L1Patch 応用ネットワークテストシステム
技術レポート

2015/11/13

内容

1 用語

表 1-1 用語一覧

用語	略語・省略表記	分類	意味
沖縄オープンラボラトリ	OOL	一般	(略語定義) 一般社団法人 沖縄オープンラボラトリ https://www.okinawaopenlabs.org/
ネットワーク	NW	一般	(略語定義)
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(特定の物理結線を集中して付け替えるための機構)。本プロジェクトでは、「ネットワーク用パッチパネルと同等のことを行えるシステム」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)
OF パッチ（OFpatch)		OOL	OOL で実装している、OpenFlow を使った L1 パッチパネル機能を持つプロダクト
Firewall	FW	一般	(略語定義)
Load Balancer	LB	一般	(略語定義)
ディスパッチャ (dispatcher)		プロジェクト	L1patch と同じ。初期モデル検討の中では L1patch という名称ではなく、任意の物理箇所にパケットを送出するものという意味でディスパッチャという語を使っていた。今回の PoC 実装のプログラム中でも使用している。
ジェネレータ (generator)		プロジェクト	「ネットワークに対して行いたいテスト」を実行するソフトウェア。ネットワークのテストで行うことは、一般的に、パケットを吐いてその応答を観測するものなので「ジェネレータ」と記載している。NW テストのためのトラフィック(パケット)生成機能/そのためのアプリケーション。
テスト対象 NW		一般	検証環境で構築する、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。
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 NW を構成するいずれかひとつの機器。
Mininet		一般	OSS のネットワークテスト用ツール
Open vSwitch	OVS	一般	Linux 上で動作する L2 スイッチ/OFS 実装
ワイヤ(wire)		プロジェクト	L1patch によって実現されるデバイス間接続(L1patch が論理的に実現する結線)

専有モード (exclusive mode)		プロジェクト	ワイヤ種類のひとつ。物理ポート単位で 1 対 1 のマッピングを行うワイヤ。テスト対象 NW 機器(DUT)間ポートを接続し、テスト対象 NW の物理トポロジを構成するために使用する。
共有モード (shared mode)		プロジェクト	ワイヤ種類のひとつ。MAC アドレス単位でマッピングを行うワイヤ。ひとつの物理ポートに複数のテスト用ノードを配置するために使用する。

2 はじめに

2.1 本書の目的

本書の目的は、本プロジェクト(L1patch 応用ネットワークテストシステムプロジェクト: 以降 L1patchPJ)で実行した実証実験について、実証実験のために作成したツールの設計および使用方法など、技術面の詳細をまとめることである。本プロジェクトの目的、実証実験のターゲット、実行結果とそれに対する評価等、以下の各項目については「試験結果レポート」に記載しているため本書では取り上げない。

- 本プロジェクトの概要、目的
- PoC のターゲット設定、PoC の実施内容と結果
  - L1patch の基本的な仕様

本書では、「試験結果レポート」で解説した、プロジェクトの目的、PoC の目的、L1patch の基本動作に対する知識を前提とした上で、実装面の詳細についての解説を行う。特に 3 章・4 章は L1patch の基本的な使い方を机上で(実機のテスト対象を使用せずに)試す際のチュートリアルとして使うことを想定しているため、実際に動作させてみる際は本書に沿って机上テストの実行を行うことを推奨する。

3 L1patch によるネットワークテスト実行の流れ

3.1 テスト計画とテスト方針の検討

一般的に、ネットワークのテスト行う際は以下の計画検討が行われる。

- テストの目的とそのための環境の設定・設計
  - 本番環境でのオペレーションや、オペレーションを行った際の実際の機材の動作を確認したい、特定のデバイスの機能設定方法、そのときの動作が想定通りのものかどうかを確認したいなど、まず実際に行いたいこと・確認したいことを決める。また、それに必要な環境や機材などを決める。
  - 目的の実現に対して必要な機材をもとに、どのようなネットワークを構築すべきか、テスト対象ネットワーク(検証環境で構築するネットワーク)を決める。
- 確認方法と手順、判断基準の設定
  - テスト対象 NW が決まったら、その上でどのような作業をし、何を確認すべきかを決める。
  - テスト目的を満たすためにどのような指標、動作、状態を確認して、どうなっていれば成功/失敗なのかを決める。

以降、今回行った PoC においてどのようなテストを設定したかを解説する。

3.2 テスト対象ネットワークの設計

本書ではテスト対象のネットワークとして以下の 2 種類を取り上げる。

- 机上テスト
  - L1patchPJ で作成した各種ツールを仮想環境(ひとつの OS 上)でテストするための環境。Mininet 上でテスト対象の環境を構成し、外部の機材やツールを利用せずに各ツールの動作確認をすることが目的のテスト。

- 5 台構成テスト
  - L1patchPJ で最終的に実施した PoC 環境。テスト対象 NW として、ルータ・スイッチ合計 5 台を使って構築した環境。
  - 以降本書では、実機環境を含む設定・設計・オペレーションなどで検討に加えるべき点(仮想環境内、机上テストだと考慮されない部分)の説明のためにこの環境を取り上げる。5 台構成テストについては「試験結果レポート」を参照すること。

以降本書では主に机上テストを対象に解説を行う。机上テストは OS の外の環境(実機のテスト対象ネットワーク)を使用しないテストなので、実機をテストする際に必要な項目の一部が現れない。そうした差分については 5 台構成テストの例をもとに解説を追加する。

3.3 机上テスト

3.3.1 机上テストに求められる要件

L1patch の目的と今回の PoC 目標設定から、L1patch としては Mininet によるテスト用のノードの生成と配置について以下の条件を満たせることが要件と考えた。

- 多数のテスト用ノードに対して同数の物理ポートを要する構成だと、物理環境制約(テスト用ノードを収容する機材の物理 NIC 数(ポート数))に左右されてしまうため、テストの拡張性が大きく制限されてしまう。
  - 多数のパターン網羅、特に物理構成の網羅を考慮し、環境(機材数)の拡張性に対応させるためには、テスト用ノード(Mininet host)機材で、少数の物理ポートに対して複数の論理ワイヤがマッピングできなければいけない。
  - テスト対象機器についても、テスト用ノードを受け入れるために多数の物理ポートを必要とする構成にしてしまうと、物理構成による制約を受けてしまう。可能な限り少数の物理ポートに対して複数のテスト用ノードを接続できることが望ましい。
- 検証環境用途のテスト対象 NW の自動構築
  - 検証環境でのテストを用途として想定すると、テスト対象機器間のトポロジ(テスト対象ネットワークのトポロジ)構成が想定される。テスト対象のネットワーク機器(DUT)間を、物理ポート 1 対 1 マッピングで接続されている状況を、L1patch を使って構成する。

3.3.2 机上テスト

L1patch の基本動作確認を行うためのテスト環境として図 3-1 のような環境を設定した。3.3.1 項であげたテスト要件をもとに、机上テストでは以下の点のテストが可能となるようにしている。

- L2 制御を用いた 1:N 接続(図 3-1/wire1～wire4)
  - DUT 側が VLAN trunk port となっており、異なるセグメントの複数のテスト用ノード接続を受け入れる: wire1, wire2
  - DUT 側が VLAN access port となっており、同一セグメント内の複数のテスト用ノード接続を受け入れる: wire3, wire4
- L1 制御を用いた 1:1 接続(図 3-1/wire5)

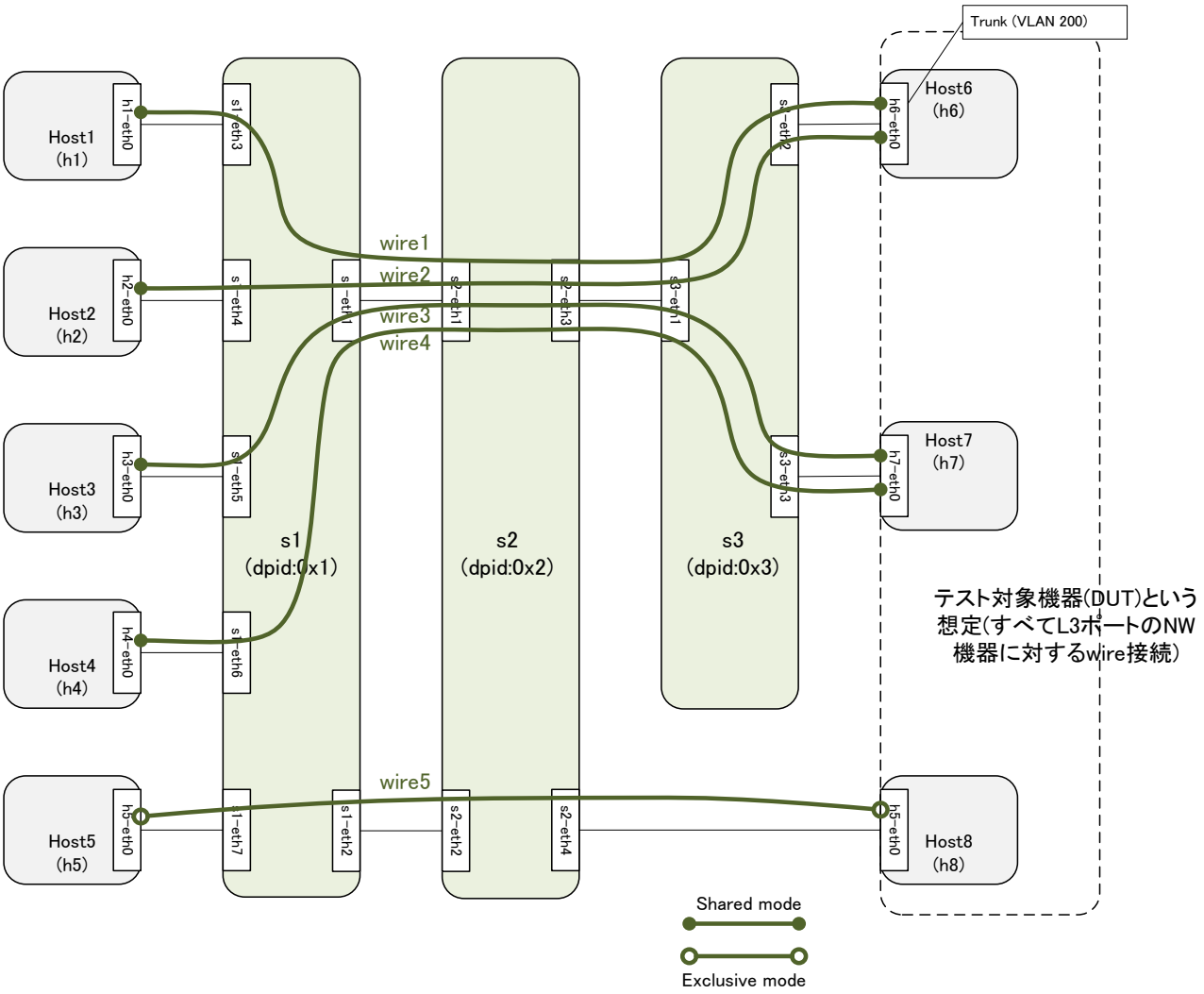

図 3-1 机上テストの構成図

机上テストでは、すべてのノード(h1-h8)は同じサブネットの IP アドレスを持つ。h6～h8 はテスト対象機器(DUT)であり、図 3-1 に記載したとおり L3 ポートを持つ NW 機器に相当するが、実際には Mininet host で代替しているだけなのでルーティングやフォワーディング動作は行われない。したがって、この環境ではワイヤで結ばれた 2 ノード間だけが通信可能になる。

3.4 設定ファイルの定義

3.4.1 設定ファイルの制約と考え方

テスト対象のネットワークの構成、テストを行うためのテスト用ノード配置などが決まったら、L1patch を使ってその構成を実現するために、L1patch の構成定義(設定ファイル作成)を行う。

L1patch の実装を行う上で、設定ファイルの定義としては以下の点を考慮した。

- データ構造表現は JSON で統一する。
- ID/Number(Datapath ID, Port Number など)は可能な限り参照名を付ける。設定ファイルの可読性を挙げるため、中は番号ではなく名前で参照する。

- 複数のファイルで同じ「参照名」を使う必要がある点に注意してください。

テスト実施にあたって必要な設定ファイルは表 3-1 の 4 種類である。机上テストの際に作成した各ファイルを取り上げ、各節で解説を加える。

表 3-1 設定ファイル一覧

設定ファイル	設定ファイル名(prefix)	内容	参照
物理構成	nodeinfo.json	OFS やテスト用ノードのネットワークパラメータなど、L1patch 物理構成情報の設定	3.4.2
論理構成	wireinfo.json	テスト用ノードと DUT ポート間ワイヤ(L1patch 論理構成)の設定	3.4.3
テストパターン	scenario_pattern.json	通信テストパターンの定義(確認したいトラフィックパターンの定義)	3.4.4
テスト自動実行	testdefs.json	テスト自動実行に必要なパラメータの定義	3.4.5

3.4.2 物理構成

表 3-2 に L1patch を使用してテスト環境を定義するための物理環境定義ファイルを示す。基本的には、最終的に L1patch 動作ルール(OFS へ設定するフロールール)を生成するために必要な、テスト用ノード、接続先 DUT のポート設定である。

表 3-2 L1patch 物理構成定義

机上テスト: L1patch 物理構成定義ファイル(nodeinfo_topo2.json)

```
{
テスト用ノード(Mininet Host)の情報定義。
 "test-hosts": {
  "test-hosts": {
    "h1": {
      "port-index": {
        "h1-eth0": {
L1patch 動作のために生成するフロールールは、テスト用ノードの MAC アドレスをキーにするため、テスト用ノード設定では MAC アドレス指定が必須となる。
          "mac-addr": "0a:00:00:00:00:01",
IP アドレスとデフォルトゲートウェイについてはオプショナル。[1]
          "ip-addr": "192.168.2.11/24",
          "gateway": "192.168.2.254"
```

[1] 本来デフォルトゲートウェイはインタフェースに対してではなくホストに対してひとつ設定するものなので、設定ファイル上のデータモデルとしてはあまり良くない。(当初テスト用ノードに対するデフォルトゲートウェイ設定が考慮漏れしたこと・テスト用ノードは 1 ホスト 1NIC を暗黙的に仮定していることによる。)

```
      }
    }
  },
  "h2": {
    "port-index": {
      "h2-eth0": {
        "mac-addr": "0a:00:00:00:00:02",
        "ip-addr": "192.168.2.12/24",
        "gateway": "192.168.2.254"
      }
    }
  },
  "h3": {
    "port-index": {
      "h3-eth0": {
        "mac-addr": "0a:00:00:00:00:03",
        "ip-addr": "192.168.2.13/24",
        "gateway": "192.168.2.254"
      }
    }
  },
  "h4": {
    "port-index": {
      "h4-eth0": {
        "mac-addr": "0a:00:00:00:00:04",
        "ip-addr": "192.168.2.14/24",
        "gateway": "192.168.2.254"
      }
    }
  },
  "h5": {
    "port-index": {
      "h5-eth0": {
        "mac-addr": "0a:00:00:00:00:05",
        "ip-addr": "192.168.2.15/24"
      }
    }
  }
```

```
  },
```

テスト対象機器(DUT)情報定義。
L1patch でテスト用ノードを接続する DUT 側ポート情報を指定する。(机上テストではここも Mininet host を使用するが実際には実機ポート情報が入る。)

```
 "dut-hosts": {
   "h6": {
```

ポート一覧(index), テスト対象の複数のポートを使う場合は列挙していく。

```
     "port-index": {
```

L1patch でテスト用ノードを接続する先、DUT 側ポートの設定として VLAN Tag を使用するかどうか(trunk port かどうか)を設定する。ポート名(h6 は trunk port 設定なので、VLAN サブインタフェースとしてポート名を定義) 複数 VLAN 使用する場合は VLAN ID ごとにサブインタフェースを定義する。

```
       "h6-eth0.200": {
         "vlan-tagged": true,
         "vlan-id": 200
       }
     }
   },
   "h7": {
     "port-index": {
       "h7-eth0": {
```

VLAN Tag を使わないポート(access port)の場合。

```
         "vlan-tagged": false
       }
     }
   },
   "h8": {
     "port-index": {
       "h8-eth0": {
         "vlan-tagged": false
       }
     }
   }
 },
```

```
Dispatcher = L1patch として動作する OpenFlow Switch の情報定義
 "dispatchers": {
スイッチ"s1"は Mininet が起動し、テスト用ノードが接続される OFS である。PoC 上は"s1/dpid:1"をテスト用ノードが接続されるデフォルトのスイッチとして仮定しているので注意。
   "s1": {
OFC Datapath ID (10 進数で指定する点に注意)
     "datapath-id": 1,
コメント
     "description": "host edge switch",
OFS のポート一覧(使うもの)
     "port-index": {
ポート名
       "s1-eth1": {
ポート番号(フロールール中で指定する OFS ポート番号)
         "number": 1
       },
       "s1-eth2": {
         "number": 2
       },
       "s1-eth3": {
         "number": 3
       },
       "s1-eth4": {
         "number": 4
       },
       "s1-eth5": {
         "number": 5
       }
     }
   },
   "s2": {
     "datapath-id": 2,
     "description": "inter switch",
     "port-index": {
       "s2-eth1": {
         "number": 1
       },
       "s2-eth2": {
```

```
        "number": 2
      },
      "s2-eth3": {
        "number": 3
      },
      "s2-eth4": {
        "number": 4
      }
    }
  },
  "s3": {
    "datapath-id": 3,
    "description": "DUT edge switch",
    "port-index": {
      "s3-eth1": {
        "number": 1
      },
      "s3-eth2": {
        "number": 2
      },
      "s3-eth3": {
        "number": 3
      },
      "s3-eth4": {
        "number": 4
      },
      "s3-eth5": {
        "number": 5
      }
    }
  }
},
```

