OpenFlow 技術のホームネットワークへの適用に関する研究

北陸先端科学技術大学院大学
情報科学研究科

迫田 紘志

平成 27 年 9 月

課題研究報告書

OpenFlow 技術のホームネットワークへの適用に関する研究

1010752 迫田 紘志

主指導教員 丹 康雄

審査委員主査 丹 康雄
審査委員 篠田 陽一
リム 勇仁

北陸先端科学技術大学院大学
情報科学研究科

平成 27 年 8 月

目次

第1章　はじめに

1.1. 背景

データセンタではサーバやストレージの仮想化技術とクラウドの進展に伴い、既存のネットワーク技術だけでは難しい問題が数多くある。代表例として VLAN_ID が足りない、マイグレーションによるポートの移動問題、新規に増設した物理スイッチや物理サーバの設定、アプリケーションやサービスの変更に合わせたネットワーク設定変更などが挙げられる。

OpenFlow 技術はひとつの装置の中に存在していた、実際にルーティングやスイッチングを行う機能とコンフィグレーションを行う機能を分離する。さらに、集中的にコンフィグレーションを行えるようにすることで、いわばプログラマブルなネットワークを実現する。データセンタで起きている様々な問題を解決するものとして期待されている。

一方、ホームネットワークは小規模ながらも制御系やストリームデータ伝送系など、様々なアプリケーションが混在し、異なる性質を有するトラヒックが発生する。また、ユーザの行動に応じてトラヒックの時間的な変動も大きいという特徴がある。ホームネットワークでは UTP ケーブルなどによる専用の情報配信線を新規に行うことが望ましくない。電力線や無線といった伝送能力が不安定な媒体を使わざるを得ないことが多い。そのため十分な通信品質が得られない状況に陥ることが珍しくない。

ホームネットワークにおいて、ユーザの利用状況に応じてネットワーク機器の設定をダイナミックに変更し、適切な QoS 制御を行うことにより、限られた伝送能力を最大限に活用できる可能性が取りざたされている。しかし、QoS の設定方法は機器ごとに異なる。さらに、網羅的に対応する必要があるなど、その実現は容易ではなかった。

本研究では、データセンタでの課題を解決するプロトコルとして開発が進みつつある OpenFlow 技術をホームネットワークへ適用することで、ホームネットワークにおける課題の解決が可能であるかについて検討を行うものである。

1.2. 目的

本研究では、従来エンタプライズネットワークやデータセンタ内で用いられてきたOpenFlow 技術のホームネットワークへの適用に関する調査と検討を行う。

OpenFlow 技術の開発経緯、現状の規格などについて調査する。さらに、ホームネットワークにおける諸課題について分析を行う。

OpenFlow 技術の適用によりホームネットワークの状況がどのように改善しうるかについて検討する。また、実際に環境を構築し、評価を行うことでその効果を明らかにする。

1.3. 構成

本報告書は全 6 章で構成されている。以下に各章の概要を述べる。

第 1 章では、本研究の背景と目的について述べる。

第 2 章では「OpenFlow 技術」と題して、OpenFlow 技術の開発経緯、規格について調査する。

第 3 章では「ホームネットワーク」と題して、ホームネットワークの概要と諸課題について分析を行う。

第 4 章では「OpenFlow ネットワーク構築」と題して、第 3 章にて分析した課題の中にOpenFlow 技術で解決出来そうなものを抽出し、仮想環境を用いネットワークを構築する。

第 5 章では「OpenFlow 技術を用いたホームネットワーク構築」と題して、第 4 章にて構築したネットワークを実際に構築し、データを取得する。

第 6 章では、「研究結果と今後の課題」と題して、本報告書をまとめ、今後の課題と展望について述べる。

第2章　OpenFlow 技術

2.1. OpenFlow 開発経緯

2.1.1. 従来ネットワークの特徴と問題点

OpenFlow 技術の開発経緯を説明するに辺り、現在主流となっているネットワークについて述べる。現在のネットワーク技術 TCP/IP は OSI 参照モデルで定義されているレイヤと呼ばれる階層に分けられている。

OSI 参照モデル	TCP/IP	代表プロトコル
アプリケーション層	アプリケーション層	HTTP SMTP POP3 SSH FTP
プレゼンテーション層		
セッション層		
トランスポート層	トランスポート層	TCP UDP
ネットワーク層	インターネット層	IP
データリンク層	ネットワーク インターネット層	Ethernet PPP
物理層		

