体的に定義したものとなる。

通常、テストコードは直接的に金銭的報酬を発生させるものではなく、開発においてテストによる品質担保とテスト対象 (プロダクト) とのバランスを適切に保つ必要がある。ユニットテストなどの詳細化された機能単位のテストでは、個々の機能としての動作は確認できるが、複数の機能を組合せ・連結して最終的にどのよ

うな動作をするか、というテストとの対応が結びつけにくい (図 3.2)。

こうした状況では、往々にして「ユニットテストをもれなく作成すること」「テストカバレッジを 100% にすること」が目的化されがちである。しかしもちろん、個々のユニットテストがパスしても、最終的にテスト対象となる製品が (利用者が求める) 仕様を満たさなければ価値がない。BDD では最終的に製品が満たすべき仕様に着目し、テスト対象が実現すべき価値の観点で無駄なテストを減らすことを想定している。

3.2　理想像とプロジェクトのターゲット

3.2.1　テストしたいネットワークの「ふるまい」

3.1.2 節に示したとおり、本プロジェクトでは BDD の考えかたにそって、ネットワークの「ふるまい」をテストする。ネットワークのふるまいとして、大きく下記の 2 点を考える。

- 静的なふるまい
- 動的なふるまい

■静的なふるまい　ネットワークが定常状態にあるときに、end-to-end でどのような通信ができるか (end-to-end の通信試験: 図 3.4)。ネットワークにもとめられ最も基本的なる機能要求は、必要なノード間/アプリケーション間で通信ができることである。静的なふるまいとして、ネットワークの状態が変わらない (一定のトポロジ/一定の状態、例えば、通常利用していて特にイベントが発生していない状況) ときに、アプリケーションレベルでの通信が実現できる (できない) ことを確認する。

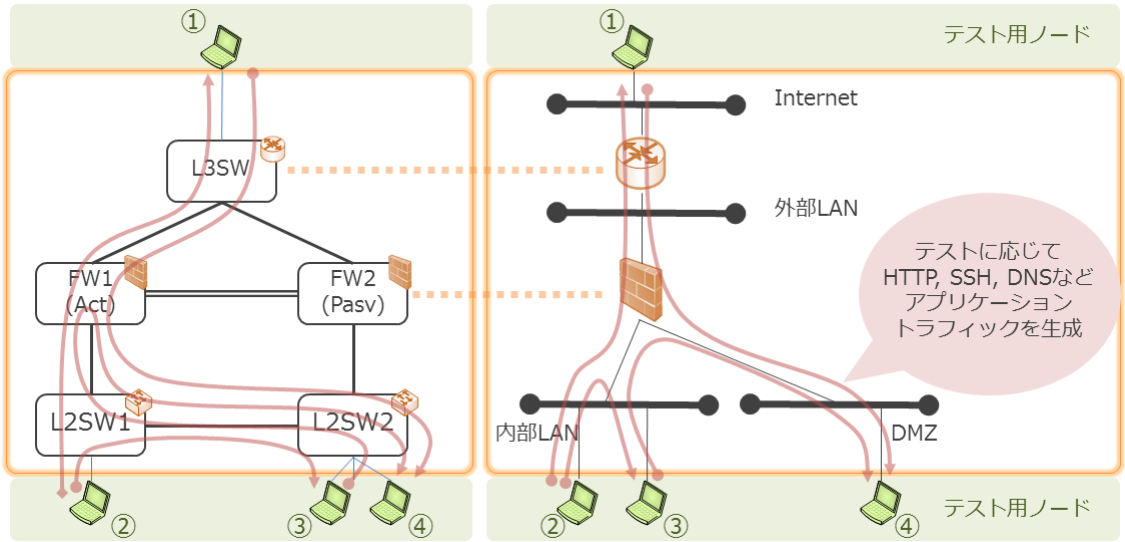

図 3.4　ネットワークの静的なふるまいのテスト

このような通信試験においては、適切な場所 (物理的・論理的な位置) にノードを設置すること、なるべく実際的な (実際利用されたときに発生するトラフィックに近い) テストトラフィックを発生させることがポイントとなる。

■動的なふるまい　ネットワークは、耐障害性・拡張性を保証するために、ネットワーク機器間で相互に制御情報を交換し、動的にネットワーク全体のトラフィック制御をおこなう[*1]。こうしたネットワーク自身の状態変化がともなう状況を、「動的なふるまい」ととらえる。

定常状態にあったネットワークで、イベントに対して動的なふるまいが発生すると、ネットワークの状態[*2]が別の定常状態へと変化する。状態変化の過程では、利用しているネットワーク機器の機能や仕様により、テストトラフィックパスが変更され、一時的に転送が中断/再開したりする。動的なふるまいのテストは、こうした状態遷移中の通信状況を確認するものである (図 3.5)。

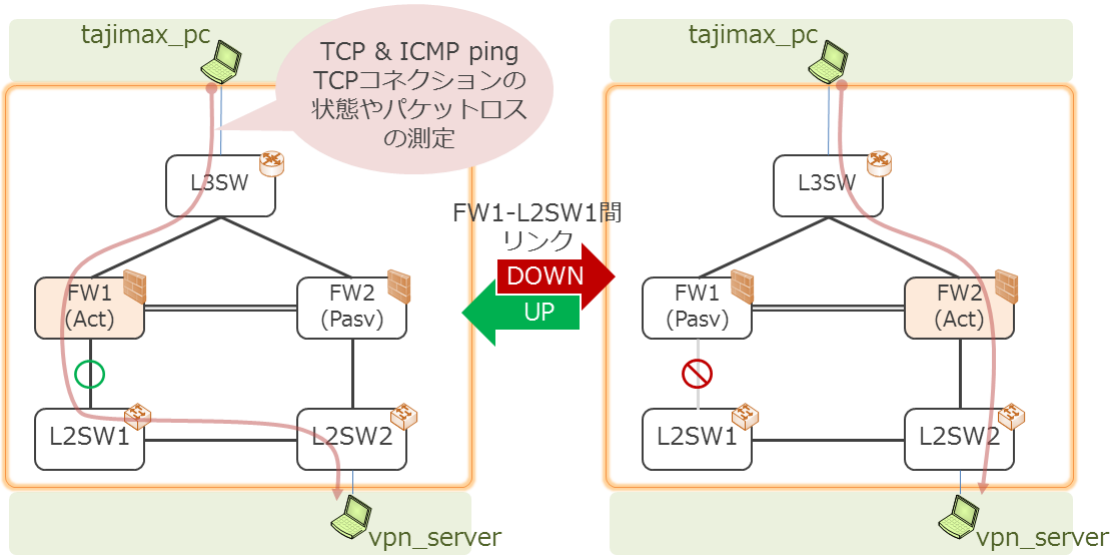

図 3.5　ネットワークの動的なふるまいのテスト

動的なふるまいのテストでは、テスト対象ネットワークでの状態変化イベントの発生、イベントをまたいだテストトラフィックの生成や通信状況の判断がポイントとなる。

3.2.2　ネットワークの自動テストと運用プロセス

2 章で解説したように、現状ネットワークのテストは人手によるところが多く、それがボトルネックになって網羅性やスケール性が低い。ここでは、もし仮に、現状人手に頼らざるをえないネットワークの操作が機械的に・自動実行できるとしたらどのような構築・運用プロセスをとることができるかについて検討する。

ソフトウェアによってネットワークの操作が自由にできる (操作の自動化ができる) と仮定した場合、ネットワークに対する構築や運用についても、ソフトウェア開発でおこなわれているベストプラクティスやノウハウがそのまま応用できるようにしたい (3.1.2 節)。これによって、図 3.6 のようなかたちでシステムの自動構成・自動テスト、本番システムデプロイという CI/CD プロセスを実現させることができる。

- システム (ネットワーク) で実現すべき要求を定義する

[*1] 例えば、ある機器に障害が発生し、その機器でトラフィックが中継できないと (周囲の機器が) 判断した場合、その機器を中継しないようにネットワークのトラフィック転送ルールが再構成される。

[*2] トポロジ、あるいは NW 機器の持つ状態 (経路情報, Acitve/Standby などの状態など)

- 要求をコード化する
  - 要求を実現するためのコード (ネットワーク設計/実装)
  - 要求を確認するためのコード (ネットワークが満たすべき「ふるまい」: ネットワーク上で実現されるべき通信やネットワーク自身の動作)
- 検証環境で自動構築・自動テストをおこなう
  - テストが失敗したら原因を分析し、コードを修正・再テストをおこなう
- 自動テストがすべてパスしたら本番環境へのデプロイをおこなう

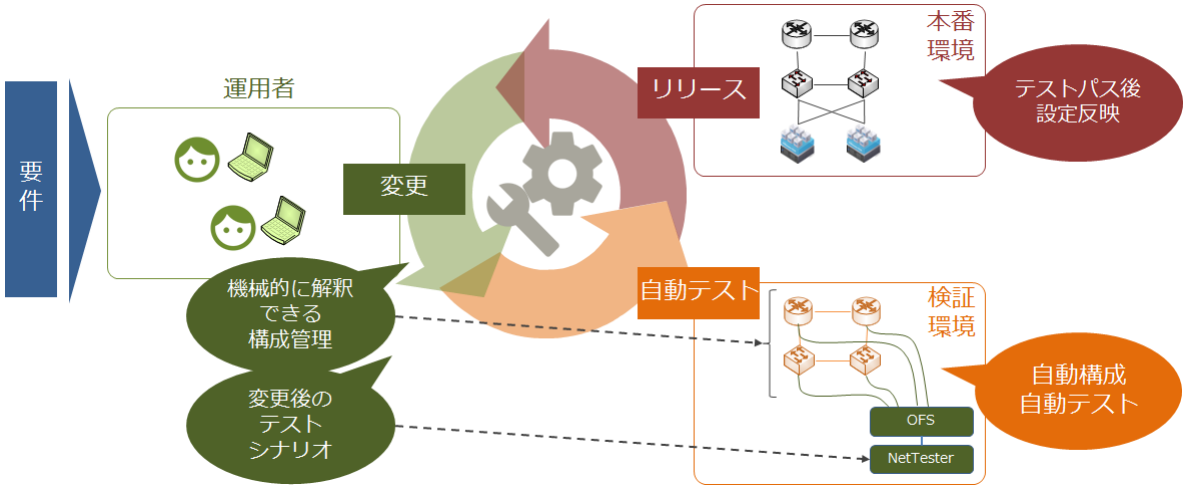

図 3.6　ネットワークの CI/CD プロセス

ネットワーク (に限らずシステム基盤) では、ハードウェア製品を多用するため本番環境とまったく同等の検証環境を整備することは通常難しい。しかし、性能面・拡張性の問題などをある程度スコープ外とすることで、仮想アプライアンスなどを利用して小規模でも機能的には同等の環境を整備することができるようになっている。このように、仮想化技術の応用をふくめて、機器/環境をソフトウェアで制御できる範囲がひろがり、自動化されるにともない、まず自動構築のための技術・ツール・ノウハウが発展しつつある。

本プロジェクトは、そこに (ネットワークの) テストという構成要素を補い、上記のようなシステム基盤の CI/CD 実現を促進させることを狙っている。

3.3　ネットワークテストシステムの検討

ネットワークのテストを自動化するにあたって、現状手作業でおこなっている操作を機械的に実行できるようにしなければならない。本プロジェクトでは、ネットワークのテストのために必要な操作を図 3.7・表 3.1 のように分類している。

■テスト対象 NW 内ノード操作　表 3.1 No.1-2 はテスト対象ネットワークを構成する機器 (ハイパーバイザなど仮想化基盤管理・操作を含む) である。こうした機器操作のための技術・ツールなどは既に多数存在しているため、本プロジェクトではここには注力しない。(既存の技術やツールを応用する; 1.3 節)

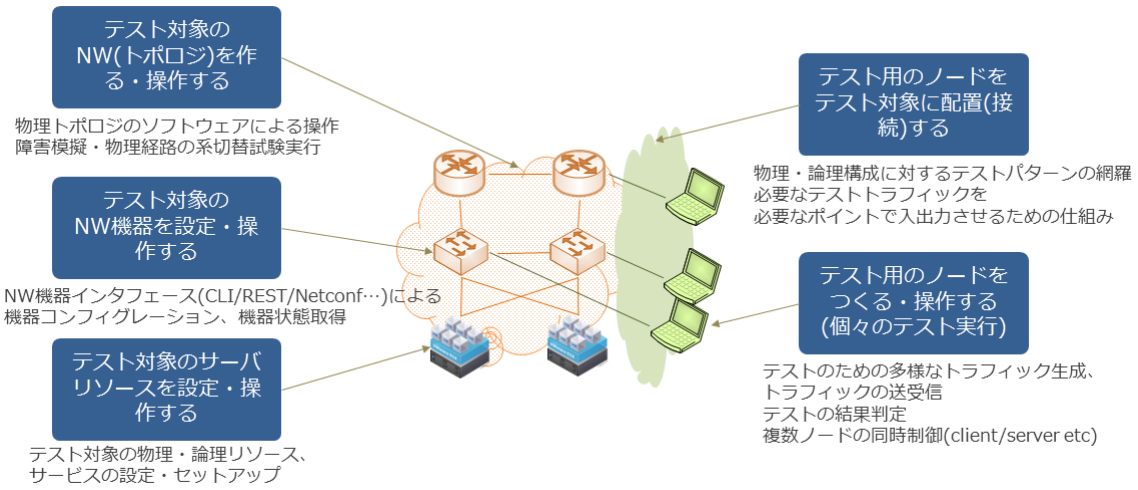

図 3.7　ネットワークテスト自動化のための機能要素

表 3.1　NW テスト自動化に必要な機能要素

No.	要素	役割
1	NW 機器を設定・操作する	テスト対象ネットワークにある NW 機器の設定を変更する
2	サーバリソースを設定・操作する	テスト対象ネットワークにある物理サーバ・仮想サーバのリソース操作などをおこなう
3	トポロジを操作する	テスト対象ネットワークのトポロジをソフトウェアによって操作する
4	テスト用ノードを配置する	テスト用のノードを生成し、テスト対象ネットワークの指定した箇所に配置 (接続) する
5	テスト用ノードを操作する	テスト用のノードを操作し、テストトラフィックの生成をおこなう

■トポロジ操作　表 3.1 No.3 はテスト対象ネットワークのトポロジ (物理結線を含む) の操作である。障害試験 (リンクダウンなど) や環境の構成拡張・縮小 (ノードの追加・削除などトポロジ変更) といった、ネットワークトポロジそのものの変更が発生する際のネットワークのふるまいをテストするために必要となる。

2.1.1 節で示したように、特に物理構成要素の操作は自動化が難しい。L1patch プロジェクトでは、OpenFlow スイッチを応用したテスト対象ネットワークの物理トポロジ構成・操作についての実証実験を実施した。本プロジェクトではそこで実証した技術を応用してテスト対象ネットワークの物理トポロジ操作をおこなう。

■テスト用ノード操作　表 3.1 No.4-5 はテスト用ノードの操作である。一般的に、ネットワークのテストではテスト用のトラフィックの生成・受信をおこなうことによって、テスト対象ネットワークが問題なく動作しているかどうかを確認する。テストトラフィックの送受信はテスト対象ネットワークを構成する機器 (NW 機器やサーバ) を利用場合もあるが、ログ取得やテスト用ツール準備などの都合から、何らかのテスト用ノード

(PC など) を別途用意することが多い[*3]。こうしたテスト用ノードを利用したテストには以下のような問題がある。

- テスト用ノード (機器台数) の上限
- テスト用ノードを操作する人 (担当者人数) の上限
- 分散作業によるテスト全体での状況把握の難しさ
- テスト用ノードのセットアップ (IP など) の手間
- テスト用ノードの配置 (テスト対象 NW への接続) の手間

トポロジ操作同様に、L1patch プロジェクトでは、Linux Namespace を利用したテスト用ノードの生成・集中制御と OpenFlow スイッチを応用したテスト用ノード配置についての実証実験をおこなった。本プロジェクトではそこで実証した技術を応用して、より実用的なテストユースケース (テストシナリオ) の実現をおこなう。

3.4　基本的なアイディア

3.3 節でネットワークのテスト自動化のために必要な機能要素について解説したが、それらをもとにネットワーク「テスタ」に求められる機能要求について解説する。要求については L1patch プロジェクト技術情報 [4] も参照のこと。

ネットワークのテスト自動化では、従来手作業でおこなっていたのと同等の操作を、ソフトウェアにより自動的かつプログラマブルに実現したい。そのために、テスト対象ネットワークの物理トポロジを、テストシナリオに応じて動的に再構成したり、テスト用のノードを動的に接続したりする (図 3.8)。そこで、様々なノード間をつなぎ合わせる機能を準備する (表 3.1 No.3-4)。

L1patch プロジェクトでは、安価かつトラフィック制御を自分で操作 (プログラミング) 可能な OpenFlow スイッチを利用して、物理配線操作をするパッチパネルと同等のものを実現し (図 3.9: 表 3.1)、簡単なテストユースケースのもとで有効性を確認した。

本プロジェクトは、L1patch プロジェクトの結果をもとに、NetTester [24] というテストツールを開発した。NetTester については 4 章・5 で解説する。また、NetTester をつかったネットワークテストのユースケース実証については 6 章・7 で解説する。

[*3] 特に新規構築の場合は、サーバ等テスト対象とするノードが利用できない (存在しない) 状態でネットワークのテストをおこなうこともある。

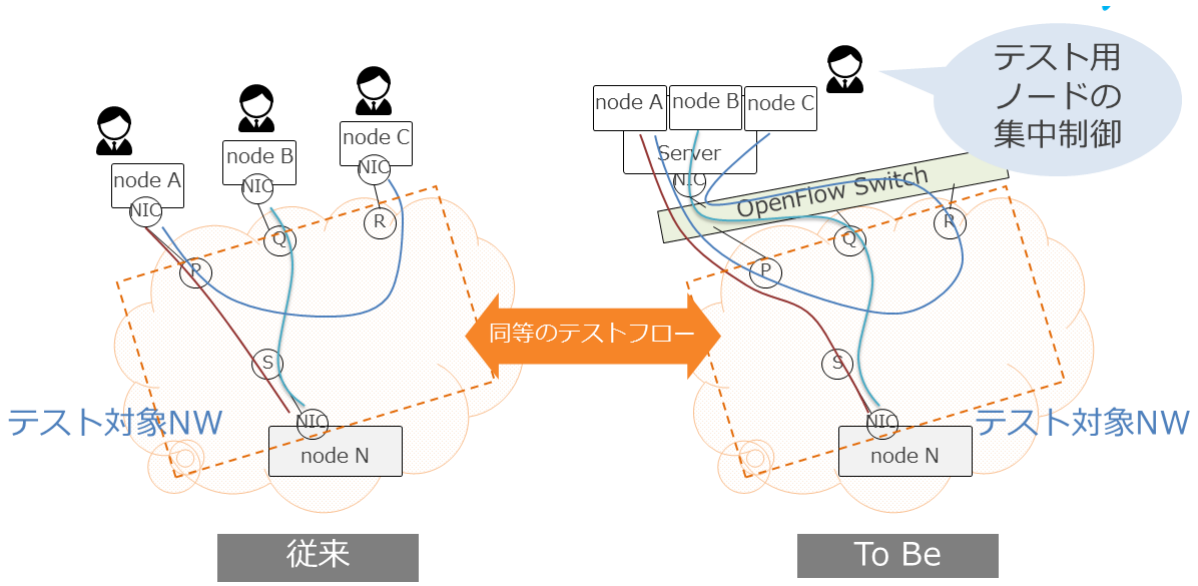

図 3.8　テスターの基本アイディア

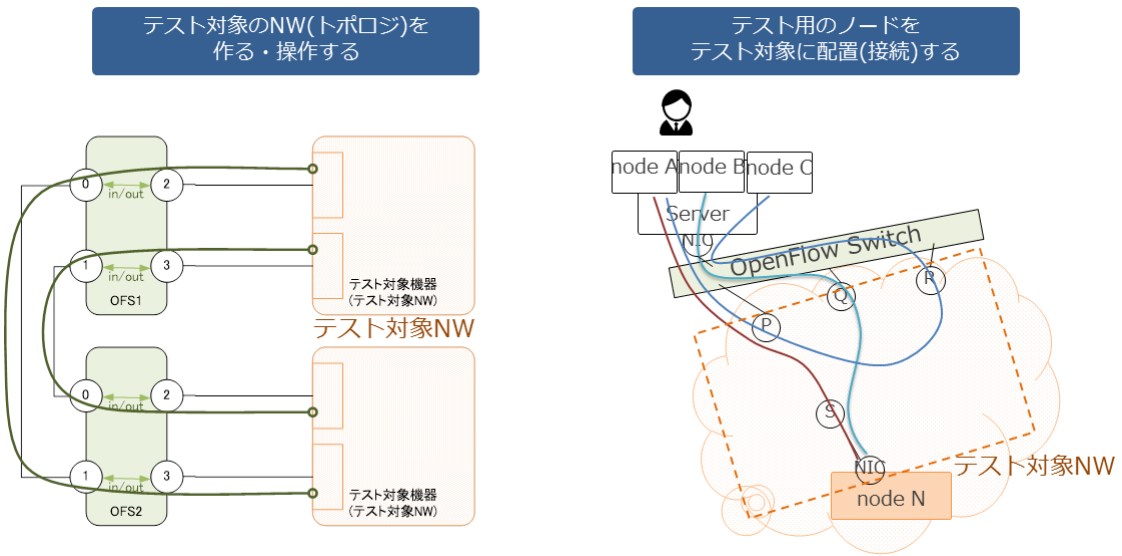

図 3.9　L1patch プロジェクト PoC で実装・実証した機能要素

第 4 章

NetTester の技術仕様

本プロジェクトで NetTester に実装した「パッチ」動作とフロー制御方式について解説する。テスト用ノードの操作については Trema/Phut を利用して実装されている。Phut の使いかたや NetTester での応用については B.3 を参照のこと。

4.1 NetTester のモデル

4.1.1 基本的な構成要素

本プロジェクトでは、3.4 節で解説したネットワークテストシステムのアイディアを、NetTester [24] というツールとして実装した。NetTester は図 4.1 の示すコンポーネントで構成される。

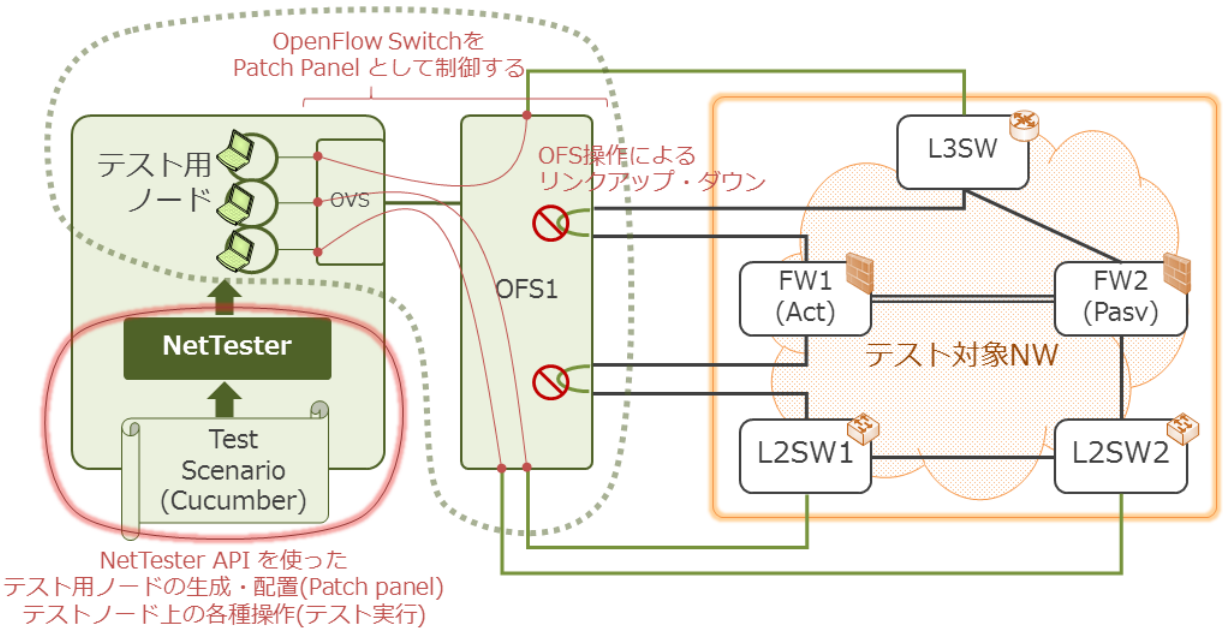

図 4.1 NetTester のモデル

物理スイッチ (OFS)　パッチパネルとして動作させる OpenFlow スイッチ[*1]。テスト対象ネットワークのトポロジ操作やテスト用ノードの配置などをおこなう、物理ポートを複数もつスイッチである。

ソフトウェアスイッチ (OVS)　パッチパネルとして動作させる OpenFlow スイッチ[*2]。仮想ノードとして作成したテスト用ノードを収容し、物理スイッチを介してテスト対象ネットワークにノードを配置するためのスイッチである。

テスト用ノード　テスト対象ネットワークのふるまいを調べるために、テストトラフィックを生成しするためのノード。実装としては、Linux OS 上の仮想ノード (Linux Network Namespace を使用したノード) で実現される。

NetTester　テスト用ノードやパッチパネルの作成・管理・制御をおこなうためのツール。テストシナリオを実行するために必要なテスト用リソース (テスト用ノードとパッチによる配置) 操作のための API を提供する。

NetTester Server(Linux)　NetTester を実行し、テスト用ノードをホストするサーバ。Network Namespace が利用可能な Linux OS を使用する。

テストシナリオ　テスト対象ネットワークのふるまいを調べるための手順 (シナリオ) 定義。本プロジェクトでは Cucumber を使ってテストシナリオの記述・実装をおこなう (7 章)。

4.1.2　構成上の制限・制約

PoC 実施時点[*3]の NetTester では、図 4.1 のようにモデルを定義した。このモデル定義では、以下のようなテストシステム構成上の制限や制約がある。

- NetTester が制御可能な物理スイッチはひとつだけである。そのため、テスト対象ネットワークで操作可能なリンクやテスト用ノード配置可能な範囲は物理スイッチの持つポート数を上限として制限される。
- NetTester が制御可能なソフトウェアスイッチはひとつだけである。また、NetTester は NetTester が実行される OS 上でのみテスト用ノードの生成・操作ができる。
- 物理スイッチおよびソフトウェアスイッチを接続するためのリンクは 1 本とし、物理スイッチ側のポート番号を Port 1 とする。また、このリンクは NetTester を実行する OS で専有して使用する。(NetTester Server を仮想マシンとして実行する場合、物理スイッチ-ソフトウェアスイッチ間を接続するための物理リンク/ポートを他の仮想マシンと共有させない。)
- ソフトウェアスイッチの Datapath ID (DPID) を固定している (SSW DPID=`0xdad1c001`[*4])。物理スイッチの DPID は環境変数により実行時指定が可能である (5.2.1 節)。

これらのモデル定義および制限の設定は、NetTester 実装をシンプルにするためのものである。本プロジェクトでは、NetTester それ自体の機能拡張や拡張性の担保ではなく、BDD ベースのネットワークテスト (ユー

[*1] 本書あるいは NetTester 関連ドキュメント中では “物理 OpenFlow スイッチ”,“Physical (OpenFlow) Switch”, “PSW” などで表記される。実装上は Pica8 スイッチを使用していることから、単に “Pica8” とすることがある。

[*2] 本書あるいは NetTester 関連ドキュメント中では、“Software (OpenFlow) Switch”, “SSW” などで表記される。あるいは、実装上は Open vSwitch を使用していることから、単に “OVS” とすることがある。

[*3] 2016 年 10 月-12 月

[*4] “daddy cool”