物理トポロジ(どのポートからどのポートに対して物理リンクが張られているか)の情報。L1patch で接続するテスト用ノード・OFS(dispatcher)・DUT ポートについて、物理リンクで直接結線されているポートのペアで定義。(ポート間対応、接続バリデーションなどで使用)

```
 "link-list": [
   [["h1", "h1-eth0"], ["s1", "s1-eth2"]],
   [["h2", "h2-eth0"], ["s1", "s1-eth3"]],
   [["h3", "h3-eth0"], ["s1", "s1-eth4"]],
   [["h4", "h4-eth0"], ["s1", "s1-eth5"]],
   [["h5", "h5-eth0"], ["s2", "s2-eth4"]],
   [["s1", "s1-eth1"], ["s2", "s2-eth1"]],
   [["s2", "s2-eth2"], ["s3", "s3-eth1"]],
   [["s2", "s2-eth3"], ["s3", "s3-eth2"]],
   [["h6", "h6-eth0.200"], ["s3", "s3-eth3"]],
   [["h7", "h7-eth0"], ["s3", "s3-eth4"]],
   [["h8", "h8-eth0"], ["s3", "s3-eth5"]]
 ]
}
```

DUT 側ポートの VLAN 指定について、ひとつの物理ポートで複数の VLAN を利用する Trunk ポートでは表 3-3 のように定義する。

表 3-3 DUT トランクポート設定例

5 台構成テスト: 物理構成定義ファイル 抜粋 (nodeinfo_topo5.json)

```
  "dut-hosts": {
    "L2SW1": {
      "port-index": {
        "gi1/0/2.2013": {
          "vlan-tagged": true,
          "vlan-id": 2013
        },
        "gi1/0/2.2015": {
          "vlan-tagged": true,
          "vlan-id": 2015
        },
        "gi1/0/2.2016": {
          "vlan-tagged": true,
          "vlan-id": 2016
        },
```

```
（省略）
        }
    },
```

3.4.3　　論理構成

表 3-4 に、L1patch としてどのようなノード間接続(論理構成: L1patch によって仮想的に実現される”ワイヤ”)を実現するかを定義するための論理構成定義ファイルを示す。ワイヤ定義なので、端点となるホスト・ポートの関連が定義の中心となるが、フロールール生成のための情報を追加で定義している。

- ワイヤが経由する OFS(dispatcher)上のポート: OFS 間接続を検出して中間経路を自動的に生成する機能は現時点では実装されていないため、端点(テスト用ノード～接続先 DUT ポート)とその間で経由する経路を手動で指定している。
- OpenFlow 制御ロジックの上で、共有モードワイヤのグループを識別するための情報が別途必要になるため、ワイヤグループ(“wiregroup”)を定義している。(制御ロジックは 5 章で解説する。)

表 3-4 L1patch 論理構成定義

机上テスト: L1patch 論理構成定義ファイル(wireinfo_topo2.json)

```
{
 "wire-index" : {
論理接続(ワイヤ)の定義
   "wire1": {
コメント
     "description": "wire between host1(h1) and h6",
ワイヤ動作モードの設定(共有モード)
     "mode": "shared",
接続元(テスト用ノード側)の端点
     "test-host-port": ["h1", "h1-eth0"],
接続先(テスト対象機器/DUT)側の端点
     "dut-host-port": ["h6", "h6-eth0.200"],
端点～端点を結ぶときに通過する OFS のポート(現在は自動検出ではなく手動設定)。共有モードワイヤなので、同じ物理ポートを複数のワイヤ(wire2～wire4)で使用する。
     "path": [
       ["s1", "s1-eth2"],
       ["s1", "s1-eth1"],
       ["s2", "s2-eth1"],
       ["s2", "s2-eth2"],
       ["s3", "s3-eth1"],
       ["s3", "s3-eth3"]
```

```
    ]
  },
  "wire2": {
    "description": "wire between host2(h2) and h6",
    "mode": "shared",
    "test-host-port": ["h2", "h2-eth0"],
    "dut-host-port": ["h6", "h6-eth0.200"],
    "path": [
      ["s1", "s1-eth3"],
      ["s1", "s1-eth1"],
      ["s2", "s2-eth1"],
      ["s2", "s2-eth2"],
      ["s3", "s3-eth1"],
      ["s3", "s3-eth3"]
    ]
  },
  "wire3": {
    "description": "wire between host3(h3) and h7",
    "mode": "shared",
    "test-host-port": ["h3", "h3-eth0"],
    "dut-host-port": ["h7", "h7-eth0"],
    "path": [
      ["s1", "s1-eth4"],
      ["s1", "s1-eth1"],
      ["s2", "s2-eth1"],
      ["s2", "s2-eth2"],
      ["s3", "s3-eth1"],
      ["s3", "s3-eth4"]
    ]
  },
  "wire4": {
    "description": "wire between host4(h4) and h7",
    "mode": "shared",
    "test-host-port": ["h4", "h4-eth0"],
    "dut-host-port": ["h7", "h7-eth0"],
   "path": [
      ["s1", "s1-eth5"],
      ["s1", "s1-eth1"],
```

```
        ["s2", "s2-eth1"],
        ["s2", "s2-eth2"],
        ["s3", "s3-eth1"],
        ["s3", "s3-eth4"]
      ]
    },
    "wire5": {
      "description": "wire between host5(h5) and h8",
専有モードワイヤ
      "mode": "exclusive",
専有モードワイヤ端点の定義[2]
      "test-host-port": ["h5", "h5-eth0"],
      "dut-host-port": ["h8", "h8-eth0"],
      "path": [
        ["s2", "s2-eth4"],
        ["s2", "s2-eth3"],
        ["s3", "s3-eth2"],
        ["s3", "s3-eth5"]
      ]
    }
  },
ワイヤグループは、共有モードワイヤの L2 broadcast domain 設定。
ひとつの DUT ポートに複数の wire を付ける場合、どの wire が同じセグメントに所属している wire なのかを別途指定する必要があるため。
  "wire-group-index": {
    "wiregroup1": {
      "description": "DUT:h6, PORT:s6-eth6.200(vlan 200)",
"id"(wire group id)の実装については 5.3 節参照。
      "id": 101,
wiregroup1 は DUT h6/h6-eth6.200 に接続されるふたつのワイヤを含むグループ
      "wires": ["wire1", "wire2"]
    },
    "wiregroup2": {
      "description": "DUT:h7, PORT:h7-eth0",
      "id": 102,
```

[2] 専有モードワイヤはテスト対象 NW 機器間(DUT-DUT)を接続するため、フィールド名”test-host-port”にも DUT のポートを設定することになる。これは実装上最初に共有モードワイヤ動作を実装し、そこで定義したフィールド名を流用してしまったことによるもので、本来はフィールド名を他の名称にした方が良い。

```
      "wires": ["wire3", "wire4"]
    }
  }
}
```

3.4.4 テストパターンとテストシナリオ

表 3-5 にテストパターン(どこからどこに対してテストトラフィック(ping)を生成するか)の定義ファイルを示す。テスト対象のネットワーク上で、どのようなテストを行うか(テストの方針、テストトラフィックのパターン定義)については「試験結果レポート/5.2 テスト方針」を参照すること。

表 3-5 テストパターン定義

机上テスト: テストパターン定義ファイル(scenario_pattern_topo2.json)

```
{
 "params": {
   "@h1@": "192.168.2.11",
   "@h2@": "192.168.2.12",
   "@h3@": "192.168.2.13",
   "@h4@": "192.168.2.14",
   "@h5@": "192.168.2.15",
   "@h6@": "192.168.2.106",
   "@h7@": "192.168.2.107",
   "@h8@": "192.168.2.108"
 },
 "scenarios": {
   "test-node to dut": {
     "shared-wire-to-h6(SUCCESS)": [
       ["h1", "h2"],
       ["@h6@"]
     ],
     "shared-wire-to-h7(SUCCESS)": [
       ["h3", "h4"],
       ["@h7@"]
     ],
     "exclusive-wire-to-h8(SUCCESS)": [
       ["h5"],
       ["@h8@"]
     ]
   },
```

```
   "dut to test-node": {
     "shared-wire-to-h1_h2(SUCCESS)": [
       ["h6"],
       ["@h1@", "@h2@"]
     ],
     "shared-wire-to-h3_h4(SUCCESS)": [
       ["h7"],
       ["@h3@", "@h4@"]
     ],
     "exclusive-wire-to-h5(SUCCESS)": [
       ["h8"],
       ["@h5@"]
     ]
   }
 }
}
```

定義ファイル中は以下の記法を使用している。

- テスト用ノードのホスト名は物理構成定義ファイル中で使用されている名前を参照すること。
- “@hostname@”はパラメータで、ファイル先頭に定義されている “params” の値に置換される。Mininet 内にあるホストについてはホスト名そのままで参照(操作)可能だが、Mininet の外にあるホストや IP を扱うためにパラメータを定義している。
  - ホスト名による参照をするために、文字列置換機能を用意している。
  - 実装上、ping 送信元(source)になれるのは Mininet host のみである(ツールとして Mininet の外部にあるホストを操作する機能はない)。テスト用ノード(h1-h5)についてもパラメータ定義しているが、これはパラメータを使うことで、送信元を選択可能にするためである。
- パターン定義による個々のタスクの生成
  - 送信元/送信先になる複数のグループを定義する。各グループの要素を取り出して、他のグループのすべてのホストに対して ping をおくるという個々のルール(テストケース)を生成する。
  - タスク名の後に想定される結果を “(SUCCESS)” または “(FAIL)” で定義する。
    - なにをもって「成功」「失敗」とするかはテスト自動実行スクリプト中(scenario_test_runner.py)の関数で定義している。今回、ping パケットを N 回送信したときに N/2 回以上(半分以上)通信成功していることを「成功」の条件とした。
- 表 3-6 のようなテスト定義であれば、h1→@h6@, h2→@h6@, @h6@→h1, @h6@→h2 のパターンを生成することになるが、パラメータ指定(@hostname@)のホストは ping 送信もとにならない(Mininet 外部にある IP の想定なので操作可能なものではない)ので、最終的に生成されるのは h1→@h6@, h2→@h6@の 2 パターンとなる。

表 3-6 パターン定義と生成されるテストケースの例

パターン定義	最終的に生成されるテストケース
"shared-wire-to-h6(SUCCESS)": [["h1", "h2"], ["@h6@"]],	"task-list": [{ "source": "h1", "destination": "192.168.2.106", "task": "[shared-wire-to-h6] ping h1 to @h6@", "command": "ping", "expect": "SUCCESS" }, { "source": "h2", "destination": "192.168.2.106", "task": "[shared-wire-to-h6] ping h2 to @h6@", "command": "ping", "expect": "SUCCESS" },

3.4.5 テスト自動実行

表 3-7・表 3-8 にテスト自動実行スクリプトで使用する定義ファイルを示す。テスト自動実行スクリプトでは、実行環境の情報とテストアプリケーションの情報からなる。

- "l1patch-defs"セクション
  - テスト対象ネットワークに対する L1patch の設定(定義ファイル)と操作方法の定義
  - フロールール生成(設定・削除)のためのコマンド
    - 実装上、フローの生成するスクリプト、生成されたフローを OFC へ投入するスクリプトなどを個別に実装している。自動実行スクリプトではこれらを組み込むのではなく別プロセスとして実行する形にしている。
  - 専有モードワイヤ・共有モードワイヤで個別に設定・削除するように分けられているのは、ひとつの環境で繰り返しテストを行う際に、すべてのテストを通して維持すべきネットワーク定義と、個々のテストアプリケーションで設定すべきネットワーク構成で分割するためである。(「試験結果レポート/6.2.1 L1patch のテスト実行フェーズの考え方」を参照。)
- "test-env-param"セクション
  - L1patc として動作させるためのフロー生成に必要なパラメータの定義。
- "ping-test-param"セクション
  - テストアプリケーション(今回は ping)のためのパラメータ定義。

表 3-7 テスト自動実行定義

机上テスト: 自動テスト定義ファイル(testdefs_topo2.json)

L1patch の設定情報

```
 "l1patch-defs": {
```

L1patch 物理構成定義ファイル(read)

```
   "physical-info-file": "nodeinfo_topo2.json",
```

L1patch 論理構成定義ファイル(read)

```
   "logical-info-file": "wireinfo_topo2.json",
```

専有モードワイヤ用のフロールールを保存するファイル(write)

```
   "exclusive-wire-flows-file": "flows_exclusive_topo2.json",
```

共有モードワイヤ用のフロールールを保存するファイル(write)

```
   "shared-wire-flows-file": "flows_shared_topo2.json",
```

専有モードワイヤ用のフロールールを生成するためのコマンド

```
   "generate-exclusive-wire-flows-command": "python run_l1patch.py -p @physical-info@ -l @logical-info@ -m exclusive > @exclusive-wire-flows@",
```

共有モードワイヤ 用のフロールールを生成するためのコマンド

```
   "generate-shared-wire-flows-command": "python run_l1patch.py -p @physical-info@ -l @logical-info@ -m shared > @shared-wire-flows@",
```

専有モードワイヤ用のフロールールを OFC に PUT するコマンド

```
   "put-exclusive-wire-flows-command": "cat @exclusive-wire-flows@ | python patch_ofc_rest_knocker.py -m put",
```

共有モードワイヤ用 のフロールールを OFC に PUT するコマンド

```
   "put-shared-wire-flows-command": "cat @shared-wire-flows@ | python patch_ofc_rest_knocker.py -m put",
```

専有モードワイヤ用のフロールールを OFC に DELETE するコマンド

```
   "delete-exclusive-wire-flows-command": "cat @exclusive-wire-flows@ | python patch_ofc_rest_knocker.py -m delete",
```

共有モードワイヤ用のフロールールを OFC に DELETE するコマンド

```
   "delete-shared-wire-flows-command": "cat @shared-wire-flows@ | python patch_ofc_rest_knocker.py -m delete"
 },
```

テスト実行環境のパラメータ定義

```
  “test-env-params” : {
```

L1patch を構成する OFS で使われる OpenFlow Version ("OpenFlow10" or "OpenFlow13")

```
   "ofs-openflow-version": "OpenFlow10",
```

Mininet サーバが外部ネットワークと接続するために使うインタフェース名(複数設定可能)

```
   "mininet-external-interfaces": []
 },
```

```
シナリオテストのパラメータ定義
 "ping-test-params": {
ping テストで使うコマンド(コマンドオプション指定注意)
   "ping-command": "ping -i 0.2 -c 5",
 }
}
```

表 3-8 自動テスト定義(オプション)

5 台構成テスト: 自動テスト定義ファイル 抜粋(testdefs_topo5.json)
"test-env-params" : { 物理 NW のテスト(5 台構成テスト)では OS 外部のネットワークと接続するために OS 上のインタフェースを指定する。 "mininet-external-interfaces": ["eth2"] },

"ping-test-params": {
"ping-command": "ping -i 0.2 -c 5",
オプショナル: 実行結果が想定(expect)と合わないときのリトライ回数(最大)。
デフォルト(指定なし)は最大 3 回。
"ping-max-retry": 5,
オプショナル: リトライ時の実行間隔(秒)
リトライするときの待ち時間の設定。リトライするたびに加算していく。この設定の場合、リトライ 1 回目は 1 秒、2 回目は 2 秒、3 回目は 3 秒と実行間隔を長くとる。デフォルト(指定なし)は 1 秒。
"ping-retry-interval": 1
}

表 3-7 では以下の記法を使用している。

- "foo-command"で使われている "@parameter@" はファイル参照用のパラメータ定義を表す。"@parameter@"は同じセクションにある "parameter-file" キーを参照して値に置き換える。
- "foo-command"で定義されているコマンドはパラメータ部分を文字列置換して、サブプロセスとして実行される。
  - "shared-wire-flows-file"など、定義ファイルから自動生成されるフロールール/テストシナリオは、テスト実行時に毎回更新(再生成)されて上書きされていくので繰り返しテストを実行する際は注意すること。

4 L1patch の使い方

4.1 L1patch 環境のセットアップ

使用している機器型番・バージョン等の詳細については「試験結果レポート/8.1 PoC 環境で使用した機器・ソフトウェア情報」を参照すること。本書では一般的なソフトウェアのインストール方法は扱わない。今回実装した

L1patch ツール各種は下記の条件を満たす Linux OS 上で動作する(PoC では Ubuntu server を使用)。

- Mininet2.2.22 以降
- Ryu3.24 以降
- Open vSwitch 2.0 以降
- Python2.7 系

なお、Mininet を動作させるサーバは、2 個以上のネットワークインタフェースを持つ必要がある。L1patch として Mininet OVS にインタフェースを組み込む必要があるが、OVS へ組み込むと、サーバに対するリモートアクセスや OFC との通信(OFC を別ホストに用意する場合)のためには使用できなくなるためである。

4.2 テスト仕様の検討

机上テストのテスト仕様の考え方については 3.3 節参照。テスト仕様の検討については従来のテスト仕様(テスト方針)検討と同様である。テスト対象ネットワークにおいて、どのようなトラフィックを発生させ、何をテストしたいのか、テストの成否を判断する指標が何かを検討する。

4.3 テスト用ネットワークの設定ファイル作成

テスト対象ネットワークの情報と、テストしたい事をもとに、L1patch の設定情報を作成する。

- 物理環境情報 (nodeinfo.json, 3.4.2 項)
  - 机上テストの場合、テスト対象が「複数パターンのワイヤを実現すること」になる。3.3 節で示した基本的なワイヤ種別のパターンと中間経路の設定(複数の OFS をホップしてワイヤが作成できること)を確認するために図 3-1 のようなネットワーク構成・テスト用ノード配置とした。
- 論理環境情報 (wireinfo.json, 3.4.3 項)
  - 図 3-1 のようにテスト用ノードを、どの DUT ポートへワイヤで接続するかを決める。
- テスト成否評価の検討
  - 本来、DUT に対するテスト用ノードとその上でのトラフィックパターンがテスト方針で決められているが、机上テストの場合はワイヤそのものを作成することがテスト対象であるため、ワイヤを作成した後のテストという考え方はない。(ワイヤで接続された 2 ノード間が相互に通信可能になることがテスト成否評価項目となる。)