表 1:OSI 参照モデルと TCP/IP 階層構造

各階層において IETF や IEEE、ITU-T などで標準化された技術仕様に従って、各ベンダがネットワーク機器を開発し、相互に通信できるようになっている。

各ネットワーク機器ベンダはネットワーク装置のハードウェアと各階層に応じた処理を行うソフトウェアの両方を開発する。その際、標準化されていない独自の機能を追加し、製品を提供している。つまり、各ベンダの製品が必ずしも標準化された機能のみ実装されているわけではない。機能によっては採用するベンダが限定される。

実現したいネットワークがあったとしても、ネットワーク装置の制約によって実現できない可能性がある。何故なら、ハードウェアとソフトウェアが一体となっているため、ベンダが想定するネットワークとは異なる場合、ベンダに対応してもらう必要があるからである。

また、近年ネットワークにつながるシステムやサービスなどの大規模化・複雑化によって既存のプロトコルでは対処できない様々な課題が浮上している。ひとつ例を挙げると、大規模データセンタではサーバ仮想化技術を導入することが多い。しかし、サーバ仮想化の利点である動的な設定変更にネットワークが自動で追従できない問題がある。

2.1.2. Software Defined Networking

従来ネットワークの問題点を解決するに辺り、近年大きなトレンドとして注目されているのが Software Defined Networking(以下 SDN)というコンセプトである。[1]

SDN とはソフトウェアによって仮想的なネットワークを構築する技術全般のことである。SDN により、ネットワーク機器の物理的な制約から離れ、目的に応じたネットワークを構築することが出来る。

OpenFlow 技術は SDN のコンセプトを実現するための一つの技術である。

2.1.3. OpenFlow 技術の歴史

OpenFlow 技術は 2007 年頃にアメリカのスタンフォード大学で研究が始まった。

ネットワーク上でデータを適切に送るための経路制御など、複雑な計算を担う「コントロールプレーン」、フレーム転送など単純な処理をする「データプレーン」、各種プロトコルの機能を提供する「アプリケーション」を分離する。そのうち「コントロールプレーン」と「データプレーン」を結ぶ標準インターフェースが OpenFlow プロトコルである。

OpenFlow プロトコルの標準化は「OpenFlow Switch コンソーシアム」によって進められた。このコンソーシアムの主要参加メンバーはアメリカのスタンフォード大学、シスコシステムズ、ヒューレットパッカード、ジュニパーネットワークス、日本電気である。コンソーシアムによって OpenFlow ver1.0.0 と ver1.1.0 の仕様策定がされた。[1][2]

2011 年 3 月に Open Networking Foundation(以下 ONF)が発足した。ONF は SDN の推進を目標に掲げる組織である。ONF のボードメンバーはドイツテレコム、フェイスブック、ゴールドマンサックス、グーグル、マイクロソフト、NTT コミュニケーションズ、ベライゾン、ヤフーである。(2013 年 4 月 27 日現在)OpenFlow 技術の仕様策定は ONF に移行し、ver1.2.0 と ver1.3.2 の仕様策定がされた。(2013 年 4 月 27 日現在)

2.2. OpenFlow プロトコル

OpenFlow プロトコルにはコントロールプレーンである「OpenFlow コントローラ」とデータプレーンである「OpenFlow スイッチ」が存在する。

2.2.1. OpenFlow コントローラの役割

OpenFlow コントローラ(以下 OFC)の最も大事な役割は「経路計算」である。OFC のアルゴリズムを基に経路が決定される。最短経路や回線速度。重み付けや、負荷分散などの条件を加味して経路を計算する必要がある。経路が決定したら、OpenFlow スイッチにフローエントリを書き込む。

2.2.2. OpenFlow スイッチの役割

OpenFlow スイッチ(以下 OFS)の役割は自身に投入されたフローエントリにしたがって動作する。

2.3. OpenFlow バージョンによる違い

OpenFlow スイッチのバージョンは現在大きく分けて ver1.0、1.1、1.2、1.3 の 4 つに分けられる。主な違いを以下の表にしめす。[13][14][15][16](2013 年 4 月 27 日現在)