スケースの実証) を目標としている (3.1.2 節)。

4.2　フロールール設計の方針

4.2.1　フロー制御対象の分類

NetTester が制御するテストシステムのトラフィックは大きく以下のように分類できる。それぞれについて NetTester で OpenFlow によるフロー制御を使用する。

■テスト対象 NW 機器間の物理結線 (トポロジ) 操作　表 3.7 No.3 相当。テスト対象ネットワークの物理構成操作のための機能。物理スイッチが持つポートを L1 レベルで 1:1 に直接マッピングする方法で制御する。これは L1patch プロジェクトで解説している「専有モードワイヤ」と同様であるため、本書では解説しない。

■テスト用ノードの配置　表 3.7 No.4 相当。テスト用ノードを、テスト対象ネットワークの任意の箇所に配置 (接続) させるための機能。NetTester サーバ上に生成し、ソフトウェアスイッチに収容したテスト用ノードを、物理スイッチを経由させてテスト対象ネットワークにパッチする。

NetTester 上の複数のノードをパッチするために、NetTester サーバと物理スイッチ間の複数の物理リンクを必要とするようにしてしまうと、サーバ物理ポート数が拡張上のネックになってしまう。そのため、テスト用ノード配置では L2 レベルでの制御をおこない、ひとつの物理ポート/リンクに複数の「パッチされた配線」が通るようにする。

L1patch プロジェクトではこれを「共有モードワイヤ」としてフロー制御を実装した。L1patch プロジェクトでは複数の物理 OpenFlow スイッチを使用する想定でフロー制御を考慮している。NetTester では、フロールール実装を簡略化するために、システム構成として物理スイッチを 1 台に限定した (4.1.1 節)。よって、L1patch プロジェクトで検討したフロー制御をもとに、より簡略化した制御ルールとなっている。以降、テスト用ノード配置のためのフロールールについて解説する。

4.2.2　フロー設計の方針と前提

■制御対象トラフィック種別　一般的に OpenFlow によってフロー制御 (IPv4 の制御) を設計する場合、以下のようにトラフィックの種別にわけてフロールールを設計する必要がある。

- Broadcast
  - ARP Request (L2 broadcast)
  - Unknown Unicast (Flooded packet)
- Unicast
- Multicast

テスト用ノードについて、NetTester では「テスト用ノードは IPv4 Unicast のテストトラフィックを生成する」という前提をおいている。したがって、マルチキャストトラフィックの利用 (制御) は想定していない。また、今回の PoC 範囲において、IPv6 のテストトラフィックを含むユースケースについては想定していない。制御ルール自体は L2 となるため、Unicast については IPv4/v6 による影響は原理的にはないが、テスト用ノードでの IPv6 ND/RA などの利用は考慮していない。

■フローテーブル設計　L1patch プロジェクト同様、OpenFlow/1.0 を利用するため、すべての OpenFlow スイッチはひとつのフローテーブルをもつ (Single Table)。複数フローテーブルによるフロー処理の分割[*5]は考慮せず、あるフローが複数のフロールールにマッチする可能性がある場合はフロールールの優先度 (priority) による制御をおこなう。

4.3　フロー制御設計

L1patch プロジェクトではテスト用ノード配置のためのフロールール設計を、複数の物理 OpenFlow スイッチを使用する想定で設計していた。本プロジェクトでは、図 4.1 のように簡略化して物理 OpenFlow スイッチを 1 台とした。これによって、テスト用ノードをテスト対象のどこに (物理スイッチのどのポートに) 配置したいかが決まれば、物理スイッチ-ソフトウェアスイッチ間の中間経路が一意に定まる。

■ユニキャストの制御　L1patch プロジェクトと同様の方法で制御可能である。本書では特に解説しない。

■ブロードキャストの制御　L1patch プロジェクトでは複数の OpenFlow スイッチを使用するため、ソフトウェアスイッチ (テスト用ノードが収容されている OpenFlow スイッチ) でパケットの複製をおこなっていた。このとき、パケット複製の範囲を指定するために wire-group とよばれる情報を付与して制御するようにした。

今回、システム構成を簡略化したことにより、wire-group を利用せず実装をある程度簡易にする方式と、L1patch プロジェクト同様の方式の 2 種類が考えられた。NetTester では簡略化したブロードキャスト制御を採用した。以降、それぞれの制御パターンおよびメリット・デメリットと NetTester での採用理由について解説する。

4.3.1　L2 Broadcast 制御の要件

例として、テスト用ノード 4 台を、ふたつの L2 セグメントがあるテスト対象ネットワークに 2 台ずつ配置 (パッチ接続) する状況を考える (図 4.2)。ノード 1 からノード 3 へ ping を撃つ場合、テスト対象ネットワーク内で各ノードの MAC アドレスを学習するために ARP Request/Reply のやりとりがおこなわれる。ここで、ARP Request はブロードキャストとなるため、ARP Request がブロードキャストされたセグメントに所属しているテスト用ノードすべてに転送されなければならない。

テスト用ノードを増やしていったときに、テスト対象ネットワークにある機器側で接続先物理ポートが都度必要になってしまうと、拡張性を制限する原因となってしまう。テストシステムでは、自由に・多数のテスト用ノードを生成し、かつ、物理ポートの利用を抑えられるようにしたい。そのために、テスト用ノードでは VLAN による複数セグメントの集約を利用できるようにする (図 4.3)。

VLAN の制御については 4.4 節で解説するが、図 4.3 のようにした場合、テスト対象機器 (DUT) の同一物理ポートに接続したテスト用ノード (ノード 1/2, 3/4) の間では相互に通信ができないという制約が発生する。これは、フラッディング動作による。たとえばノード 1 が送信したブロードキャストはノード 2 にはフラッディングされない。したがって、ノード 1/2 は相互に ARP をやりとりすることができない。

この問題を解決するためには、同一機器 (DUT)・同一セグメントでの通信をおこないたい場合は図 4.4 の

[*5] Multiple Table: OpenFlow/1.3 以降で対応。

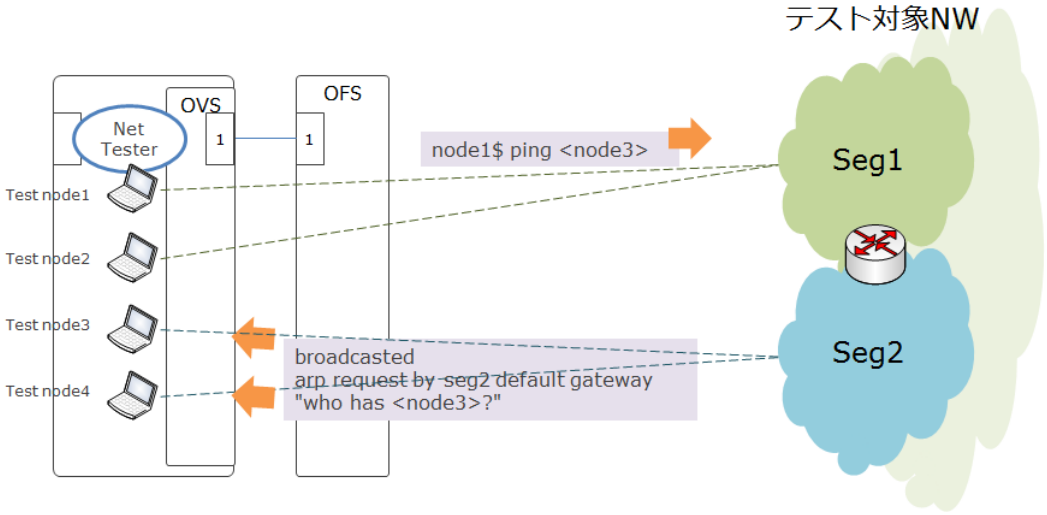

図 4.2　L2 Broadcast 制御の要件

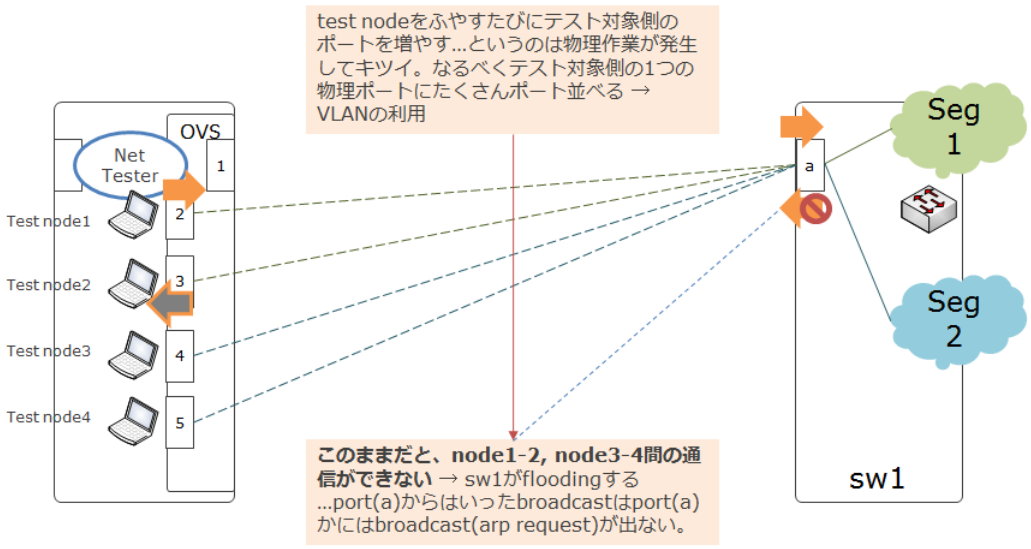

図 4.3　パッチパネル動作上の制約

ような構成をとる。詳細については L1patch プロジェクト技術情報 [4] を参照すること。

ブロードキャストは、実際には物理スイッチとソフトウェアスイッチの間の物理リンクを経由して転送される。4 台のテスト用ノードがひとつのセグメントに所属していて、ノード 4 のみが異なる物理ポートでパッチ接続されている状況を考える (図 4.5)。このとき、Seg.1 で発信されたブロードキャストは、ポート (a) とポート (b) からそれぞれ出力される。このふたつのブロードキャスト (フレーム) は物理スイッチからソフトウェアスイッチへ転送され、ノード 1-4 へ転送されなければならない。

システム構成上、物理スイッチとソフトウェアスイッチの間はひとつの物理リンクのみとする (4.1.2 節)。OpenFlow では、同一のポートにパケットの複製して複数ながすことはできないため、ソフトウェアスイッチで各テストノードむけにパケットを複製して送信する必要がある。

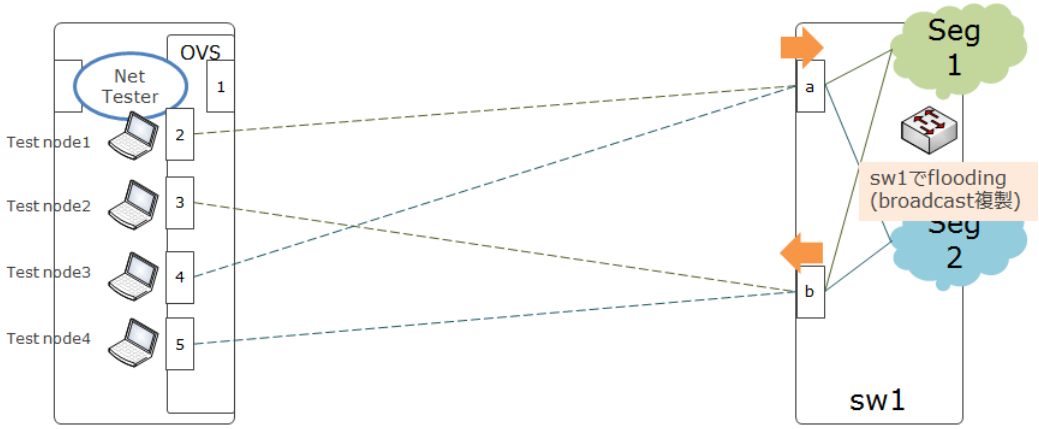

図 4.4　テスト用ノード配置の要求

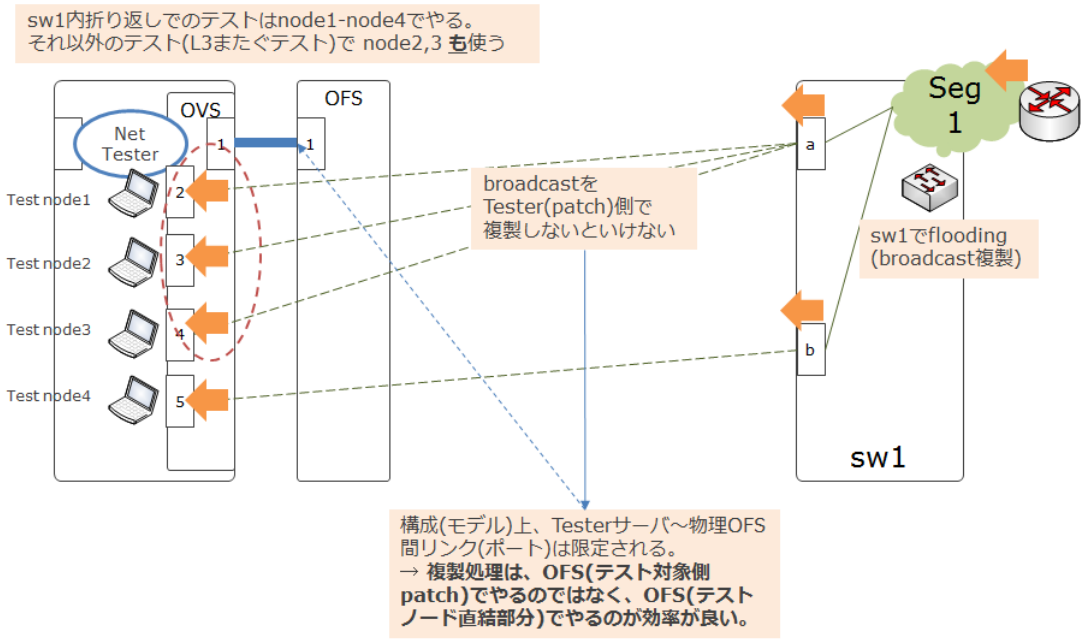

図 4.5　L2 Broadcast パケット複製ポイント

「パッチパネル」では、接続された 2 点 (2 ポート) 間が 1:1 に接続される。図 4.5 ではポート (a)(b) から送信された Seg.1 内のブロードキャストがふたつソフトウェアスイッチに転送されるが、「パッチパネル」としてはこれを同じものとして扱うのは望ましくない。パッチ接続の対応関係に基づいて、ポート (a) から送信されたブロードキャストフレームはノード 1-3 へ、ポート (b) から送信されたブロードキャストフレームはノード 4 へと、それぞれ転送されるのが「パッチパネル」としての理想的な動作となる。

4.3.2　Broadcast 制御 案 1/簡易制御

4.3.1 節に、「パッチパネル」動作に基づいてブロードキャスト制御に制御されるのが理想的かを示した。しかし、この理想的な動作をやや緩和することで、ブロードキャスト制御の実装が非常にシンプルにできる。

案 1/簡易制御では、図 4.5 と同様の状況で、ポート (a)(b) から送信されるブロードキャストフレーム (a)(b) を区別しない。これらはいずれも Seg.1 内にブロードキャストであり、どちらも同様に Seg.1 に属するノード (ノード 1-4) へ複製・転送する (図 4.6)。

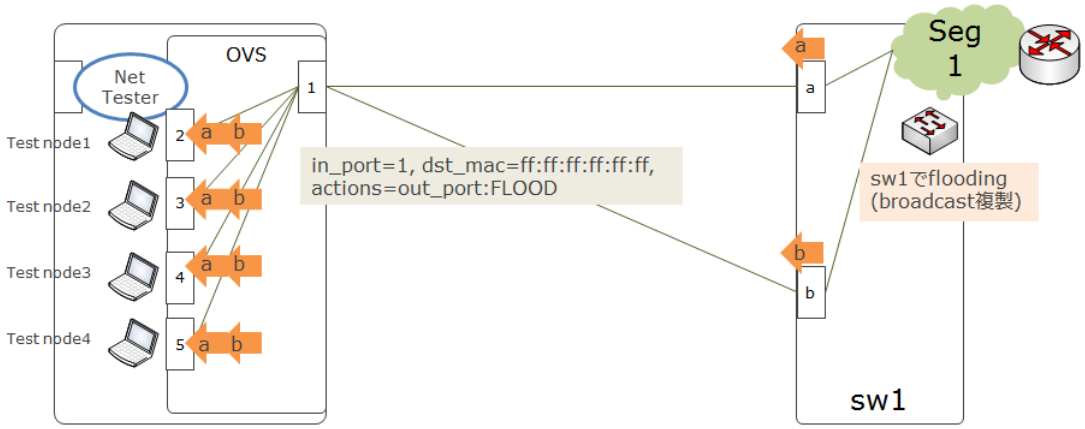

図 4.6　Broadcast 制御 案 1/簡易制御

そのため、Seg.1(テスト対象 NW) 内で複製されたブロードキャストフレーム (a)(b) は都度複製され各テスト用ノードに送信される。この動作は、ソフトウェアスイッチ上ではひとつのルールでノード数によらず固定で記述できるため、フロールールの実装上は非常にシンプルになる。

4.3.3　Broadcast 制御 案 2/Wire-group 制御

4.3.1 節に示したように、ブロードキャストは対応するセグメントおよびパッチ接続先 (DUT 物理ポート) に応じて複製・転送されるのが望ましい。L1patch プロジェクト [2] ではこの理想的な動作を実現するために、wire-group という情報を付与して、パッチ接続単位のブロードキャスト制御を実装した (図 4.7)。

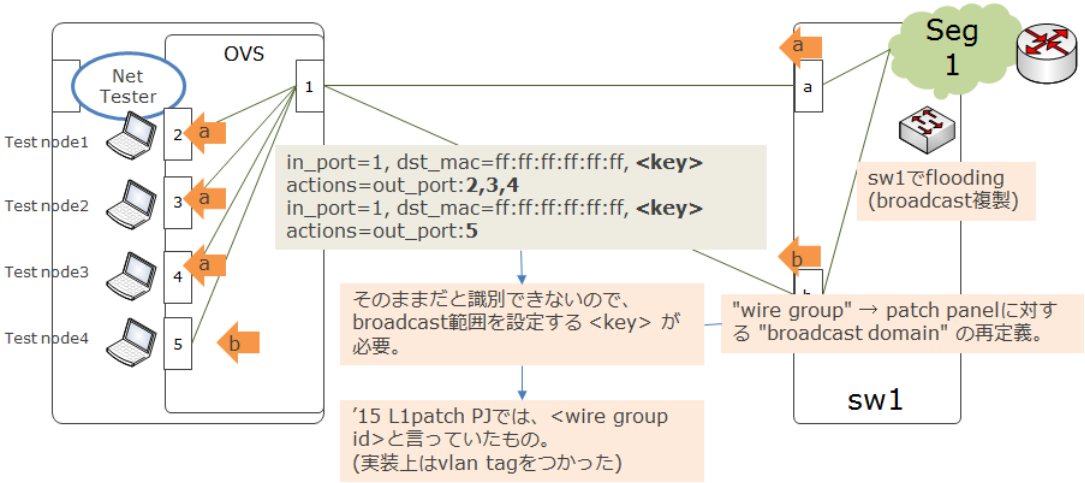

図 4.7　Broadcast 制御 案 2/Wire-group 制御

Wire-group には VLAN Tag を使用している。テスト対象ネットワーク内でつかう VLAN Tag とは異な

る使いかたなので、タグのつけはずしなどの処理をおこなう (詳細は L1patch プロジェクト技術情報 [4] を参照すること)。したがって、テスト用ノードの設定・パッチ設定に応じてフロールールが変動し、案 1 にくらべてフロー制御は複雑になる (図 4.8)。

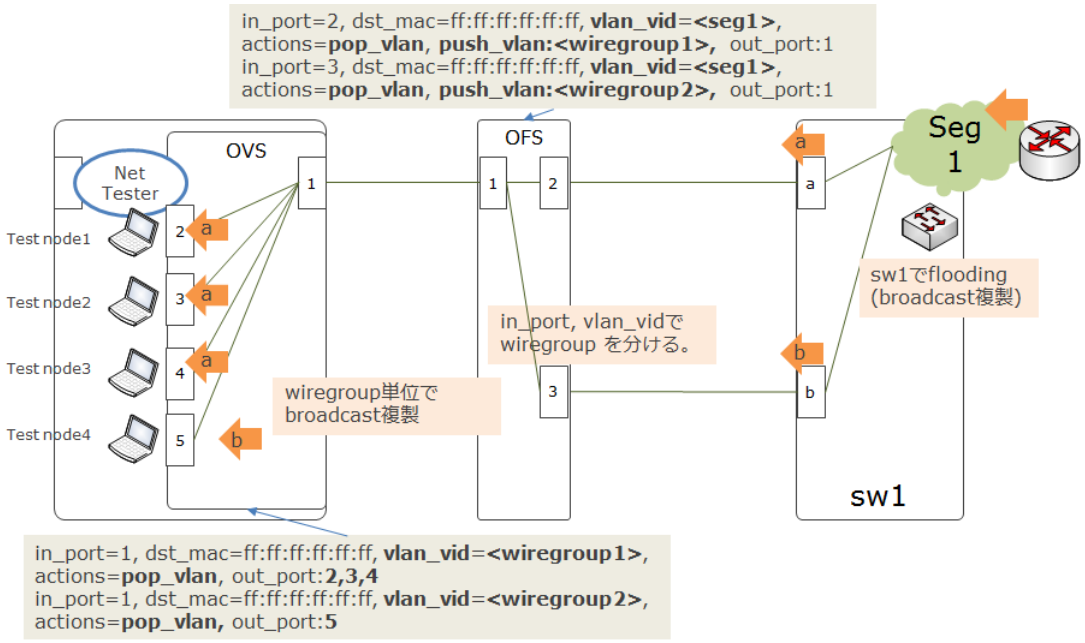

図 4.8　Broadcast 制御 案 2/Wire-group 制御 (実装)

4.3.4　Broadcast 制御案の比較と選択

案 1・案 2 のメリット・デメリットをまとめてみると表 4.1 のようになる。本プロジェクトでは、その目的 (3.1.2 節) からツールとしての機能・拡張性よりもユースケース実証に重きをおく。そのため、構成のシンプルさにみあった制御ルールのシンプルさが得られる案 1 を採用した。

表 4.1　L2 Broadcast 制御案 比較

案	メリット	デメリット
案 1	実装がシンプル	「パッチパネル」としての動作としては理想的ではない。ブロードキャストが多数あるとパケット複製処理のオーバーヘッドがおおきく性能面での影響が予想される。
案 2	「パッチパネル」として理想的な動作であり不要なブロードキャストの複製を抑制できる。	NetTester のシステム構成のシンプルさに対して実装が複雑になる。

複数 OpenFlow スイッチ・DUT ポート単位でのブロードキャスト制御 (案 2) については、L1patch プロジェクトで実証できている。本プロジェクトでユースケースを実証し、より複雑なシステム構成、拡張性が必要になった際は、案 2 を使用して制御ロジックを再実装することで実現が可能である。

4.3.5　フロー制御設計の注意事項

■Broadcast 制御の選択肢　テスト対象ネットワーク (Testee) からテスト用ノード (Tester) へ送信されるブロードキャストの複製についてみてきた。検討点としては Tester から Testee へのブロードキャストについて、ソフトウェアスイッチ内で複製させることも可能である (図 4.9)。

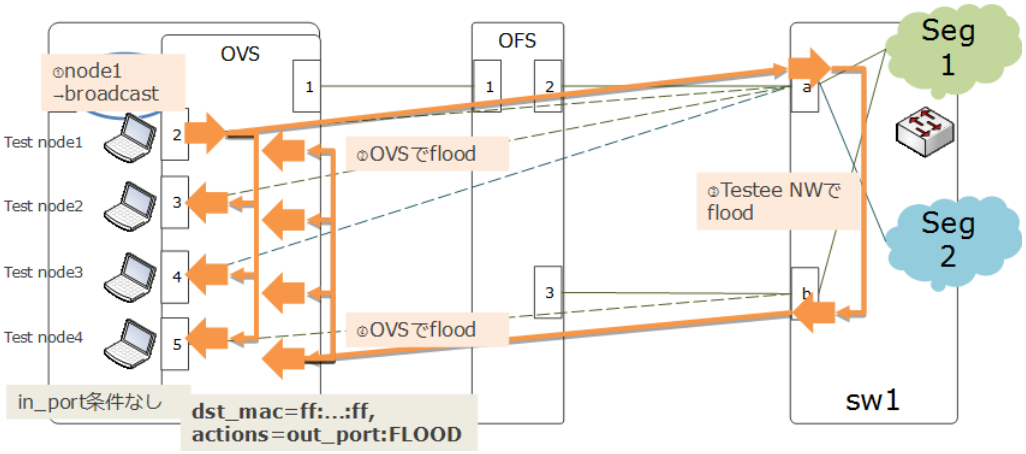

図 4.9　テスト用ノードのブロードキャスト制御パターン

図 4.9 ではソフトウェアスイッチ (OVS) では入力ポート (`in_port`) 条件を指定せず、ブロードキャストを常にフラッディングする。フロー制御ルールとしてはよりシンプルになるが、今回は以下の理由からソフトウェアスイッチ内でのブロードキャスト折り返しは採用していない。

- ソフトウェアスイッチ内だけでなく、テスト対象ネットワーク内でもブロードキャストが複製され、テスト用ノードに多数の (不用な) ブロードキャストフレームがコピーされる。
- 「パッチパネル」をモデルとしたトラフィック制御をしているため、理想としては、ある「パッチ」でやりとりされているデータは異なる「パッチ」で接続されるノードには見えてはいけない。
- テスト対象ネットワーク内でブロードキャストが正しくおこなわれ、テスト用ノードに送信されるかどうかも、テストとして確認したいことに含まれる。本来テスト対象ネットワークでおこなわれるべき処理を、テストシステム (ソフトウェアスイッチ) 側で代行してしまうのは、「ネットワークのテスト」という目的からは避けなければいけない。[*6]
- ソフトウェアスイッチと物理スイッチの間のリンクについて、NetTester では 1 本と設定しているが、複数本に拡張すると L2 ループとなる。構成次第ではあるが、無条件のフラッディングはブロードキャストストームに発展しやすいというリスクがある。

■設計上の注意事項　テストシステムとしては「パッチパネル」でテスト対象にテスト用ノードを直接接続した状況を忠実に再現できることが理想である。しかし、今回選択したフロー制御ルール (4.3.2 節; 案 1) では、

[*6] ソフトウェアスイッチでテスト用ノード間のブロードキャスト折り返しを許可してしまうと、テスト対象ネットワーク側の不具合などで L2 の接続ができない状態になっていても、あるテスト用ノードから隣にいるテスト用ノードの MAC が見えるという状況が発生し得る。これは従来の、独立した機器をテスト対象に直接結線していた状況では発生しない。テストシステムとしては、「本来パッチパネルで直接接続されていたときに想定されること」を再現することが望ましい。

テスト対象ネットワークからテスト用ノードへのブロードキャストについては本来の「パッチパネル」の動作からははずれた動作をする。具体的には、あるテスト用ノードが接続されている「パッチ」には流れないはずのブロードキャストも、ソフトウェアスイッチで複製されてしまう。そのため、本来見えないはずの隣接ノードの情報 (MAC アドレス) なども見えてしまう。

本プロジェクトでは、PoC(ユースケースの実証) を主要な目的とし、実装上は可能な限りシンプルにすることを方針としたため、こうした動作については許容することとした。また、L1patch プロジェクトでは wire-group によるブロードキャスト範囲の制限を試みており、より詳細なコントロールも可能であることも理由として挙げられる。