4.4 テストシナリオの設定ファイル作成

設定したテスト対象ネットワーク、テスト用ノード配置をもとに、どのノードからどのノードに対して通信を行うかを定義する。

- テストパターン定義(scenario_pattern.json, 3.4.4 項)
  - 4.3 節「テスト成否評価検討」で示したように、机上テストの場合はワイヤで接続された 2 点間のみが通信可能になる。共有モードの場合 L2 の処理(ARP ブロードキャストの処理)が必要になるため、片方向ではなく双方向での通信を行う必要がある。(フロールール設計については 5 章で解説する。)

4.5 テスト定義ファイルの作成

テスト環境やテストシナリオの定義ファイルをもとに、テスト自動実行のための定義ファイルを作成する。

- テスト自動実行定義(testdefs.json, 3.4.5 項)

4.6 OpenFlow コントローラの起動

(別ホストあるいは別ターミナルで)OpenFlow コントローラを起動する。(表 4-1)

今回の PoC では、OFC は Mininet と同一のサーバ上で起動している。そのため、OFC の REST API URL は localhost:8080 となっている。

表 4-1 OpenFlow Controller の起動

hagiwara@prjexp01:~/PycharmProjects/l1patch-dev$ ryu-manager --verbose patch_ofc.py
loading app patch_ofc.py loading app ryu.controller.ofp_handler (省略) (24249) wsgi starting up on http://0.0.0.0:8080/

4.7 L1patch 設定の確認

L1patch 物理情報定義(nodeinfo.json)作成にあたって、OFS のポート番号が必要になる。Mininet は通常ノード接続順にポートが指定されるが、あらかじめ内容を確認しておく必要がある。そのため、まず自動実行スクリプトを CLI モードで実行して Mininet 側の情報確認を行う(表 4-2)。[3]

表 4-2 物理構成情報確認

hagiwara@prjexp01:~/PycharmProjects/l1patch-dev$ sudo python run_scenario_test.py -f testdefs_topo2.json --manual
● 手動テスト実行 (--manual) ● Layer オプション指定なし: L1patch の設定は行われない
フローデータの生成。データ生成はおこなうが OFS への投入は行われない。 INFO - exec command: python run_l1patch.py -p nodeinfo_topo2.json -l wireinfo_topo2.json -m exclusive > flows_exclusive_topo2.json シナリオファイルの生成。データ生成はおこなうが CLI モードでは使わない。 INFO - exec command: python scenario_generator.py -f scenario_pattern_topo2_simple.json > scenario_topo2.json -f オプションで指定されたテスト定義ファイル内のパラメータとして、"test-scenario-defs" - "class" 指定がある場合は、指定された class で再読込する[4]。 INFO - reload runner class: scenario_pinger_topo2.ScenarioPingerTopo2 INFO - exec command: python run_l1patch.py -p nodeinfo_topo2.json -l wireinfo_topo2.json -m exclusive > flows_exclusive_topo2.json

[3] 自動テスト実行スクリプト(run_scenario_test.py)のオプションについては 6.5.3 項参照。

[4] 机上テスト構成で Mininet の OVS Bridge 構成を再定義する派生クラスのインスタンスを生成しなおす処理。設定ファイルを一度読んでから、指定されたクラスで再度作り直す処理をしている。実機(外部 NW)のテスト行う際には使われない。

```
INFO - exec command: python run_l1patch.py -p nodeinfo_topo2.json -l wireinfo_topo2.json -m shared
> flows_shared_topo2.json
INFO - exec command: python scenario_generator.py -f scenario_pattern_topo2_simple.json >
scenario_topo2.json
テストを行う test-runner class を確認
INFO - run scenario test with runner-class: ScenarioPingerTopo2
テスト用ノードを生成
INFO - Start run_test()
INFO - build test host: test host h1[h1-eth0] = MAC:0a:00:00:00:00:01, IP:192.168.2.11/24,
Gateway:192.168.2.254
INFO - build test host: test host h2[h2-eth0] = MAC:0a:00:00:00:00:02, IP:192.168.2.12/24,
Gateway:192.168.2.254
INFO - build test host: test host h3[h3-eth0] = MAC:0a:00:00:00:00:03, IP:192.168.2.13/24,
Gateway:192.168.2.254
INFO - build test host: test host h4[h4-eth0] = MAC:0a:00:00:00:00:04, IP:192.168.2.14/24,
Gateway:192.168.2.254
INFO - build test host: test host h5[h5-eth0] = MAC:0a:00:00:00:00:05, IP:192.168.2.15/24,
Gateway:None
CLI モードに入る
mininet>
```

```
mininet topology を確認する(狙った順序で mininet ovs に接続されているかどうか)
mininet> net
h1 h1-eth0:s1-eth3
h2 h2-eth0:s1-eth4
h3 h3-eth0:s1-eth5
h4 h4-eth0:s1-eth6
h5 h5-eth0:s1-eth7
h6 h6-eth0.200:s3-eth2
h7 h7-eth0:s3-eth3
h8 h8-eth0:s2-eth4
s1 lo: s1-eth1:s2-eth1 s1-eth2:s2-eth2 s1-eth3:h1-eth0 s1-eth4:h2-eth0 s1-eth5:h3-eth0
s1-eth6:h4-eth0 s1-eth7:h5-eth0
s2 lo: s2-eth1:s1-eth1 s2-eth2:s1-eth2 s2-eth3:s3-eth1 s2-eth4:h8-eth0
s3 lo: s3-eth1:s2-eth3 s3-eth2:h6-eth0.200 s3-eth3:h7-eth0
c0
mininet>
```

```
OVS のポート番号を確認する
(机上テストでは外部 NW 接続用インタフェースを指定していないが、外部ネットワーク(実機のテスト対
```

象 NW)でテストを実行する場合は、外部接続用のインタフェースが含まれることを確認すること) mininet> dpctl show

4.8 手動テスト実行

物理構成の確認ができたら、生成されるフロールールを OFC に投入し、テスト対象ネットワークを構成して手作業で動作を確認する。

机上テストではテスト対象ネットワークをすべて OVS Bridge でひとつの OS 上に模擬するため不要だが、外部の実機環境をテストする場合は、フロールールを設定して DUT 間接続が成立してはじめて、テスト対象ネットワークの物理構成や簡単な通信テストが行えるようになる。したがって、この手順で専有モードワイヤのフロールールを設定した状態ではじめて、DUT 間は意味のある物理接続が行われた状況になる。「試験結果レポート/5.4.2 L1patch の設定による NW 機器間の接続」で示したが、L1patch では DUT 間が直接結線されているわけではないので、目視による物理構成の確認ができないことに注意が必要である。(物理構成確認のために CDP 等のプロトコルを使う必要がある。)

表 4-3 フロールールの設定と手動動作確認

hagiwara@prjexp01:~/PycharmProjects/l1patch-dev$ sudo python run_scenario_test.py -f testdefs_topo2.json --manual --all-layers
● 手動テスト実行(--manual) ● すべてのワイヤを設定(--all-layers)
フロールール生成・テストシナリオ生成を行う。 (省略) INFO - reload runner class: scenario_pinger_topo2.ScenarioPingerTopo2 INFO - exec command: python run_l1patch.py -p nodeinfo_topo2.json -l wireinfo_topo2.json -m exclusive > flows_exclusive_topo2.json INFO - exec command: python run_l1patch.py -p nodeinfo_topo2.json -l wireinfo_topo2.json -m shared > flows_shared_topo2.json INFO - exec command: python scenario_generator.py -f scenario_pattern_topo2_simple.json > scenario_topo2.json INFO - run scenario test with runner-class: ScenarioPingerTopo2 テスト用ノードを生成 INFO - Start run_test() INFO - build test host: test host h1[h1-eth0] = MAC:0a:00:00:00:00:01, IP:192.168.2.11/24, Gateway:192.168.2.254 INFO - build test host: test host h2[h2-eth0] = MAC:0a:00:00:00:00:02, IP:192.168.2.12/24, Gateway:192.168.2.254 INFO - build test host: test host h3[h3-eth0] = MAC:0a:00:00:00:00:03, IP:192.168.2.13/24, Gateway:192.168.2.254 INFO - build test host: test host h4[h4-eth0] = MAC:0a:00:00:00:00:04, IP:192.168.2.14/24,

```
Gateway:192.168.2.254
INFO - build test host: test host h5[h5-eth0] = MAC:0a:00:00:00:00:05, IP:192.168.2.15/24, Gateway:None
生成したフロールールを OFC REST API に設定する[5]
INFO - put exclusive-wire-flow-rules
INFO - exec command: cat flows_exclusive_topo2.json | python patch_ofc_rest_knocker.py -m put
INFO - Set API URL: http://localhost:8080/patch/flow
INFO - Send PUT: node:s2, rule:{"priority": 65535, "outport": 4, "inport": 2, "dpid": 2}
INFO - Response: {'date': 'Tue, 27 Oct 2015 11:59:29 GMT', 'status': '200', 'content-length': '0', 'content-type': 'text/html; charset=UTF-8'}
INFO - Content:
(省略)
設定し終わったら CLI モードに入る
mininet>
```

```
L1patch 定義に問題がなければ、テスト用ノードの生成と DUT への配置が行われ、手作業でテストが可能になる。
mininet> h6 ping -c3 h1
PING 192.168.2.11 (192.168.2.11) 56(84) bytes of data.
64 bytes from 192.168.2.11: icmp_seq=1 ttl=64 time=0.498 ms
64 bytes from 192.168.2.11: icmp_seq=2 ttl=64 time=0.059 ms
64 bytes from 192.168.2.11: icmp_seq=3 ttl=64 time=0.057 ms

--- 192.168.2.11 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1998ms
rtt min/avg/max/mdev = 0.057/0.204/0.498/0.208 ms
mininet>
```

4.9 テスト自動実行

手動テスト実行で基本的な動作を確認し、問題なければ自動実行を行う(表 4-4)。

表 4-4 テスト自動実行

```
hagiwara@prjexp01:~/PycharmProjects/l1patch-dev$ sudo python run_scenario_test.py -f testdefs_topo2.json --all-layers
```

- ユースケースオプション指定なしでテスト自動実行
- すべてのワイヤを設定(--all-layers)

[5] 今回の PoC 環境では、OFC を Mininet(テスト自動実行スクリプト)と同一ホストで実行している。そのため、OFC REST API へ JSON データを送信ためのスクリプト内に、REST URL が直接コーディングされている。

4.10 ツールを使ったテストデバッグ・応用

テスト自動実行にあたって、テスト結果をもとにした動作の不具合(テスト対象の問題、L1patch 側の問題)を調査するために以下のような機能を実装した。

- 自動実行の中断と手動チェック作業エントリ(4.10.1 項)
- 部分的なトポロジ変更とテストの繰り返し実行(4.10.2 項)

4.10.1 テスト自動実行のデバッグ

当初作成したテスト自動実行スクリプトでは、一度実行を開始すると強制終了しかできない、かつ、実行が終わったあとはすべてのフロールールを削除して終了する、という動作で実装した。そのため、テスト実行中の動作不具合や明らかにテスト対象ネットワーク側の設定としておかしなところがあったとしても、ネットワークの状態を維持したまま中断できない(強制終了することでフロールールがクリアされる)という問題があった。

そこで、PoC では以下の 2 パターンの手動操作移行方法を用意している。

- 自動テストを中断して CLI モードに移行(表 4-5)
  - テストシナリオの自動実行中に異常を発見した場合に、DUT 間接続や NW の状態を保存したまま手動チェック可能な状態に移行する。
  - テスト自動実行モード(自動実行中)に "^C"(Control-C, SIGINT) を送ることで、自動実行を中断して CLI モードに移行する。
- 自動テスト終了後にプロセスを終了せず、CLI モードに移行して待機する。(表 4-6)
  - テストの自動実行、テスト結果レポートをチェックしたうえで不明点や再確認したい点を個別にチェックできるよう、テスト対象環境の状態を維持したまま、手動作業可能な状態で待機する。

表 4-5 テスト自動実行中の中断

自動実行中のシグナル送信によるテスト中断と CLI モード移行
hagiwara@prjexp01:~/PycharmProjects/l1patch-dev$ sudo python run_scenario_test.py -f testdefs_topo2.json --all-layers
(省略) INFO - run sub scenario: main tasks INFO - [20.0%/current:1/total:5] run task: [shared-wire-to-h6] ping h1 to @h6@ INFO - run @`h1`: `ping -c 5 -i 0.2 192.168.2.106` INFO - result: SUCCESS INFO - [40.0%/current:2/total:5] run task: [shared-wire-to-h6] ping h2 to @h6@ INFO - run @`h2`: `ping -c 5 -i 0.2 192.168.2.106` INFO - result: SUCCESS INFO - [60.0%/current:3/total:5] run task: [shared-wire-to-h7] ping h3 to @h7@ INFO - run @`h3`: `ping -c 5 -i 0.2 192.168.2.107` INFO - result: SUCCESS INFO - [80.0%/current:4/total:5] run task: [shared-wire-to-h7] ping h4 to @h7@ INFO - run @`h4`: `ping -c 5 -i 0.2 192.168.2.107`

```
^CINFO - change mode to CLI by signal:2

Control-C(SIGINT)送信でテスト実行を中断し、CLI モードに移行する。
mininet>
L1patch 状態(DUT 間接続やテストノード配置)は維持される。
mininet> net
h1 h1-eth0:s1-eth3
h2 h2-eth0:s1-eth4
h3 h3-eth0:s1-eth5
h4 h4-eth0:s1-eth6
h5 h5-eth0:s1-eth7
h6 h6-eth0.200:s3-eth2
h7 h7-eth0:s3-eth3
h8 h8-eth0:s2-eth4
s1  lo:  s1-eth1:s2-eth1  s1-eth2:s2-eth2  s1-eth3:h1-eth0  s1-eth4:h2-eth0  s1-eth5:h3-eth0
s1-eth6:h4-eth0 s1-eth7:h5-eth0
s2 lo:  s2-eth1:s1-eth1 s2-eth2:s1-eth2 s2-eth3:s3-eth1 s2-eth4:h8-eth0
s3 lo:  s3-eth1:s2-eth3 s3-eth2:h6-eth0.200 s3-eth3:h7-eth0
c0
mininet>
```

表 **4-6** テスト実行後の手動作業以降

自動テスト終了後、手動作業が可能な状態を維持する
hagiwara@prjexp01:~/PycharmProjects/l1patch-dev$ sudo python run_scenario_test.py -f testdefs_topo2.json -test-cli --all-layers
● 自動実行後、手動実行(CLI モード)に移行する (--test-cli) ● すべてのワイヤを設定(--all-layers)
(省略) INFO - run @`h6`: `ip neigh show` INFO - run task: check host h7 arp table INFO - run @`h7`: `ip neigh show` INFO - run task: check host h8 arp table INFO - run @`h8`: `ip neigh show` mininet> テスト終了後に CLI モードに入るので、テスト結果をもとに個別の再チェックが可能。 mininet> net h1 h1-eth0:s1-eth3 h2 h2-eth0:s1-eth4

```
h3 h3-eth0:s1-eth5
h4 h4-eth0:s1-eth6
h5 h5-eth0:s1-eth7
h6 h6-eth0.200:s3-eth2
h7 h7-eth0:s3-eth3
h8 h8-eth0:s2-eth4
s1  lo:  s1-eth1:s2-eth1  s1-eth2:s2-eth2  s1-eth3:h1-eth0  s1-eth4:h2-eth0  s1-eth5:h3-eth0
s1-eth6:h4-eth0 s1-eth7:h5-eth0
s2 lo:  s2-eth1:s1-eth1 s2-eth2:s1-eth2 s2-eth3:s3-eth1 s2-eth4:h8-eth0
s3 lo:  s3-eth1:s2-eth3 s3-eth2:h6-eth0.200 s3-eth3:h7-eth0
c0
再チェック実行例
mininet> pingall
*** Ping: testing ping reachability
h1 -> X X X X h6 X X
h2 -> X X X X h6 X X
h3 -> X X X X X h7 X
h4 -> X X X X X h7 X
h5 -> X X X X X X h8
h6 -> h1 h2 X X X X X
h7 -> X X h3 h4 X X X
h8 -> X X X X h5 X X
*** Results: 82% dropped (10/56 received)
mininet>
mininet> quit
CLI モード終了後にフロールールを削除して終了する。
（省略）
hagiwara@prjexp01:~/PycharmProjects/l1patch-dev$
```

4.10.2 部分的なトポロジ操作とテストの再実行

「試験結果レポート/6.2.1 L1patch のテスト実行フェーズの考え方」に記載したように、テスト自動実行を繰り返す場合には、ベースとなるネットワーク構成とテストごとに設定すべきネットワーク構成を切り離して考える必要がある。

今回の PoC 実装上は、L1patch 上の連続テスト実行にあたって以下のように設定した。

- すべてのテストを通して保持されるネットワーク構成
  - 専有モードワイヤを使って構成されるテスト対象ネットワーク機器間の接続(物理トポロジ)
  - テスト対象ネットワーク機器間の接続は最初に設定し、OFC 実行している限りする。(テスト実行ごとに設定の変更を行わない)

- テストごとに設定されるネットワーク構成
  - 共有モードワイヤを使って接続されるテストノードとテスト対象機器間の接続
  - テスト用ノードはテスト単位で配置・操作するものなので、テストの実行ごとにセットアップと削除(クリーニング)を行う。

「試験結果レポート/6.2.1 L1patch のテスト実行フェーズの考え方」に記載したとおり、本来はこのようなワイヤ種別による分類は正しくないが、PoC 上は簡易的にこの分類とし、繰り返しテストを試行している。繰り返しテストを実行する場合には以下のように操作を行う。