Version	項目	変更内容
1.0	ベースバージョン	-
1.1	複数テーブル対応	OFS の複数のテーブル処理
	グループ化処理対応	OFS の複数ポートをひとつのグループとする処理
	MPLS/VLAN Tag 対応	MPLS および VLAN タグの追加・修正・削除対応
	マスク表記対応	MAC アドレス、ネットワークアドレスのマスク表記対応
	TTL 処理対応	IPv4 及び MPLS の TTL の変更・複製・減算対応
	SCTP 対応	SCTP ヘッダによるマッチング。書き換え処理対応
1.2	OXM 概念導入	OXM(OpenFlow Extensible Match)定義
	実験用エラーコード追加	実験用エラーコード追加
	IPv6 サポート	IPv6・ICMPv6・IPv6 TTL 対応
	FlowMod 時動作変更	複数コントローラ対応
	転送種別の概念変更	ALL・FLOOD・CONTROLLER などの予約済みポート追加
1.3	フローが存在しなかった場合の挙動変更	一致しなかった場合、Drop 処理に変更
	Merter Table 追加	QoS の対応
	IPv6 Extension ヘッダ対応	IPv6 Extension ヘッダサポート
	OFC/OFS 間複数コネクション対応	複数の TCP コネクション対応
	MPLS BoS マッチング対応	MPLS BoS でのマッチングサポート
	Tunnel-ID メタデータ追加	マッチングに Tunnel-ID の metadata を利用可能
	Cookie 追加	フローヒットが区別できるよう cookie 追加
	フローカウンタ選択	カウンタを持たないフローの記述対応

表 2:OSI 参照モデルと TCP/IP 階層構造

現在の主流は OpenFlow ver1.0、1.1 である。(2013 年 4 月 27 日現在)

OpenFlow ver1.3 から QoS 機能に対応しているが、OpenFlow ver1.3 機能を完全に実装

した OFS がない(2013 年 4 月 27 日現在)ので、本実験は OpenFlow ver.1.0 を使用する。

2.4. OpenFlow ver1.0.0 プロトコル

OpenFlow Switch Specification 1.0.0 や参考文献を基に OpenFlow ver1.0.0 のプロトコルを調査する。[7][9][13]

2.4.1. Switch Components

「フローテーブル」と「セキュアチャネル」の 2 つの要素で構成される。OFC は OpenFlow プロトコルを利用し、セキュアチャネル経由でスイッチを管理する。

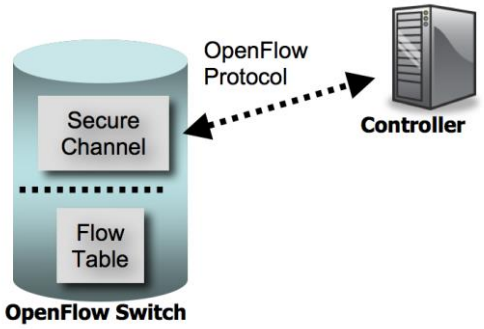

図 1:OpenFlow 通信図 …OpenFlow Switch Specification 1.0.0 P1 の Figure1 引用

フローテーブルは「フローエントリ」、「アクティビティカウンタ」、「アクション」から構成する。

OFS を通過する全てのパケットはフローテーブルと照合する。一致する場合はアクション処理を実行する。一致しない場合は OFC へ転送する。OFC にてパケット処理を決定後、OFS のフローテーブル書き換えを行う。

OpenFlow ポートは「up」、「down」、「スパニングツリープロトコルのパケットを可否」の 3 つの状態を持つ。

2.4.2. Flow Table

フローテーブルは Header Fields、Counters、Actions の 3 つの要素からなる。

この 3 つをまとめてフローエントリと呼ぶ。

2.4.2.1. Header Fields

Header Fields は「タプル」と呼ばれる 12 種類の情報とパケットが一致するかを判断するのに用いる。「タプル」は特定の値または ANY が設定可能である。

Ingress Port	Ether source	Ether dst	Ether type	VLAN id	VLAN pri-or-ity	IP src	IP dst	IP proto	IP ToS bits	TCP/ UDP src port	TCP/ UDP dst port

表 3:OSI 参照モデルと TCP/IP 階層構造…OpenFlow Switch Specification 1.0.0 P3 の table2 引用

2.4.2.2. Counters

Counters は統計情報の為に利用する。パケットが一致した場合にアップデートされ、テーブル、フロー、ポート、キューごとに保存される。必要なカウントは以下表の通りである。

Counter	Bits
Per Table	
Active Entries	32
Packet Lookups	64
Packet Matches	64
Per Flow	
Received Packets	64
Received Bytes	64
Duration (seconds)	32
Duration (nanoseconds)	32
Per Port	
Received Packets	64
Transmitted Packets	64
Received Bytes	64
Transmitted Bytes	64
Receive Drops	64
Transmit Drops	64
Receive Errors	64
Transmit Errors	64
Receive Frame Alignment Errors	64
Receive Overrun Errors	64
Receive CRC Errors	64
Collisions	64
Per Queue	
Transmit Packets	64
Transmit Bytes	64
Transmit Overrun Errors	64