4.4　トラフィック転送と VLAN の制御設計

4.3.1 節に示したように、構成上の物理リソース (物理ポート/リンク) 消費を抑えるために、テスト対象ネットワーク側で VLAN を利用し、ひとつの物理ポートで複数のセグメントを集約する。このときのユニキャストの制御およびブロードキャストの制御 (案 1: 4.3.2 参照) を検討する。

4.4.1　VLAN 利用の要件

例として、4 台のテスト用ノードがテスト対象ネットワークの 2 つのセグメントにパッチ接続されている状況を想定する (図 4.10)。このとき、ポート (a) は vlan tagged port (trunk port)、ポート (b) は untagged port (access port) である。ノード 1-2 は Seg.1、ノード 3 は Seg.2 に所属している。また、sw1/Seg.1 内の折り返し通信をおこなうため、ノード 4 をポート (b) に配置する。

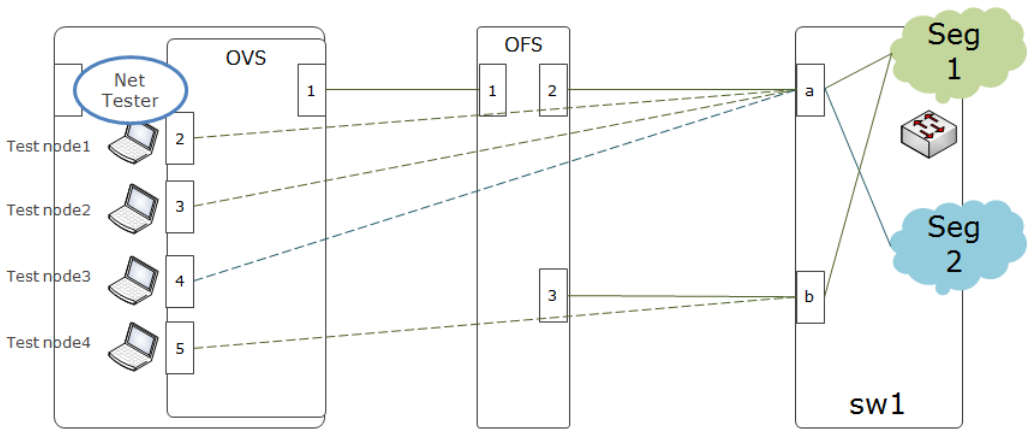

図 4.10　VLAN 制御の要件

VLAN Tag をどこでテスト用ノードで制御することも可能だが、L1patch プロジェクトでは以下の理由から OpenFlow スイッチ (物理スイッチおよびソフトウェアスイッチ) で VLAN Tag の制御をおこなっていた。NetTester の実装もこれに従っている。

- テスト用ノードとして VLAN Tag を利用できないノードも想定する。
- Wire-group(4.3.3 節) をつかった制御では、wire-group の情報を VLAN Tag を使用して実装する。そのため、VLAN Tag の制御を物理スイッチ・ソフトウェアスイッチの両方でおこなう必要がある。

4.4.2　Tester から Testee へ向かうトラフィックの VLAN 制御

Tester(テスト用ノード) が送信したトラフィックを Testee(テスト対象ネットワーク) に送信する場合、ユニキャスト/ブロードキャストともに送信元の情報 (Source MAC) が一意に定まる。パッチとしての接続先 (物理スイッチのポート番号) は与えられるため、これをもとに転送制御できる (図 4.11)。

ポート (a) に接続するパッチについて、VLAN Tag は物理スイッチ/ソフトウェアスイッチどちらで設定してもよい。

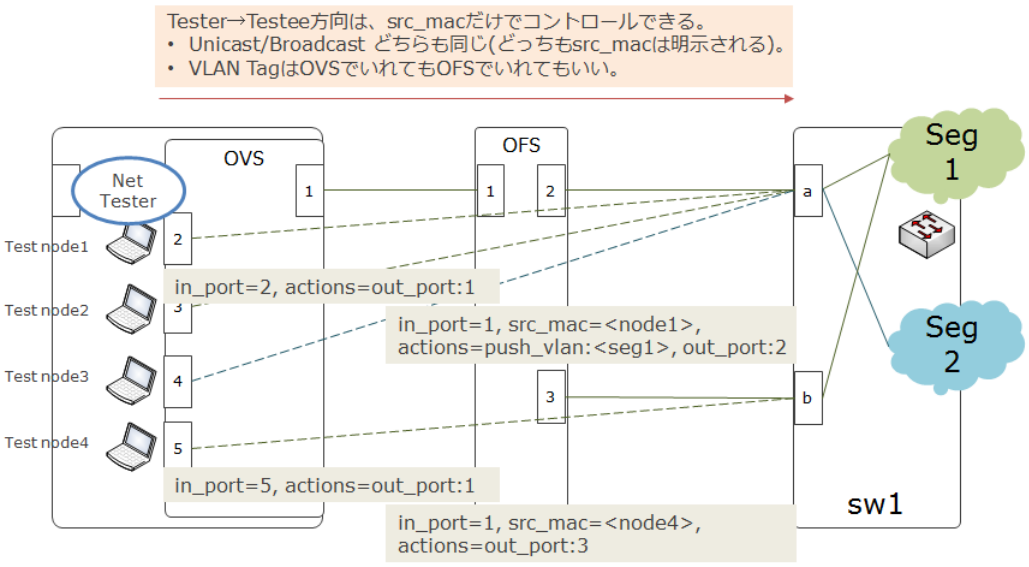

図 4.11　Testee へ向かうユニキャスト/ブロードキャストの VLAN 制御

4.4.3　Testee から Tester へ向かうユニキャストの VLAN 制御

Testee から Tester(テスト用ノード) へ向かうトラフィックは、ユニキャストとブロードキャストで制御が異なるため分けて扱う。Tester 宛てのユニキャストは、送信先 (Destination MAC) が一意に識別できるため、これをもとに転送制御できる (図 4.12)。

ポート (a) に接続するパッチについて、VLAN Tag は物理スイッチ/ソフトウェアスイッチどちらで設定してもよい。

4.4.4　Testee から Tester へ向かうブロードキャストの VLAN 制御

Tester(テスト用ノード) に向かうブロードキャストは、宛先 (Destination MAC) がブロードキャストアドレスとなるため、一意には定まらない。パッチ (送信元情報: どこの Testee port から送信されるか) とセグメント (VLAN ID) をもとにブロードキャストの複製・転送先を判断する必要がある (図 4.13)。

ブロードキャスト制御として案 1(4.3.2 節) を採用するため、ソフトウェアスイッチに設定するブロードキャスト制御のルールは、原則 `in_port=1, actions=FLOOD` のルールのみとなる。

図 4.13 では物理スイッチ側で Testee から送信されたフレームの VLAN Tag をはずす処理をしているが、

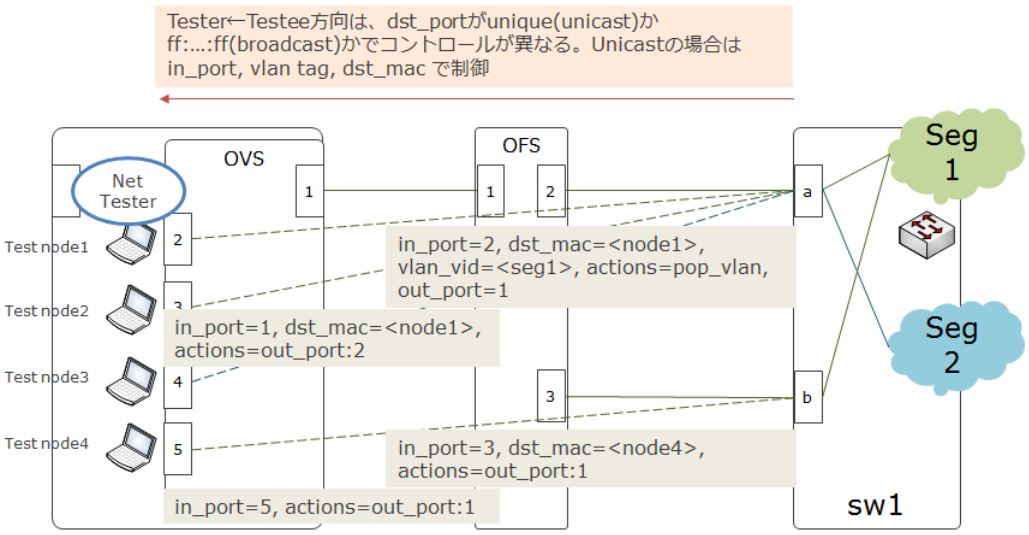

図 4.12　Tester へ向かうユニキャストの VLAN 制御

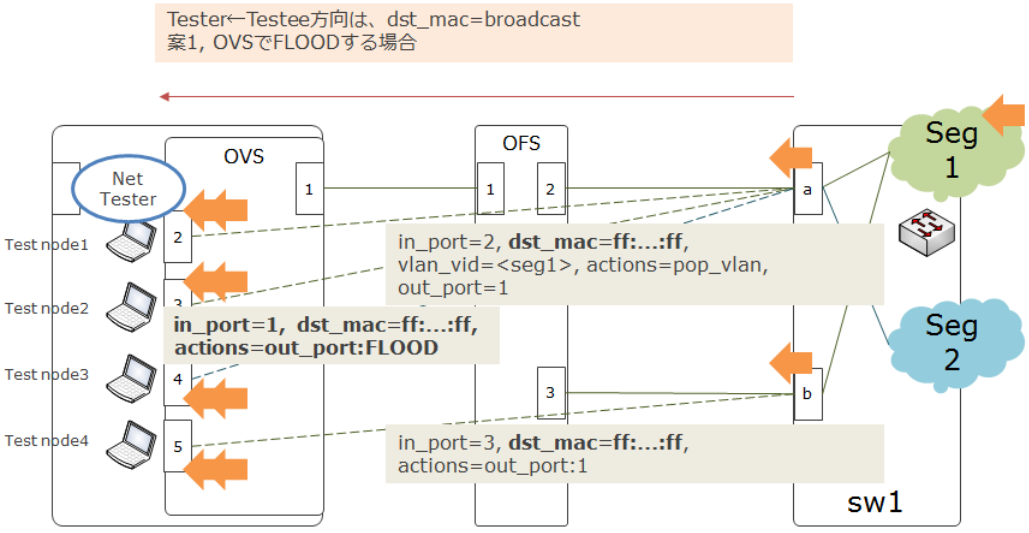

図 4.13　Tester へ向かうブロードキャストの VLAN 制御

どこで VLAN Tag のつけはずしをおこなうかによって実装が変化する (VLAN 制御ポイントや実装比較については 4.4.5 節)。いずれにせよソフトウェアスイッチで `actions=FLOOD` とするため、ブロードキャストはセグメントによらず全てのテスト用ノードへ送信される。

4.4.5　VLAN 制御ポイントの比較

4.4.1 節-4.4.4 節で示したように、ブロードキャスト制御案 1(4.3.2 節) の元では、VLAN tag の制御は物理スイッチ/ソフトウェアスイッチのどちらでも実装が可能である。表 4.2 にソフトウェアスイッチ (SSW) で VLAN Tag 操作する場合のフロールールを、表 4.3 に物理スイッチ (PSW) で VLAN Tag 操作する場合のフロールールを示す。NetTester では、以下の理由から SSW による VLAN Tag 操作を採用している。

- 物理スイッチはスイッチ OS やハードウェア (ネットワークチップ) などにより、製品ごとに機能差があることがある。今回は OpenFlow でも最も基本的な OpenFlow/1.0 および VLAN Tag 操作程度の機能しか使っていないため、リスクは少ないと考えられるが、複雑な処理が確実に動作するソフトウェア処理の SSW 側に寄せる。
- SSW 側で VLAN Tag 操作をおこなうほうが、システム全体のフロールールをシンプルにできる (フローの設定項目を少なくできる)。

表 4.2　SSW で VLAN Tag 操作をおこなう場合

Direction	Type	VLAN	Flow rule @SSW	Flow rule @PSW
Tester to Testee	Unicast & Broadcast	None	(1) `in_port=#p, actions=out_port:1`	(2) `in_port=1, src_mac=#host_mac, actions=out_port:#q`
		Exist	(3) `in_port=#p, actions=SetVlanVid:#v,out_por`	(4) `in_port=1, src_mac=#host_mac, vlan_vid=#v, actions=out_port:#q`, (2) に含まれる。
Testee to Tester	Unicast	None	(5) `in_port=1, dst_mac=#host_mac, actions=out_port:#p`	(6) `in_port=#q, actions=out_port:1`
		Exist	(7) `in_port=1, dst_mac=#host_mac, vlan_vid=#v, actions=StripVlanHeader,out_por`	(8) `in_port=#q, dst_mac=#host_mac, vlan_vid=#v, actions=out_port:1`, (6) に含まれる
	Broadcast	None	(9) `in_port=1, dst_mac=[broadcast], actions=out_port:FLOOD`, per-switch rule	(10) `in_port=#q, dst_mac=[broadcast], actions=out_port:1`, (6) に含まれる
		Exist	(11) `in_port=1, dst_mac=[broadcast], vlan_vid=#v, actions=StripVlanHeader, out_port:FLOOD`, per-wire rule, non-tag broadcast rule (9) より優先にならないと tag strip されずにフラッディングされる。	(12) `in_port=#q, dst_mac=[broadcast], vlan_vid=#v, actions=out_port:1`, (6) に含まれる

`#p`, `#q`: ポート番号
`#v`: VLAN ID
`#host_mac`: テスト用ノードの MAC アドレス
`[broadcast]`: Broadcast MAC Address (`ff:ff:ff:ff:ff:ff`)

表 4.3　PSW で VLAN Tag 操作をおこなう場合

Direction	Type	VLAN	Flow rule @SSW	Flow rule @PSW
Tester to Testee	Unicast & Broadcast	None	(1) `in_port=#p, actions=out_port:1`	(2) `in_port=1, src_mac=#host_mac, actions=out_port:#q`
		Exist	(3) `in_port=#p, actions=out_port:1`, (1) 同様。	(4) `in_port=1, src_mac=#host_mac, actions=SetVlanVid:#v, out_port:#q`
Testee to Tester	Unicast	None	(5) `in_port=1, dst_mac=#host_mac, actions=out_port:#p`	(6) `in_port=#q, actions=out_port:1`
		Exist	(7) `in_port=1, dst_mac=#host_mac, vlan_vid=#v, actions=out_port:#p`, (5) に含まれる。	(8) `in_port=#q, dst_mac=#host_mac, vlan_vid=#v, actions=StripVlanHeader,out_port:1`
	Broadcast	None	(9) `in_port=1, dst_mac=[broadcast], actions=out_port:FLOOD`, per-switch rule	(10) `in_port=#q, dst_mac=[broadcast], actions=StripVlanHeader, out_port:1`, per-wire rule
		Exist	(11) `in_port=1, dst_mac=[broadcast], actions=out_port:FLOOD`, (9) 同様。	(12) `in_port=#q, dst_mac=[broadcast], vlan_vid=#v, actions=StripVlanHeader,out_port:1`, per-wire rule, non-tag broadcast rule (10) より優先にならないと tag strip されずにフラッディングされる。

`#p`, `#q`: ポート番号
`#v`: VLAN ID
`#host_mac`: テスト用ノードの MAC アドレス
`[broadcast]`: Broadcast MAC Address (`ff:ff:ff:ff:ff:ff`)

4.5　フロー優先度設計

NetTester は OpenFlow/1.0 を使用するため、ひとつのフローテーブル内で優先度によるマッチ条件の使いわけをおこなう (4.2.2 節)。一般的には、マッチ項目が多いルール (より複雑なマッチ条件を持つルール) を優先させる必要がある。

NetTester のフロー制御ルールは表 4.2 を採用するが、VLAN Tag がある場合・ない場合でブロードキャストのルールを使いわけ、Tag をはずすアクションを優先して実行する必要がある。そのため、表 4.2(11) は VLAN Tag をもたないブロードキャスト制御ルールとの優先度をわける。OpenFlow ではマッチルールにワイルドカードのような機能がないため、使用する VLAN ID それぞれについてマッチさせる必要がある。

表 4.4　フロールール優先度設計

優先度	フロールール種別	Flow rule @SSW [1]	Flow Rule @PSW [1]
高	Tagged broadcast/unicast	(3),(7),(11)	(6)
中	Untagged broadcast/unicast	(1),(5),(9)	(2)
低	Layer1 [2]		
0(最小)	default(match-any)	`actions=DROP` or `actions=CONTROLLER`	

[1] 表 4.2 No.
[2]「専有モードワイヤ」(4.2.1 節)

OpenFlow/1.0 では、テーブルミス[*7]のとき、デフォルトでコントローラに送信[*8]される。この動作はこのあとの OpenFlow バージョンでは変更されているので、NetTester OFC では動作を統一させるために明示的に指定する。当初、デバッグ用途もあり `actions=CONTROLLER` としていたが、Trema のパケットパーサ (Trema/Pio) で特定のパケットがパースできず、コントローラが停止するバグがあった[*9]ため、NetTester では default DROP を採用している。

4.6　リンク操作方式

表 3.7 No.3 では物理リンク (物理トポロジ) 操作をあげた。これは、複数のポートをつなぎあわせて (「パッチ」して) テスト対象ネットワークの物理トポロジを構成するという観点だけでなく、テスト中に発生する物理リンク操作という観点も含まれる (3.2.1 節)。ここでは「動的なふるまい」のテストのための物理リンク操作方法について解説する。

テスト対象ネットワークでは図 4.14 のように、物理 OpenFlow スイッチを仲介させることで物理トポロジ操作を可能にする。このとき、「動的なふるまい」を発生させるために、NW 機器間のリンクを操作する。あたらしくリンクを追加する場合は同様に物理 OpenFlow スイッチへの結線とフロールールの追加をおこなう。リンクを削除する場合には以下の 2 パターンが考えられる。

直接リンクダウン　物理 OpenFlow スイッチのポート自体を up/down させることでリンクを操作する。こ

[*7] フローテーブル内のどのフロールールにもマッチしない場合。
[*8] `actions=CONTROLLER` : OFC への Packet-in 送信
[*9] 修正済。デバッグについての補足情報については B.2 節を参照。

の場合、テスト対象ネットワーク機器側でもポートの up/down を直接検出することができる。

関節リンクダウン　物理 OpenFlow スイッチで、リンクを成立させているポート間マッピングのためのフロールールを add/delete する。この場合、テスト対象ネットワーク機器側ではリンクの状態変更 (up/down) は発生せず、トラフィックが送受信できない状態となる。

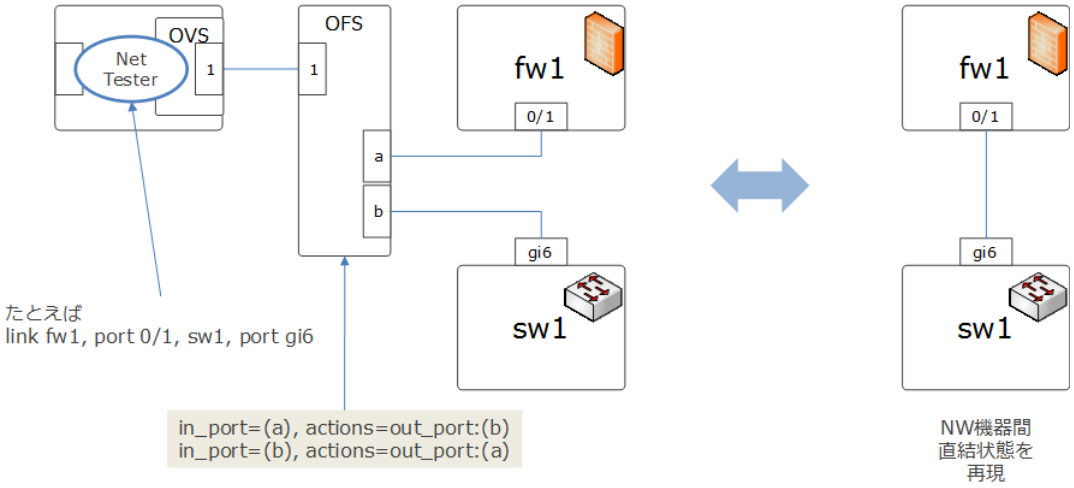

図 4.14　テスト対象の物理トポロジ操作

本プロジェクトでは、最も基礎となる「動的なふるまい」とそのテストをターゲットとし、リンク障害試験をそのユースケースとしてとりあげた。リンク障害は通常、テスト対象ネットワーク内 (該当するリンクをもつネットワーク機器) でポートの up/down を検出するため、直接リンクダウン方式を採用する。

直接リンクダウンの実装としては、以下の 2 つの方法がある。

- 物理 OpenFlow スイッチ上でポート状態操作コマンドを実行する。
- OpenFlow Message (PortMod) を物理 OpenFlow スイッチに送信し、ポート状態変更操作をおこなう。

今回、NetTester では、物理 OpenFlow スイッチに Expect ベースのツール (B.1 節) でログイン/コマンド実行し、直接スイッチ上でポート状態変更する方法をとった。これは、PoC 実施時点[*10]の Trema ではまだ PortMod メッセージの送信が実装されていなかったためである。

[*10] 2016 年 10 月-12 月

第 5 章

NetTester の使いかた

NetTester によるテスト環境の構築方法やシステム要求、本 PoC での実際に使用した構成や機器・ソフトウェアの情報、基本的な使用方法について解説する。

5.1 NetTester のデプロイ

5.1.1 物理 OpenFlow スイッチ

4.1.1 節に示したとおり、NetTester は 1 台の物理 OpenFlow スイッチを利用して、テスト対象ネットワークへの接続をおこなう (物理 OpenFlow スイッチを使用してテストトラフィックを “distribute” する)。本節では物理 OpenFlow スイッチの選択と設定について解説する。

■物理 OpenFlow スイッチの要件と選択 NetTester が使用する物理 OpenFlow スイッチには表 5.1 の要件 [3] が求められる。

表 5.1 OpenFlow スイッチ要件 (OpenFlow/1.0)

分類	項目
Match	In-Port
	VLAN ID
	Source/Destination MAC Address
Action	Out-Port
	Push/Pop VLAN ID
Other	Priority

本プロジェクトでは昨年度実施した L1patch プロジェクトの環境を継続しており、表 5.2 に示す OpenFlow スイッチを使用している。(2 台の OpenFlow スイッチを使用して Tester set[*1] を 2 セット構築している。)

■物理 OpenFlow スイッチの設定 PicOS は OVS Mode (Open vSwitch) で使用する。物理スイッチ (PicOS OVS) では以下の設定をおこなう。下記設定事項については PicOS マニュアルおよび Open vSwitch ドキュメント [29] も参照すること。

[*1] 6.4 節参照。

表 5.2　物理 OpenFlow スイッチ

Host name	Hardware	OS/Version
OFS1	Quanta T1048-LB9	PicOS 2.5.2 / Revision 19975
OFS2	Pica8 P-3290	PicOS 2.2.1S3 / Revision 14775

- パッチとして使用するすべてのポートの設定 [4]
  - VLAN の操作をおこなうため、VLAN Mode を trunk として設定 (`vlan_mode=trunk`) する:
  - テスト対象ネットワークで発生するブロードキャスト転送の抑制・STP への干渉をさけるため、フラッディングの無効化 (`no-flood`) および STP の無効化 (`no-stp`) を設定する。
- 物理 OpenFlow スイッチ (OVS Bridge) 全体の設定
  - Bridge で STP を処理しない (`stp_enable=false`)。
  - OFC との接続が切断された際にテスト対象ネットワークから届くトラフィックを処理しない (`fail_mode=secure`)[*2][*3]。
  - NetTester では、利用する物理 OpenFlow スイッチの DPID を指定できる (5.2.1 節)。PicOS ではリスト 5.1 のように OFC(NetTester Controller) 情報および DPID の設定をおこなう[*4]。
  - (Optional) OFS と OFC の接続に時間がかかる場合、OVS の `max_backoff`[*5]値を調整する [30]。
  - (Optional) OFS と OFC との接続がタイミングにより切断されるなどの現象がある場合には、`inactivity_probe`[*6] の値を調整する。

リスト 5.1　物理スイッチの OpenFlow 設定

```
1 ovs-vsctl set bridge br0 other-config:datapath-id=0000000000000001
2 ovs-vsctl set-controller br0 tcp:[NetTester Server Mgmt IP]:6653
```

5.1.2　NetTester Server の構成選択

NetTester サーバは物理 OpenFlow と物理リンクで直結される必要がある。サーバの構成方法として、ベアメタル構築する場合と仮想マシンで構築する場合の注意事項について解説する。

■ベアメタル構成　NetTester サーバをベアメタルで構成する場合は図 5.1 のようになる。4.1.1 節に示したとおり、NetTester サーバと物理スイッチ (PSW) 間は物理リンクで直結される必要がある。また、OpenFlow チャネルや NetTester-PSW 間の制御・管理通信のために、NetTester サーバと PSW は管理ネットワークを介して通信をおこなう。

[*2] `fail_mode` には secure と standalone の 2 種類がある。Standalone を指定した場合、OFS(OVS Bridge) は OFC との接続がきれた際に独立した L2 スイッチ (Learning Switch) として動作する。テスト対象ネットワークとの物理結線がある状態で L2 スイッチとして動作してしまうと、テスト対象ネットワークをまきこんで L2 ループが発生してしまうため注意が必要である。

[*3] PicOS では、PicOS バージョンが異なると (PicOS 上の OVS のバージョンが同一でも)OVS の `fail_mode` デフォルト設定が異なっているため注意が必要である。利用する場合は明示的に `fail_mode=secure` を設定すること。

[*4] PoC では物理 OpenFlow スイッチの DPID を `0x1` としている。

[*5] OFS が OFC に接続を試みる際のインターバルの指定。ミリ秒単位で指定でき、最小は 1 秒 (`max_backoff=1000`) である。

[*6] OFS-OFC 間接続の中断時間 (idle time) の最大値。Inactivity probe で設定した時間 OFS-OFC 間での通信が発生していない場合、OFS は OFC へ probe を送信する (OFC が probe に応答しない場合はコネクション切断と見なして再接続をおこなう)。ミリ秒単位で指定する。0 を指定することで設定が無効化される。

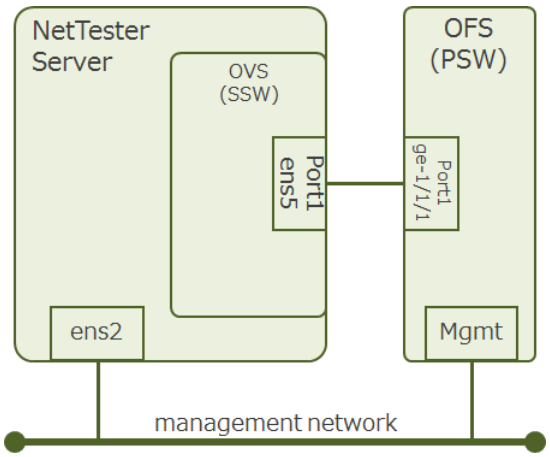

図 5.1　ベアメタル構成

■仮想マシン構成　NetTester サーバを仮想マシンとして構成する場合は図 5.2 のようになる。管理ネットワークについてはベアメタル構成の場合と同様に L2/L3 で PSW とのコネクティビティがとれればよい。NetTester(SSW)-PSW 間の接続については、ホスト OS 側で物理ポート (リンク) を NetTester サーバ (VM) へ直結させる必要がある。SSW-PSW 間は OFC(NetTester) によって制御するため、ハイパーバイザ側では L2 以上の制御はおこなわない。パススルーあるいはプロミスキャスモードで接続する。