- 専有モードワイヤ・共有モードワイヤの両方を設定してテスト自動実行(表 4-7)
  - テスト終了あとは共有モードワイヤの設定のみ削除される。
- 1 回目のテスト実行後に必要な操作を行う。
  - 今回の PoC の場合は OFS のポート設定操作による DUT 間リンクダウンの発生。
- 共有モードワイヤのみを設定してテスト自動実行(表 4-8)
- 以降繰り返し

テストごとの操作範囲を限定することで、(異なる)テストを、同じ環境(テスト対象 NW)の上で複数回繰り返し実行可能になる。

表 4-7 テスト自動実行スクリプトの繰り返し実行(初回)

hagiwara@prjexp01:~/PycharmProjects/l1patch-dev$ sudo python run_scenario_test.py -f testdefs_topo2.json --manual --all-layers
● レイヤオプションによる動作の違いを見るために手動テスト実行(--manual)を行う。 ● すべてのワイヤを設定(--all-layers)
(省略) mininet> pingall *** Ping: testing ping reachability h1 -> X X X X h6 X X h2 -> X X X X h6 X X h3 -> X X X X X h7 X h4 -> X X X X X h7 X h5 -> X X X X X X h8 h6 -> h1 h2 X X X X X h7 -> X X h3 h4 X X X h8 -> X X X X h5 X X *** Results: 82% dropped (10/56 received) mininet>
オプション --all-layers を付けると、専有モードワイヤ・共有モードワイヤ両方を設定して環境セットアップを行う。手動テスト(--manual)を指定して実行すると、共有モードワイヤ・専有モードワイヤ両方が設定されていることが pingall の結果からわかる。 ● 手動テストではなく自動テストを実行した場合、テスト実行が終わると共有モードワイヤのルール

のみ削除する。 ● 専有モードワイヤのフロールールはそのまま残す。よって、OFC のプロセスを落とさなければそのまま接続は維持できる。

表 4-8 テスト自動実行スクリプトの繰り返し実行(2 回目以降)

hagiwara@prjexp01:~/PycharmProjects/l1patch-dev$ sudo python run_scenario_test.py -f testdefs_topo2.json --manual --layer2
● レイヤオプションによる動作の違いを見るために手動テスト実行(--manual)を行う。 ● 共有モードワイヤのみ設定(--layer2)
(省略) mininet> pingall *** Ping: testing ping reachability h1 -> X X X X h6 X X h2 -> X X X X h6 X X h3 -> X X X X X h7 X h4 -> X X X X X h7 X h5 -> X X X X X X X h6 -> h1 h2 X X X X X h7 -> X X h3 h4 X X X h8 -> X X X X X X X *** Results: 85% dropped (8/56 received) mininet>
--layer2 オプションで実行した場合、専有モードワイヤ(--layer1)のフロールール設定は行わない。手動テスト(--manual)を指定して実行すると専有モードワイヤの設定が行われないため、h5-h8 間の接続が成立しない。 ● テスト用ノード接続(共有モードワイヤ)の操作のみ行う。

レイヤオプションによって環境共通(専有モードワイヤ)・テスト固有(共有モードワイヤ)の操作が可能なので、テストを繰り返し実行する場合には表 4-9 の操作でもよい。

表 4-9 自動テストの繰り返し実行(オプション指定の別例)

0 回目	sudo python run_scenario_test.py -f testdefs_topo2.json --layer1
	専有モードワイヤ(--layer1)のみ指定: 専有モードワイヤの設定のみ実行してテスト自動実行は行わない。(テスト用ノードの配置を行わないので、テスト自動実行は行わない。)
1 回目以降	sudo python run_scenario_test.py -f testdefs_topo2.json --layer2
	共有モードワイヤを設定してテスト自動実行

5 L1patch 動作設計

5.1 L1patch の機能要件

機能要件については「試験結果レポート/4.1 L1patch 仕様策定」を参照。以降、定義した L1patch 機能をもとに、下記 2 種類の「ワイヤ」をどのように OpenFlow を使って実現したかを解説する。

- 専有モードワイヤ
- 共有モードワイヤ

L1patch で実現するノード(ポート)間の直接接続を「ワイヤ」とする。L1patch は「ワイヤ」をエミュレートするネットワーク制御を行うことになる。ワイヤのエミュレートにあたっては、基本的に end-to-end での in-port/out-port の対応を取る(入ったものを指定された箇所に出す)動作で実現できる。つまり、操作対象となるパケット(フレーム)がどの「ワイヤ」にあたるのか識別できれば、あとは入力に対して一意に出力箇所が決定できる。

OpenFlow 制御を考えるにあたり、今回は packet-in による動的なルール設定(リアクティブ動作)は使用しないことを方針とした。上記の L1patch 動作は、「ワイヤ」の考え方として接続する 2 点間が決まれば固定的なものであること、また、フロールールが一意に決まることで L1patch のテストが容易になるためである。

5.2 専有モードワイヤの OpenFlow ルール設計

専有モードでは物理ポート単位に入出力(in-port/out-port)を決定する。そのため、接続したい端点の中間経路(OFS)すべてで物理ポート単位の 1 対 1 マッピングを行う(図 5-1)。ポート間入出力(in/out port)の対応は物理ポートマッチでポート単位に行われるため、専有モードが利用する経緯路上のポートはそのほかの用途では使用できない(物理ポート単位で専有される)。

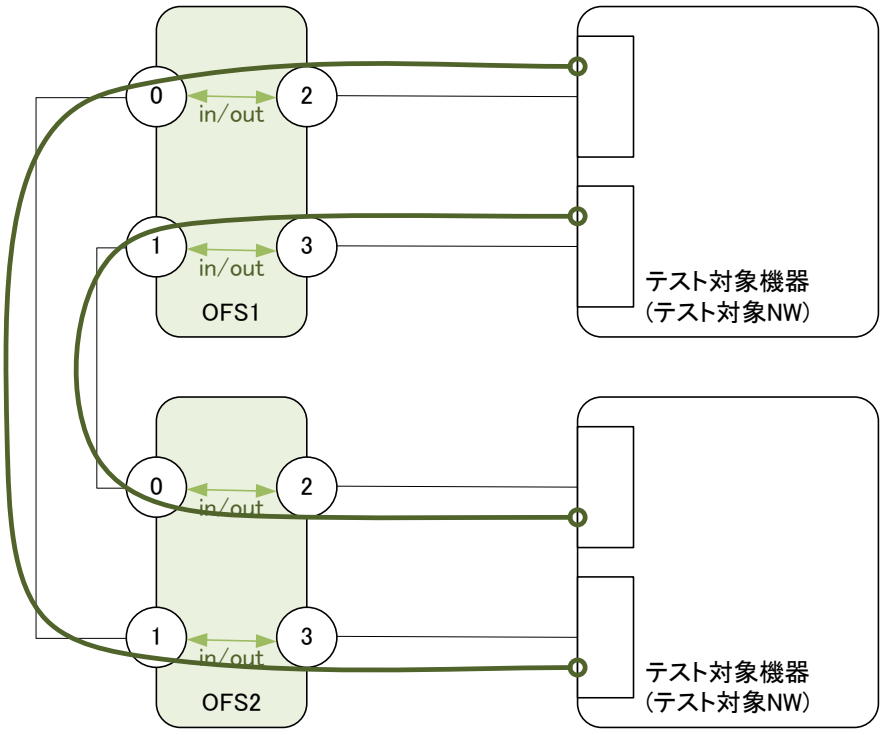

図 5-1 専有モードワイヤのフロー設計

5.3 共有モードワイヤの OpenFlow ルール設計

5.3.1 コントロール対象の検討

共有モードワイヤではひとつの物理ポートに複数のワイヤをマップ可能にするため、物理ポート単位のマッピングを行わない。PoC では MAC アドレスベースの L2 制御を行っている。

はじめに、動作仕様をもとに、制御対象となるパケット(フレーム)種別と、その制御方法を検討する。

- L2 での Wire の識別
  - テスト用ノード(L1patch を使って DUT につなぎこむ機材)は MAC/IP アドレスは既知(given)とする。
    - IP アドレスは重複がありうるので一意の ID にはならない。
    - MAC アドレスは重複なしという前提にする。
      - Mininet によるノード生成を想定する上では問題のない前提設定と判断した。また、テスト用ノード(Mininet host 以外のノードをテストに使う可能性を含めて)は通常、事前に MAC アドレスは調査可能(あるいは任意に設定可能)であるため問題がないと考えた。
- ワイヤ動作制御
  - MAC アドレスでワイヤ識別可能な L2 フレームの処理:
    - 送信元 MAC アドレス、または宛先 MAC アドレスにテスト用ノードの MAC アドレスが指定されているもの(ユニキャストまたはテスト用ノードからのブロードキャスト送信))は、MAC アドレスをもとにワイヤを識別できる。(1Host-1Wire という前提にする)
  - テスト対象 NW(テスト対象機器:DUT)側から送られる BUM の処理:
    - DUT 側からの Unknown Unicast: DUT がフラッディングした Unknown Unicast。DUT が送信先を学習できていなかっただけで、送信先 MAC が指定されているのであればユニキャストと同等。
    - DUT 側からの Broadcast: テスト用ノードが持つ IP アドレスの MAC アドレスを解決するために送信される ARP Request は対処が必要。送信元/宛先 MAC アドレスにどのテスト用ノードなのかを識別するための情報はない。Broadcast は L2 セグメント(Broadcast Domain)単位でコピーして複数のノードに送信しなければならない。そのため何らかの形で「L2 セグメントの情報」が必要になる。
    - マルチキャストは今回の PoC では考えない。

以上の検討から、制御としては、送信元/送信先 MAC アドレスのいずれかが既知で、一意に接続されるノード(ワイヤ)が識別できるユニキャスト/テスト用ノードからのブロードキャストと、送信元/送信先 MAC アドレスに特定のノード(ワイヤ)を識別できる情報がないブロードキャストについての考慮が必要となることがわかる。

そこで、共有モードワイヤの制御において考慮すべき点を具体的な実装イメージをもとに検討する(表 5-1, 図 5-2)。DUT 側から送信されるブロードキャストの処理では、以下の点を検討する必要があることがわかる。

- VLAN Tag の有無(DUT 側ポートの VLAN 設定: trunk/access)
- Broadcast のコピー箇所(どこで複数 packet-out 指定が必要か: ワイヤの重ね合わせの検討)
- “セグメント”識別子の設定
  - DUT から送信された broadcast(arp)は、そのポートに接続している複数のワイヤにコピーされる。コピー対象となるワイヤをまとめたもの(L1patch 制御における“セグメント”)の識別が必要。

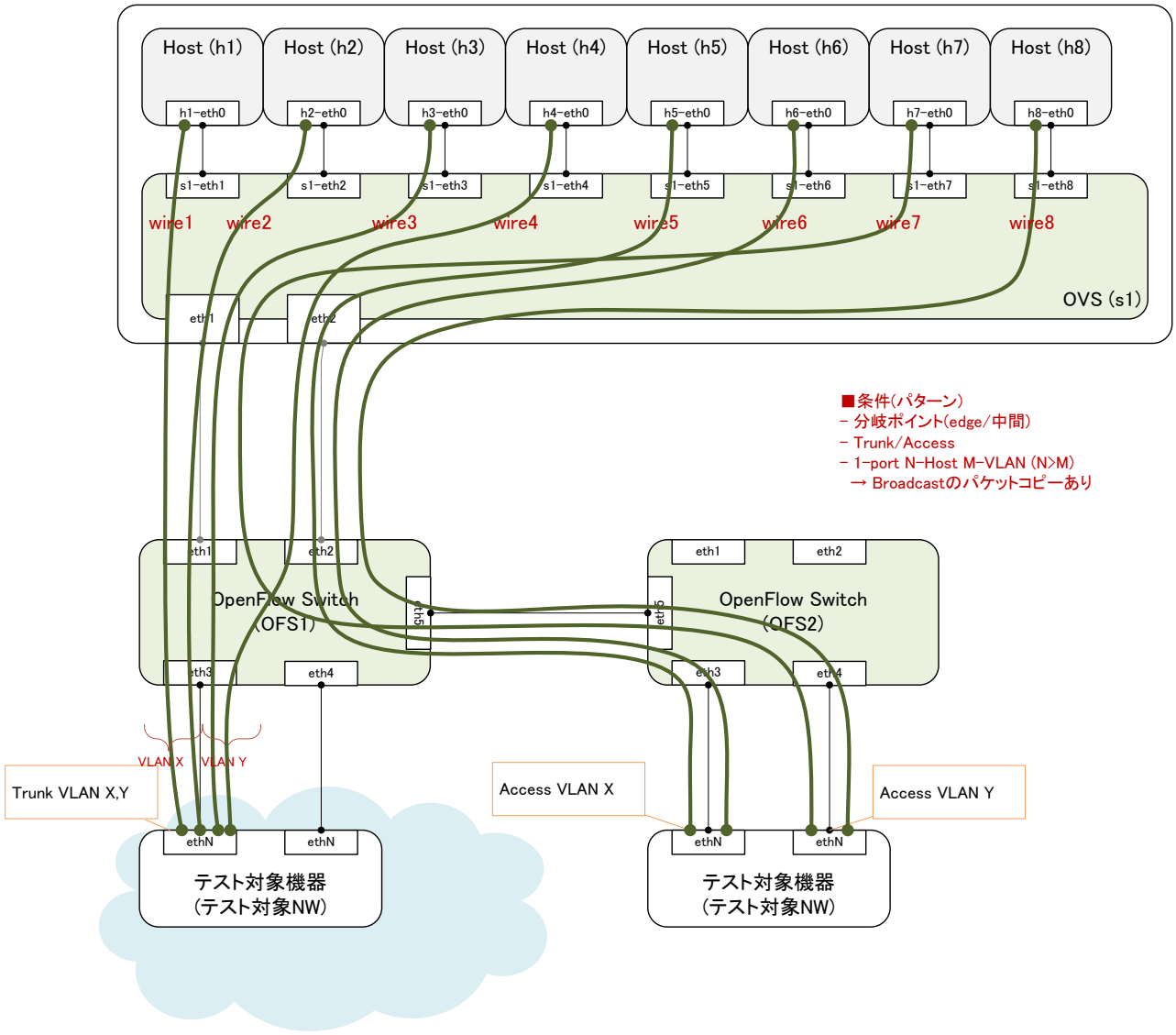

図 5-2 共有モードワイヤの構成パターン検討

表 5-1 DUT からの arp request (broadcast) に対する L1patch のコントロール(図 5-2)

Wire	DUT から L1patch へ送信される Broadcast (ARP request)	アクション
wire1, wire2	vlan_vid=X eth_dst=broadcast	● OFS1 では out_port=eth1, OVS では broadcast をコピーして out_port=s1-eth1、s1-eth6 する。 ● OVS で Host に渡す時に pop VLAN する。
wire3, wire4	vlan_vid=Y eth_dst=broadcast	● OFS1 で eth1/eth2 にコピーして packet-out(分岐ポイントで out_port を複数設定)する、つまりワイヤの重ね合わせの考慮が必要。 ● OVS で Host に渡す時に pop VLAN する。
wire5, wire6	vlan_vid なし	DUT 側が access port の場合、同一 VLAN のノードが複数いるものと想

	eth_dst=broadcast	定。(DUT 対向はシンプルな L2SW があり h5,h6 が接続する) ● DUT 側 access port の場合、そのポートから受けたという以外に L2 Seg を識別することができないので、この場合 OFS2/in_port=eth3 な broadcast には”セグメント ID”を付加して次のスイッチに渡す。 ● “セグメント ID”で識別できれば、中間 OFS では入ってきたもの派をどこに出すかを設定するだけで良い。
wire7, wire8	vlan_vid なし eth_dst=broadcast	Wire5/6 同様。ただし OFS1 で broadcast をコピー/分岐させる必要がある。(Wire の重ね合わせ考慮)

5.3.2 ブロードキャストトラフィックの処理方針検討

5.3.1 項にあげたとおり、共有モードワイヤの実装では、DUT 側からの broadcast (arp request)についてのみ特定のホスト(テスト用ノード)を識別する状態がないため、別途フレーム制御のルールを設定する必要がある。これは、L2 スイッチがブロードキャストを特定のセグメントに対してフラッディングする動作に相当する。

L1patch は DUT 側からのブロードキャスト処理とそのほかの処理(ユニキャスト)は分けて制御するものとし、DUT 側からのブロードキャストの制御方法については図 5-3 のように設計した。(なお図 5-3 は机上テスト:図 3-1 で示した共有モードワイヤ設定をもとにしている。)

- DUT 側から L1patch に入るブロードキャストについては、入ってきた DUT ポートと接続するワイヤで、かつ、同じ L2 セグメントに所属するものだけがブロードキャストドメインとなる。
  - 図 5-3 のワイヤ A とワイヤ B は、テスト対象ネットワーク内のセグメントとしては同一の L2 セグメント(VLAN X)にある。しかし、それぞれのワイヤが接続されている機器(DUT)の物理ポートが異なるため、L1patch としては同一の”セグメント”とはみなさない。OFS N のポート 1 で受信した VLAN X のブロードキャストがコピーされるのは、ポートに接続されているワイヤの端点である h1/h2 だけである。
  - ワイヤ A は、ひとつの DUT 物理ポートに接続されている共有モードワイヤ(図 3-1 の wire1,wire2)を束ねたものをイメージしている。L1patch で設定されるブロードキャストドメインを表す。(これを「ワイヤグループ」と設定している。)
- フレーム(パケット)に対するコントロールは L1patch の端点で行う。
  - 端点:L1patch に対する inbound/outbound の境界ポート。すなわち、テスト用ノードの接続ポート、DUT の接続ポートのこと。
  - 中間経路でのパケットコピーは L1patch のフロー制御の複雑さにつながる(OpenFlow packet-out によるパケットコピーは異なる OFS ポートに対して行うもので、共有モードのように物理ポートを共有するものに対して(何らかの加工をして)packet-out させることは難しい[6]。そのため、ブロードキャストフレームのコピーはテスト用ノード側端点(Mininet OVS: s1)で行うものとする。
    - L1patch 中間での分岐(コピー)は考えない。そのため、ワイヤ C のように、異なるテスト用ノードスイッチ(s2)から同一の DUT ポートにワイヤを接続する状況は、今回は想定しない。
  - 1 台の OFS だけでテスト用ノードと DUT を中継することを想定していない。
    - OpenFlow スイッチへのルール設定上、同時にパケット(フレーム)ヘッダフィールドを複数操作させ