表 4:カウンタ必要項目…OpenFlow Switch Specification 1.0.0 P5 の table4 引用

2.4.2.3. Actions

Actions はパケットが一致した場合に実施する処理である。各フローエントリに 0 個以上の Action が記載できる。転送する Action がない場合、パケットは破棄される。Action は記述された順に処理される必要が必ずあるが、パケット送信順序が変わらない保証はない。

処理が出来ない Action リストに対して、OFS は「unsupported flow error」パケットを OFC に送信する。

OFS は「Required Action」は必ず実装されているが、「Optional Action」は実装されていなくても良い。OFC と初めて接続する際、どの「Optional Action」が実装されているか通知する。

OpenFlow ver.1.0 では、以下の 4 種類の Action が定義されている。

- ・Forward(Required Action)
- ・Enqueue(Optional Action)
- ・Drop(Required Action)
- ・Modify-Field(Optional Action)

Forward アクションは複数の動作がある。Forward アクションの動作一覧を以下に記載する。

アクション名	種別	動作説明
ALL	Required	受信ポート以外へパケットを送信する。
CONTROLLER	Required	カプセル化をして、OFC へパケットを送信する。
LOCAL	Required	ローカルスイッチへパケットを送信する。
TABLE	Required	フローテーブルに登録されている Action に従う。(パケットアウトメッセージのみ)
IN_PORT	Required	受信ポートへパケットを送信する。
NORMAL	Optional	既存の L2/VLAN/L3 に沿ったパケットを送信する
FLOOD	Optional	受信ポート以外に最小のスパニングツリーに沿ったパケットを送信する。

表 5:Forwad アクションリスト一覧

Enqueue アクションは各ポートに設定された queue の最後にパケットを追加する。

Drop アクションはパケットを破棄する。

Modify-Field アクションは厳密には必須機能とはしていないが、OpenFlow 技術の有効性を高めるためには必要な機能である。Modify-Field アクションのアクションリストを以下表通りである。

Action	Associated Data	Description
Set VLAN ID	12 bits	If no VLAN is present, a new header is added with the specified VLAN ID and priority of zero. If a VLAN header already exists, the VLAN ID is replaced with the specified value.
Set VLAN priority	3 bits	If no VLAN is present, a new header is added with the specified priority and a VLAN ID of zero. If a VLAN header already exists, the priority field is replaced with the specified value.
Strip VLAN header	-	Strip VLAN header if present.
Modify Ethernet source MAC address	48 bits: Value with which to replace existing source MAC address	Replace the existing Ethernet source MAC address with the new value
Modify Ethernet destination MAC address	48 bits: Value with which to replace existing destination MAC address	Replace the existing Ethernet destination MAC address with the new value.
Modify IPv4 source address	32 bits: Value with which to replace existing IPv4 source address	Replace the existing IP source address with new value and update the IP checksum (and TCP/UDP checksum if applicable). This action is only applicable to IPv4 packets.
Modify IPv4 destination address	32 bits: Value with which to replace existing IPv4 destination address	Replace the existing IP destination address with new value and update the IP checksum (and TCP/UDP checksum if applicable). This action is only applied to IPv4 packets.
Modify IPv4 ToS bits	6 bits: Value with which to replace existing IPv4 ToS field	Replace the existing IP ToS field. This action is only applied to IPv4 packets.
Modify transport source port	16 bits: Value with which to replace existing TCP or UDP source port	Replace the existing TCP/UDP source port with new value and update the TCP/UDP checksum. This action is only applicable to TCP and UDP packets.
Modify transport destination port	16 bits: Value with which to replace existing TCP or UDP destination port	Replace the existing TCP/UDP destination port with new value and update the TCP/UDP checksum This action is only applied to TCP and UDP packets.