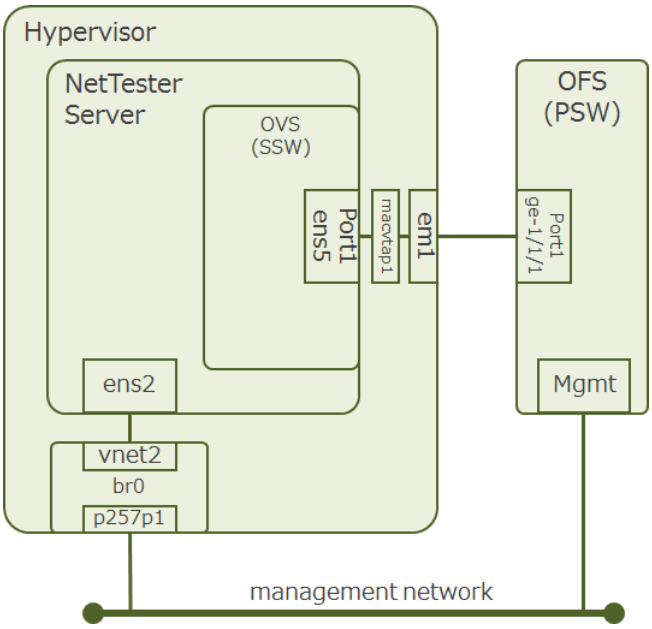

図 5.2　仮想マシン構成

本プロジェクトは仮想マシン構成を採用した。PoC では表 5.3 に示すサーバを使用している。SSW-PSW 間接続では、ハイパーバイザ (KVM Host) が持つ物理 NIC を直接仮想マシンに接続 (`macvtap/passthrough`) させる (リスト 5.2)。管理ネットワークについては bridge 接続でよい (他の VM やハイパーバイザと L2 で

接続する，リスト 5.3)。

表 5.3　NetTester サーバ情報

Host	OS	Virtualization
Hypervisor (KVM Host)	Ubuntu 14.04.5 LTS (GNU/Linux 3.16.0-30-generic x86_64)	qemu-kvm/2.0.0+dfsg-2ubuntu1.25, libvirt/1.2.2-0ubuntu13.1.17
NetTester Server (VM)	Ubuntu 16.04.1 LTS (GNU/Linux 4.4.0-31-generic x86_64)	

リスト 5.2　PSW 接続用ポート設定

```
<interface type='direct'>
  <mac address='52:54:00:12:56:0e'/>
  <source dev='em1' mode='passthrough'/>
  <target dev='macvtap1'/>
  <model type='virtio'/>
  <alias name='net1'/>
  <address type='pci' domain='0x0000' bus='0x00' slot='0x05' function='0x0'/>
</interface>
```

リスト 5.3　管理ポート設定

```
<interface type='bridge'>
  <mac address='52:54:00:64:ee:38'/>
  <source bridge='br0'/>
  <target dev='vnet2'/>
  <model type='virtio'/>
  <alias name='net0'/>
  <address type='pci' domain='0x0000' bus='0x00' slot='0x02' function='0x0'/>
</interface>
```

5.1.3　NetTester Server のソフトウェア構成

■NetTester を使用するために必要なソフトウェア　NetTester [24] を使用するための必須ソフトウェアと PoC で使用したバージョンについて表 5.4 に示す。使用するソフトウェアの導入にあたっての注意事項を以下に示す。

- 表 5.4 以外に必要な一般的な開発ツール等については Build Essential [32] などで導入しておく[*7]こと。
- NetTester で使用する Ruby package(gem) は bundler で自動的にインストールされる。Gem については NetTester リポジトリ内 `Gemfile` 等を参照すること。
- Ruby については rbenv [33] などで複数の ruby バージョンを管理する環境でもよい。
- NetTester はバックエンドで `ip` コマンドを使用して Network Namespace の操作をおこなっている[*8]。

[*7] `sudo apt install build-essential`
[*8] 正確には NetTester が利用している Trema/Phut が `ip` コマンドを使用している。B.3 節参照。

このとき root 権限が必要になり sudo を使用して実行する。sudo 利用にあたって PATH に関するエラーが発生する場合は sudo 設定ファイル (/etc/sudoers) の secure_path 設定を修正すること。

表 5.4　NetTester Server のソフトウェア構成

Software	Version (PoC)	Notes
OS (Linux)	Ubuntu 16.04 (GNU/Linux 4.4.0), 表 5.3 参照	Linux Network Namespace が使用できること。
iproute2	4.3.0	
ethtool	4.5	NIC オフロード機能操作
Git	2.7.4	
Open vSwitch	2.5.0	SSW として使用する。表 5.1 を満たすこと。
Ruby	2.3.1p112	2.0 以上が必要。2.3 以上を推奨。

■Open vSwitch(SSW) の設定　NetTester サーバ上のソフトウェアスイッチは、NetTester 起動時に都度 NetTester が設定をおこなうため、事前に設定等をおこなう必要はない。NetTester を起動すると、4.1.2 節に示した DPID で Open vSwitch のブリッジが作成され、OFC との接続設定がおこなわれる。

■NIC のハードウェアオフロード機能操作　NetTester をつかったテストシナリオの実装・テスト実行にあたって、以下の条件でテスト用ノードが生成している TCP パケットのチェックサムが incorrect となり、正しく通信がおこなえないという事象がおきた。

- テスト用ノードをテスト対象ネットワークに trunk port で接続させる。(OVS で VLAN Tag を操作する。)
- テストトラフィックとして TCP 通信をおこなう。
  - SYN/FIN フラグがついたトラフィックについては問題が発生しない。(TCP の特定のフラグがついたパケットで問題がおきる。)
  - ICMP 通信については問題が発生しない。

根本的な原因究明はできていないが、ワークアラウンドとして、NetTester でテスト用ノード (Netns) を生成した際に、テスト用ノードへ接続する NIC(veth) でチェックサム計算のハードウェアオフロードをオフにしている [35]。

5.1.4　テストシナリオ (net-tester/examples) のインストール

本プロジェクトでは、NetTester を使用したネットワークのテストシナリオを NetTester Examples [25] として整備している。

テストシナリオをインストールする場合はリスト 5.4 の手順を実行する。テストシナリオの中で NetTester はテスト用ノード操作をおこなう際に使用するツールのひとつとなるため、bundler により自動的にインストールがおこなわれる。NetTester 以外にテストシナリオ (Cucumber で記述) 内で使用する ruby gems についても、リポジトリ内 Gemfile 等で管理される。

リスト 5.4　テストシナリオのインストール

```
1 git clone https://github.com/net-tester/examples.git
2 cd examples
3 bundle install --binstubs --path=vendor/bundle
```

このほかに、テストシナリオでは、NetTester を使って生成・配置したテスト用ノードの上で、さらにテストトラフィックを生成し end-to-end の通信状況を見ていくためのツールが必要になる。PoC テストシナリオでは表 5.5 に示すツールを使用している。2017 年 1 月時点で、テストシナリオ内で使用する (bundler で管理できる ruby package ではない) ツールについては統一して管理することはできておらず、別途手動で必要なツールをインストールする必要がある。

表 5.5　PoC テストシナリオ内で使用するツール

Tool(package)	Version	テストシナリオ内での使いかた
netcat(nc)	1.105	汎用/簡易 TCP サーバとして使用
dnsmasq	2.75	DNS 通信テスト用の軽量 DNS サーバとして使用
curl	7.47.0	HTTP(HTTPS) アクセス用クライアントとして使用
openssl	1.0.2g	HTTPS サーバで使用
openssh	7.2p2	SSH サーバ/クライアントで使用

5.1.5　NetTester のインストール

テストシナリオを中心にネットワークのテストを開発する場合は 5.1.4 節のとおり、テストプロジェクト (リポジトリ) が使用するツールのひとつとして NetTester が含まれる。

NetTester を直接 (単体で) 使用して簡単なテスト操作を行なうこともできる。そうした場合にはリスト 5.5 の手順で NetTester 単体でのインストールをおこなう。

リスト 5.5　NetTester のインストール

```
1 git clone https://github.com/net-tester/net-tester.git
2 cd net-tester
3 bundle install --binstubs --path=vendor/bundle
```

5.2　基本的な使い方

5.2.1　NetTester が使用する環境変数

NetTester は以下の 2 つの環境変数を使用する。

`DEVICE`　NetTester サーバ上で、SSW-PSW 間を接続するために使用する NIC を指定する。
`DPID`　物理 OpenFlow スイッチ (PSW) の Datapath ID を指定する。

図 5.2 の場合は `DEVICE=ens5`、`DPID=0x1` となる。テストシナリオ (NetTester Examples) では `rake` により複数のテストシナリオの一括実行 (回帰テスト) を実行できる。`rake` 実行時には次のように環境変数を指定

して実行する。

```
rake DEVICE=ens5 DPID=0x1 cucumber
```

この他にも、実際のテストシナリオ実行の際に、テスト対象ネットワーク機器操作のために使うツール (B.1 節) でも環境変数を使用するので注意すること。

5.2.2　NetTester API の基礎

NetTester を使った処理の基本的な流れと、API の基本的な使用方法を例をもとに解説する。

■サンプル環境構成　いま図 5.3 のように、テスト対象ネットワークにあるふたつの L2 スイッチ (L2SW) に、それぞれ 1 台ずつテスト用ノードを接続してテスト作業をおこなうことを想定する。このとき、テスト用ノードの生成および配置・接続は NetTester を使用しておこなう。また、テスト用ノードで実施するテスト作業は Pry [34] を使用して ad-hoc に (手動で) 実施する。

これを NetTester API を用いて実装するとリスト 5.6 のようになる[*9]。

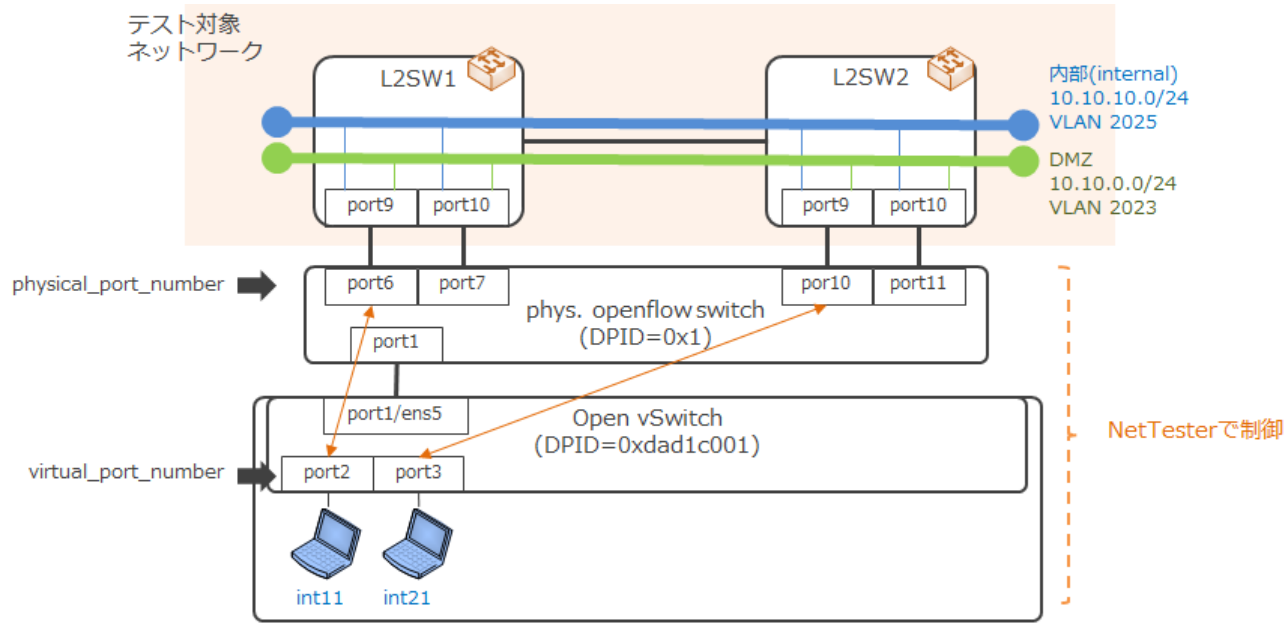

図 5.3　NetTester 利用例 (構成図)

リスト 5.6　NetTester 基礎

```
#!/usr/bin/env ruby
# frozen_string_literal: true

require 'net_tester'
require 'pry'
```

[*9] このスクリプトは解説用にパラメタ設定などを冗長に記載している。実際にテストシナリオ等を実装する場合は、より無駄のない形で記述することができる (7.4.5 節参照)。

```
6
7  # parameter definition
8  nwdev = 'ens5'
9  pss_dpid = 0x1
10 mac_base = '00:ba:dc:ab:1e:'
11
12 p '# run net_tester'
13 NetTester.run(network_device: nwdev,
14               physical_switch_dpid: pss_dpid)
15 p '# wait ssw/psw connection'
16 sleep 10
17
18 p '# Create host11 in internal segment (ip:10.10.10.7, vlan:2025)'
19 host11 = NetTester::Netns.new(name: 'host11',
20                               mac_address: mac_base + '01',
21                               ip_address: '10.10.10.7',
22                               netmask: 24,
23                               gateway: '10.10.10.254',
24                               virtual_port_number: 2,
25                               physical_port_number: 6,
26                               vlan_id: 2025)
27
28 p '# Create host21 in internal segment (ip:10.10.10.9, vlan:2025)'
29 host21 = NetTester::Netns.new(name: 'host21',
30                               mac_address: mac_base + '02',
31                               ip_address: '10.10.10.9',
32                               netmask: 24,
33                               gateway: '10.10.10.254',
34                               virtual_port_number: 3,
35                               physical_port_number: 10,
36                               vlan_id: 2025)
37
38 # into pry console
39 binding.pry
40
41 # cleanup
42 NetTester.kill
```

■NetTester によるテスト操作の流れ　NetTester を使ったネットワークのテスト操作は次のステップで実装される。

1. NetTester を起動する
   - `NetTester.run` (引数については 5.2.1 節参照)
   - 起動すると、SSW の設定と OFC の起動をおこなう。リスト 5.6 では OFC 起動後に SSW/PSW の接続を待つために `sleep` を設定している。
2. NetTester でテスト用ノードを生成する

- `NetTester::Netns.new` によって Network Namespace を使ったテスト用ノードを生成する。(詳細については B.3 節参照)
- 生成と同時にテスト対象ネットワークへの配置 (パッチ接続) をおこなう。引数 `virtual_port_number` および `physical_port_number` は「パッチ」の両端点となるポート番号をあらわしている。ここで指定された端点情報をもとに、NetTester は OpenFlow スイッチ (PSW, SSW) にフロールールを設定する。
- 図 5.3 で、L2SW 側のテスト用ノード接続ポートは Trunk Port (vlan tagged port) になっている。そのため、`vlan` オプションを指定して、テスト用ノードが生成するトラフィックに VLAN Tag を追加することを指定している。

3. 生成したテスト用ノード上で作業をおこなう
   - `NetTester::Netns#exec` (後述)
4. NetTester を終了する
   - `NetTester.kill`
   - NetTester が生成した SSW, テスト用ノードなどを削除する。テスト用ノード (`NetTester::Netns`) の実態は network namespace であるため、NetTster プロセスが消去されても OS 上に設定が残る。適切な終了 (削除) 処理がおこなわれない場合、OS 上にこれらの設定が残りつづけてしまう (他のテストシナリオ実行時に同名の namespace を作成しようとするとエラーになる) ため注意が必要である。

テスト用ノード上での操作については、このスクリプトでは pry によるインタラクティブシェル上での操作となる。生成したテスト用ノード上 (テスト用ノードの namespace 上) でコマンドを実行することでテスト作業をおこなう。テスト用ノード上でのコマンド実行には `NetTester::Netns#exec` を使用する。たとえば次のような操作を実施することができる。

テスト作業例

```
[31] pry(main)> puts host11.exec("ping -c3 #{host21.ip_address}")
PING 10.10.10.9 (10.10.10.9) 56(84) bytes of data.
64 bytes from 10.10.10.9: icmp_seq=1 ttl=64 time=0.514 ms
64 bytes from 10.10.10.9: icmp_seq=2 ttl=64 time=0.336 ms
64 bytes from 10.10.10.9: icmp_seq=3 ttl=64 time=0.311 ms


--- 10.10.10.9 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1998ms
rtt min/avg/max/mdev = 0.311/0.387/0.514/0.090 ms
=> nil
[32] pry(main)>
```

■Pry 応用　リスト 5.6 を拡張してテスト用ノードを多数配置し、Pry と NetTester を併用してインタラクティブにテスト作業をおこなうといった応用もできる。このような応用については本書では扱わない。別途資料 [28] を参照すること。

第 6 章

PoC ターゲット設計

PoC としてユースケース実証実験をおこなう際に設定した PoC シナリオ (状況設定)、PoC 環境について解説する。

6.1 PoC の概要

6.1.1 PoC の目的

L1patch プロジェクト [2] では network namespace によるテスト用ノード生成と OpenFlow スイッチによる配置 (パッチ) という、表 3.1 の No.3-4 に相当する技術の基礎検証を実施した。本プロジェクトでは、それらの基礎技術をもとに NetTester を実装し、実際のテスト (ユースケース) としてどういったテストが自動化可能か (表 3.1, No.5) を中心に実証していく。テストのユースケースとして、静的なふるまい・動的なふるまいのふたつの観点でテストシナリオの実装を進める (3.1.2 節)。

6.1.2 PoC ターゲットユースケース検討

ネットワークテストのユースケースとしてとして表 6.1 のような事例について検討した。

表 6.1 の事例をもとに、汎用性があり、かつ「テストの自動化」という観点で技術的に基礎となる機能を盛り込めるユースケースとして、以下の 2 ケースを選択した。いずれもテスト自動化の有効性を出しやすい条件を満たす (頻繁にくりかえしおこなう・複雑度が高く人によるレビュー等でのミスの発見が難しい・従来は人力にたよっており自動化が難しい) という観点で選択している。

- FW のパケットフィルタポリシ運用
  - 「静的なふるまい」の代表例として選択。
  - FW ポリシ管理 (通信制御ポリシ管理) は、パターン数 (ルール) が多く、順序の依存関係など複雑度の高い操作が求められるため。また日々の変更頻度が高く、運用管理コストの高い事例であるため。
  - 特にアプリケーションレイヤ (L7) を検査する DPI 機能を持つ FW のテストは単純な L3/L4 のツールでは自動化が難しい動作であるため。
- 冗長化構成 FW のリンク障害試験
  - 「動的なふるまい」の代表例として選択。(最も基本的な「動的なふるまい」の例としての物理トポ

ロジ (物理リンク) 操作。)

- 一般的に、FW(ハードウェアアプライアンス) はその機能や性能上の理由から製品ごとに固有のアーキテクチャを持つ。製品機能や性能を維持しつつ、FW を経由するセッションなどの情報 (状態) をクラスタとして保持するために、FW のクラスタ化では製品ごと固有の機能実装を持つ。そのため、製品ごとに固有の機能やバグがあり、実際本番環境で使用する実機 (ハードウェア) を使用したテストが重要になる。
- クラスタ化された FW のフェイルオーバー/フェイルバックはステートフルな動作となり、初期状態の設定、指定された手順を実行してネットワークの状態を遷移させていくオペレーションが必要となる。また、状態遷移中の動作を複数並列してチェックしていく必要がある。いずれも従来は複数人で作業することでおこなっていたものだが、複数人による作業は全体の動作などが掴みにくいという問題があった。

表 6.1　テストユースケース案

ユースケース	詳細 (事例)
ネットワーク移設時の通信不能	ある拠点から他の DC へのシステム移設の際、現行システムの IP を継続するために L2 延伸をおこなっていたが、上流側ネットワークでの経路消失により通信ができなくなった。
FW 通信誤許可	外部から http でアクセス不能なはずのシステムについて、実際には外部からの http でのアクセスが許可されていた。
システム拡張時の通信不安定	システムの拡張時に L2 ループが発生し通信が不安定になった。
FW の再起動による通信障害	Active/Standby 構成の FW で Standby 側が再起動したときに、Active 側が機能停止・冗長構成の情報交換ができず通信が停止した。
通信遅延の発生、ネットワークのスローダウン	L3 スイッチのリソース (メモリ・TCAM) 枯渇による重大なネットワーク性能低下、通信遅延が発生した。
経路制御設定ミスによるアクセス不能	サーバリプレースによりシステム側の IP アドレスが変更となった際、一部機器でルーティング追加作業が漏れていたためにシステムへアクセス不能となった。
FW のリプレース	古い FW 機器 (ハードウェア) を新しい世代の機器にリプレースする際、既存のルールのコンバートやインポートに問題があり一部の通信が停止した。
FW フィルタ (ポリシ) のミスチェック	パケットフィルタはルール (ポリシ) としては設定されているが、ルール作成者の認識違いやミスなどで最終的に実現したいポリシになっておらず、必要な通信の遮断・不要な通信の透過が発生してしまった。

6.1.3　PoC 方針

ここまでで既にいくつか方針をあげているが、あらためて PoC の実施方針をまとめる。

- 本 PoC で実施すること
  - テストシステムの機能・特定の自動化実装ではなく、「テストとして実行可能なユースケース」に

注目する。
  - ネットワークテストの基本的な考えかたとして「静的なふるまいのテスト」「動的なふるまいのテスト」の観点で実際のテストシナリオを実装する。
  -「ネットワークのふるまい」として、end-to-end の通信の実現可否に着目する。ネットワークの設計・内部構造・使用している機材に関わらず、ネットワークとしてどういった通信が実現されるべきか、という要求とそのテストに注目する。これは、ネットワークの利用者から見たネットワークに対する期待 (要求)、あるいはネットワーク (サービス) 提供者の視点から見たネットワークに対する期待[*1]を確認することである。
- 本 PoC で実施しないこと
  -「できないこと」のテスト: 利用者あるいはサービス提供者の視点では、ネットワーク上「できてはいけないこと」の要求がある。例えば、隣接する他者顧客環境と通信ができてはいけない、など。本 PoC はまず利用者に対して最低限提供すべきサービス (「できなければいけないこと」) についてのテストに焦点をあてる。
  - 非機能要件のテスト (性能・拡張性・冗長性): 上記のとおり、本 PoC ではネットワークテスト (ユースケース) の実現可否、機能的な実現性に注目する。そのため、性能や拡張性などについては考慮せず、ユースケースを実現可能な最低限のシステム構成・実装をとることを方針としている。

6.2　PoC ターゲットの設定

6.2.1　登場人物

PoC にあたって図 6.1 のように登場人物を設定した。

- ヨーヨーダイン社[*2]
  - タジマックス通信工業社と共同でソフトウェア開発をおこなっている。
  - 社内ネットワークの新規構築をおこなうにあたり、ネットワークの設計・構築についてもタジマックス通信工業社に発注することになった。
- タジマックス通信工業社[*3](以降 タジマックス社)
  - ソフトウェア開発および情報システム基盤の設計・構築などをおこなっている。
  - ヨーヨーダイン社の社内ネットワーク設計構築を受注している。

以降、ネットワークのテストシナリオについては、タジマックス社社員がテストをする視点をとる。(タジマックス社社員がネットワークの設計・構築をおこない、ヨーヨーダイン社へ納入する前に、ヨーヨーダイン社が業務をおこなう上で実現したいこと、ヨーヨーダイン社の要求すべてが問題なく実現できているかどうかをテストする。)

7 章で実際のテストシナリオ実装について解説するが、PoC では タジマックス社社員に相当するメンバとして次のように設定した。

[*1] サービス提供者として、ネットワーク利用者にどういったサービスが提供できなければいけないか。
[*2] 架空の企業名 [39]
[*3] 架空の企業名, プロジェクトメンバの氏名から。

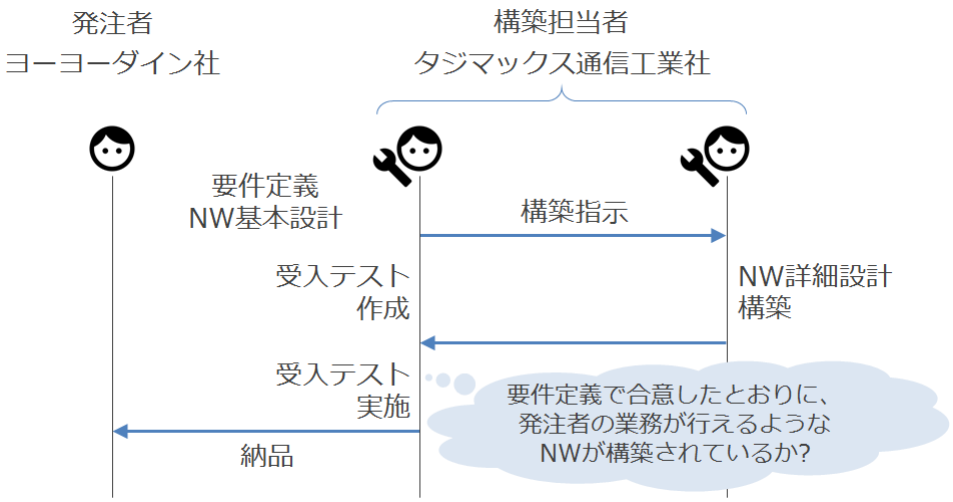

図 6.1　PoC 設定: 登場人物の位置付け

- ネットワークの要件定義・基本設計を行なうネットワークエンジニア
- 決められたネットワーク要求に基づいてネットワークの構築をおこなうネットワークエンジニア
- 決められたネットワーク要求に基づいてテストシナリオの実装・実行をおこなうソフトウェアエンジニア

ネットワークエンジニア担当の PoC メンバは、プログラミングの知識があり、テストシナリオの実装やテストシナリオ実行時に発生した問題のトラブルシュートが可能なスキルを持つ。テストシナリオ実装をおこなうソフトウェアエンジニアについては、ネットワーク (TCP/IP, Ethernet) の基本的な知識はあり、ネットワーク要件やそれに応じてどのようなテストを実行すればよいかを考えることができるが、ネットワーク機器の設定などを実際には (ほとんど) おこなったことのないメンバを設定している。

6.2.2　サービス要件定義

以下、PoC 上の状況設定をおこなう。なお、PoC を複雑化させないため、原則 L1-L4 までの機能的な要件設定とし、(可用性を除く) 非機能要件や主要な要件に付随する業務上の要求 (アクセスログ取得等セキュリティ観点の要求など) は設定していない。

■機能要件　ヨーヨーダイン社および タジマックス社が共同開発をおこなうための機能要件は次のようになる[*4]。

- ヨーヨーダイン社は社内に開発環境を持つ。開発環境は社内からのみアクセス可能とする。
- タジマックス社はインターネット経由で ヨーヨーダイン社社内にアクセスし、開発環境を共用してヨーヨーダイン社との開発業務をおこなう。
  - タジマックス社からのアクセスはセキュアな通信方法をとること (通信を暗号化すること)
  - タジマックス社からのアクセスについて個人単位での認証・アクセスコントロールができること。
- ヨーヨーダイン社が社内外に提供するサービスは、インターネット環境に直接は設置せず、上流ネット

[*4] 平成 25 年度のネットワークスペシャリスト試験の問題 [40] をもとに、架空の中小企業ネットワークとして設定している。

ワークで ヨーヨーダイン社による L4(以上) のアクセス制御を行なうこと。
- インターネット側から ヨーヨーダイン社社内への通信を許可しない。(ヨーヨーダイン社がインターネット側に提供するサービスを除く。)

■可用性　ネットワークの可用性については次のように設定する。

- インターネット回線の冗長化はおこなわない[*5]。
- 社内では中核をなす NW 機器の冗長化をおこなう。(機器メンテナンス作業や障害発生時の タジマックス社開発業務継続のため。)

6.3　環境構成 (Target Network)