[6] PoC で利用可能な OpenFlow 機能の制約などもあるため、極力複雑なパケット加工処理などは利用しないようにしたかったという事情もある。

ることが難しい(実装上複雑になる)ためである(5.3.3 項参照)。フロー設定上の役割を分担するため、テスト用ノード収容スイッチ・DUT 収容スイッチの 2 段あることを最小の構成とする。PoC でテスト用ノード収容スイッチは必ず Mininet OVS(s1) となるため、L1patch システムの物理構成としては、Mininet サーバと物理 OFS(1 台)が最小構成となる。

- ワイヤグループの L1patch 経路
  - DUT 側からのブロードキャストとそのほかユニキャストで制御(フォワーディングルール)を変更するため、共有モードワイヤについては、ユニキャストの経路(経由する OFS とポート)とブロードキャストの経路の 2 種類を設定することになる。
    - ブロードキャストの経路は、L1patch としてひとつのブロードキャストドメインとなる複数のワイヤに対してひとつ設定する。

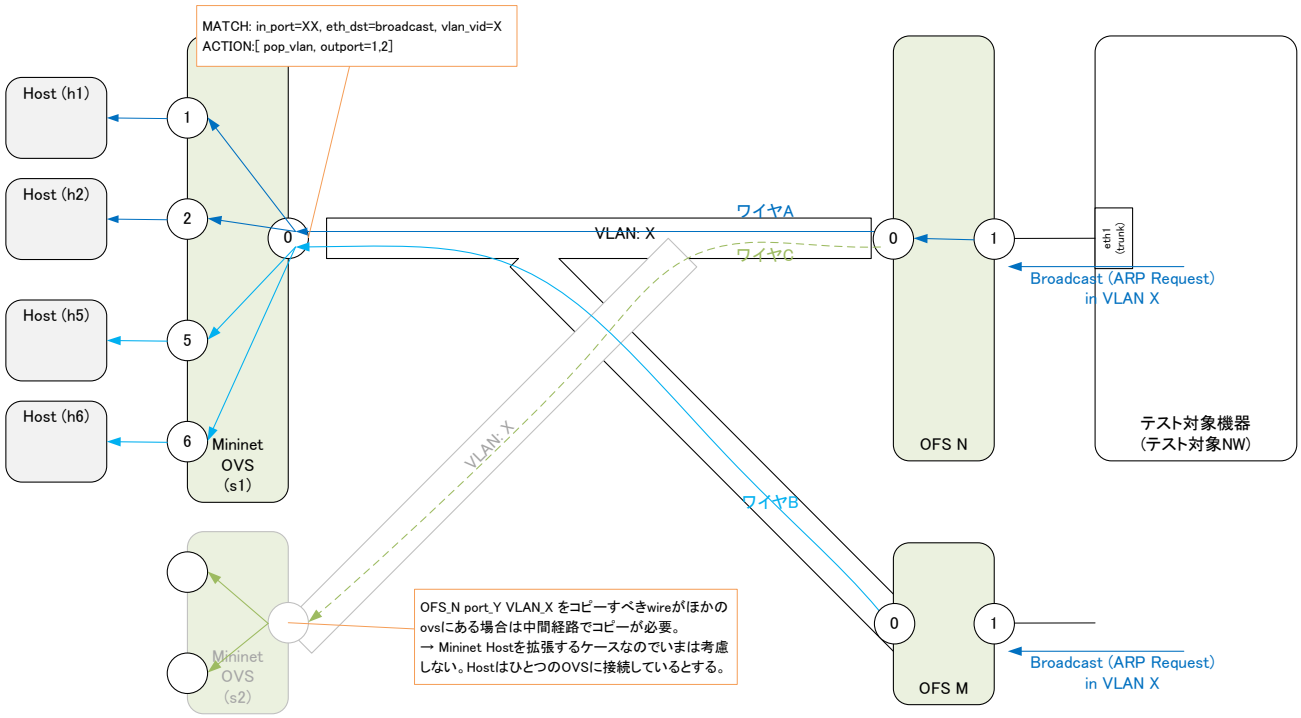

図 5-3 DUT 側からのブロードキャストフレームの処理

5.3.3 フロールール設計

5.3 節でこれまで検討してきた制御ルールをもとに、L1patch のモデルを表 5-2・図 5-4 のように設定した。

表 5-2 L1patch オブジェクト一覧

オブジェクト種類	オブジェクト（クラス名）	役割
Host	TestHost	テスト用ノード。PoC では Mininet 上で構成するホスト。
	Dispatcher	L1patch として制御される OpenFlow Switch。
	DUTHost	テスト対象のネットワーク機器
Port	TestHostPort	テスト用ノード(TestHost)のポート。

	HostEdgePort	OFS(dispatcher)のポート。	TestHostPort が接続するポート。
	InterSwitchPort		Dispatcher 間のポート。
	DUTEdgePort		DUT Port が接続するポート。
	DUTPort	テスト対象機器(DUTHost)のポート。	
Wire	ExclusiveWire	専有モードワイヤ	
	SharedWire	共有モードワイヤ	
Wire Group	WireGroup	L1patch として同一ブロードキャストドメインに所属するとみなす共有モードワイヤの集合。	

ワイヤグループ(Wire Group)は 5.3.2 項で解説した DUT 側からのブロードキャストをコントロールするためのオブジェクトである。ワイヤグループが必要になる理由は、「フレームヘッダに転送すべき共有ワイヤを識別する情報を持たない」ことである。したがって、DUT からのブロードキャスト転送にあたっては、最終的にどのホスト(wire)にコピーされるべきなのかを識別するための情報を持たせる必要がある。これを”Wire Group ID”とした。ワイヤ中間経路(Dispatcher, InterSwitchPort)ではブロードキャストのフレームヘッダに指定された Wire Group ID を参照して、フレームの転送先を定義する。

- Wire Group の識別
  - DUTPort が VLAN Trunk の場合: 対向の DUT Edge Port で VLAN ID に応じた Wire Group ID の設定を行う。
  - DUT Port が VLAN Access の場合: 対向の DUT Edge Port で固定の Wire Group ID の設定を行う。
- フレームヘッダに設定する”Wire Group”の選択
  - OpenFlow の機能から表 5-3 のパターンを検討し、最終的には”3. VLAN ID (Swap)”を選択した。
  - PoC ツール実装にあたっては当初”1. MPLS Tag”で実装した。しかし、PoC 環境で実機 OFS を使用しての動作テストを行う際に、借用していた一部のスイッチで OpenFlow1.3 対応あるいはフィールド操作機能(feature)対応ができないなどの制約があり、最も OpenFlow スイッチとして基本的な機能で実装可能な 3.を選択した。

表 5-3 Wire Group ID の実装検討

No.	Wire Group ID 候補	OpenFlow		特徴
		1.0	1.3	
1.	MPLS Tag	×	○	● 単純なフィールドの追加だけで Wire group を識別可能になるため転送ルールとしてわかりやすくなる。 ● DUT 側ポートが Trunk の場合は最終的にテスト用ノードへ渡す際に VLAN ID の削除(pop)処理が必要になる。単一の OFS(Mininet OVS, s1)で複数タグのつけ外し処理を行うのは OpenFlow 実装的には複雑になる。また、機能的に対応している機器があるため、実機使用 PoC 上を考えた際には制限が発生する。
2.	VLAN ID (Q-in-Q)	×	○	
3.	VLAN ID	○	○	● Wire Group ID として VLAN Tag を使用する。

	(Swap)			● DUT port が trunk だった場合には、テスト対象 NW 側の VLAN Tag を削除して、Wire Group ID を表す VLAN Tag に付け替える。(テスト用ノードに送信する際に VLAN tag はすべて削除(pop)する) ● ひとつのフィールドを、テスト対象 NW 側と L1patch 側で異なる意味に使用するため、考え方はやや複雑になる。ただし、ほとんどの OpenFlow スイッチで VLAN push/pop アクションが使用可能であるため、実機使用 PoC 上はメリットがある。

最終的な共有モードワイヤ実現のためのフロールールは図 5-5・図 5-6 のようになる。

図 5-4 に示したように、共有モードワイヤを実現するためには、表 5-4 の 3 種類の制御が必要になる。3 種類それぞれについて、中間経路(ワイヤが経由する OFS ポート)を選択することができる。今回実装にあたっては、ワイヤ単位で設定するルール(表 5-4: 1.,2.)は同一の経路を往復するものとした。また、3.ブロードキャスト制御ルールでは、ワイヤグループの要素にあるひとつのワイヤを選択し、そのワイヤが使用する経路(2.)と同じ経路を使用するものとした(最後 Mininet OVS 上で、ワイヤグループで指定する複数のテスト用ノードに対してブロードキャストフレームをコピーする)。

表 5-4 共有モードワイヤを実現するためのラフィック制御ルール種別

No	方向	設定単位	制御対象	
			Unicast	Broadcast
1.	行き (テスト用ノード→DUT)	ワイヤ (テスト用ノード)	送信元 MAC でどのホスト(どのワイヤ)か識別できる	
2.	帰り (DUT→テスト用ノード)		送信先 MAC でどのホスト(ワイヤ)か識別できる	(対象外)
3.		ワイヤグループ	(対象外)	送信元・先 MAC でワイヤは識別できない。 DUT エッジでワイヤグループ ID を付与して制御