表 6: Modify-Field アクションリスト一覧…OpenFlow Switch Specification 1.0.0 P7 の table5 引用

2.4.2.4. Matching

OpenFlow ver.1.0 では以下の状態遷移を規定する。

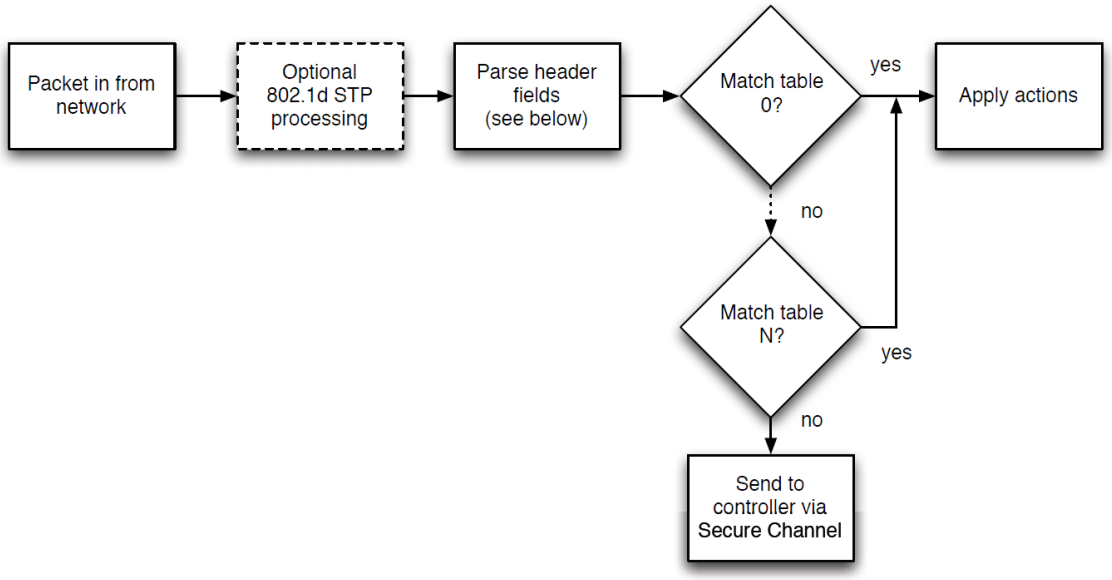

図 2:フローマッチ状態遷移図 …OpenFlow Switch Specification 1.0.0 P8 の Figure2 引用

OFS で STP 機能が有効かつ受信したパケットが IEEE802.1D の場合、フローマッチ処理は行われずに STP として処理される。

STP 処理後、以下の状態遷移にてヘッダフィールド解析処理が実施される。

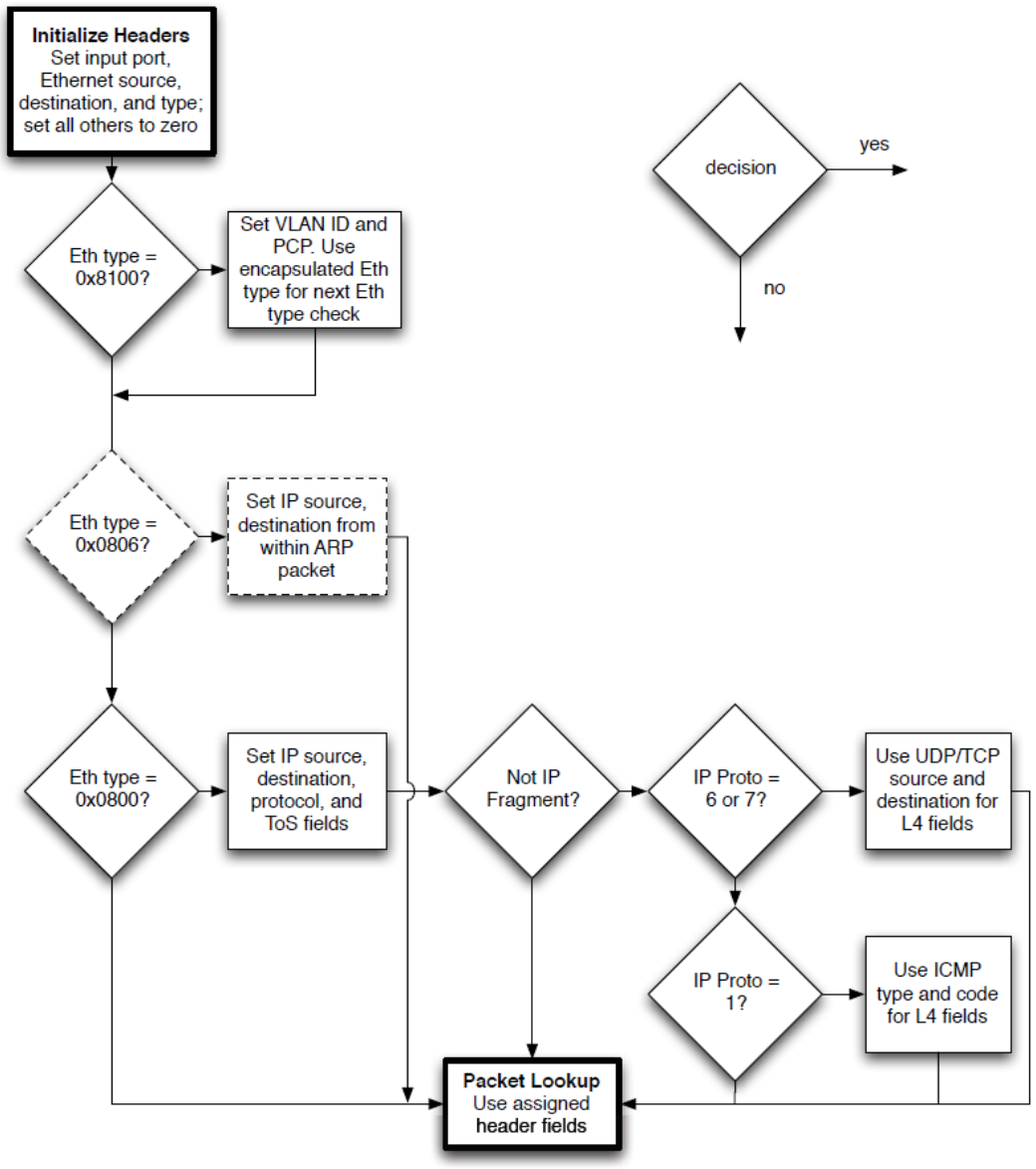

図 3:ヘッダフィールド解析状態遷移図 …OpenFlow Switch Specification 1.0.0 P8 の Figure3 引用

OpenFlow ver.1.0 で規定するヘッダフィールド解析は以下の処理を行う。

1. Eth Type=0x8100(VLAN)である場合、VLAN_ID と PCP を探索に利用する。
2. Eth Type=0x0806(ARP)である場合、Source IP アドレスと Destination IP アドレスを探索対象に追加する。処理は Optional である。
3. Eth Type=0x0800(IPv4)である場合、IPv4 ヘッダを探索対象に追加する。
4. IPv4 パケットかつ、TCP または UDP の場合、トランスポート層のポート番号を探索対象に追加する。

5. IPv4 パケットかつ、ICMP の場合、ICMP タイプと ICMP コードを検索対象に追加する。
6. IPv4 パケットかつ、フラグメントオフセットが 0 以外、または More Fragments ビットが設定されている場合、トランスポート層のポート番号は 0 番に設定される。

また、フローテーブルフィールドはANY値を保持する場合、ワイルドカード扱いとなり、すべての値が Match する。