6.3.1　論理ネットワーク設計

■利用者要件　タジマックス社は ヨーヨーダイン社とのネットワーク要件検討をもとに、図 6.2・表 6.2 のようなネットワークを構築することとした。

- ヨーヨーダイン社ネットワークは、外部 (社外/インターネット)・DMZ・内部 (社内) のセキュリティゾーンに分割する。
- 外部 (external) ゾーン: インターネット接続用の固定 IP セグメントをひとつ持ち、インターネット経由で外部へのサービスを提供できるようにする。
  - グローバル IP アドレスは外部ゾーンのみで使用する。DMZ および内部ゾーンではプライベート IP アドレスを使用する。社外 (インターネット) との必要な通信についてのみ、外部ゾーン境界 (FW) で NAT をおこなう。
- DMZ ゾーン: 外部に公開するサービスと内部に公開するサービスの中継をおこなう。
  - SSL VPN サーバ: タジマックス社との共同開発業務のために、インターネット経由で タジマックス社からのリモートアクセスを受け付ける。タジマックス社 VPN クライアントは VPN 接続後、VPN サーバから割り当てられた DMZ 内 IP を使用して内部ゾーンの開発環境にアクセスする。
  - DNS サーバ: ヨーヨーダイン社内部向けに提供する名前解決・サービス
- 内部 (internal) ゾーン: 社内のみ利用可能な開発系サーバおよび社員 (開発用 PC 等) を設置する。
  - 開発環境として、資産管理サーバ (Git)・内部テストサーバ (Telnet/Jenkins) を起く。
- アクセスポリシとして、原則、内部-外部ゾーン間の通信はおこなえないものとする。ただし、現状は内部ゾーン内機器からの NTP および HTTP/HTTPS 通信に関しては直接外部ゾーンへ通すことを許可する[*6]。

なお、タジマックス社の設計・構築範囲は ヨーヨーダイン社ネットワークである。タジマックス社ネットワークについては PoC の対象には含まれない。

■管理者要件　ネットワークや各種サービスを運用・管理するための要求を以下のように設定する。

[*5] PoC 設定上の都合。障害試験ポイントとして NW 機器 (FW) にのみ注目する。
[*6] タジマックス社は、ヨーヨーダイン社 NW 規模およびセキュリティポリシを検討した結果、現時点では Proxy サーバ導入をおこなわないものとした。

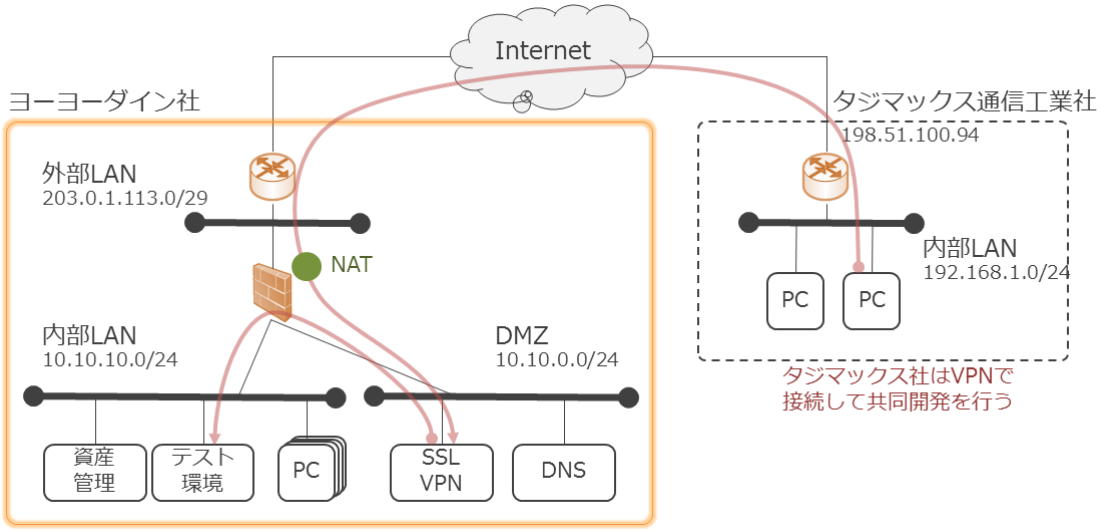

図 6.2　PoC 環境: 論理構成図

- サーバ管理アクセス: 内部/DMZ ゾーンにおかれるサーバについては、ヨーヨーダイン社開発者が管理者を兼任しており、内部ゾーンから直接リモートアクセスすることで運用・管理作業をおこなう。(in-band management)
- ネットワーク管理アクセス: ネットワーク機器の運用管理については、構築した タジマックス社社員をおくことを想定し、ヨーヨーダイン社ネットワークとは分離した設計とする。詳細については 6.4 節参照。

6.3.2　通信要件

利用者要件および管理者要件をふまえて、ネットワークで提供する通信要件は表 6.3・表 6.4・表 6.5 のようになる。

表 6.2　セグメント/IP アドレス一覧

Zone/Segment	IP Subnet	Local Host[1]	Host
外部	203.0.113.0/29	0	[Network]
		1	Router
		2	Firewall
		5	SSLVPN (タジマックス社社 VPN アクセス)
		6	NAPT (内部/DMZ から Internet へのアクセス)
		7	[Broadcast]
DMZ	10.10.0.0/24	0	[Network]
		1	Firewall (default gateway)
		10	DNS サーバ
		11	SSLVPN サーバ
		128-254	SSLVPN Client IP Pool
		255	[Broadcast]
内部	10.10.10.0/24	0	[Network]
		1	資産管理サーバ
		2	テスト環境サーバ
			開発用 PC
		254	Firewall (default gateway)
		255	[Broadcast]

[1] 第 4 オクテット

表 6.3　PoC 通信要件 (ヨーヨーダイン社内部セグメント起点)

No.	ポリシ	アプリケーション	Source IP		Destination IP		Destination Port
1	PC →開発環境	Git	PC	10.10.10.0/24	資産管理サーバ	10.10.10.1	tcp/11000
2	PC →開発環境	Telnet	PC	10.10.10.0/24	テスト環境サーバ	10.10.10.2	tcp/23
3	PC →開発環境	Jenkins	PC	10.10.10.0/24	テスト環境サーバ	10.10.10.2	tcp/13000
4	PC →開発環境	サーバ管理: ssh	PC	10.10.10.0/24	資産管理サーバ	10.10.10.1	tcp/22,80,443
5	PC →開発環境	サーバ管理: ssh	PC	10.10.10.0/24	テスト環境サーバ	10.10.10.2	tcp/22,80,443
6	PC → DNS サーバ	DNS Query	PC	10.10.10.0/24	DNS サーバ	10.10.0.10	tcp,udp/53
7	PC → Internet	Web browsing	PC	10.10.10.0/24	Internet	ANY	tcp 80,443
8	PC → Internet	NTP Query	PC	10.10.10.0/24	Internet	ANY	udp/123
9	PC → DNS サーバ	応答確認: ping/-traceroute	PC	10.10.10.0/24	DMZ	10.10.0/24	icmp
10	PC → Internet	応答確認: ping/-traceroute	PC	10.10.10.0/24	Internet	ANY	icmp
11	PC → DNS サーバ	サーバ管理: ssh	PC	10.10.10.0/24	DNS サーバ	10.10.0.10	tcp/22
12	PC → SSLVPN サーバ	サーバ管理: ssh, webui	PC	10.10.10.0/24	SSLVPN サーバ	10.10.0.11	tcp/22,80,443

ヨ社: ヨーヨーダイン社
タ社: タジマックス通信工業社

表 6.4　PoC 通信要件 (ヨーヨーダイン社 DMZ セグメント起点)

No.	ポリシ	アプリケーション	Source IP		Destination IP		Destination Port
13	DMZ → Internet	package update (web)	DMZ 内サーバ	10.10.0.0/25	Internet	ANY	tcp/80,443
14	DNS サーバ → DNS Query	上位 DNS へのクエリ	DNS サーバ	10.10.0.10	Internet	ANY	tcp,udp/53
15	DMZ → DNS サーバ	DNS Query	DMZ 内サーバ	10.10.0.0/25	DNS サーバ	10.10.0.10	tcp,udp/53
16	DMZ → NTP	NTP Query	DMZ 内サーバ	10.10.0.0/25	Internet	ANY	udp/123
17	PC → DNS サーバ	ヨ社内部	PC	10.10.10.0/24	DNS サーバ	10.10.0.10	tcp,udp/53
18	PC → DMZ	サーバ管理: ssh	PC	10.10.10.0/24	DMZ 内サーバ	10.10.0.0/25	tcp/22,80,443
19	VPNPOOL →開発環境	Git	DMZ VPN Pool	10.10.0.128/25	資産管理サーバ	10.10.10.1	tcp/11000
20	VPNPOOL →開発環境	Telnet	DMZ VPN Pool	10.10.0.128/25	テスト環境サーバ	10.10.10.2	tcp/23
21	VPNPOOL →開発環境	Jenkins	DMZ VPN Pool	10.10.0.128/25	テスト環境サーバ	10.10.10.2	tcp/13000
22	DMZ → Internet	応答確認: ping/-traceroute	DMZ 内サーバ	10.10.0.0/25	Internet	ANY	icmp
23	DMZ →ヨ社内部	応答確認: ping/-traceroute	ヨ社内部	10.10.10.0/24	DMZ 内サーバ	10.10.0.0/25	icmp
24	ヨ社内部→ DMZ	応答確認: ping/-traceroute	DMZ 内サーバ	10.10.0.0/25	ヨ社内部	10.10.10.0/24	icmp

ヨ社: ヨーヨーダイン社
タ社: タジマックス通信工業社

表 6.5　PoC 通信要件 (Internet/タジマックス社セグメント起点)

No.	ポリシ	アプリケーション	Source IP		Destination IP		Destination Port
26	Internet →外部	応答確認: ping/-traceroute	Internet	ANY	Router	203.0.113.1	icmp
27	Internet →外部	応答確認: ping/-traceroute	Internet	ANY	Firewall	203.0.113.2	icmp
28	外部→ Internet	応答確認: ping/-traceroute	ヨ社外部	203.0.113.0/29	Internet	ANY	icmp
29	PC →ヨ社 VPN	SSLVPN	タ社 (Global)	198.51.100.94	SSLVPN サーバ	203.0.113.5	tcp/80,443
30	PC →ヨ社 VPN	応答確認: ping/-traceroute	タ社 (Global)	198.51.100.94	SSLVPN サーバ	203.0.113.5	icmp

ヨ社: ヨーヨーダイン社
タ社: タジマックス通信工業社

6.3.3　物理ネットワーク設計

■物理設計の基本的な考えかた　論理ネットワーク設計 (6.3.1 節) をもとに、物理ネットワーク設計を図 6.3 のようにした。使用した機器の詳細については表 6.6 参照。

- セキュリティゾーンおよびゾーン間通信ポリシは Firewall によっておこなう。
  - 外部: WAN/Untrust Zone
  - DMZ: DMZ Zone
  - 内部: LAN/Trust Zone
- 外部ゾーンは FW 上流側 L3SW で収容する。L3SW は internet 境界 (キャリア回線終端) である。
- 内部ゾーンおよび DMZ は L2SW1/L2SW2 で収容する。物理リンクは共有し、VLAN によって論理的に分離する。

管理系ネットワークについてはサービス系ネットワークと分離する (6.4 節)。

■冗長性　社内ネットワーク冗長化の要求から Firewall を冗長化する。

- Firewall は Active/Passive 方式とする。障害が発生していない場合は FW1 が通常 Active となり、トラフィックの転送をおこなうものとする。
- Active 側 FW(FW1) のもつすべてのリンクについて、いずれかひとつでもリンクダウンが発生した場合に自動的に処理を Passive 側 (FW2) へ切り替える。
- 切替のトリガとなったリンクダウンが復旧し、FW1 のすべてのリンクが正常になった場合は、トラフィック処理を FW1 へ切り戻す。
- ステートフルフェイルオーバー: FW1/FW2 は常に処理しているセッションの情報を同期し、切替・切り戻しが発生した際でも処理中のトラフィック (セッション) を維持する。

表 6.6　機器一覧

Host	Vendor	Device	Version
FW1/2	Juniper	SSG20	ScreenOS 6.3.0r22.0
L3SW, L2SW1/2	Cisco	Catalyst3750G-24TS	IOS 12.2(50)SE5 (ipservicesk9)

6.4　環境構成 (管理系およびテストシステム)

管理者要件 (6.3.1 節) のとおり、ネットワーク管理については ヨーヨーダイン社サービスネットワークと分離した構成をとる。図 6.3 のネットワーク機器 (FW, L2/L3 スイッチ) 管理アクセスについては、次のように out-of-band な管理ネットワークを構成する (図 6.4)。

- Firewall: 管理用 VR を設定し、管理用 VR へ接続するインタフェースおよび VLAN をサービス用のものと分離する。
- L2/L3 スイッチ: 管理用 VRF を設定し、管理用 VRF へ接続するインタフェースおよび VLAN をサー

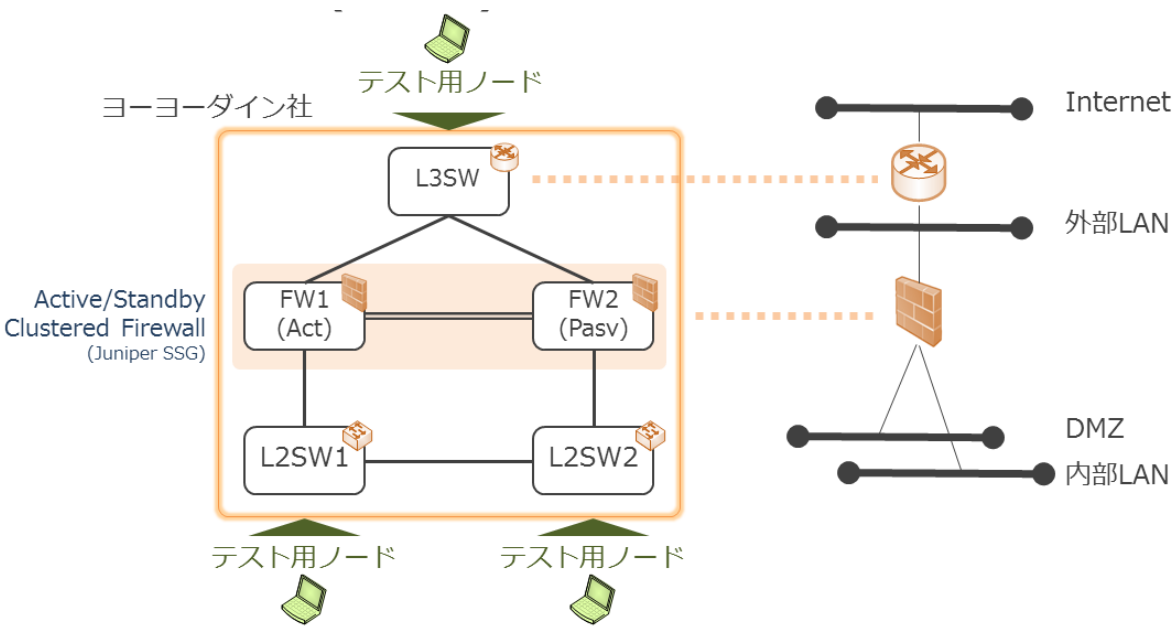

図 6.3　PoC 環境: 物理構成図 (概要)

ビス用のものと分離する。

■Tester set　図 4.1 のように、NetTester server と対応する物理 OpenFlow スイッチのペアを、本プロジェクトでは「Tester Set」と呼ぶ。NetTester によるテスト作業は tester set の範囲内でおこなわれる。

PoC にあたって、検証環境 (図 6.4) 内ではふたつの tester set を構築した。これは次の理由によるものである。

- テストシナリオ実装のうち、並行して実装・テスト実行[*7]が可能なものについて並行作業可能にするため。
- NetTester のデバッグ、バグ再現調査などトラブルシュート対応のための切りわけ手段として。機器故障や環境依存のある問題を切りわけ、作業中断リスクを回避するため。

■Tester set によるテスト実装作業の制限　テスト実行は原則としてひとつの tester set で実行されることを想定する[*8]。並行作業や同時実行については以下のような条件について考慮する必要がある。

- テストシナリオ中で使用するテスト用ノード等のパラメタ重複: 例えば、同一 IP のテスト用ホストを生成するテストシナリオを、ひとつのテスト対象ネットワーク内で同時に実行するような場合 (デバッグや調査などで同一シナリオを異なる tester set で同時に実行するといったケース)。
- テスト実行中 (実行前後) のテスト対象ネットワーク状態遷移: 例えば、リンク障害試験 (「動的なふるまいのテスト」) では、操作対象となる物理リンクが 1 箇所となるためその物理リンクを制御可能な

[*7] テストシナリオの動作テスト
[*8] ひとつのサーバ (OS) 上で複数の NetTester は実行できない。複数の NetTester server をまたいで排他制御をおこなう制御は NetTester ではなく外部オーケストレータなどで実行する必要がある (7.4.2 節)。

tester set はひとつだけに限られる。

– 同様に、テスト実行中にトポロジなどが変化する想定をおいていない「静的なふるまいのテスト」も、「動的なふるまいのテスト」と同時に実行することはできない。

図 6.4 では tester set 1 を「動的なふるまいのテスト」用に使用するように設計している。テスト対象が物理ネットワークであり、ひとつのインスタンスからのみ操作可能なリソースがある点に注意が必要である[*9]

■物理構成図補足　図 6.4 にある SSG5 は FW(FW1/2) の設定や動作確認用に使用したものである (OS は FW1/2 と同様: 表 6.6)。NetTester 開発やテストシナリオ実装にあたって、FW の設定や挙動をテストシナリオ上から確認し、問題点の切りわけをおこなうために利用した。

[*9] NetTester に外部から操作可能な API を実装して連携させるといった応用も考えられるが、こうした応用は本プロジェクトの目標範囲外となる。

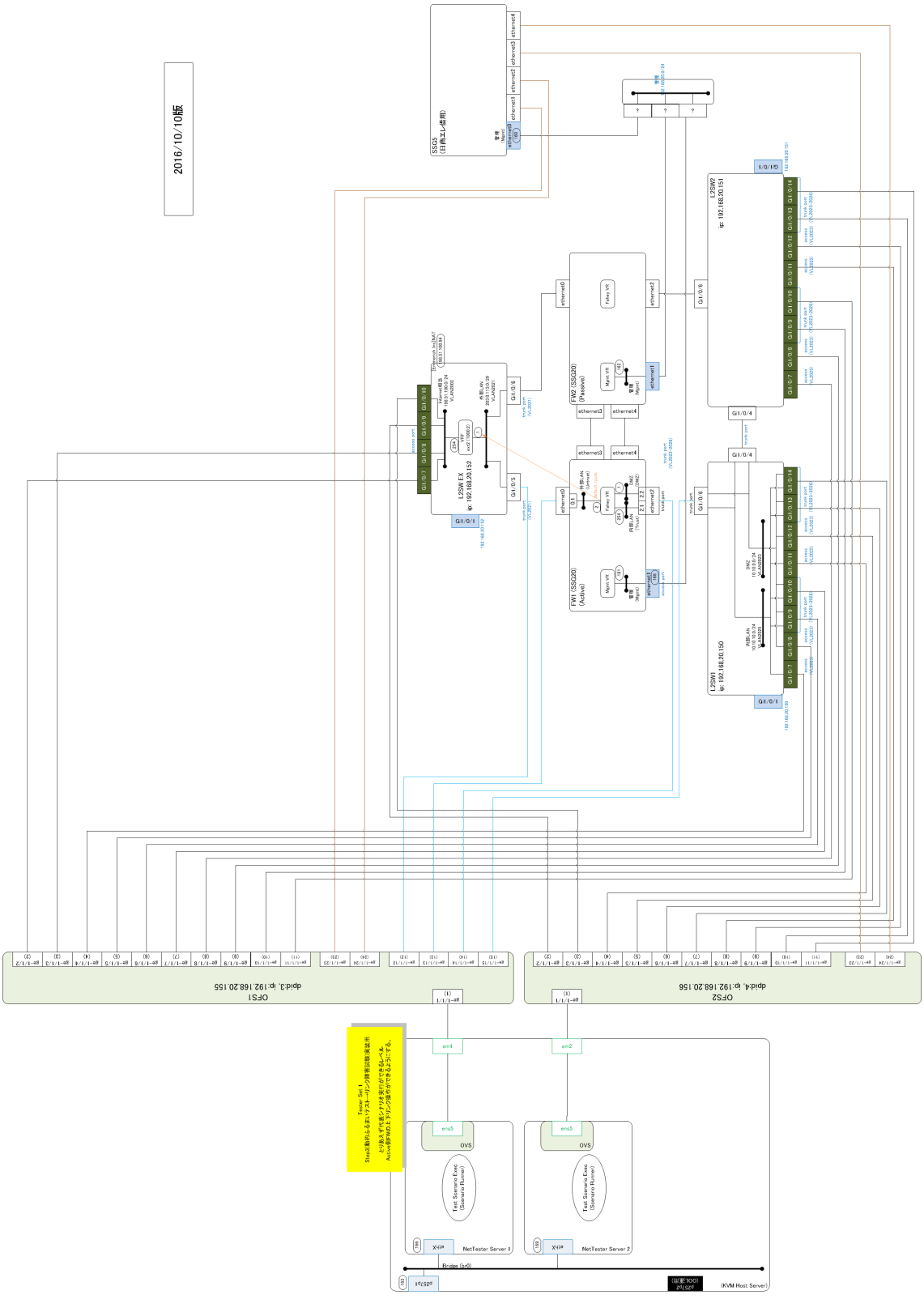

図 6.4　PoC 環境: 物理構成図 (詳細)

第 7 章

テストシナリオ実装

PoC で実施したテストシナリオの実装、実際にテスト自動化をおこなってわかったことや注意事項などについて解説する。

7.1 テストシナリオの概要

7.1.1 ネットワークテストの方針

6 章で設定したとおり、ヨーヨーダイン社ネットワークのテストを考える。そのためのテストシナリオとして「静的なふるまいのテスト」「動的なふるまいのテスト」の 2 種類を実装する。テストシナリオ実装・テスト実行を通して、物理ネットワークのテスト自動実行のポイント検討や問題点の抽出、従来の人手によるテスト作業との比較をおこなう。

7.1.2 BDD テストシナリオの基礎

■テストツール　NetTester はテスト用ノードの生成・テスト対象ネットワークへの配置 (パッチ接続) をおこなうための API を提供する。提供される API を使用して ヨーヨーダイン社ネットワークのテストを実装する。[*1]

テストシナリオの記述については、アプリケーションのテストでつかわれる既存のツールと連携することを想定している。本 PoC では、「BDD によるネットワークのテスト」を考えていること (3.1.2 節)、広く使われているツールであること、Ruby ベースで NetTester との親和性が高いことをもとに、BDD ツールである Cucumber[41] を採用した。

■Narrative　BDD では、テストのストーリー (フィーチャ) を以下のような構造で記述する [36, 37]。

タイトル　どのストーリーについて説明するのかを示す。タイトルは一般に、ユーザがシステムに要求するかもしれないアクティビティを短い言葉で表したものになる。

ナラティブ　ストーリーの内容について説明をする。一般的には Connextra フォーマットと呼ばれる形式の短い文章で記述される。このテンプレートは、誰がシステムを使っていて、そのユーザは何をしていて、なぜそのことに関心があるのか、を明確にする。

[*1] 実際に作成した自動テストのためのコードは NetTester Examples [25] として公開している。

- “As a (role)”: [誰のために (role)] として
- “I want (feature)”: [何を (feature)] したい
- “So that (bussiness value)”: なぜなら [なぜ (bussiness value)] のためだ

受け入れ基準　ストーリーの完了・完成を定義する受け入れ基準、シナリオの定義。

Cucumber におけるフィーチャとは、システムを利用するユーザーまたは別のコンピュータの視点に立っておおまかに表現された要件のことを指す。Cucumber のフィーチャはタイトルと簡単なナラティブ、受け入れ基準としての役割をはたす自動化されたシナリオによって構成される。

- フィーチャはタイトルとナラティブで構成され複数のシナリオを含む。(本プロジェクトでは `features/*.feature` ファイル)
  - シナリオはそれぞれ、シナリオ内でおこることを任意のステップで記述する。
- 個々のステップ定義は開発で使っているシステムの言語で記述される。(本プロジェクトでは Ruby: `features/step_definitions/*.rb` ファイル)

7.2　静的なふるまいのテスト

7.2.1　テストとして実現したいこと

ネットワークに対して最低限要求されることは、ネットワークを介して必要な通信が実現できることである。ネットワークの目的は、大きく言えばコミュニケーションを実現することである。あるノードが、他のノードと、必要な手段 (アプリケーション) で通信できるかどうか、という点がまず初めにネットワークに求められる機能要求となる。

「ネットワークの静的なふるまい」(3.2.1 節) では、ネットワークが定常状態にある (一定の状態にあって状態変化しない) ときに、ネットワーク利用者が実現したい end-to-end の通信がすべて実現可能かどうかをテストする。

7.2.2　テスターに求められる機能

テストで確認したい end-to-end の通信、すなわち、実現したい通信要求は、通信をおこなうノード (エンドポイント) の論理的・物理的な配置の組合せに応じて異なる。また、通信をおこなうアプリケーションによっても別途制御がおこなわれる。例えば、DPI をおこなう FW や LB などがネットワーク内にある場合は、単純な宛先 (ヘッダ) 情報だけではなく、やりとりされるアプリケーションレベルの情報によって通信の可否などが変化する。

したがって、単純な end-to-end の通信試験だとしても、配置やアプリケーションの組合せによって大量のテストケースが発生してしまう、テストのためのリソース確保などが充分にできない、などの課題があった (3.3 節)。

そこで、テスター (NetTester) では以下の機能が要求される。

- テスト用ノードの生成
- テスト用ノードの配置
- テスト用ノード上でのタスク実行

- 複数のテスト用ノード操作/集中管理

NetTester はこうしたノードの生成・配置・集中制御の機能 (API) を提供している。テストシナリオ (Cucumber) では BDD の考えかたに基づいて、ノード間でどういった通信を実現する必要があるかを個別の通信要件 (6.3.2 節) ごとに定義する。

7.2.3　静的なふるまいのテストで実際に発見できた問題点

静的なふるまいのテストでは、ヨーヨーダイン社ネットワーク通信要件 (6.3.2 節) それぞれについてテストシナリオを実装し、個々の要件に対して end-to-end の通信試験が自動化できることを実証した。テストの実行によって実際に発見できた物理ネットワークの問題点について解説する。

テスト対象ネットワークの設定不備の発見

一般的な (人手による) 疎通試験と同様に、ping やその他のアプリケーションによる通信確認によって、テスト対象ネットワーク上の不備 (要求される仕様と異なる) 点あるいはそれにつながる事象を発見できた。

■テスト対象ネットワークの設定ミス　テスト対象ネットワークを構築した際の設定ミスによる問題を発見した。主要なものは FW のパケットフィルタルールの設定不備、NAT(NAPT) の設定ミスによるものである。これらは「できなければいけないこと」ができていない (テストが失敗する) ことによって発見されたものである。以下に例を示す。

- インターネットから NAT(VIP) を経由して DMZ へアクセスする通信失敗 (パケットフィルタルールの設定漏れ)
- ヨーヨーダイン社内部ゾーンから外部ゾーン (internet) への通信失敗 (NAPT 設定ミス)
- FW が持つ IP に対する ping 失敗 (FW デフォルトの ping 応答ポリシ設定漏れ)

■テスト作業者側 (構築・運用側) の通信仕様の認識違い　登場人物 (6.2.1 節) に示したように、テストシナリオの実装や実行をおこなった担当者は必ずしもネットワークやネットワーク機器のスペシャリストではない。そのため、FW のゾーン間通信についての FW のポリシの認識不足[*2]から、ヨーヨーダイン社外部ゾーン (インターネット) から内部ゾーンへの ping 通信が可能と判断してテストシナリオを実装してしまったケースがあった。実際には要件どおり設定された FW によってこのテストが失敗し、認識違いがあったことが判明した。