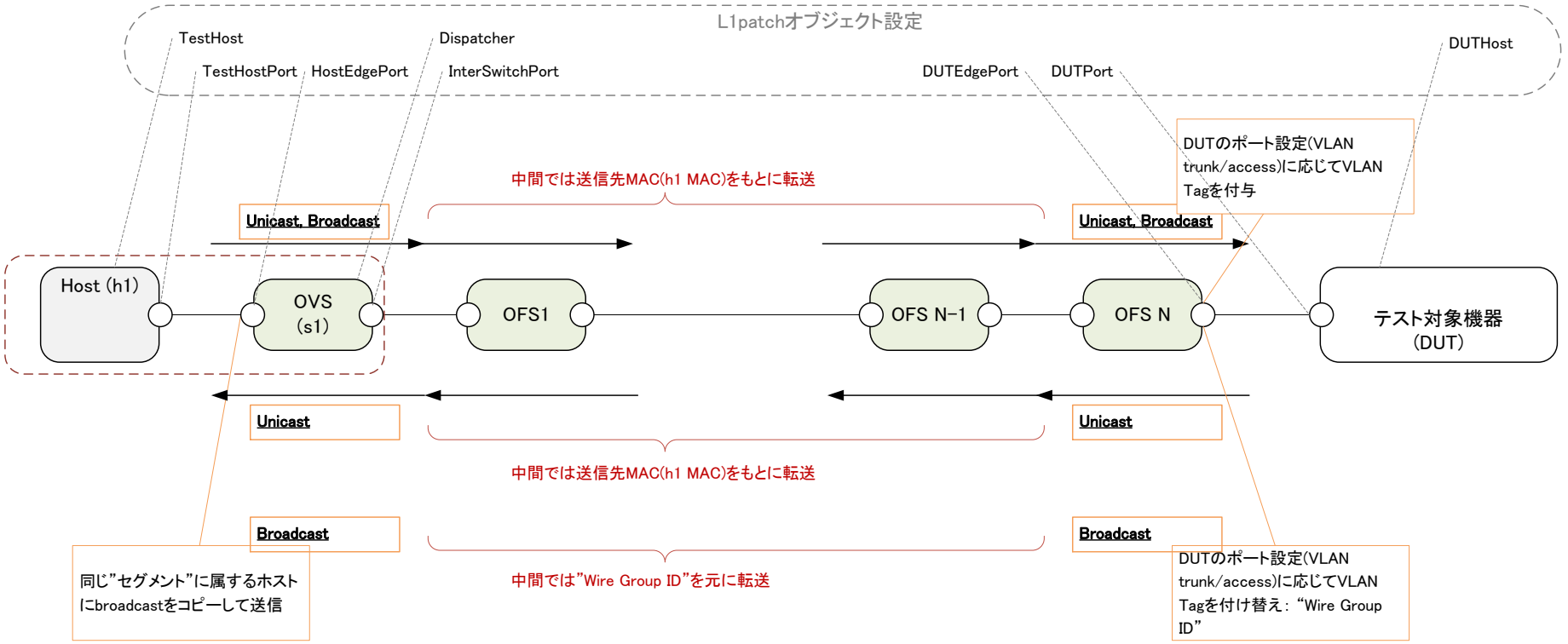

図 5-4 L1patch のトラフィック制御方針と役割設定

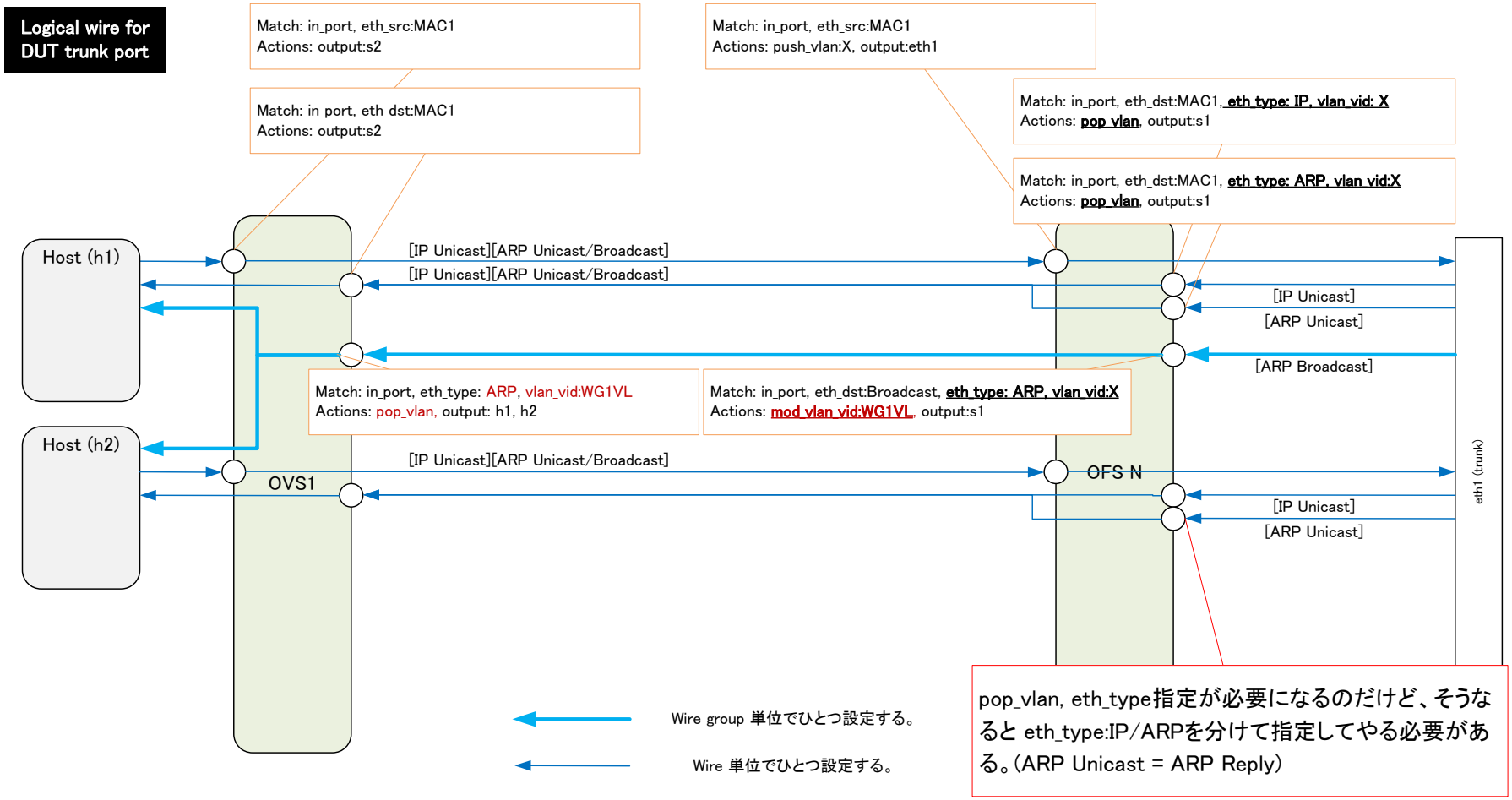

図 5-5 DUT トランクポートに対する共有モードワイヤのフロー設計

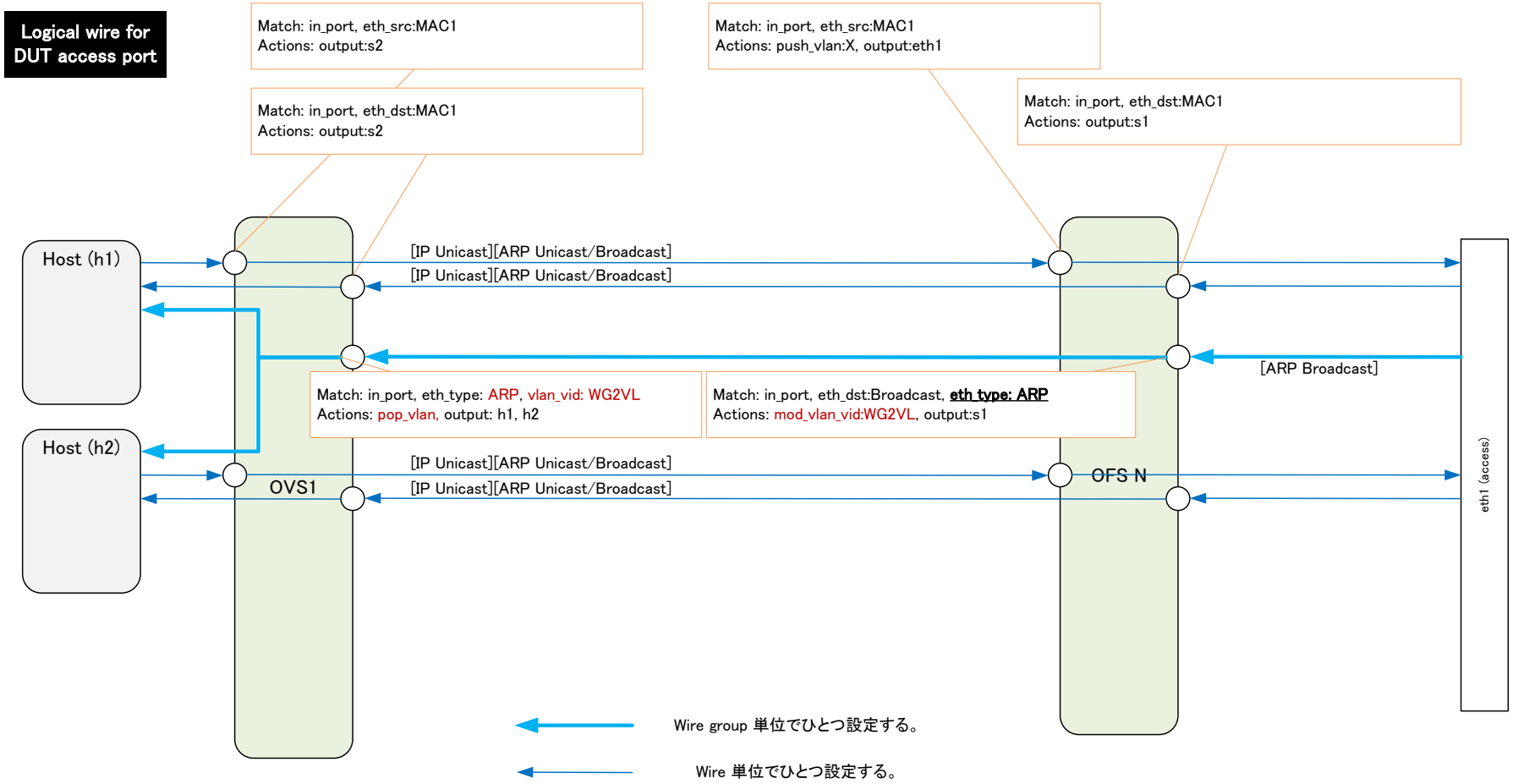

図 5-6 DUT アクセスポートに対する共有モードワイヤのフロー設計

5.4 デフォルトのフロールールとフロールールの優先度設定

5.3.2・5.3.3 項で解説したように、L1patch では共有モードワイヤを実現するためのトラフィック制御に、VLAN ID フィールドを 2 種類の用途で使用する。

- ①テスト対象ネットワーク内の VLAN 識別子として(通常の VLAN ID の用途として)
  - DUT port が VLAN trunk port の場合、VLAN Tag (VLAN ID)をもとに Wire group ID を決定する。
- ②Wire Group ID として
  - L1patch としてコントロールするブロードキャストドメインの識別

共有モードワイヤとは別に、専有モードワイヤは L1 の条件(in/out port)のみでトラフィック制御を行う。そのため、共有モードワイヤ・専有モードワイヤを混在して利用している場合、L1patch を構成する OFS では以下の通り①②それぞれの用途のフレームを取り扱う必要がある。また、VLAN ID 用途①②はそれぞれここに 1-4095 までの ID を管理することになるため、ID が重複する可能性も想定される。

- 専有モードワイヤとして処理される、①の用途の VLAN Tag を持ったフレーム
- 共有モードワイヤとして扱われる、②の用途の VLAN Tag を持ったフレーム

こうした VLAN ID の取り扱いの混同などが起こらないよう、L1patch のフロールール生成では表 5-5 のような優先度設定を行うことにした。

今回、実装上は OpenFlow1.0/1.3 両方での対応が可能なように実装を行っており、かつ、プロアクティブな制御を行う方針としているため(「試験結果レポート/4.1.2 OpenFlow 制御方式」参照)、設定したフロールールのいずれにもマッチしないパケットは drop するよう明示的に設定し、不要な packet-in の発生を防いでいる。

表 5-5 フロールールの優先度設定

優先度	Flow priority	設定対象
高	65535	占有モードワイヤのフロールール
中	32767	共有モードワイヤのフロールール(ユニキャスト)
低	16383	共有モードワイヤのフロールール(DUT からのブロードキャスト)
最低	0	“Default drop”ルール: match-any , action-drop スイッチと OFC との接続が確立した段階で自動的に設定する。

6 L1patch 実装

6.1 前提および制約の整理

5 章に示した L1patch 設計上の前提、実装上の技術的な制約などによって設定している制約などをまとめると以下のようになる。

- テスト用ノード
  - テスト用ノードは MAC アドレスが既知であるとする。(5.3.1 項)
    - テスト用ノードの MAC アドレスはシステム全体(テスト対象 NW 内、L1patch および L1patch に接続するテストノード)で一意とする。
  - ひとつのホスト用ノードに対してひとつのワイヤ(専有モードワイヤ・共有モードワイヤ)が設定される。(5.3.1 項)

- L1patch 内で制御可能な通信
  - マルチキャストは考慮しない。(5.3.1 項)
  - 前提として、マルチキャスト以外の IPv4 通信利用を想定している。L1patch の通信制御は L1/L2 のみで行っているため、IPv6 通信についても利用可能なはずだが PoC では IPv6 の利用可否はテストしていない。
- L1patch 構成
  - PoC では、複数の Mininet サーバを同時に使用することは想定していない。(5.3.2 項)
    - ひとつの DUT ポートに異なる Mininet OVS からワイヤを張ることを想定しない。(PoC ではテストしていないが、こうしたワイヤ利用上の制約に反しない範囲で複数の OFS にテスト用スイッチを置くことは可能だと考えられる。)
  - テスト用ノード収容スイッチと DUT 収容スイッチで最低 2 段のスイッチが必要。(5.3.2 項)
  - VLAN ID 操作可能な OpenFlow スイッチを使用する。OpenFlow1.0/1.3 いずれでも動作可能だが、バージョンが混在することは想定していない。(「試験結果レポート/8.1.2 OpenFlow バージョン」参照。)
- テストアプリケーション
  - 「別ドキュメント/5.2 PoC で実行した内容」参照

6.2 全体の構造

今回 PoC で実装したツールは大きく以下の 3 つに分類できる。

- L1patch フロールール生成
- OpenFlow Controller
- テストシナリオ操作(テストシナリオの生成とテスト自動実行)

各ツールと実際の機器へのデプロイ、今回使用しているソフトウェアとの関係を示すと、図 6-1 のようになる。図 6-1 の”physical view”は L1patch を構成するハードウェア、ソフトウェアの実体、”logical view”は L1patch が実現するパケット制御によって実現されるネットワーク接続を示している。

上記の 3 つのツール間のデータフローを図 6-2 に示す。ツールは基本的にそれぞれ単体で動作する。テスト自動実行ツールは各ツールをサブプロセスで起動することで全体のテスト環境操作とテスト実行を行う。

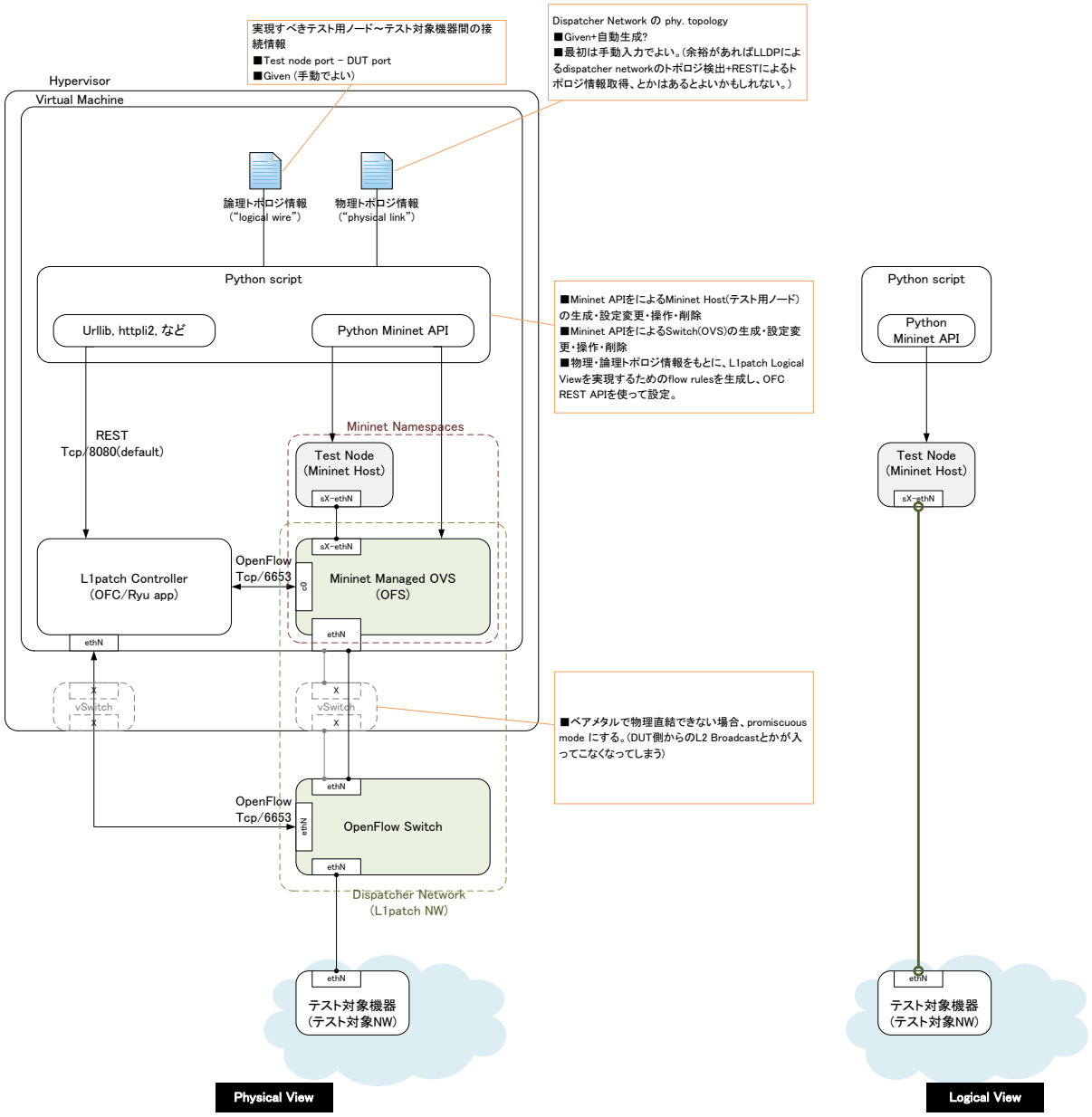

図 6-1 L1patch システムアーキテクチャ

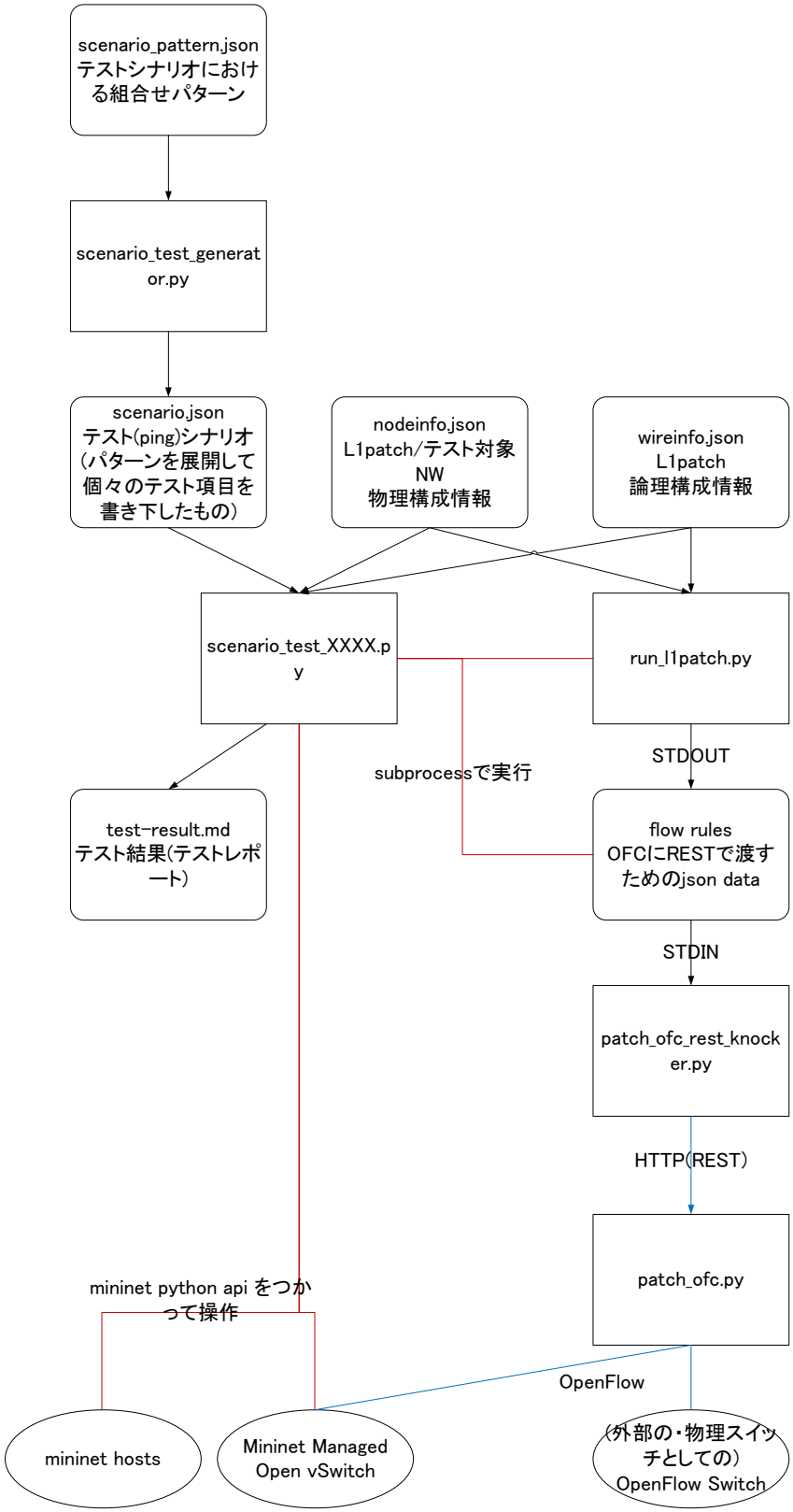

図 6-2 L1patch ツール間データフロー

6.3 OpenFlow Controller

6.3.1 ベースコントローラ

今回の PoC で使用した OpenFlow コントローラは、下記の実装をもとに機能拡張をしている。

- nmasao/OFPatchPanel-SDNHackathon2014 · GitHub
  https://github.com/nmasao/OFPatchPanel-SDNHackathon2014

OFPatchPanel で実装されているコントローラは、今回の PoC でいう専有モードワイヤを実現するためのフロールールを設定可能な機能を持っている。PoC では、共有モードワイヤを実現するためのフロールールに必要な、VLAN の設定やマッチ条件指定などの機能追加を行っている。また、Wire group ID の実装パターンを複数試す(5.3.3,表 5-3 参照)うえで、MPLS タグ操作や OpenFlow1.0/1.3 対応の VLAN ID 操作機能なども実装されている。

6.3.2 静的構造

OpenFlow コントローラのクラス図を図 6-3 に示す。[7]

- patch_ofc_flowrule
  - REST API で REST API で受信したフロールール(json)データのをもとに、個々の Match/Action を生成する。REST API で定義するフロールールと Ryu API で使用するフロールールは同一ではないため、OpenFlow のバージョンごとのフィールド名の違いや一部のフィールドの省略、データ構造の単純化などをおこなうため別途定義・変換をしている。(6.3.3 項参照)
- patch_ofc_flow_builder
  - REST API で受信したフロールール(json)データをもとに、受信したデータのバリデーションやエラー処理、Ryu API に送信するフローデータの構築などを行う。
- patch_ofc_rest_knocker
  - フロールール(json)のデータを OpenFlow コントローラの REST API に設定する(送信する)ツール。

[7] 以降、クラス図についてはソースコードを元に自動生成した図を記載している。そのため、基本的な関連や継承関係を示しているのみで、関連の多重度などは記載されない簡易的なクラス図となっている。