Ethernet Ⅱフレーム、802.3 フレーム(SNAP ヘッダ付き、OUI 0x000000)、および 802.2 フレーム(SNAP ヘッダなし、OUI x000000 以外)は異なる方法で処理する。

パケット探索の優先度順位はワイルドカードを保持しない完全に一致するエントリが常に最優先である。ワイルドカードを保持する場合、優先度を番号で判断し、数値が高いほど優先される。

フローエントリが Match した場合、対応したカウンタが更新される。Match しない場合、セキュアチャンネル経由で OFC にパケットを送信する。

2.4.3. Secure Channel

セキュアチャンネルは OFC と OFS を接続するためのインターフェースである。このインターフェースを使用して、OFC は OFS の設定と管理、イベント受信、OFS に対してパケットを送信する。

Datapath とセキュアチャンネル間のインターフェースは実装依存だが、セキュアチャンネルのメッセージは OpenFlow プロトコルのフォーマットに従う必要がある。

2.4.3.1. OpenFlow プロトコル概要

2.4.3.1.1. Controller-to-Switch

OFC で生成され、OFS の状態確認、管理に使用する。

- Features:TLSセッション確立時にOFCはOFSのスイッチ情報を取得する。
- Configurations：OFC は OFS の設定変更、設定確認をする。
- Modify-State：OFS のフローテーブル追加・修正・削除を設定する。
- Read-State：OFS の統計情報を取得する。
- Send-Packet：OFS の指定されたポートからパケットを送信する。
- Barrier：処理が完了した場合、通知される。

2.4.3.1.2. Asynchronous

OFS で生成され、OFC へ状態変化やエラー通知などを送信する。代表する例を以下に記載する。

- Packet-in：フローテーブルに Match しないパケットを受信。
- Flow-Remove：タイマによりフロー消滅。
- Port-status：ポート設定状態が変更。
- Error：エラー発生。