ネットワーク機器の動作不具合

当初、静的なふるまいのテストのテストシナリオを実装し、動作させたときに、同一のテストシナリオが実行されるたびに成功・失敗をくりかえすという事象があった。調査したところ、FW の ARP 処理の挙動について以下のような動作をしていることがわかった。

- NetTester が生成するテスト用ノードでは、IP アドレスは一定だが、MAC アドレスはテストシナリオ実行ごとにランダムに変更する。

[*2] ゾーン間通信のデフォルトポリシについて、当該機器を使用したことがあるネットワークエンジニアとしては自明だったために明確に伝達がおこなわれていなかった。

- FW に ARP 前のシナリオで実行したテスト用ノードの MAC エントリが残っていると、次のシナリオで実行した (IP は同一だが MAC アドレスが異なる)ARP に対して応答しない。(ARP エントリのクリアをおこなっても応答しない。)
- FW の ARP エントリをクリアして、クライアントから TCP 通信をおこなっても FW(Gateway) からサーバ側への ARP が確認できず、MAC アドレスを学習していないにもかかわらず TCP SYN が送信されることがある。

これらの事象および当初使用していた FW の OS バージョンが古かったことから、FW(OS) の不具合と判断し、FW の OS を更新する[*3]ことで解決した。

L7 レベルの FW 挙動変化

今回のテストシナリオ実装では、当初 L4 レベルの通信確認から実装し、可能なものについては実際のサーバ/クライアントを使用した L7 レベルでの実装をおこなっている。L4 レベルの通信確認としては、netcat を使った任意の TCP/UDP port listen とクライアントからの接続 (tcp セッション確立/単純なサーバからの応答確認) を使用している。L7 レベルの end-to-end テスト実装としては以下のシナリオがある。

- インターネットへの SSL アクセス (openssl, curl を使用したテストシナリオ実装[*4])
- SSH サーバへのアクセス (openssh を使用したテストシナリオ実装[*5])
- DNS サーバへのアクセス (dnsmasq を使用したテストシナリオ実装[*6])

特に、DNS のテストについては、当初 netcat による L4 レベル確認として実装したが、テストが成功しなかった (FW を経由して通信ができなかった)。今回使用した FW(SSG) の詳細なフィルタ仕様については調査していないが、このとき `dig` コマンド等による実際の名前解決はできていたことから、テストシナリオとしても dnsmasq による L7 レベルでのトラフィック生成シナリオを実装している。

7.3　動的なふるまいのテスト

7.3.1　テストとして実現したいこと

ネットワークは状況に応じて状態を変化させる。変化のトリガとして、例えば障害発生による冗長経路への切替、メンテナンスのための一部のデバイスの停止や切り離し・そのための通信経路迂回、ネットワーク機器の追加 (拡張) や削除 (縮小) などがある。こうしたイベントに対して、ネットワーク (全体) は、ネットワークの状態と発生したイベントに応じて自らの情報 (状態) を自律的に更新し、トラフィックの経路を変更するなどの制御をおこなう。

「ネットワークの動的なふるまい」(3.2.1 節) では、ネットワークが状態変化するときに、ネットワーク利用者へ与える影響が許容範囲内かどうかをテストする。ネットワーク利用者への影響とは、ネットワーク状態遷移中に発生するトラフィックの継続可否、影響度・影響範囲 (場所的な範囲や時間) である。例を以下にあ

[*3] 表 6.6 参照。FW は更新版 6.3.0r22.0(2016/4 月リリース) に対して当初は 6.3.0r5.0(2010/9 月リリース) を使用していた [49]。
[*4] https://github.com/net-tester/examples/blob/feature/ood_demo/features/step_definitions/google_steps.rb
[*5] https://github.com/net-tester/examples/blob/feature/ood_demo/features/step_definitions/ssh_steps.rb
[*6] https://github.com/net-tester/examples/blob/feature/ood_demo/features/step_definitions/dns_steps.rb

げる。

- トラフィックの継続可否
  - ロードバランサなど L7 の情報にしたがってトラフィックを操作するような機器では、冗長系切替のときに、セッション情報などを引きついでアプリケーション通信を維持することが求められる。同様に NAT などの変換テーブルに基づいてトラフィックを操作する機器でも冗長系機器間で状態の同期が必要になる。
  - 系切替には隣接する機器も関連する。例えば、対象とする冗長系の L1/L2 の切替に Gratuitous ARP を使用する機器で、隣接する L2 スイッチのバグによって Gratuitous ARP 処理が正しくおこなわれず、全体としてトラフィックの継続に失敗した事例などがある。
- 影響度・影響範囲
  - 状態変化によって予期しない現象が予期しない範囲でおこることがある。範囲としては、物理的な場所 (特定のスイッチの配下など)・論理的な場所 (特定の L2 セグメントなど) 複数の要素が考えられる。
  - 仮想化によって物理構成と論理構成は分離されることが一般的である。この際、物理構成操作のオペミスや論理構成操作ミス (記述ミスや勘違い) の見落としがセグメント全体をまきこむ L2 ループを引き起すことがある。
  - 機器のバグや設定ミス、そのほかの機能との併用による機器 CPU 専有状況の発生などがあると、こうした系切替によるトラフィックの維持・継続ができなかったり、想定した時間内に完了できないことがある。例として、多数の VLAN を収容していた L3 スイッチで、系切替の際の STP トポロジ再計算処理による CPU 専有が発生し、スイッチ配下のセグメント全域で広範囲にわたって切替時間の遅延・それにともなうトラフィックの維持失敗 (タイムアウト発生) がおこってしまった事例などがある。

いくつか事例を挙げたが、これらの問題はネットワーク全体 (系全体) の相互作用に影響される。ネットワークの場合はベンダや機器ごとに独自の OS やアーキテクチャを持つものが多く、特定機能や製品を組み合わせたときの相性問題やキャパシティ問題、特定ハードウェアやソフトウェア (バージョン) のトラブルといった情報を共有することは難しい。また、複数の冗長化機能 (複数の機器にまたがるものも含む) の連動する際、状態遷移のタイミングなどで機能単体では問題なく動作する機能が複合した結果、問題が発生することもある[*7]。こうした理由により、ネットワークの操作・状態変化において、すべての影響を予想するのは非常に困難である。

動的なふるまいのテストでは、上に挙げた「トラフィックの継続可否」「影響度・影響範囲」のテストを自動化することを目標とする。

7.3.2　テスターに求められる機能

「動的なふるまいのテスト」では、次のような機能が必要となる。

イベントの生成　ネットワークの状態変化を発生させるためのイベントを発生させる機能。今回の PoC ではリンク障害試験をターゲットとするので、何らかの形でリンク障害を発生させる必要がある。

[*7] 例として: OSPF/BGP 収束時間差によるブラックホールやループの発生 [38] など。

イベント発生時のトラフィック影響調査　ネットワークが状態変化するタイミングで、ネットワーク上のトラフィックがどの程度の影響をうけているかを調査する必要がある。

一般的には、ネットワークの状態変化イベントによって影響をうけることが予想されるトラフィックをあらかじめ生成しておき、イベント発生・ネットワーク状態変化の収束をまって、トラフィックに最終的にどのような影響があったのかを調査する。したがって、複数のノード間において・同時並行で・トラフィックを継続生成 (送受信) しながらイベントを発生させる機能が必要となる。

7.3.3　システム構成

PoC 環境 (詳細は図 6.4) では図 7.1 のように tester set 1 を「動的なふるまいのテスト」用に使用するよう設計している。PoC では、ヨーヨーダイン社内部ネットワークの中心となる FW の冗長化機能試験をおこなう。そのため、テストシナリオでは FW の Active/Passive 切替トリガとなる Active 側 (FW1) リンクダウン/リンクアップ操作を実装する。物理リンク操作のため物理 OpenFlow スイッチ (OFS1) へ以下の 2 リンクをそれぞれ引き込んでいる。

- L3SW-FW1 間リンク (FW1 Uplink)
- FW1-L2SW1 間リンク (FW1 Downlink)

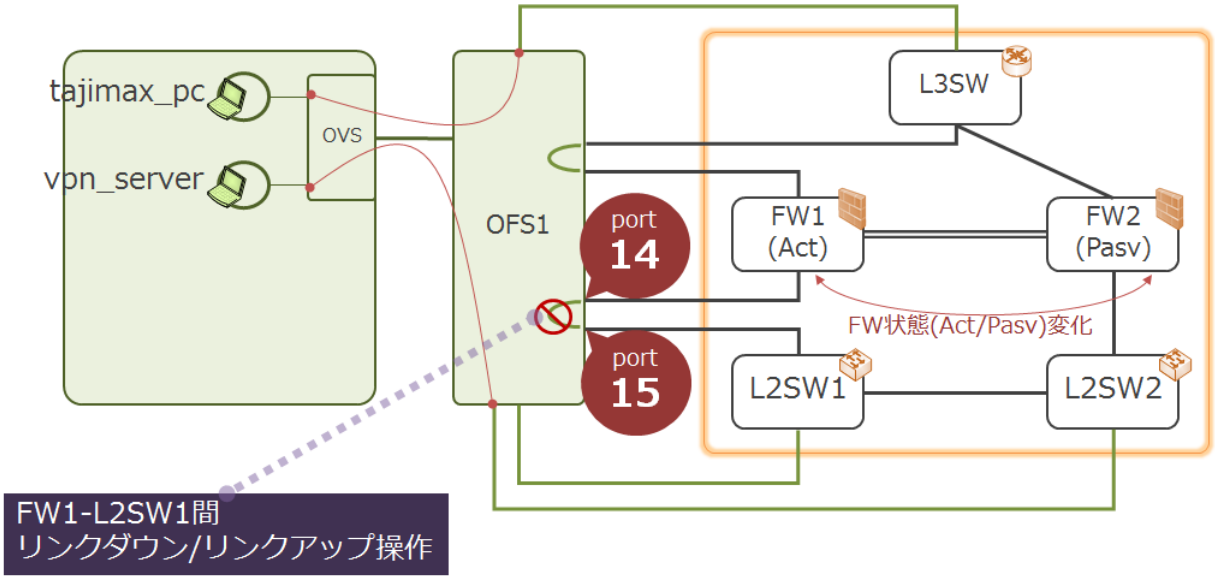

図 7.1　PoC 環境: NetTester によるリンクダウン操作

7.3.4　継続的通信テストの実装

テスト対象ネットワークの状態変化を調査する際によく使用されるのは、Ping(ICMP) を一定間隔で送受信しておき、通信断等の影響がどの程度発生したかを測定することである。本 PoC 環境には、L4(TCP) レベルの情報を同期してフェイルオーバー/フェイルバックする FW があるため、L3(ICMP) レベルでの通信維持

確認に加えて、L4 での状態変化通信影響も測定する (7.3.1 節)。そのために、TCP セッションを張ったまま Ping 同等の動作をおこなうためのツールを自製[*8] して使用している。

リンク障害発生時のトラフィック影響としては、単純に CIMP/TCP ping ログにおいてパケットが受信できなかった部分の時間差分を計算する[*9]ことでおこなっている。

7.3.5　動的なふるまいのテストとして実現できたこと

動的なふるまいのテストとして、ICMP/TCP でトラフィックを継続的に流しながら FW のフェイルオーバー・フェイルバックをくりかえすテストを実装することができた。PoC 環境において、FW フェイルオーバー・フェイルバックの動作で問題 (テスト対象 NW の不備) は特に発生しなかった。テスターおよびテストシナリオ実装上のポイントについては 7.4 節にまとめる。

テストシナリオ実装を通して、ネットワークの通信仕様変更が発生した際の対応プロセスについても検討している。今回は、タジマックス社からの SSLVPN NAT IP の変更を題材としている。IP 変更要求に対して、以下の手順でテストファーストに対応を進めるためのデモンストレーション [26] を作成した。

- 変更要求にしたがって、対応するテストシナリオを修正する。
- テストを実行する。(失敗するテストを書く)
- テスト対象ネットワークの設定を変更要求に基づいて変更する。(テスト対象の実装)
- テストを実行する。(テストが通ることを確認する)
- 変更対象のサービス (要求仕様/ふるまい) に影響を与えていないことを確認するため、その他のテストシナリオをすべて再実行する。(回帰テストの実行)

これにより、要求変化への追従に対するテストファーストな対応、回帰テストによる仕様変更対応の影響範囲の調査 (予期しない影響が発生していないこと) を実際に実装し、確認することができた。

7.4　テストシナリオ実装における検討ポイント

7.4.1　Teardown 処理

CI/CD といったプロセスを実現する上で、複数・任意のテストシナリオをまとめて実行することが求められる (回帰テスト含む)。

テストシナリオはいずれも、テスト対象が所定の初期状態にあるところから実行を開始しなければ狙った「ふるまい」を調査することができない。ソフトウェアの自動テストでは、テスト対象となるアプリケーション (プロセス) やインスタンスは初期状態で起動しなおすことが容易なため、常にアプリケーションやインスタンスを起動しなおして初期状態からのテストをおこなうのが一般的である。

しかし、ネットワーク (特に物理ネットワーク) を対象としている場合はテスト対象が物理的に存在している。よって、そのままでは、前のテストシナリオ実行直後のネットワーク状態が次のテストシナリオ実行開始時の初期状態となってしまう。個々のテストシナリオ実行のたびにネットワーク (機器) の初期化・再構築することは、不可能ではないが以下の理由により難しい。

[*8] https://github.com/net-tester/examples/blob/feature/ood_demo/features/support/echo_server.pl

[*9] https://github.com/net-tester/examples/blob/feature/ood_demo/features/step_definitions/util.rb, `check_connection` 参照。

- 機器によっては、コンフィグ消去・再起動に数分から十数分かかるものがある。
- 再起動による teardown 処理をおこなう場合、ネットワーク機器の起動順序によりネットワーク全体の状態が変化し得る。
- コンフィグリセット操作に失敗すると再起動後にリモートアクセスできず、復旧作業 (現地作業) が必要になるリスクがある。

特に本 PoC では、静的なふるまいのテストとして数十の通信要件をテストするためのシナリオを連続実行したいという要求があるため、単一テストシナリオ実行の時間的なオーバーヘッドが大きいことが問題となる。

そのため、テストに応じて何らかの形でネットワークの状態を初期状態に戻すための処理 (teardown 処理) を実装する必要がある。本 PoC では L2-L4 の状態に着目して、以下の teardown 操作を実装している。

■テスト対象ネットワークの状態操作　テスト対象ネットワーク各機器の L2-L4 の状態をクリアする (CLI による各種 `clear` コマンドの実行)。

- MAC アドレステーブル/ARP キャッシュのクリア
- FW の NAT Table のクリア
- FW の active/standby 状態のクリア
  - 今回、動的なふるまいのテストにあたって、FW は障害が発生したリンクの復旧にあわせて active/passive を自動復旧するように設計したため、特に操作していない (6.3.3 節)。

■Netns の /etc 配下の Setup/Teardown　NetTester では、Network namespace でテスト用ノードを生成する。このときバックエンドでは iproute2(`ip` コマンド) を使用している。iproute2 によって生成されたホスト (netns) では、`/etc/netns/<netns>/` が etc ディレクトリとしてマウントされる [31]。そのため、テスト用ノード内部での名前解決などでは、実体としては `/etc/netns/<netns>/hosts`, `resolv.conf` を参照する。これらの設定ファイル等が正しく設定されていないと、名前解決がタイムアウトするなどの問題が発生する。

■物理 OFS の flow table のクリア　物理 OpenFlow スイッチには、テストのたびにフローエントリが登録される。そのため、テストの実行のたびにフローエントリのクリア処理が必要となる。

本 PoC では、テスト用ノードの MAC はテスト実行時にランダムに設定するよう実装したため、テスト実行をくりかえすと多数の不要なフローエントリが物理スイッチ上に残る。テスト用ノードの MAC アドレスを固定にすることでフローテーブルの消費は抑えられるが、異なる用途で同一の MAC アドレスを使いまわすなどのケースで、予期しない動作不具合が発生する恐れがある[*10]点に注意すること。

■テスト用ノードのパラメタ設定の考えかた　テスト用ノードの MAC アドレス設定による不具合が見付かった際 (7.2.3 節)、一時的にテスト用ノードの MAC アドレス設定をテストシナリオ上で固定にすることで回避した。しかし、実際には teardown 処理を実装し、テスト用ノードの MAC アドレスはシナリオ実行時にランダムに設定されるようにした。これは次のような考えかたによる。

- 通常、テスト対象に接続する機器の MAC アドレスについて、テスト作業者が意識していることはほとんどない。テストシナリオを記述する際にも同様に、意識しなくてよいものを意識させなければいけな

[*10] NetTester のテーブル設計では、テスト用ノードの MAC アドレスをキーとして、フロー優先度 (priority) による制御をおこなう (4.5 節)。同一 MAC アドレスを異なる用途で使いまわす場合、他の用途として設定したフローエントリとマッチしてしまい、狙ったパッチ動作が実現できない恐れがある。

い実装は避け、テストシナリオの本質的な部分の実装に注力できるようにすべきである。
  - もちろんテストシナリオとして常に指定した MAC アドレスを使うことも可能である。テスト対象ネットワーク内の個々の機器について、ユニットテストで詳細な動作を調査するような場合にはこうした機能が必要になるが、本 PoC では「ふるまい」つまり要求仕様ベースでのテストを主眼においており、そのレベルで本題でないパラメタ設定はテストシナリオ実装者に極力意識させないようにする。
- 物理的なテスト対象では、テスト対象インスタンスの状態を低コストでクリアすることが難しく、teardown 処理はテスト自動化において重要な処理となる。一時的に MAC アドレス等の設定で回避してしまうのではなく、PoC を実行するうえで実用性があるかどうかを見極める必要がある。

7.4.2　NetTester およびテストシナリオの並列実行と排他制御

物理ネットワークをテストする場合、テスト対象 (物理ネットワーク) はひとつしかない。テストの数が多くなる場合、複数のテストを同時に実行することが考えられるが、現状 NetTester を使ったテスト自動化では、並行実行は原則できない。以下にその理由を列挙する。

状態操作の競合　テストシナリオごとに、想定しているテスト対象ネットワークの状態がある。それらを変化させるテストシナリオは同時に実行できない。
- テストシナリオ A を実行している途中で、テストシナリオ B が teardown 処理を実行して、テスト対象ネットワークの状態クリアをしてしまうと、テストシナリオ A で想定していた状態を変化させてしまう[*11]。
- 動的なふるまいのテストのように、ネットワーク全体の状態に影響をあたえるようなテストシナリオは同時に実行できない。

ネットワークリソースの競合　テスト対象ネットワークで一意でなければならないリソースの操作については同時に実行できない。
- テストシナリオ A/B が同じ IP/MAC を持つテスト用ノードを同時に生成してしまうと、テスト対象ネットワーク内で IP/MAC の重複が発生してしまう。
- 物理リンクの操作など、ひとつしか存在しないリソースの制御が必要なテストシナリオは同時に実行できない。

本 PoC では図 6.4 のようにふたつの tester set を使用しているが、これはあくまでもデバッグ用途のためである。Tester set を複数用意することで、静的なふるまいのテスト (NW が定常状態にある) で、かつ同じ IP/MAC のテスト用ノードを使用せず、teardown 処理をテストシナリオ単位で実行しないようなケースに限って複数のテストが実行可能となる。

上記の制約の一部については、NetTester で実行されるタスクの排他制御のしくみを導入することで対応できるものがあるが、現状はそうした対応については本 PoC の範囲外としている。本 PoC では、ひとつのテスト対象ネットワークに対して tester set があり、同時にひとつのテストシナリオを実行することが前提となっ

[*11] Teardown 処理として `clear` コマンドを使用しているが、特定シナリオで使うエントリだけを消去するのではなく、NW 機器全体の情報をまとめて消去してしまうため。そのシナリオに依存するエントリだけをねらって消去できるのであればこの制約は外れるが、テスト対象ネットワーク内の機器それぞれについて対応可能かどうかという点が問題となるだろう。あるいは、teardown 処理を他のテストシナリオ実行がおわるまで待機し、まとめて実行できればよいが、現状はそうした実装をおこなっていない。

ている。

7.4.3　テストダブルの考えかた

一般的に、ソフトウェアテストにおいて、テスト対象が依存しているコンポーネントを置き換える代用品のことをテストダブル [50] という。

本 PoC のように、end-to-end のネットワークテストを行なう場合、テストトラフィック生成のためにクライアント/サーバを用意する必要がある。テスト対象ネットワーク内に既存のサーバがあるケースもあれば、本 PoC のように実際にはサーバがなく、代替となるサーバをテストシナリオで準備しなければならないケースもある。

テストダブルとしてのサーバを導入する場合、導入することによってその他のテストコードの書き換えが発生する (実際のサーバがある場合のテストコードと代替サーバを使う場合のテストコードが大きく異なる) のは好ましくない。

例として、インターネット検索 (google) の可否をテストする場合、テスト対象ネットワークが直接インターネットアクセス可能で直接 google のウェブサイトを見ることができる場合、リスト 7.1 のようなステップにできる。テスト対象ネットワークが直接インターネットアクセスできない場合は、テストダブルを建ててリスト 7.2 としたとする。

リスト 7.1　実際のサービスを利用する場合

```
1 Whenブラウザで (/^ Google のページを開く$/) do
2   cd('.') do
3     @browser_pc.exec 'curl -L https://google.com/ > log/google.log'
4   end
5 end
6
7 Then(/^Web サイトへのアクセスに成功$/) do
8    step %(the file "log/google.log" should contain "<title>Google</title>")
9 end
```

リスト 7.2　テストダブルを利用する場合

```
1 Whenブラウザで (/^ Google のページを開く$/) do
2   cd('.') do
3     @browser_pc.exec "nc 192.0.2.100 80 > log/nc_web.log"
4   end
5 end
6
7 Then(/^Web サイトへのアクセスに成功$/) do
8   step %(the file "log/nc_web.log" should contain "OK")
9 end
```

リスト 7.2 では、テストダブルを使用することでサーバについてのコードを変更しているが、それだけでなくクライアント側の処理に関するコードにも影響をあたえてしまっている。「ふるまい」として「google に接続して検索したい」という目的をふまえると、リスト 7.1 では直接的に「検索サービスに接続」していること

がわかるが、テストダブルを導入したリスト 7.2 では、本来テストしたい「ふるまい」を間接的にしか表現できていない。

しかし実際には、テストダブルで代替したい元のサービスを完全に実現することは不可能である (簡単にオリジナルのサーバを準備できるのであればテストダブルを準備する必要がない)。本 PoC では、例えば表 6.3 No.1 では git を使用する要件がある。実際に `git clone` コマンドでリポジトリ操作可能かどうか (L7) というレベルでテストするのか、TCP(L4) レベルの接続性だけをテストするのかなど、テストとしてどこまでを保証するのかという線引きをテスト自動化システムのポリシとして設定しておく必要がある。

7.4.4　コマンドのバックグラウンド実行

テスト用ノードでのサーバプロセスの実行 (テストダブルの生成)、動的なふるまいのテストにおける複数トラフィックの同時生成などでは、並列でコマンドを実行する・クライアント操作が終了したらサーバを終了する、などの処理が必要になる。単発の同期コマンド実行 (`exec`) ではなく、非同期で実行し、join させるような手段が求められる。

NetTester を使ったテストステップの実装では、バックグラウンド実行の方法として以下のふたつの方法がある[*12]。

スレッドを使う　`#exec` 自体を別スレッドで動かす。

```
1 Thread.start { @server.exec "bash -c 'echo OK | nc -l 80'" }
```

直接 ip コマンドを使う　`Netns#exec` と同様のことを `ip` コマンドを使って直接バックグラウンド実行する

```
1 run "sudo ip netns exec #{@server.name} bash -c 'echo OK | nc -l 80 &'"
```

本 PoC ではスレッドを使う方法を採用し、`AsyncExecutor` クラス[*13] を用意してテスト用ノード上でのバックグラウンドコマンド実行をおこなっている。

7.4.5　テスト用ノードのパラメタ管理

リスト 5.6 で、NetTester の基本的な仕様方法をサンプルコードで示した。NetTester でテスト用ノードを生成する (`NetTester#new`) 場合、テスト用ノードが持つ IP アドレス等各種パラメタを設定する必要がある。これらのパラメタは複数のテストシナリオにまたがって使用されたり、テスト結果によっては変更されることがあるため、リスト 5.6 に示すような直接的なパラメタ記述では管理が非常に煩雑になってしまう。

そこで、テストデータの管理に Factory Girl [51] を使用する。各テストシナリオで使用するテストデータを `features/factories.rb`[*14] で管理する。`factories.rb` では次の情報を記述する。

仮想ホスト共通属性の定義　ネットワークアドレスやゲートウェイのように、各テスト用ノードが共有する属性を記述する。

[*12] `Netns#exec` でバックグラウンド実行することはできない。これは `Netns#exec` が内部的に利用している `Open3.popen3` の仕様によるものである。

[*13] https://github.com/net-tester/examples/blob/develop/features/support/async_executor.rb

[*14] https://github.com/net-tester/examples/blob/develop/features/factories.rb

```
FactoryGirl.define do
  trait :internal_network_host do
    netmask '255.255.255.0'
    gateway '10.10.10.254'
    mac_address Faker::Internet.mac_address('00')
  end

 ...
end
```

仮想ホストの定義　すでに定義した :internal_network_host 使って仮想ホストを生成する。

```
FactoryGirl.define do
  ...

  factory :ntp_client, class: Netns do
    internal_network_host

    name 'ntp_client'
    ip_address '10.10.10.3'
    virtual_port_number 2
    physical_port_number 2
  end
end
```

Factory Girl で定義されたテストデータを使用してテスト用ノードを生成すると次のようになる。このように、テスト用ノードの共通パラメタを分離し、パラメタの設定や管理を効率的に実装することができる。

```
Given(/^NTP クライアントとなる開発者 PC$/) do
  @ntp_client = Netns.new(attributes_for(:ntp_client))
end
```

7.4.6　テストシナリオのサイズ

動的なふるまいのテスト (リンク障害試験) について、手作業で実施する場合には、1 回のイベント (リンクダウンまたはリンクアップの発生) に対して複数の操作をおこなう。例えば、あらかじめ複数のテストトラフィックを生成しておき (今回の PoC シナリオでは複数パスの ICMP/TCP ping の実行)、1 回のイベントに対してそれらのトラフィックがどう変化したのかを確認する。

これをそのままテストシナリオとして記述すると、ひとつのシナリオのステップ数が増大し、テストシナリオ全体の見通しが悪くなってしまう。そこで本 PoC では、テストが繰り返し実行可能であることを利用して、イベント・トラフィックごとにテストのシナリオを分割し、ひとつひとつのテストシナリオが単純かつ理解しやすいものになるようにした。

手作業でテストシナリオを実行する場合には、くりかえし実行 (作業) する時間的コストが大きいためイベントの発生回数自体をおさえる。しかし、テストシナリオを自動化した場合は、繰り返し実行することのコス

トは低い。そのため手作業で実行する際の一般的なやりかたに準拠させる必要はなく、個々のテストシナリオのメンテナンス性を重視して考えることができる。

7.4.7　Sleep tuning

テストシナリオ (テストステップ) の実装にあたっては、処理のタイミングを同期させるために sleep によるタイミング調整が必要になる。

- 処理の待機
  - 物理 OFS と OFC との接続・再接続の待機
  - テスト対象機器に対するコマンドの発行、処理実行の待機
  - コマンド実行後のログファイル生成等の待機
  - スレッドによるコマンドの同時実行 (7.4.4 節) を使用する場合は、スレッドのスケジューリングのためにコマンド実行までにタイムラグがあるため sleep を設定する。
- テストシナリオ上必要な操作タイミング設定
  - 動的なふるまいのテストの場合は、イベント発生前・発生後の定常状態遷移待機 (テスト対象ネットワークのトポロジの収束など) のように、テスト対象の状態変化にかかる時間にあわせたテストオペレーション実行タイミング調整が必要になる。