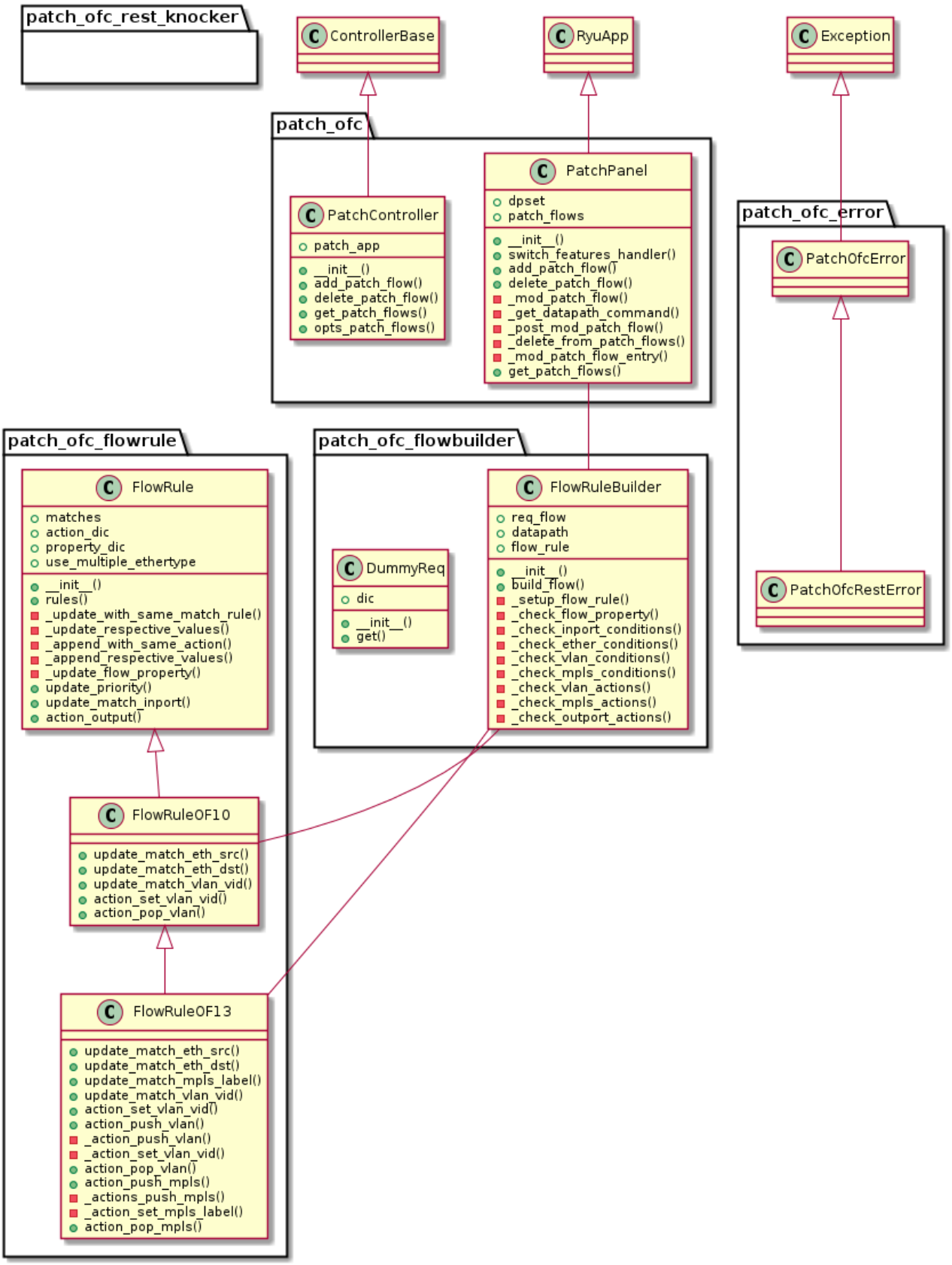

図 6-3 クラス図(OpenFlow Controller)

6.3.3 API 設計

L1patch OpenFlow コントローラの REST API を表 6-1 に示す。

- HTTP Method
  - PUT でフロールールの設定、DELETE でフロールールの削除を行う。
  - GET を送信すると、現在設定されているフロールールの一覧を返す。
- HTTP Response
  - 問題なくリクエストが受け付けられると 200 番を返す。
  - 送信するデータや送信先 URL などに問題がある場合は 400/500 番台を返す。
- 送受信するデータについて
  - フロールールは特定の OFS に対して設定する。そのため、設定対象となるスイッチのデータパス ID(DPID)を指定する。
  - 専有/共有モードワイヤの設定に必要な操作に限定して OpenFlow フローエントリ(フロールール)操作を行うため、Match/Action 等のデータ構造は簡略化している(Match/Action などで構造化したデータにしていない)。表 6-1”Flow type”列は、そのフィールドが Match 条件設定のフィールドなのか、Action 条件設定のフィールドなのかを示している。
  - 表 6-1”internal”列は、コントローラ内部で使用する Ryu API(mod_flow_entry())に対して渡すデータで使用するフィールド名である。OpenFlow バージョン(OF10/13)で Ryu API で設定するフィールド名などの違いがあるが、L1patch OFC REST で操作するデータではそれらの違いを意識しないようになっている。
  - MAC アドレスの Matci, VLAN ID の Match/Action は共有モードワイヤでのみ使用される。

表 6-1 L1patch OpenFlow Controller REST API

REST REQUEST							
PUT /patch/flow							
	REST	internal (mod_flow_entry)	Value type	Flow type	Exclusive (L1)	Shared (L2)	
{							
	priority		<int>	----	○	○	
	dpid		<int>	----	○	○	必須(10 進数指定注意)
	inport		<int>	match	○	○	必須
	outport		<int>	action	○	○	どちらかは必須
	outports		[<int>]	action	○	○	outport/outports 両方を同時には使わない(両方ある場合は outports を使う)。
	eth_src	dl_src	<string>	match	----	○	どちらかは必須
	eth_dst	dl_dst	<string>	match	----	○	
	vlan_vid	dl_vlan	<int>	match	----	○	match vlan &&

							dl_dst=broadcast
	set_vlan		<int>	action	----	○	vlan tag が既にあれば変更(modify)，なければ追加(push)
	pop_vlan		TRUE	action	----	○	value = true であれば実行
}							

6.4 L1patch フロールール生成スクリプト

6.4.1 静的構造

L1patch フロールール生成ツールのクラス図を図 6-4 に示す。L1patch フロールール生成ツールは 5 章に示した L1patch 設計に基づいて L1patch のモデル(図 5-4)を実装している。

- patch_flowgen
  - L1patch の物理構成・論理構成情報をもとに、ホスト・ポート・ワイヤなどの各インスタンスを生成する。構成情報から OpenFlow コントローラに送信するフロールール(json)を生成する。

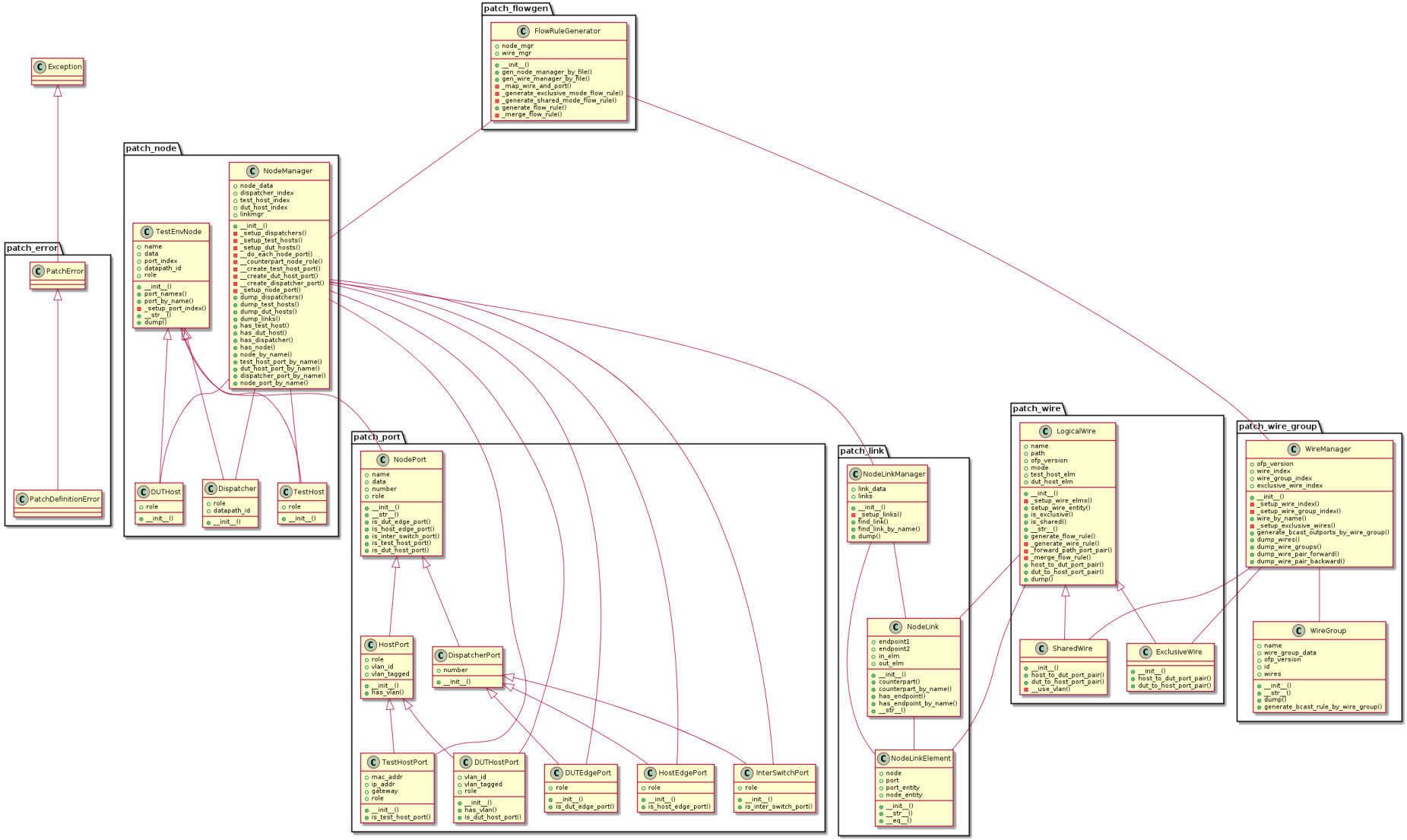

図 6-4 クラス図(フロールール生成)

6.5 テストシナリオ操作スクリプト

6.5.1 静的構造

テストシナリオ操作ツールのクラス図を図 6-5 に示す。テストしなり操作は、テストパターン定義から個々のテストケースを生成するツールと、生成されたテストケースをもとにテスト用ノードを操作し、テストレポートを生成するツールのふたつのツールがある。

- scenario_generator
  - テストパターン定義をもとに、テストケースを自動生成する。
- scenario_tester
  - テストケースを読み込んで、テスト用ノード(Mininet host)の操作を行う。
  - クラス”ScenarioTesterBase”は、テスト自動実行を行う際に必要となる共通的なテスト環境の操作、すなわち、Mininet 環境の起動、テスト用ノードの生成とネットワークパラメータ設定などを行う。
  - 派生クラスで、Mininet 環境上での特定のアプリケーション実行操作を定義する。
- scenario_pinger
  - 今回の PoC で実行するテストアプリケーション(テストケースデータをもとに ping コマンドを実行する)の処理を実装している。
- scenario_pinger_topo2
  - 机上テスト校正用のテスト用ネットワーク構築処理の実装。
  - 実機環境に対するネットワークテストでは、Mininet 環境にはテスト用ノードを接続するひとつの OFS(OVS, s1)だけがあればよい。しかし、机上テストでは、机上テスト環境で使用するテスト用ネットワークをすべて OVS Bridge で準備する必要がある。そのため、Mininet 環境を準備する処理をオーバーライドして再実装している。
- scenario_result_manager
  - テスト自動実行にあたって、テスト結果の保持とレポーティングを行うための機能実装。

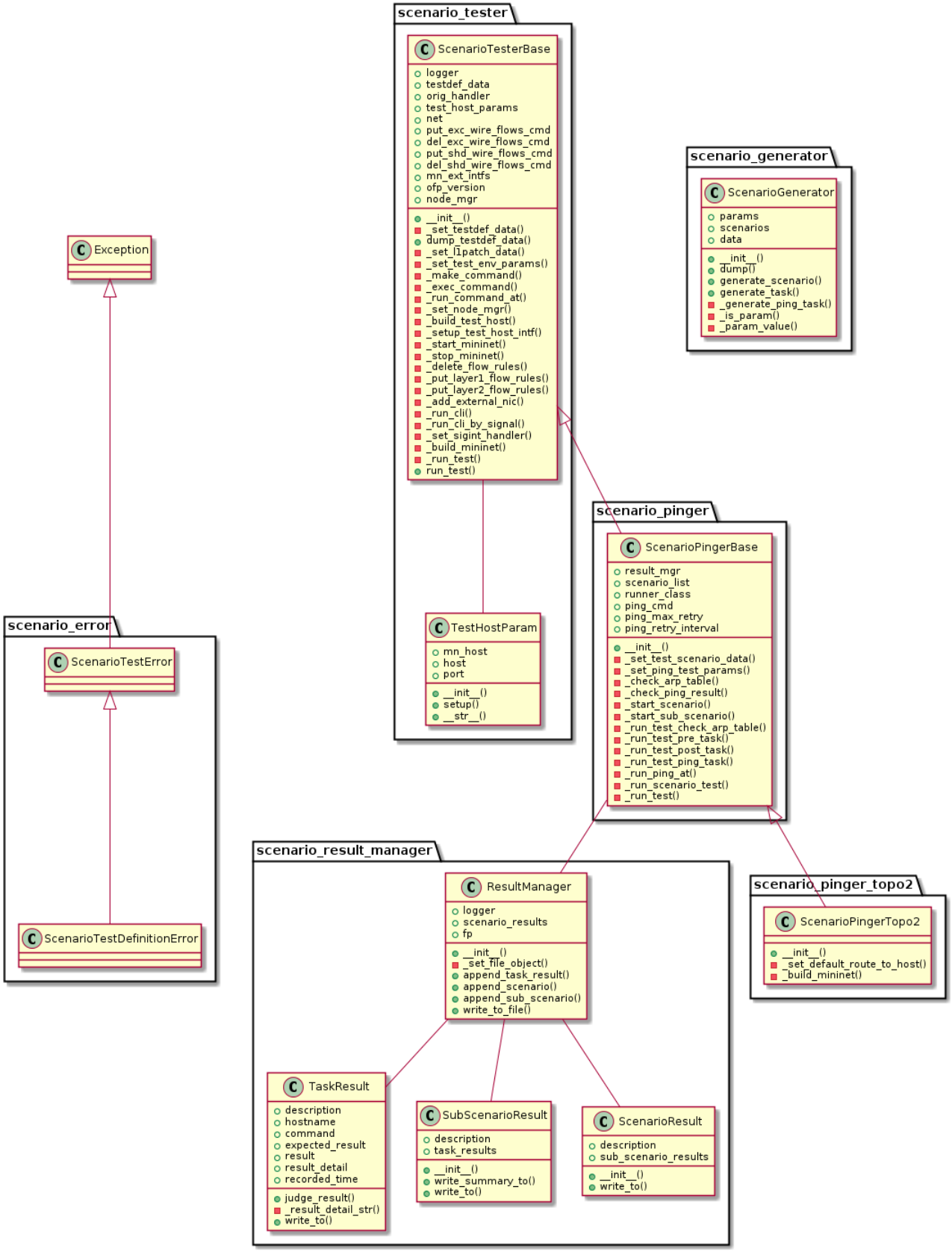

図 6-5 クラス図(テストシナリオ生成・テスト自動実行)

6.5.2 ロギング

テスト自動実行スクリプトでは、処理のエビデンス取得・または実装でバッグのために動作ログ取得機能を組み込んでいる。ログレベル指定は一般的なロガー(logger)におけるレベル指定となっている(表 6-2)[8]。

表 6-2 L1patch ツールで使用しているログレベル設定

ログレベル	L1patch での用途	例
DEBUG		
INFO	一般的な動作ログ(想定されていた動作のログ)は informational として出力される。	表 6-3
WARNING	テストのリトライなど、動作しているが注意が必要なものについては warning で出力される。	表 6-4
ERROR	OFC REST で 400 番台などが帰っているなど動作上の不具合が起きているときに出力される。	表 6-5
CRITICAL		

表 6-3 INFO: 想定されていた動作の確認

シナリオテスト実行時のメッセージ類
2015-09-27 11:49:56,039 - scenario_tester - INFO - exec command: python run_l1patch.py -p nodeinfo_topo5.json -l wireinfo_topo5.json > flows_topo2.json 2015-09-27 11:49:56,106 - scenario_tester - INFO - exec command: python scenario_generator.py -f scenario_pattern_topo5.json > scenario_topo5.json 2015-09-27 11:49:56,147 - __main__ - INFO - run scenario test with runner-class: ScenarioPingerBase

表 6-4 WARNING: 問題となり得る動作を行っている場合

テストリトライ
2015-09-27 11:50:09,580 - scenario_tester - INFO - run @`ext1`: `ip neigh show` 2015-09-27 11:50:09,581 - scenario_tester - INFO - run sub scenario: main tasks 2015-09-27 11:50:09,581 - scenario_tester - INFO - [8.3%/current:1/total:12] run task: [L2SW1-dmz1] ping dmz11 to dmz12 2015-09-27 11:50:09,581 - scenario_tester - INFO - run @`dmz11`: `ping -i 0.2 -c 5 10.13.1.2` 2015-09-27 11:50:12,582 - scenario_tester - INFO - result: FAIL 2015-09-27 11:50:12,583 - scenario_tester - WARNING - task: [L2SW1-dmz1] ping dmz11 to dmz12, (retry:1/5, after wait 1[sec]) 2015-09-27 11:50:13,584 - scenario_tester - INFO - run @`dmz11`: `ping -i 0.2 -c 5 10.13.1.2` 2015-09-27 11:50:16,586 - scenario_tester - INFO - result: FAIL