2.4.3.1.3. Symmetric

OFC、OFS の双方で生成され、必要に応じて送信される。

- Hello：接続開始時に OFC・OFC 間に交換する。
- Echo：受信したら必ず返信する。遅延、帯域測定、および死活監視に使用。
- Vendor：将来の Version 更新のための予約領域。

2.4.3.2. Connection Setup

OFS は IP アドレスとポート番号を指定して通信を確立する。セキュアチャンネル上のトラヒックはフローテーブルの照合を行わない。

OFC と OFS の最初の通信確立時に OFPT_HELLO メッセージを送信する。このメッセージを交換するとサポートする最新のバージョン情報がわかる。

ネゴシエーションが可能な場合、OFPT_HELLO を送信する。不可能な場合、タイプを OFEPT_HELLO_FAILED、コードを OFPHFC_COMPATIBLE、そして Optionally として ASCII コードで説明を記載した OFPT_ERROR メッセージを送信する。

2.4.3.3. Connection Interruption

OFS からの echo 応答タイムアウト、TLS セッションタイムアウトなど切断した場合、バックアップコントローラへ接続する。

OFC との接続に失敗した場合、OFS は「emergency mode」に遷移し、TCP コネクションを切断する。「emergency mode」に遷移した OFS は通常のエントリがすべて削除される。

OFC との接続が回復しても非常用フローテーブルは維持する。

2.4.3.4. Encryption

OFC と OFS は TLS コネクションを使用する。TCP 6633 番をデフォルトし、OFS から OFC に接続開始する。OFC は OFS の証明書を所持し、OFS は OFC の証明書を所持する。

2.4.3.5. Spanning Tree

OFS は optionally として 802.1D スパニングツリー(以下 STP)に対応する。STP に対応する場合、フローテーブルの探索前に STP に対応する処理をする。OFS は OFPT_FEATURES_REPLY の OFPC_STP のビットを設定する必要がある。OFS の STP を有効にする場合、物理ポートはすべて STP を有効にしなければならいが仮想ポートは有効にする必要はない。

2.4.3.6. Flow Table Modification Messages

フローテーブルの変更メッセージは以下の 5 種類ある。

- OFPFC_ADD
- OFPFC_MODIFY
- OFPFC_MODIFY_STRICT
- OFPFC_DELETE
- OFPFC_DELETE_STRICT

OFPFF_CHECK_OVERLAP フラグが設定された ADD リクエストの場合、OFS は重複するフローエントリを確認する必要がある。もし予盾するフローエントリが存在した場合、OFPET_FLOW_MOD_FAILED と OFPFMFC_OVERLAP を設定した ofp_error_msg を送信する。

同一なヘッダフィールドが OFS に存在する場合、新たなフローを優先し、カウンタ情報を初期化する。

フローエントリを追加するスペースがない場合、OFPET_FLOW_MOD_FAILED と OFPFMFC_ALL_TABLES_FULL を設定した ofp_error_msg を送信する。

ポートが無効な場合、OFPET_BAD_ACTION と OFPBAC_BAD_PUT_PORT を設定した ofp_error_msg を送信する。

Modify リクエストの対象フローが存在しない場合、ADD リクエストと同じ動作をし、カウンタ情報を初期化する。対象フローが存在する場合、対象フローのアクションは変更されるが、カウンタ情報は更新されない。

Delete リクエストの対象フローエントリが存在しない場合、エラーが記録されず、フローテーブルに変化は起きない。フローエントリが一致し、通常のフローエントリを削除する場合、OFPFF_SEND_FLOW_REM を設定した、フロー削除メッセージを作成すべきである。非常用フローテーブルは削除されてもフロー削除メッセージを作成する必要はない。

Modify と Delete リクエストには STRICT が存在する。STRICT 処理の場合、ワイルドカードを含め、すべてが一致するフローのみ処理される。

非常用フローテーブルのタイマ値は”0”に設定する必要がある。失敗した場合、OFPET_FLOW_MOD_FAILED と OFPFMFC_BAD_EMERG_TIMEOUT を設定した ofp_error_msg を送信する。

処理できない場合、OFPET_FLOW_MOD_FAILED と OFPFMFC_UNSUPPORTED を設定した ofp_error_msg を即座に送信する必要がある。

2.4.3.7. Flow Removal

各フローエントリは idle_timeout と hard_timeout を持っている。指定された秒数が経過するとフローは削除される。フローを削除した場合、OFC にフローを削除したメッセージを送信する。