テストシナリオ上必要なタイミング設定についてはテストステップとして明示的にテストシナリオ中に待機時間を明示する。

```
1 When(/^(\d+) 秒待つ$/) do |seconds|
2   sleep seconds.to_i
3 end
```

処理の待機について、テストシナリオの実行と本質的に無関係のものであればテストシナリオ上は明示させる必要がない。テストステップ中での sleep 実装は意図や必要性がわかりにくくなりがちであるため乱用には注意が必要である。また、デバッグのために一時的に sleep を設定することがある (一時的にテストシナリオ実行を遅延させて別途テスト対象ネットワーク上での調査や作業をおこなう) ようなケースでは不要な sleep チューニングが残らないようにする必要がある。

7.4.8　標準出力 (stdout) の操作

テストステップ実装では、テスト用ノードのコマンド実行結果 (stdout) を取得したいケースがある。このとき、`$stdout` の操作をおこなってしまうと、teardown 処理等で実行する expect 系ツールへ影響を与えてしまうことがある。

今回、テスト対象ネットワークの操作には expectacle(B.1 節) を使用しているが、内部的には `spawn` されたコマンド d(`telnet`,`ssh`) の標準出力を `pty`/`expect` で操作している。テストシナリオ (テストステップ) 側で `$stdout` などを操作してしまうと、こうしたプロセスに影響が及んでしまう点に注意すること。

7.5　テストシナリオのデバッグ

テストシナリオの実装・実行をした際、テストが失敗したときには原因として以下の要因が考えられる (7.2.3 節)。また、これらの原因が複合して発生することがある。

テストシナリオ実装とテスト対象ネットワーク実装の認識違い　テスト実装者とテスト対象ネットワーク構築者との認識違い。

テスト対象ネットワークの設計や設定の不備　テストの目的通り、テスト対象が仕様を実現できていないこと、予期しない動作を発見できている状態。

テストシナリオの実装上の不備　テストシナリオ・テストステップ実装上の不備。テストに仕様するツールやコマンドの実行時エラーやシナリオ全体としてみたときのテストオペレーション実行タイミングのずれ・ログ取得不備など。

テスター (NetTester) の動作不具合　NetTester 実装においてはテスター自体の動作の問題やバグなどが発生する。

テストが失敗した場合は、まずは問題がテスター側にあるのかテスト対象ネットワーク側にあるのかを切り分ける必要がある。以下に問題切り分け・デバッグ時の確認ポイントを示す。

- テスト対象ネットワーク側での操作
  - ARP/MAC アドレステーブルの確認: テスト用ノードが生成したトラフィックがテスト対象ネットワークに届いているかどうかを確認する。テスター側の VLAN 設定とテスト対象ネットワーク機器側のポート VLAN 設定とのミスマッチに注意すること。
  - テスト用ノード接続ポートのパケットカウンタ (エラーカウンタの確認)
  - パケットキャプチャ: テスト対象ネットワーク機器でテスト用ノードが生成したトラフィックが送受信できていない場合はパケットキャプチャしてヘッダ等の情報を確認する。(NetTester の場合、L2(MAC, VLAN Tag) の操作をおこなうので注意。)
- テスター側 OVS での操作
  - OVS のポート通信量の確認: テスト対象ネットワーク機器側とあわせて確認することで、L1/L2 レベルでの接続状態確認を実施する。

```
1 ovs-ofctl dump-ports <bridge> <port-number>
```

  - フローテーブル、フローエントリに対するパケットカウンタの確認: フローテーブル設計・実装のデバッグをおこなう[*15]。

```
1 ovs-ofctl dump-ports <bridge>
```

  - OFS-OFC 間通信 (セキュアチャネル) 状態
- テスター (NetTester) サーバでの操作

[*15] 2016 年 12 月時点では、NetTester は OFS の teardown 処理を実装していない。過去のテストの際にフローテーブルに残ったエントリによる影響なども考慮に入れる必要がある (7.4.1 節)。

  - PSW-SSW 間リンクでのパケットキャプチャ: NetTester サーバの OFS スイッチ間接続用物理ポート (図 5.2 の ens5 相当のインタフェース) ですべてのテスト用ノードがやりとりするトラフィックをキャプチャできる。
  - テスト用ノードの ARP テーブルの確認
  - NetTester OFC(trema) でのデバッグ: B.2 節参照。
  - OS 上での不要な network namespace や veth インタフェース等の確認
- テストシナリオでの操作
  - Pry [34] によるデバッグ: テストシナリオ (ステップ) 中に pry を挿入することで、シナリオ中の特定の箇所でインタラクティブシェルに以降し、テストシナリオ上の変数やテスト対象ネットワークの状態などを確認する。
  - 各テストオペレーションでの操作ログの取得
  - Sleep tuning, テストシナリオの連続 (繰り返し実行) や実行間隔の調整: キャッシュ依存の動作やタイマベースの状態変化イベントなどでは、テストの実行タイミングあるいは実行順序によってテスト実行結果が変化することがある。

こうした作業では、テスター側の動作状況・ログだけでなく、テスト対象ネットワーク内の各機器に対するオペレーションや状態変化ログもあわせて一元管理できることが望ましい。そのために、一般的なネットワークのテスト実行や検証作業と同様、テスト対象ネットワーク機器のログ (Syslog) の収集、時刻同期 (NTP) などを実施しておくことが必要である。この他、テスト環境としては以下のような点が検討事項となる。

- テスト自動化として物理構成操作を極力避けたいという狙いがあるため、テスト環境構成の際にはあらかじめ、パケットキャプチャのためのリソースや予備配線などを想定した設計をしておくことが望ましい。
- 問題切り分け用の対照設定・対照機器の準備: 今回テストシナリオ実装にあたっては、FW のパケットフィルタを設定していない (permit any)FW を 1 台用意し、テスト用ノード (server/client) が直接通信してテストが実行可能かどうかを切り分けられるようにした。テストシナリオを一度に作成してテスト実行可否を見極めるのが難しい場合は、段階的なテストシナリオ自体の動作確認や切り分けが可能な工夫が求められる。

7.6　PoC 結果に関する定性的な評価

本 PoC では、BDD の考えかたのもとに、静的・動的なふるまいのテストを実装した。実際に、これまで手作業でおこなってきたのと同等の end-to-end 通信試験やリンク障害試験を自動化することができた。また、teardown 処理の難しさなど、ソフトウェア開発における自動テストでは存在しない (物理ネットワークのテスト固有の) 問題などもあきらかにすることができた。本節ではテスト自動化システムの運用に付随する検討事項や当初目的に対する今後の課題について解説する。

7.6.1　ネットワークの構築・運用プロセス

本プロジェクトの目的としてはネットワークの構築・運用プロセスの CI/CD 化 (3.2.2 節) がある。今回、7.3.5 節に示したように、テストファーストな運用という観点での操作を実際に試している。この中で、テス

トシナリオによる回帰テストの実行と影響範囲のチェックはできている。CI のための最も基本的なオペレーションについては実現性が見えているが、より高度な CI プロセスへ発展させるためには、今後下記のような点の検討が必要と考えられる。

■開発プロセスと自動テスト実行　ソフトウェア開発における CI でよく実施されるベストプラクティスに、コードコミット等のイベントをトリガとして自動的にビルド・回帰テストを実行することがある。テストシナリオの開発 (シナリオの修正や追加)、テスト対象ネットワークの実装 (設定変更等) に応じて、同様にテスト対象ネットワークを自動構成、テスト自動実行をおこなうような仕組みやプロセスが必要になると予想される。

少なくとも、テスト対象ネットワークの構成や設定情報、テストシナリオの開発プロセスとして、作成・作成者によるテスト、レビュー・レビュアーによるテスト、試験環境側でのマージ、本番デプロイなどのフェーズわけやテスト実行のための仕組み (7.4.2 節) などの整備が必要となる。

■テスト対象ネットワークの自動構築・操作　上記のような CI 開発プロセスを想定すると、テスト対象ネットワークの構成 (トポロジや設定等) 自体をバージョンコントロールし、任意の時点・構成・設定・状態に巻き戻したうえでテストを実行するような仕組みが求められる。

テスト対象が物理実体、すなわち物理ネットワークを想定すると、こうした任意の状態操作は難しい。また、テスト対象がひとつに制限されることによる同時テスト実行回避 (排他制御) など、一般的なソフトウェア開発では発生しない要求などを考慮する必要がある。

状態のコントロールとして、現状は teardown 処理による初期化をおこなっているが、特定の障害がおきた状態に戻すなど、指定した状態への復帰が簡単にできるようになると、テストシナリオのデバッグ、あるいはテスト対象ネットワークのデバッグ (再現試験) などへの応用がよりしやすくなると考えられる。

7.6.2　役割分担・スキルセット

テスト対象ネットワークの設計・構築、テストシナリオの実装・テストの実行にあたって、表 7.1 のような役割・スキルセットが求められる。テスター (NetTester) 自体の開発やデバッグでは、Ruby/Trema/OpenFlow/OpenFlow Switch 等の知識が求められるが、テストシナリオの策定や実装については一般的なネットワークの知識とある程度のプログラミングの知識[*16]があれば対応が可能である。

実際に各テストシナリオ・テストステップ実装を進めるにあたっては、個々のテストシナリオ担当者のプログラミングスキルというよりは、テストシナリオ全体の方針設計、レビューなどによるテストシナリオ実装の品質のコントロールができることが重要だった。特に、ネットワークエンジニア観点でのテストシナリオ実装をおこなうと、ユニットテストに近いテストシナリオや仕様ベースではなく NW 機器の知識をベースにしたテストシナリオを実装してしまう (過剰・冗長なテストを実装してしまう) 傾向が見られた。また、テストシナリオやテストステップ間の機能重複の排除、ファイル構成の見直しなど、テストシステム全体のリファクタリングの検討などをするうえでも、ソフトウェアによるテスト自動化のノウハウが必要となる。「ネットワークの BDD として必要十分なテストは何か」をコントロールするために、テスト自動化についての知識や経験を持ち、テストシナリオのありかた・方向性を設定できるソフトウェアエンジニアがいることが望ましい。

[*16] Cucumber と ruby によるテストステップの実装: 数行-数十行の ruby スクリプトが作成できるレベル。本 PoC のシナリオでは、外部コマンド実行や結果の取得・パースなどが ruby で実装できればよい。

表 7.1　テストシナリオ実装で求められる役割とスキルセット

役割	スキル
テストシナリオ内容の決定	テスト対象ネットワーク仕様の理解; シナリオ作成担当に対するシナリオ仕様を決定する (feature file レベルのシナリオの策定)
テスト対象 NW の整備	テスト対象ネットワーク仕様の理解; テスト対象ネットワークの設計・構築
テスターおよびテストシナリオ開発環境の整備	テスト対象ネットワーク構成 (物理構成) の理解; テスター (NetTester) のデプロイとメンテナンス; テストシステム構成情報をシナリオ作成担当へ渡す
シナリオ作成	NetTester API の理解; Ruby/Cucumber の理解; テストシナリオ (feature) の理解; テストステップの実装
シナリオ実行	テストシナリオの実行; テスト対象ネットワーク仕様の理解; テスト結果の判断
物理環境の操作・調査	テスト対象ネットワーク側原因調査 (デバッグ); ネットワーク機器の操作・設定変更
テストシナリオ実装の調査	テストシナリオ側原因調査 (デバッグ)
テスター実装の調査	テスター側原因調査 (デバッグ)

7.6.3　テストシナリオの実装コスト

テスト自動化に限らず「自動化」システムはイニシャルの実装コストがどうしても発生してしまう。また、テスト自体もシステムの変化にあわせて更新・修正していく必要があり、そうしたメンテナンスコストについても検討が必要になる。

参考までに一部のテストシナリオ実装事例についての開発コスト指標を以下に示す。ただし、本 PoC ではゼロからのテストシナリオ検討およびテスター (NetTester) 自体の開発を並行して実施しているため、テストシナリオ実装の一般的な初期コストとして直接利用できる値ではない。

静的なふるまいのテストの実装コスト　テストシナリオ方針がきまって、テストステップの共通項目がそろってきた後半でのシナリオ開発では、ひとつの通信要件に対するテストシナリオの実装に 0.5-1.0 人日程度が必要だった。テストツールでの確認を複雑にする (L7 レベルの動作を見る) ようなケースでは、テストで使用するツール (コマンド) に関する予備知識の有無などで実装コストは変動する。

動的なふるまいのテストの実装コスト　最初の 1 シナリオの構築にあたっては、NW 機器設定の担当者・テストシナリオ実装の担当者をそろえて実際の動作を見ながら実装を進める必要がある。初回実装では、動的なテストを手動で実行するのと同等のオペレーションと、それを自動化するための知見や調整[*17]が必要であり、2-4 人日程度を要した。一度基本となるシナリオが実装できて動作確認がとれると、あとは基本シナリオの応用となって 1 シナリオ 0.5-1.0 人日程度で実装できた。

[*17] Sleep Tuning (7.4.7 節) や Syslog 等によるテスト結果判定の実装など。

第 8 章

おわりに

8.1 まとめ

複数の機器や技術の整合性をとらなければいけないネットワークは、複雑になりがちで設定変更や障害の影響範囲を予測しにくい。しかしサービス提供者は、利用者の要求変化に応じて、安定した通信サービスを迅速かつ柔軟に提供していくことが求められる。そのためには、検討可能範囲や許容されるリードタイムに限界がある人によるレビューではなく、ネットワークが機械的・自動的にテストできること、テストによってネットワークが設計通りの機能を提供していることを確認できることが重要となる (2 章)。本プロジェクトでは特に、ネットワークの「ふるまい」に注目して (3 章)、物理ネットワークの自動テストをおこなうシステムを開発し (4 章・5 章)、いくつかのユースケースについて実際にシナリオテストの実装をおこなった (6 章・7 章)。

テストシナリオ実装においては、ネットワークが定常状態に状況での end-to-end 通信を「静的なふるまいのテスト」、リンク障害による通信経路切替がおきてネットワークの状態が変化する状況での end-to-end 通信を「動的なふるまいのテスト」とした。これらのテスト実装によって、定常状態でのアプリケーションレベルでの通信試験だけでなく、障害試験のように物理構成操作が必要で従来は人手で作業しなければいけなかった試験についても自動化ができるようになった。本書ではこれらの物理ネットワーク試験自動化における実装例と実装の際に発生した課題に対する対処などについて解説している。

本プロジェクトで実証したユースケースは単純化したものになっているが、物理ネットワークでおこなうテストとして通常実施される、基本的な作業を自動化したものである。本プロジェクトで整備した「ふるまい」レベルでのネットワークテスト機能により、3.3 節で示したネットワークテスト自動化のための基本機能は一通りそろえることができたと考える。

8.2 今後の課題

現時点では、架空の企業ネットワークを想定した小規模かつシンプルなネットワークでの PoC を行なっている。しかし、実際に実環境・実運用で使用する場合は、より複雑な環境・要件に対応していくことが求められる。

8.1 節で示したように、ネットワークテストの基本機能はおおむね整備できた。今後は実環境での利用を想定した実用トライアルにむけた活動を進めていく。そのなかで、3.2 節で示したようなネットワークにおける CI/CD プロセスの確立、そのためのツールチェーンのありかたや実装などについての検討をおこなう。また、開発したツールについて、実際の問題解決のために必要な機能・非機能面での機能強化を進める。

付録 A

用語

表 A.1 に本書で使用している用語・略語の一覧を示す。一般的な用語については特に解説をくわえない。

表 A.1: 用語定義

用語	英語表記	略語	分類	意味
沖縄オープンラボ	Okinawa Open Laboratory	OOL	一般	(略語定義) 沖縄オープンラボラトリ [1]
ネットワーク	Network	NW	一般	(略語定義)
ファイアウォール	Firewall	FW	一般	(略語定義)
ロードバランサ	Load Balancer	LB	一般	(略語定義)
	Virtual Private Network	VPN	一般	(略語定義)
L1patch プロジェクト			プロジェクト	2015 年度プロジェクト名 (略称) [2]
L1 パッチ	L1patch		プロジェクト	一般的なネットワーク用パッチパネル (物理結線を集中して付け替えるための機構)。本書では「ネットワーク用パッチパネルと同等のことをおこなえるシステム」として使う。
	OpenFlow	OF	一般	(略語定義)
	OpenFlow Switch	OFS	一般	(略語定義)
	OpenFlow Controller	OFC	一般	(略語定義)
テスト対象ネットワーク	Target/Testee Network		一般	動作確認を行いたいネットワーク。複数のネットワーク機器から構成される。

表 A.1: 用語定義

用語	英語表記	略語	分類	意味
テスター	Tester		プロジェクト	本書で単に「テスター」とした場合はネットワークテスター (ネットワークテストシステム): NetTester のことを指す。
テスト対象機器	Device Under Test	DUT	一般	テスト対象となる個々の機器。テスト対象ネットワークを構成するいずれかひとつの機器。
	Open vSwitch	OVS	一般	Linux 上で動作する L2 スイッチ/OFS 実装。
	NetTester		プロジェクト	本プロジェクトで開発したネットワークテストツール。
	Cucumber		一般	BDD テストツール/テストシナリオ記述言語。
	Deep Packet Inspection	DPI	一般	(略語定義)
継続的インテグレーション	Continuous Integration	CI	一般	(略語定義)
継続的デリバリ	Continuous Delivery	CD	一般	(略語定義)
ふるまい駆動開発	Behavior Driven Development	BDD	一般	(略語定義)
テスト駆動開発	Test Driven Devleopment	TDD	一般	(略語定義)

付録 B

関連ソフトウェア

B.1 Expectacle

テストシナリオ実装の際、テスト対象機器へのコマンド発行 (teardown 処理; 7.4.1 節) をおこなうために Expectacle [42] というツールを使用している。Expectacle はあらかじめ作成したコマンド (列) を指定した機器情報に基づいて順に送信する単純なツールである。詳細な使用方法については Expectacle README にあるためここでは扱わない。本 PoC で利用した際の注意事項について解説する。

Expectacle が使用する操作対象機器情報・コマンド列では ERB を使った変数の展開ができる。本 PoC では、操作対象情報のうちそのまま設定ファイルに記入してリポジトリへ登録してはいけない秘密情報 (ログインユーザ名・パスワード) を設定するために環境変数展開を使用した。例として、L2/L3 スイッチの操作に関しての操作対象情報はリスト B.1[*1] のようになる。

リスト B.1　L2 スイッチ (L2SW1) ログイン情報

```
- :hostname : 'l2sw1'
  :type : 'c3750g'
  :ipaddr : '192.168.20.150'
  :protocol : 'ssh'
  :username : "<%= ENV['L2SW_USER'] %>"
  :password : "<%= ENV['L2SW_PASS'] %>"
  :enable : "<%= ENV['L2SW_PASS'] %>"
```

リスト B.1 ではログインユーザ名 (L2SW_USER) とパスワード (L2SW_PASS, ログイン/イネーブルが同一パスワード) を環境変数を参照して設定している。そのため、テストシナリオ実行前にリスト B.2 のように環境変数を定義しておく。

リスト B.2　ログイン情報環境変数の設定

```
export L2SW_USER=username
export L2SW_PASS=password
```

[*1] https://github.com/net-tester/examples/blob/feature/ood_demo/features/support/expectacle/hosts/c3750g_hosts.yml

環境変数によるパスワード等の設定は、そのままではコマンドヒストリ等に残ってしまうことがあるため、実行方法に注意が必要である[*2]。Expectacle の機能は表 3.1 の No.1 (NW 機器の設定・操作) に相当する。この機能は本プロジェクトの主要なスコープとしていないため、このような簡易な方法を採用している。

B.2　Debugging Trema

本プロジェクトの実行にあたって発生した OpenFlow 関連の問題についての対処方法 (Tips) をまとめる。

B.2.1　Trema.logger

当初、NetTester OFC のテーブルミスアクションを `CONTROLLER` として packet-in を発生させるようにしていた。このとき、特定パケット (フレーム) の packet-in に対して OFC (Trema) がエラーで停止しまうという問題が発生した (4.5 節)。この問題にあたって、Trema 開発チームの協力により、Trema の内部ログを取得するための、`Trema.logger` API が追加されている [43]。

`Trema.logger` の使用方法は一般的なロガーと同様であり、Trema 内部にログメッセージを挿入していくことで、Trema の内部情報のログを取得することができる。たとえば使用例はリスト B.3 のようになる。

リスト B.3　`Trema.logger` 使用例

```
1 # lib/trema/switch.rb
2 def expect_receiving(expected_message_klass)
3   loop do
4     message = read
5     # コントローラが受け取った OpenFlow メッセージをデバッグプリント
6     Trema.logger.debug "Received an OpenFlow message: #{message.inspect}"
```

出力先のログファイルは `[ログディレクトリ]/trema.log` となる。`Trema.logger` を実行し内部ログを取得する場合、次のように実行時ログレベルを指定する必要がある。

```
1 trema run -l debug <controller>.rb
```

または verbose オプションを指定する (ログレベルは debug になる)。

```
1 trema -v run <controller>.rb
```

B.2.2　Packet-in Binary Analysis

節の事例 (特定 packet-in による OFC の停止) については、実際に packet-in するバイナリを解析し、原因が Trema/Pio [44] パケットパーサのバグであることを突き止めた。ここではその際のデバッグ方法を記載する。

[*2] Expectacle では標準入力から環境変数の値を設定するスクリプトを同梱している: https://github.com/stereocat/expectacle/blob/develop/exe/readne

■OFC パケットキャプチャ　そもそも Trema がどういったタイミング・原因で停止するのかを調査するために、OFS-OFC 間でやりとりされる OpenFlow メッセージのパケットキャプチャをとり、OFC 停止タイミングとの突合せをおこなっている。これによって、あるパケット (フレーム) の packet-in のタイミングで OFC 停止が発生しているという仮説をたてた。

■Trema.logger による packet-in のダンプ　Trema 内部の問題を取得するために、節に示した Trema.logger を追加して、コントローラ停止時に処理しようとしていた packet-in のダンプを取得する (リスト B.4)。

リスト B.4　問題となる Packet-in の取得

```
diff --git a/lib/trema/switch.rb b/lib/trema/switch.rb
index 5a73005..d8c222e 100644
--- a/lib/trema/switch.rb
+++ b/lib/trema/switch.rb
@@ -34,7 +34,9 @@ module Trema
     end

     def read
-      OpenFlow.read read_openflow_binary
+      openflow_binary = read_openflow_binary
+      Trema.logger.debug openflow_binary.unpack('C*').inspect
+      OpenFlow.read openflow_binary
     end

     private
```

これにより、リスト B.5 のように packet-in ダンプが取得できた。

リスト B.5　得られた Packet-in dump

```
D, [2016-09-14T11:29:41.312256 #21392] DEBUG -- : [1, 10, 0, 146, 0, 0, 0, 0, 0, 0,
   1, 24, 1, 216, 0, 4, 0, 0, 1, 0, 12, 204, 204, 204, 0, 30, 73, 27, 157, 7, 1, 202,
    170, 170, 3, 0, 0, 12, 32, 0, 2, 180, 19, 106, 0, 1, 0, 25, 76, 49, 80, 74, 95,
   76, 50, 83, 87, 49, 46, 108, 49, 112, 106, 46, 108, 111, 99, 97, 108, 0, 5, 0,
   251, 67, 105, 115, 99, 111, 32, 73, 79, 83, 32, 83, 111, 102, 116, 119, 97, 114,
   101, 44, 32, 67, 51, 55, 53, 48, 32, 83, 111, 102, 116, 119, 97, 114, 101, 32, 40,
    67, 51, 55, 53, 48, 45, 73, 80, 83, 69, 82, 86, 73, 67, 69, 83, 75, 57, 45, 77,
   41, 44, 32, 86, 101, 114, 115, 105, 111, 110, 32, 49, 50, 46, 50, 40, 53]
D, [2016-09-14T11:29:41.313604 #21392] DEBUG -- : in Controller#start_switch_main(dpid
   =0x1), rescue (and unregister switch). error class:IOError, message:data truncated
```

■BinData による解析　リスト B.5 を BinData(Trema/Pio は BinData をベースに実装されている) でパースして原因を調査する。パースはリスト B.6 のように実行することができる。

リスト B.6　BinData によるパース

```
irb> BinData.trace_reading do
```

```
irb>   Pio::OpenFlow.read [1, 10, 0, 146, 0, 0, 0, 0, 0, 0, 1, 24, 1, 216, 0, 4, 0, 0,
     1, 0, 12, 204, 204, 204, 0, 30, 73, 27, 157, 7, 1, 202, 170, 170, 3, 0, 0, 12,
    32, 0, 2, 180, 19, 106, 0, 1, 0, 25, 76, 49, 80, 74, 95, 76, 50, 83, 87, 49, 46,
    108, 49, 112, 106, 46, 108, 111, 99, 97, 108, 0, 5, 0, 251, 67, 105, 115, 99, 111,
     32, 73, 79, 83, 32, 83, 111, 102, 116, 119, 97, 114, 101, 44, 32, 67, 51, 55, 53,
     48, 32, 83, 111, 102, 116, 119, 97, 114, 101, 32, 40, 67, 51, 55, 53, 48, 45, 73,
     80, 83, 69, 82, 86, 73, 67, 69, 83, 75, 57, 45, 77, 41, 44, 32, 86, 101, 114,
    115, 105, 111, 110, 32, 49, 50, 46, 50, 40, 53].pack('C*')
irb> end
obj.ofp_version => 1
obj.message_type => 10
obj.message_length => 146
obj.transaction_id-internal-.xid => 0
obj.transaction_id => 0
obj.body => "\x00\x00\x01\x18\x01\xD8\x00\x..."
obj.ofp_version => 1
obj. => nil
obj.message_type => 10
obj. => nil
obj.message_length => 146
obj.transaction_id-internal-.xid => 0
obj.transaction_id => 0
obj.buffer_id => 280
obj.total_len => 472
obj.in_port => 4
obj.reason-internal-.reason => 0
obj.reason => :no_match
obj.padding => 0
IOError: data truncated
        from /home/yasuhito/play/net-tester/vendor/bundle/ruby/2.3.0/gems/bindata
    -2.1.0/lib/bindata/io.rb:83:in `readbytes'
        from /home/yasuhito/play/net-tester/vendor/bundle/ruby/2.3.0/gems/bindata
    -2.1.0/lib/bindata/string.rb:110:in `read_and_return_value'
        from /home/yasuhito/play/net-tester/vendor/bundle/ruby/2.3.0/gems/bindata
    -2.1.0/lib/bindata/base_primitive.rb:124:in `do_read'
        from /home/yasuhito/play/net-tester/vendor/bundle/ruby/2.3.0/gems/bindata
    -2.1.0/lib/bindata/trace.rb:58:in `do_read_with_hook'
        from /home/yasuhito/play/net-tester/vendor/bundle/ruby/2.3.0/gems/bindata
    -2.1.0/lib/bindata/struct.rb:131:in `block in do_read'
        from /home/yasuhito/play/net-tester/vendor/bundle/ruby/2.3.0/gems/bindata
    -2.1.0/lib/bindata/struct.rb:131:in `each'
```

ここから、packet-in メッセージの `total_len` の値が不正だったことが判明した。

- OpenFlow ヘッダの全体の長さ (`message_length` = 146) は正しい。
- 一方で `PacketIn` の `total_len` (PacketIn データの長さ) が 471 となっている[*3] 。

[*3] 原因詳細まで調査していない。

- Pio はデータの長さを total_len から取得するので、471 バイト読もうとして失敗している。

OpenFlow 仕様によると、データ長は total_len ではなく message_len から取得するように記述があった[*4]ことから、Pio の PacketIn を修正することで問題は解決した [47]。なお、修正後にパースをするとリスト B.7 ようになる。