[8] ロギング HOWTO — Python 2.7ja1 documentation http://docs.python.jp/2/howto/logging.html

```
2015-09-27 11:50:16,586 - scenario_tester - WARNING - task: [L2SW1-dmz1] ping dmz11 to dmz12, (retry:2/5, after wait 2[sec])
2015-09-27 11:50:18,588 - scenario_tester - INFO - run @`dmz11`: `ping -i 0.2 -c 5 10.13.1.2`
2015-09-27 11:50:21,586 - scenario_tester - INFO - result: FAIL
2015-09-27 11:50:21,586 - scenario_tester - WARNING - task: [L2SW1-dmz1] ping dmz11 to dmz12, (retry:3/5, after wait 3[sec])
2015-09-27 11:50:24,589 - scenario_tester - INFO - run @`dmz11`: `ping -i 0.2 -c 5 10.13.1.2`
2015-09-27 11:50:27,590 - scenario_tester - INFO - result: FAIL
2015-09-27 11:50:27,590 - scenario_tester - WARNING - task: [L2SW1-dmz1] ping dmz11 to dmz12, (retry:4/5, after wait 4[sec])
2015-09-27 11:50:31,595 - scenario_tester - INFO - run @`dmz11`: `ping -i 0.2 -c 5 10.13.1.2`
2015-09-27 11:50:34,594 - scenario_tester - INFO - result: FAIL
2015-09-27 11:50:34,594 - scenario_tester - WARNING - task: [L2SW1-dmz1] ping dmz11 to dmz12, (retry:5/5, after wait 5[sec])
2015-09-27 11:50:39,600 - scenario_tester - INFO - run @`dmz11`: `ping -i 0.2 -c 5 10.13.1.2`
2015-09-27 11:50:42,598 - scenario_tester - INFO - result: FAIL
2015-09-27 11:50:42,598 - scenario_tester - INFO - [16.7%/current:2/total:12] run task: [L2SW1-dmz1] ping dmz12 to dmz11
```

表 6-5 ERROR: 問題の発生

OFC REST で 400 が返っている

```
2015-09-27 11:50:00,018 - __main__ - INFO - Send PUT: node:l1pjofsw3, rule:{"outport": 1, "dpid": 6790914079155444010, "eth_dst": "ff:ff:ff:ff:ff:ff", "inport": 5, "priority": 16535, "set_vlan": 108, "vlan_vid": 2016}
2015-09-27 11:50:00,018 - __main__ - INFO - Response: {'date': 'Sun, 27 Sep 2015 02:50:00 GMT', 'status': '200', 'access-control-allow-origin': '*', 'content-length': '0'}
2015-09-27 11:50:00,019 - __main__ - INFO - Content:
2015-09-27 11:50:00,120 - __main__ - ERROR - Send PUT: node:l1pjofsw2, rule:{"priority": 65535, "outport": 10, "inport": 11, "dpid": 4}
2015-09-27 11:50:00,120 - __main__ - ERROR - Response: {'date': 'Sun, 27 Sep 2015 02:50:00 GMT', 'status': '400', 'content-length': '0', 'content-type': 'text/html; charset=UTF-8'}
2015-09-27 11:50:00,120 - __main__ - ERROR - Content:
2015-09-27 11:50:00,221 - __main__ - ERROR - Send PUT: node:l1pjofsw2, rule:{"priority": 65535, "outport": 11, "inport": 10, "dpid": 4}
2015-09-27 11:50:00,222 - __main__ - ERROR - Response: {'date': 'Sun, 27 Sep 2015 02:50:00 GMT', 'status': '400', 'content-length': '0', 'content-type': 'text/html; charset=UTF-8'}
2015-09-27 11:50:00,222 - __main__ - ERROR - Content:
```

6.5.3 オプション

3.4 章でテスト自動実行ツールの使用方法について解説した。ここではオプションと動作についての概要を示す。オプションには下記のふたつのグループがある。オプションと各オプションによる動作の一覧は表 6-6、表 6-7 のようになる。

- Use-case オプション ： 行いたいテストの指定
  - 手動テスト実行のみ
  - テスト自動実行のみ
  - テスト自動実行後に手動テスト実行
- Layer オプション ： 操作したい L1patch ワイヤ種別(「試験結果レポート/6.2.1 L1patch のテスト実行フェーズの考え方」参照)
  - Layer1(専有モードワイヤのみ)
  - Layer2(共有モードワイヤのみ)
  - 専有・共有モードワイヤ双方

表 6-6 テスト自動実行ツールのオプション

```
hagiwara@prjexp01:~/PycharmProjects/l1patch-dev$ sudo python run_scenario_test.py
--help
usage: run_scenario_test.py [-h] -f JSON [-m | -t] [-1 | -2 | -a]

Run Scenario Test

optional arguments:
  -h, --help            show this help message and exit
  -f JSON, --testdef JSON
                        Test definition file
  -m, --manual          Go to Mininet CLI with L1patch setup.
  -t, --test-cli        Run auto test and go to CLI when test finished.
  -1, --layer1          Setup L1(exclusive mode wire) flow rules
  -2, --layer2          Setup L2(shared mode wire) flow rules
  -a, --all-layers      Setup all flow rules
hagiwara@prjexp01:~/PycharmProjects/l1patch-dev$
```

表 6-7 テスト自動実行ツールのオプション別動作

	ユースケースオプション		
	-m --manual	-t --test-cli	(なし/default)
用途(Use-case)	L1patch 設定情報	テスト自動実行後の	テスト自動実行後

			の確認(4.6 節)、Mininet CLI 上の手作業テスト実行(4.8 節)	手動作業実行(4.10 節)	終了(4.9 節)
Mininet のセットアップ			実行する		
ワイヤオプション	すべてのワイヤを設定(-a, --all-layers)	専有モードワイヤの設定(-1, --layer1)	--layer1 or --all-layers 指定があれば実行する		
		共有モードワイヤの設定(-2, --layer2)	--layer2 or --all-layers 指定があれば実行する		
テスト自動実行 (Control-c で中段/CLI モード移行)			実行しない	実行する	実行する
テスト手動実行 (CLI モード移行)			実行する	実行する	実行しない

6.5.4 テスト結果レポート

テスト自動実行後、テスト結果レポートが生成・出力される。テスト結果レポートは Markdown 形式となっている(表 6-8)ため、テキストファイルとしての可読性もあり、かつ、各種 Markdown 対応ツールを使って表示あるいはフォーマット変換することができる。図 6-6 に IDE(PyCharm[9],Markdown plugin)でのテスト結果レポート表示例を示す。

今回は直接 Markdown に出力している(図 6-2)が、複数のテストアプリケーションを使う場合には、テスト結果を共通のデータフォーマット(中間形式)で出力し、テスト結果の加工用ツールを別途準備するなどの拡張が考えられる。

表 6-8 テスト結果レポート(抜粋)

机上テスト テスト結果レポート(test_result_topo2.md)
Report created date: 2015-10-11 12:55:28 # run scenario: test-node to dut ## run sub scenario: pre-tasks ### summary No.|host|command|result|expected|judge ---|----|-------|------|--------|-----

[9] Python IDE & Django IDE for Web developers : JetBrains PyCharm https://www.jetbrains.com/pycharm/

```
1|h1|`ip neigh show`|SUCCESS|SUCCESS|*OK*
2|h2|`ip neigh show`|SUCCESS|SUCCESS|*OK*
3|h3|`ip neigh show`|SUCCESS|SUCCESS|*OK*
4|h4|`ip neigh show`|SUCCESS|SUCCESS|*OK*
5|h5|`ip neigh show`|SUCCESS|SUCCESS|*OK*
6|h6|`ip neigh show`|SUCCESS|SUCCESS|*OK*
7|h7|`ip neigh show`|SUCCESS|SUCCESS|*OK*
8|h8|`ip neigh show`|SUCCESS|SUCCESS|*OK*

### run task: clear host h1 arp table

- host: h1
- command: `ip neigh show`
- recorded: 2015-10-11 12:54:46
- judge: *OK*
  - expected result: SUCCESS
  - result: SUCCESS
- result detail:


(省略)
```

```
## run sub scenario: main tasks

### summary

注: このテストは--laye1 オプションを指定せずに机上テストのテスト自動実行を行ったケースの結果レポートである。(4.10.2 項,表 4-8 参照)

No.|host|command|result|expected|judge
---|----|-------|------|--------|-----
1|h1|`ping -c 5 -i 0.2 192.168.2.106`|SUCCESS|SUCCESS|*OK*
2|h2|`ping -c 5 -i 0.2 192.168.2.106`|SUCCESS|SUCCESS|*OK*
3|h3|`ping -c 5 -i 0.2 192.168.2.107`|SUCCESS|SUCCESS|*OK*
4|h4|`ping -c 5 -i 0.2 192.168.2.107`|SUCCESS|SUCCESS|*OK*
5|h5|`ping -c 5 -i 0.2 192.168.2.108`|FAIL|SUCCESS|**NG**

### run task: [shared-wire-to-h6] ping h1 to @h6@
```

```
- host: h1
- command: `ping -c 5 -i 0.2 192.168.2.106`
- recorded: 2015-10-11 12:54:47
- judge: *OK*
  - expected result: SUCCESS
  - result: SUCCESS
- result detail:

    PING 192.168.2.106 (192.168.2.106) 56(84) bytes of data.
    64 bytes from 192.168.2.106: icmp_seq=1 ttl=64 time=0.393 ms
    64 bytes from 192.168.2.106: icmp_seq=2 ttl=64 time=0.051 ms
    64 bytes from 192.168.2.106: icmp_seq=3 ttl=64 time=0.055 ms
    64 bytes from 192.168.2.106: icmp_seq=4 ttl=64 time=0.057 ms
    64 bytes from 192.168.2.106: icmp_seq=5 ttl=64 time=0.045 ms
    --- 192.168.2.106 ping statistics ---
    5 packets transmitted, 5 received, 0% packet loss, time 799ms
    rtt min/avg/max/mdev = 0.045/0.120/0.393/0.136 ms

(省略)
```

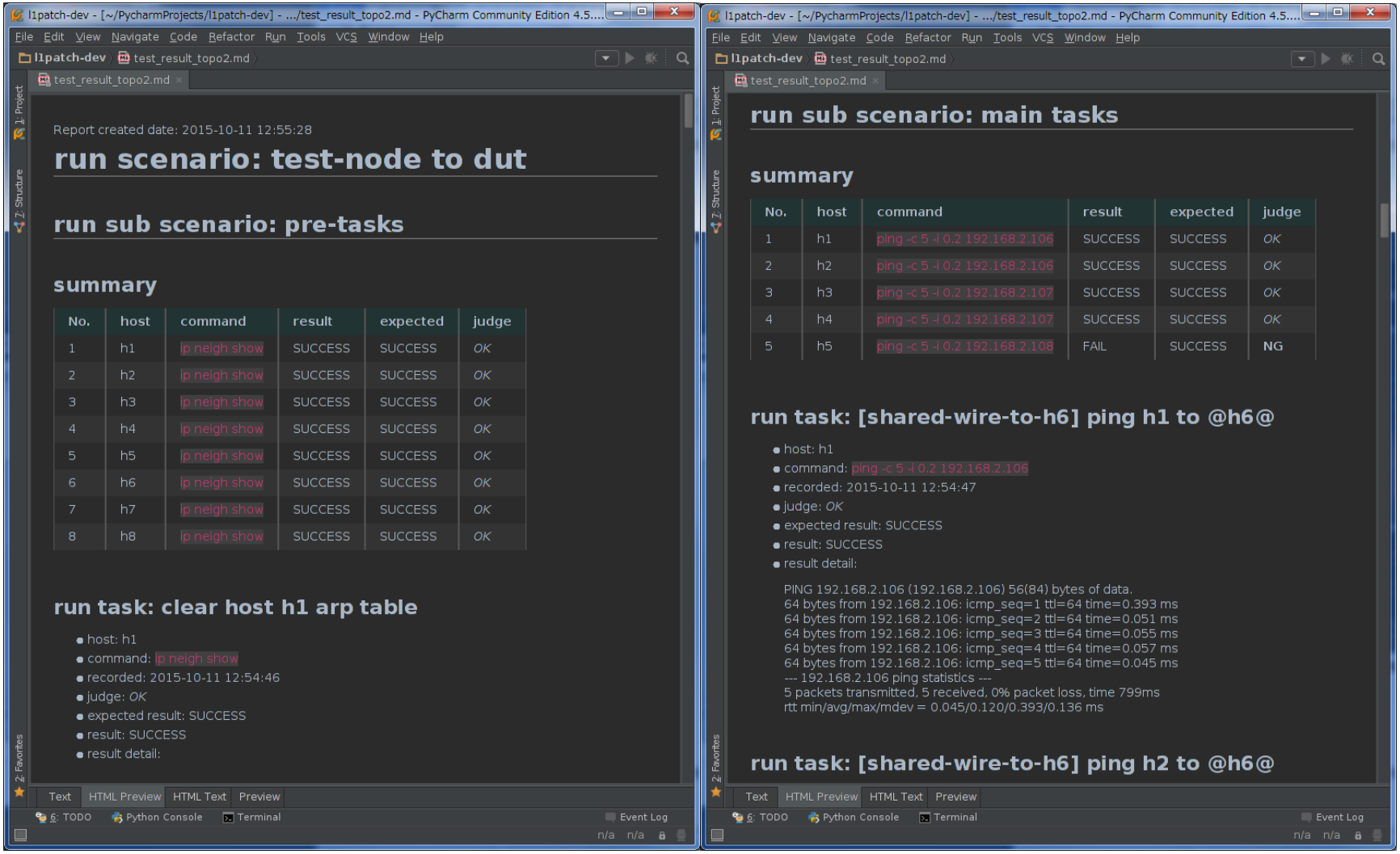

図 6-6 テスト結果レポート(pycharm による表示例)

7 おわりに

「L1patch 応用ネットワークテスト」で実装・構築したテスト用のシステム、特に”L1patch”の動作仕様・通信制御設計、ツールの使い方や実装上の設計について記載した。

3 章・4 章では L1patch それ自体の機能テスト(机上テスト)を中心に、L1patch を使ったテスト実施の流れや設定ファイルの定義、ツールの使い方などを解説した。机上テストは、単一の OS 上で実行可能なテストであり、仮想環境(Mininet で構成する仮想 NW)としてテスト対象を構成するため、すぐに L1patch の試用が可能な構成としてある。机上テスト実行を行うことで L1patch の基本的な動作や使い方を理解することができるだろう。

実際の動作や使い方を把握した上で、必要に応じて動作設計(5 章/通信制御方法)、実装(6 章/アーキテクチャや実装されたツールの構造)を参照すること。8 章には、今回 PoC で使用した機材における利用上(設定上)の注意点をまとめた。実機を使用したテスト環境構築(実機ベースの L1patch)を構築する際に参考にしてほしい。

8 補足

8.1 機器セットアップ時の注意事項

使用している機器型番・バージョン等の詳細については「試験結果レポート/8.1 PoC 環境で使用した機器・ソフトウェア情報」参照すること。

8.1.1 PicOS ポート設定

PoC では、OFS として Pica8/PicOS の OVS モードを使用している。Pica8 OVS で VLAN Tag を操作するフロールールを設定する場合、ポートに対して vlan_mode=trunk 指定が必要になる点に注意すること(表 8-1, vlan_mode=trunk を指定しない場合、VLAN Tag のついたパケットは処理されない)。

表 8-1 PicOS OVS の VLAN Mode 指定

設定	admin@PicOS-OVS$ ovs-vsctl set Port {ポート名} vlan_mode=trunk 例) admin@PicOS-OVS$ ovs-vsctl set Port ge-1/1/1 vlan_mode=trunk
確認	admin@PicOS-OVS$ ovs-vsctl list Port (省略) vlan_mode : trunk (省略)

また、今回は同じテスト対象機器(L2 スイッチ・ルータ)の複数のポートを同じ OpenFlow スイッチに接続するため、通常ではループする構成となってしまう。これらの接続を実現するために、フラッディングとスパニングツーリーの設定を無効にする必要がある。(表 8-2、表 8-3)

無効にした設定は再起動により消えてしまうため、起動のたびに設定が必要となる。設定スクリプトを作成して起動時に実行して設定させる場合には PicOS のプロセスより後の順番で起動させる必要がある。PicOS は Init

Level 2 として/etc/rc2.d に登録されているため、それ以降に実行されるようにスクリプトを登録する必要がある。なお、Linux として非推奨となっているため、rc.local にコマンドを登録することはできないので注意が必要である。

表 8-2 PicOS OVS のフラッディングの動作無効の設定

設定	admin@PicOS-OVS$ ovs-ofctl mod-port {ブリッジ名} {ポート名} no-flood 例） admin@PicOS-OVS$ ovs-ofctl mod-port br0 ge-1/1/1 no-flood
確認	admin@PicOS-OVS$ ovs-ofctl {ブリッジ名} show 例） admin@PicOS-OVS$ ovs-ofctl br0 show （省略） 1(ge-1/1/1): addr:e8:9a:8f:d2:81:1a config: NO_STP NO_FLOOD （省略）

表 8-3 PicOS OVS のポートのスパニングツーリーの動作無効の設定

設定	admin@PicOS-OVS$ ovs-ofctl mod-port {ブリッジ名} {ポート名} no-stp 例） admin@PicOS-OVS$ ovs-ofctl mod-port br0 ge-1/1/1 no-stp
確認	admin@PicOS-OVS$ ovs-ofctl {ブリッジ名} show 例） admin@PicOS-OVS$ ovs-ofctl br0 show （省略） 1(ge-1/1/1): addr:e8:9a:8f:d2:81:1a config: NO_STP NO_FLOOD （省略）

8.1.2 Mininet サーバの NIC 設定

実機環境をテストする 5 台構成テストでは、Mininet OVS が外部環境と接続するためのインタフェースが必要になる(eth2, 表 3-8 参照)。今回、Mininet サーバはハイパーバイザ上の仮想マシン(VM)ではなく、ベアメタルサーバとして用意しているため、eth2 については特にプロミスキャスモードへの変更などを行う必要はない。(プロミスキャスモードに設定しても特に問題はない。)

図 6-1 では、VM として Mininet サーバを構築した場合の想定を記載した。今回の PoC ではテストしていないが、VM として Mininet サーバを構築した場合、”eth2”相当のインタフェースが接続する仮想スイッチ(vSW)ではプロミスキャスモードの設定が必要になると考えられる。(L1patch が転送してきた ARP 等がそのまま VM に対して転送されなければならないため。Hypervisor 上の vSW が OpenFlow でコントロール可能な場合は除く。)