リスト B.7　Pio 修正後の再確認

```
irb> Pio::OpenFlow.read [1, 10, 0, ... , 40, 53].pack("C*")
=> #<PacketIn open_flow_version: 1, message_type: 10, message_length: 146,
    transaction_id: 0x0, buffer_id: 0x118, total_length: 128, in_port: 4, reason: :
    no_match, data: #<Pio::EthernetFrame destination_mac: "01:00:0c:cc:cc:cc",
    source_mac: "00:1e:49:1b:9d:07", ether_type: 0x01ca, rest: "\xAA\xAA\x03\x00\x00\f
     \x00\x02\xB4\x13j\x00\x01\x00\x19L1PJ_L2SW1.l1pj.local\x00\x05\x00\xFBCisco IOS
    Software, C3750 Software (C3750-IPSERVICESK9-M), Version 12.2(5">>
```

B.2.3　Trema 送受信メッセージログ

最新版の Trema [48] では、B.2.2 節・B.2.2 節で実施したデバッグ作業をもとに、OFC が送受信した OpenFlow メッセージをログで取得できるようになっている。リスト B.8 のように-v(verbose) オプションを指定することで利用できる。(B.2.2 節で解説したようにログファイル (trema.log) にも出力される。)

リスト B.8　OpenFlow メッセージのデバッグプリント

```
% bundle exec trema -v run ./lib/hello_trema.rb -c trema.conf
sudo ovs-vsctl add-br br0xabc
sudo /sbin/sysctl -w net.ipv6.conf.br0xabc.disable_ipv6=1 -q
sudo ovs-vsctl set bridge br0xabc protocols=OpenFlow10 other-config:datapath-id
    =0000000000000abc
sudo ovs-vsctl set-controller br0xabc tcp:127.0.0.1:6653 -- set controller br0xabc
    connection-mode=out-of-band
sudo ovs-vsctl set-fail-mode br0xabc secure
Trema started.
Sending #<Pio::OpenFlow10::Hello:0x000000011190a8 @format={:version=>1, :type=>0, :
    _length=>8, :transaction_id=>0}>
Received #<Pio::OpenFlow10::Hello:0x0000000132c750>
Sending #<Pio::OpenFlow10::Echo::Request:0x0000000131ccd8 @format={:version=>1, :type
    =>2, :_length=>8, :transaction_id=>0, :body=>""}>
Received #<Pio::OpenFlow10::Echo::Reply:0x000000012ea288>
Sending #<Pio::OpenFlow10::Features::Request:0x000000012c9560 @format={:version=>1, :
    type=>5, :_length=>8, :transaction_id=>0}>
Received #<Pio::OpenFlow10::Features::Reply:0x000000012943d8>
Hello 0xabc!
Received #<Pio::OpenFlow10::Echo::Request:0x000000012099b8 @format={:version=>1, :type
    =>2, :_length=>8, :transaction_id=>0, :body=>""}>
```

[*4] OpenFlow1.0 Spec[46] “5.4.1 Packet-in Message” 参照。

```
16 Sending #<Pio::OpenFlow10::Echo::Reply:0x000000011ef338 @format={:version=>1, :type
       =>3, :_length=>8, :transaction_id=>0, :body=>""}>
17 ...
```

B.3　Phut Basics

NetTester の内部実装および NetTester 自身のテストコードで使用されているコードを元に、仮想ノード/仮想ネットワーク操作の処理実装の基礎について解説する。NetTester 内部では、Phut [45] を使用して Linux の仮想ネットワーク機能を操作している。なお、記載内容は 2016 年 9 月時点のものである。

B.3.1　Phut による仮想ノード/仮想ネットワーク操作の概要

まず、Phut による virtual link/host/switch の操作のながれを理解する。

- vLink を生成する
- vHost/vSwitch を生成する
  - Phut では vHost/vSwitch を生成するときに interface device (vLink の端点) を指定するので、通常は先に vLink を生成ことになる。
- vLink と vHost/vSwitch を接続してトポロジを組み立てる。

vLink/vHost/vSwitch のいずれも、インスタンスを使うときは `#create` method を使用する。(`create` は `new` + `run`, `start` という形になっている。単純に `new` するだけでは active/enable にならない。)

インスタンスを操作したい場合は、インスタンス名 (`name` attribute) で `find_by` する。

```
1 instance = Phut::<class>.find_by('instance_name')
```

B.3.2　vLink

`Phut::Link` で vLink をつくる。これにより、図 B.1 のような vLink が作成される。

図 B.1　vLink のモデル

これには「Phut で扱うための名前」と「OS 上で使われる実際のデバイス名」がある。例えば、`Phut::Link('sport', 'dport')` というコードに対しては、`sport`, `dport` が Phut で扱うための名前となる。

Phut の内部処理で、Phut が識別するインタフェース名をもとに、OS 上で使われる実際のデバイス名 (`L1_sport`, `L1_dport`) が決められる (図 B.2)。OS 上 (NetTester 外) から Phut で生成したインスタンスを操作する場合は、実際のデバイス名を使用する必要がある。また、デバイス名として使用できる文字列には上限があるため、名前 (文字列) の長さに注意する必要がある。

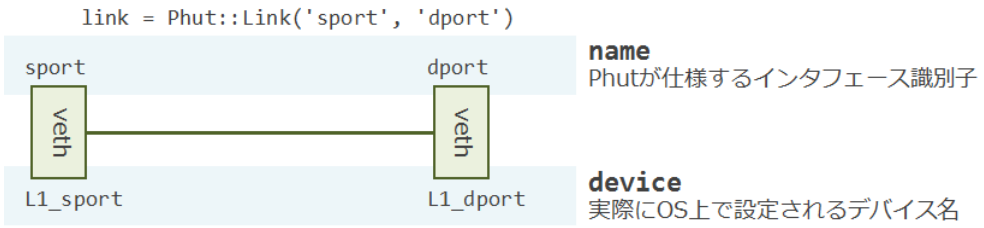

図 B.2　インタフェースの名前とデバイス名

生成した vLink(veth pair) をそのほかのインスタンス (vHost/vSwitch) に接続することで仮想ネットワークを構成する。

B.3.3　vHost

Phut では 2 種類のホストを取り扱う。

■Phut::Vhost　Ruby で実装された仮想ノードである。下記のように動作がシンプルで、直接的な動作チェックに向いている。

- UDP パケットの送受信ができる (`#send_packet`)。
  - ARP は送信せず、直接 IP(UDP) Unicast を送信する。
- ARP 処理しないので、あらかじめ ARP table を与えておく必要がある (`arp_entries` で "`IP1/MAC1,IP2/MAC2,...`" のような文字列を与える)。パケット送信 (`#send_packet`) するときに ARP entry が見つからなければ何もしない (パケットを送信しない)。

```
1 arp_entries = "192.168.0.1/00:ba:dc:ab:1e:01,192.168.0.2/00:ba:dc:ab:1e:02"
```

インスタンスを作るときに、`device` でこのホストに接続する veth を指定する (リスト B.9)。

リスト B.9　Phut::Vhost インスタンスの作成

```
1 host = Phut::Vhost.create(name: 'host_name',
2                           ip_address: ip_address,
3                           mac_address: mac_address,
4                           arp_entries: arp_entries,
5                           device: link.device('interface_name'))
```

■Phut::Netns　Linux namespace で作成したホスト。何らかのコマンド (プロセス) を namespace 上で実行する形をとる。これは、ネットワークのみホスト OS から分離した (namespace をわけた) 状態で、OS 上でコマンド実行するのと同様である。実行結果の処理などは自分で作りこみをする必要があるが、その分自由度がおおきく、OS 上で実行可能な処理は原則そのまま利用できる。

Network Namespace は Linux OS の機能であるため、NetTester で作成した namespace であっても、NetTester の外側 (OS) から使用可能である[*5]。

[*5] OS 上で `sudo ip netns exec <host namespace> <command>` する。

複雑なデバッグ作業をやりたい場合は Netns を使う必要がある。インスタンスを作ったあと `#device=` でこのホストに接続する veth を指定する (リスト B.10)。

リスト B.10　Phut::Netns インスタンスの作成

```
host = Phut::Netns.create(name: 'host_name',
                          ip_address: ip_address,
                          netmask: '255.255.255.0',
                          mac_address: mac_address)
host.device = link.device('interface_name')
```

B.3.4　vSwitch

NetTester 自体をテストするためのテストシナリオ[*6] 実装の中で、NetTester 本体の起動では以下のような処理をしている (リスト B.11)。

リスト B.11　NetTester の起動

```
Given(/^DPID が (\S+) の NetTester 物理スイッチ$/) do |dpid|
 @physical_test_switch = PhysicalTestSwitch.create(dpid: dpid.hex)
end

Given(/^NetTester を起動$/) do
 main_link = Phut::Link.create('ssw', 'psw')
 NetTester.run(network_device: main_link.device(:ssw),
               physical_switch_dpid: @physical_test_switch.dpid)
 @physical_test_switch.add_numbered_port(1, main_link.device(:psw))
end
```

これにより、図 B.4 のように仮想ネットワークが構成される。ここでは、NetTester 自身のテストのために、物理スイッチに相当するものを vSwitch(ソフトウェアスイッチ, `@physical_test_switch`) として起動している。

`NetTester.#run` の中では以下の処理をしている。

- trema の起動 (NetTesterController の起動)
- ssw の起動
- ssw に veth (`main_link.device(:ssw)`) を接続する。(port number 1)

[*6] https://github.com/net-tester/net-tester/blob/develop/features/step_definitions/net_tester_physical_switch_steps.rb, https://github.com/net-tester/net-tester/blob/develop/features/step_definitions/net_tester_steps.rb

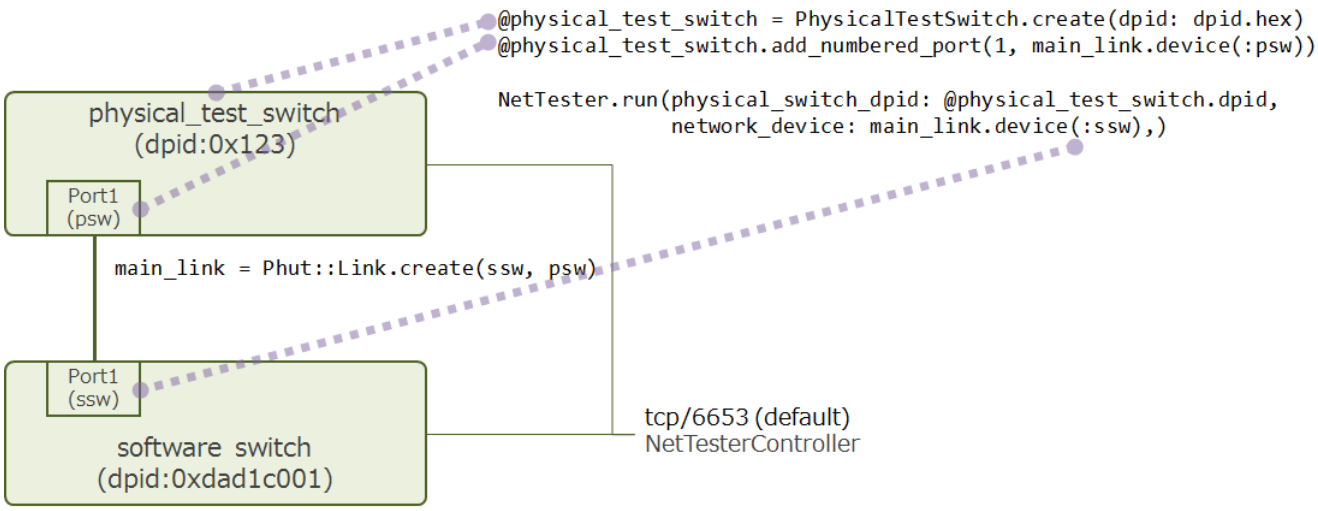

図 B.3　テスト用物理スイッチ (相当) の起動と接続

B.3.5　テスト対象としての vSwitch の利用

NetTester 自身をテストする場合、テスト対象ネットワークに相当するものをテストシナリオ中で用意する必要がある。テスト対象として vSwitch を起動する場合はリスト B.12 のようにおこなう[*7] 。リスト B.12 では、テスト対象として使用する vSwitch の起動およびコントローラとの接続 (Learning Switch として動作させるため) をおこなっている。

リスト B.12　vSwitch の起動

```
1 Given(/^$/) do
2   @testee_switch = TesteeSwitch.create(dpid: 0x1, tcp_port: 6654)
3   step %(I successfully run `trema run ../../vendor/learning_switch/lib/
      learning_switch.rb --port 6654 -L #{Phut.log_dir} -P #{Phut.pid_dir} -S #{Phut.
      socket_dir} --daemon`)
4 end
```

テストシナリオ内部[*8] では、テスト対象ネットワーク (@testee_switch) と物理スイッチ (@physical_test_switch) を接続するため、リスト B.13 のような処理をおこなう。

リスト B.13　vSwitch 間接続

```
1 Given(/^NetTester 物理スイッチとテスト対象のスイッチを次のように接続:$/) do |table|
2   table.hashes.each do |each|
3     pport_id = each['Physical Port'].to_i
4     tport_id = each['Testee Port'].to_i
```

[*7] https://github.com/net-tester/net-tester/blob/develop/features/step_definitions/ethernet_switch_steps.rb

[*8] https://github.com/net-tester/net-tester/blob/develop/features/step_definitions/net_tester_physical_switch_steps.rb

```
5       port_name = "pport#{pport_id}"
6       tport_name = "tport#{tport_id}"
7       link = Phut::Link.create(tport_name, port_name)
8       @physical_test_switch.add_numbered_port(pport_id, link.device(port_name))
9       @testee_switch.add_numbered_port(tport_id, link.device(tport_name))
10    end
11  end
```

これによって図 B.4 のように仮想ネットワークが構成される。

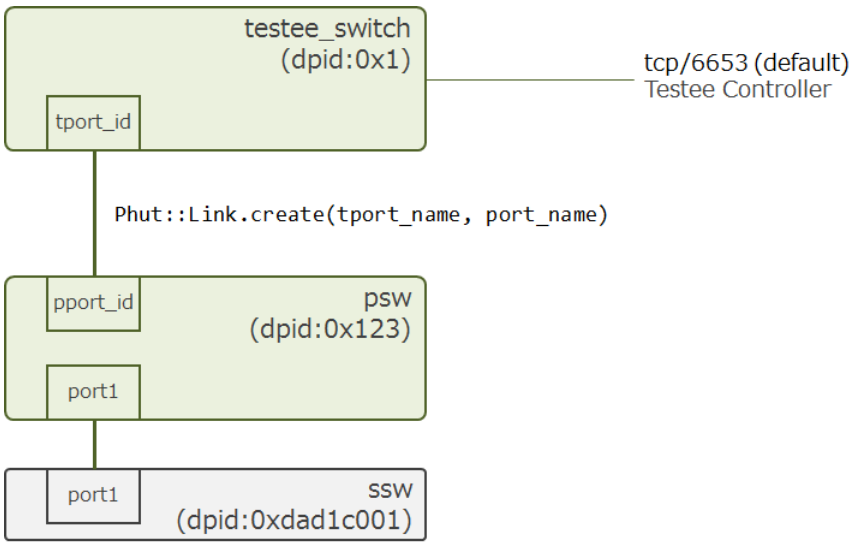

図 B.4　テスト対象機器 (スイッチ) の生成と接続

B.3.6　テスト対象としての vHost の利用

ポート間パッチのテスト[*9] では、テスト対象ネッ NetTester のテストシナリオネットワークに属するノードの生成・操作はリスト B.14 のように記述している。

リスト B.14　テスト用ノードの生成

```
1   Background:
2     Given DPID が 0x123 の NetTester 物理スイッチ
3     And NetTester を起動
4     And NetTester 物理スイッチとテスト対象ホストを次のように接続:
5       | Physical Port | Host |
6       |             2 |    1 |
7       |             3 |    2 |
8       |             4 |    3 |
```

[*9] https://github.com/net-tester/net-tester/blob/develop/features/internal_tests/p2p_patch.feature

テストシナリオ内部[*10](リスト B.15) では、テストシナリオ (リスト B.14) であたえられたテスト用ノードのパラメタを元に以下のようにノードを生成し、テストを実行する。

リスト B.15　テスト用ノードの生成と操作

```
Given(/^NetTester 物理スイッチとテスト対象ホストを次のように接続:$/) do |table|
  ip_of_host = {}
  mac_of_host = {}
  vhost_arp_list = []

  table.hashes.each do |each|
    host_id = each['Host']
    ip_address = "192.168.0.#{host_id}"
    ip_of_host[host_id] = ip_address
    mac_address = "00:ba:dc:ab:1e:#{sprintf('%02x', host_id)}"
    mac_of_host[host_id] = mac_address
    vhost_arp_list.append "#{ip_address}/#{mac_address}"
  end

  arp_entries = vhost_arp_list.join(',')
  table.hashes.each do |each|
    pport_id = each['Physical Port'].to_i
    pport_name = "pport#{pport_id}"
    host_id = each['Host']
    host_name = "host#{host_id}"
    link = Phut::Link.create(host_name, pport_name)
    Phut::Vhost.create(name: host_name,
                       ip_address: ip_of_host[host_id],
                       mac_address: mac_of_host[host_id],
                       device: link.device(host_name),
                       arp_entries: arp_entries)
    @physical_test_switch.add_numbered_port(pport_id, link.device(pport_name))
  end
end
```

リスト B.15 では Phut::Vhost を使用している。すべてのテスト用ノードの arp_entries を作るために IP/MAC を用意するところと、Phut instance を生成するパートに分割している。これにより、図 B.5 のように複数のテスト用ノードが作成され、テスト対象ネットワークとして物理スイッチ (psw) に接続される。

[*10] https://github.com/net-tester/net-tester/blob/develop/features/step_definitions/p2p_patch_steps.rb

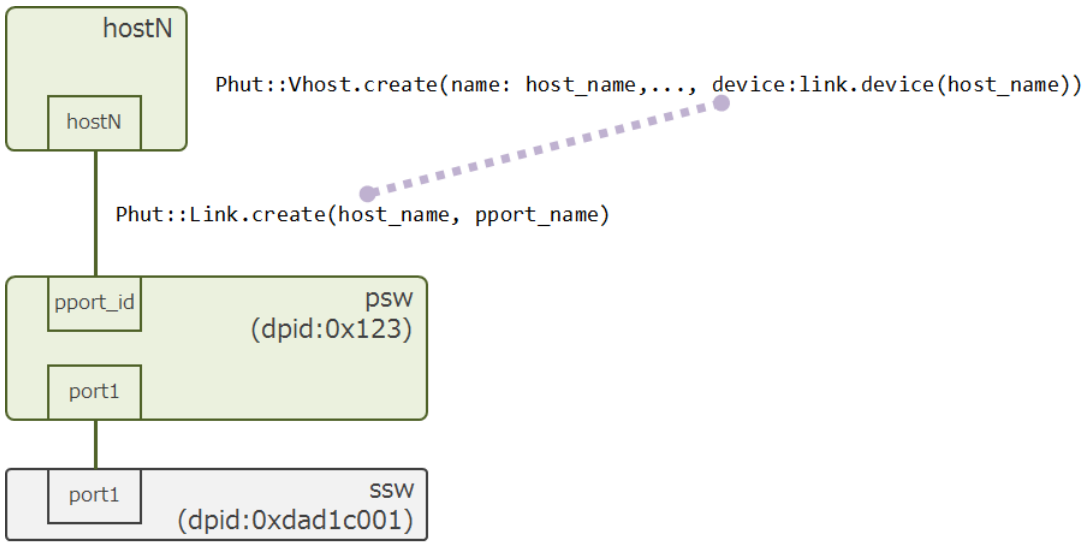

図 B.5　テスト対象機器 (ホスト) の生成と接続

参考文献

[1] “一般社団法人沖縄オープンラボラトリ | Okinawa Open Laboratory”, http://www.okinawaopenlabs.org/

[2] “L1Patch 応用 NW テストシステム プロジェクト”, http://www.okinawaopenlabs.org/archives/research2014/150410

[3] “L1patch 応用ネットワークテストシステム 試験結果レポート”. https://github.com/oolorg/ool-l1patch-dev/blob/master/report/l1pj-poc-report-20151114.pdf. 2015.

[4] “L1patch 応用ネットワークテストシステム 技術レポート”. https://github.com/oolorg/ool-l1patch-dev/blob/master/report/l1pj-tech-report-20151114.pdf. 2015.

[5] “OF-Patch 拡張 プロジェクト”, https://www.okinawaopenlabs.org/archives/research2014/141006

[6] 田部英樹, “VNF テストシナリオ自動化 PJ について”, https://www.okinawaopenlabs.org/wp/wp-content/uploads/8_VNF%E3%83%86%E3%82%B9%E3%83%88%E3%82%B7%E3%83%8A%E3%83%AA%E3%82%AA%E8%87%AA%E5%8B%95%E5%8C%96%E3%83%97%E3%83%AD%E3%82%B8%E3%82%A7%E3%82%AF%E3%83%88%E3%81%AB%E3%81%A4%E3%81%84%E3%81%A6.pdf, 2015.

[7] 渋谷惠美, 川上秀彦, 長谷川輝之, 山口弘純, “ホワイトボックススイッチと OSS を活用したネットワークに対する検証自動化システム設計に関する一提案”, 信学技報, vol. 116, no. 111, NS2016-34, pp. 35-40, 2016 年 6 月.

[8] “NEEDLEWORK | AP Communications”, http://www.ap-com.co.jp/ja/needlework/

[9] 鈴木飛鳥,“FW のポリシーテストを自動化してみた”, https://www.slideshare.net/tanksuzuki/fw-59102915, NetOpsCoding#2, 2016.

[10] “Infrataster by ryotarai”, http://infrataster.net/

[11] “toddproject/todd: A highly extensible framework for distributed capacity and connectivity testing (Testing on Demand....Distributed!)”, https://github.com/toddproject/todd

[12] “The Power of Test-Driven Network Automation”, https://keepingitclassless.net/2016/03/test-driven-network-automation/

[13] “OpenVNet”, https://openvnet.org/

[14] 山崎泰宏, “OpenVNet - SDN で物理ネットワークアプライアンスをプロビジョニングしよう”, https://www.slideshare.net/yasuhiro_yamazaki/openvnet-sdn, ネットワークプログラマビリティ勉強会#10, 2016.

[15] 村木暢哉, “SDN で始めるネットワークの運用改善（2）：SDN で物理ネットワークのテストを楽にする方法 (1/4) - @ IT”, http://www.atmarkit.co.jp/ait/articles/1612/27/news014.html, @IT,

2017.

[16] “ktbyers/netmiko: Multi-vendor library to simplify Paramiko SSH connections to network devices”, https://github.com/ktbyers/netmiko

[17] “trigger/trigger: Trigger is a robust network automation toolkit written in Python that was designed for interfacing with network devices.”, https://github.com/trigger/trigger

[18] “napalm-automation/napalm: Network Automation and Programmability Abstraction Layer with Multivendor support”, https://github.com/napalm-automation/napalm

[19] “Ansible is Simple IT Automation”, https://www.ansible.com/

[20] “Ansible 2.2、コンテナ、ネットワーク、クラウドサービス向けの新自動化機能を提供”, https://www.redhat.com/ja/about/rhjapan-press-releases/ansible-22-delivers-new-automation-capabilities-containers-networks-and-cloud-services

[21] “OpenConfig - Home”, http://www.openconfig.net/

[22] “OpenConfig - News”, http://www.openconfig.net/news/

[23] “Should TDD and BDD be used in conjunction? - Stack Overflow”, http://stackoverflow.com/questions/33746804/should-tdd-and-bdd-be-used-in-conjunction

[24] “net-tester/net-tester: 物理ネットワークのための受け入れテストツール”, https://github.com/net-tester/net-tester.

[25] “net-tester/examples: NetTester 用のサンプルテストコード”, https://github.com/net-tester/examples

[26] “NetTester でテスト自動化！ Network Test System Project - YouTube”, https://www.youtube.com/watch?v=C7z3aaWgsf4

[27] “ビヘイビア駆動開発 - Wikipedia”, https://ja.wikipedia.org/wiki/%E3%83%93%E3%83%98%E3%82%A4%E3%83%93%E3%82%A2%E9%A7%86%E5%8B%95%E9%96%8B%E7%99%BA.

[28] “NetTester で ad-hoc なテスト作業を拡張する - Qiita”, http://qiita.com/corestate55/items/d6a8cdc03de09a46877c

[29] “ovs-vswitchd.conf.db.5.txt”, http://openvswitch.org/support/dist-docs/ovs-vswitchd.conf.db.5.txt

[30] “[ovs-discuss] Forcing Switch to Reconnect”, https://mail.openvswitch.org/pipermail/ovs-discuss/2012-January/006368.html

[31] “ip-netns(8) - Linux manual page”, http://man7.org/linux/man-pages/man8/ip-netns.8.html

[32] “Ubuntu – xenial の build-essential パッケージに関する詳細”, http://packages.ubuntu.com/ja/xenial/build-essential

[33] “rbenv/rbenv: Groom your app ’ s Ruby environment”, https://github.com/rbenv/rbenv

[34] “pry/pry: An IRB alternative and runtime developer console”, https://github.com/pry/pry

[35] “implement a work-around of tcp checksum error in vlan trunk · Pull Request #7 · net-tester/net-tester”, https://github.com/net-tester/net-tester/pull/7

[36] David Chelimsky, Dave Astels , Zach Dennis “The RSpec Book”, 株式会社クイープ (訳), 角谷信太郎, 豊田祐司 (監修), Professional Ruby Series, 翔泳社, 2012.

[37] “クライアントの要望にこたえる Web サービス開発 ～「らせん型ワークフロー」のススメ～”. http://www.slideshare.net/mayuco/css-nite-in-sapporo-vol5-14085124

[38] 吉田友哉, 松崎吉伸, “幸せなパケット転送 ○○編”, https://www.janog.gr.jp/meeting/janog14/src/janog14-yoshida-maz.pdf, Janog14, 2004.

[39] “ヨーヨーダイン - Wikipedia”, https://ja.wikipedia.org/wiki/%E3%83%A8%E3%83%BC%E3%83%A8%E3%83%BC%E3%83%80%E3%82%A4%E3%83%B3

[40] IPA 独立行政法人 情報処理推進機構, “平成 25 年度 秋季 ネットワークスペシャリスト試験 午後 I 問題”, https://www.jitec.ipa.go.jp/1_04hanni_sukiru/mondai_kaitou_2013h25_2/2013h25a_nw_pm1_qs.pdf, 2013.

[41] “Cucumber”, https://cucumber.io/

[42] “expectacle”, https://rubygems.org/gems/expectacle

[43] “Logger - Trema - Relish”, http://www.relishapp.com/trema/trema/docs/logger

[44] “trema/pio: Packet parser and generator in Ruby”, https://github.com/trema/pio

[45] “trema/phut: Virtual network in seconds”, https://github.com/trema/phut

[46] “OpenFlow Switch Specification Version 1.0.0”, http://archive.openflow.org/documents/openflow-spec-v1.0.0.pdf

[47] “Fix the read_length of PacketIn#raw_data · Pull Request #320 · trema/pio”, https://github.com/trema/pio/pull/320

[48] “Debug print messages that are sent and received · Pull Request #433 · trema/trema”, https://github.com/trema/trema/pull/433

[49] “ScreenOS Release 6.3.0 Software Documentation for SSG, ISG, and NetScreen Series - Technical Documentation - Support - Juniper Networks”, http://www.juniper.net/techpubs/en_US/screenos6.3.0/information-products/pathway-pages/screenos/index.html

[50] “テストダブル - Wikipedia”, https://ja.wikipedia.org/wiki/%E3%83%86%E3%82%B9%E3%83%88%E3%83%80%E3%83%96%E3%83%AB

[51] “thoughtbot/factory_girl: A library for setting up Ruby objects as test data.”, https://github.com/thoughtbot/factory_girl