2.4.4. Appendix A: The OpenFlow Protocol

OpenFlow プロトコルの構造体は include /openflow/openflow.h にて定義されている。8byte 単位で padding する。OpenFlow メッセージはビッグエンディアンで送信される。

2.4.4.1. OpenFlow Header

OpenFlow メッセージは以下の OpenFlow ヘッダより開始される。

型	変数名	内容
uint8_t	version	OpenFlow プロトコルバージョン
uint8_t	type	メッセージタイプ
uint16_t	length	ofp_header を含むパケット長
uint32_t	xid	トランザクション ID

表 7:ofp_header 構造体

OpenFlow ver.1.0 では version は”0x01”となる。OpenFlow の最終バージョンは 0x00 になる予定である。メッセージタイプには以下の列挙体になる。

値	定数名	メッセージタイプ
0	OFPT_HELLO	Symmetric
1	OFPT_ERROR	
2	OFPT_ECHO_REQUEST	
3	OFPT_ECHO_REPLY	
4	OFPT_VENDOR	
5	OFPT_OFPT_FEATURES_REQUEST	Switch configuration
6	OFPT_OFPT_FEATURES_REPLY	
7	OFPT_GET_CONFIG_REQUEST	
8	OFPT_GET_CONFIG_EWPLY	
9	OFPT_SET_CONFIG	
10	OFPT_PACKET_IN	Asynchronous
11	OFPT_FLOW_REMOVED	
12	OFPT_PORT_STATUS	
13	OFPT_PACKET_OUT	Controller command
14	OFPT_FLOW_MOD	
15	OFPT_PORT_MOD	
16	OFPT_STATS_REQUEST	Statistics
17	OFPT_STATS_REPLY	
18	OFPT_BARRIER_REQUEST	Barrier
19	OFPT_BARRIER_REPLY	
20	OFPT_QUEUE_GET_CONFIG_REQUEST	Queue Configuration
21	OFPT_QUEUE_GET_CONFIG_REPLY	

表 8:ofp_type 列挙体

2.4.4.2. Common Structures

複数のメッセージで使用されている構造体について解説する。

2.4.4.2.1. Port Structures

物理ポートに関する構造体は以下になる。

型	変数名	内容
uint16_t	port_no	ポート番号
uint8_t	hw_addr[OFP_ETH_ALEN]	MAC アドレス
char	name[OFP_MAX_PORT_NAME_LEN]	インターフェース名
uint32_t	config	STP や管理に関する設定
uint32_t	state	STP や物理リンクの存在に関する状態
uint32_t	curr	現在のポート設定値
uint32_t	advertised	広報する機能
uint32_t	supported	サポートしている機能
uint32_t	peer	接続相手が広報している機能

表 9:ofp_phy_port 構造体

OFP_MAX_ETH_ALEN の値は 6、OFP_MAX_PORT_NAME_LEN の値は 16 である。

config 値は以下の列挙体になる。

値	定数名	内容
1<<0	OFPPC_PORT_DOWN	ポートダウン
1<<1	OFPPC_NO_STP	802.1D スパニングツリー無効
1<<2	OFPPC_NO_RECV	802.1D スパニングツリー以外の全パケットを DROP
1<<3	OFPPC_NO_RECB_STP	802.1D スパニングツリーパケットを DROP
1<<4	OFPPC_NO_FLOOD	フラッティングから除外
1<<5	OFPPC_NO_FWD	転送されたパケットを DROP
1<<6	OFPPC_NO_PACKET_IN	Packet-In メッセージを送信しない

表 10:ofp_port_config 列挙体

state 値は以下の列挙体になる。なお state は読み取り専用であり、OFC は設定することはできない。

値	定数名	内容
1<<0	OFPPS_LINK_DOWN	物理リンクが存在しない
0<<8	OFPPS_STP_LISTEN	スパニングツリーパケットを転送しない
1<<8	OFPPS_STP_LEARN	スパニングツリーパケットを学習する
2<<3	OFPPS_STP_FORWARD	スパニングツリーパケットを転送する
3<<4	OFPPS_STP_BLOCK	スパニングツリーの一部とならない
3<<5	OFPPS_STP_MASK	OFPPS_STP_*のビットマスク

表 11: ofp_port_state 列挙体

port_no 値は以下の番号が予約されている。

値	定数名	内容
0xff00	OFPP_MAX	物理ポートの最大値
0xfff8	OFPP_IN_PORT	受信ポート
0xfff9	OFPP_TABLE	フローテーブルに登録されている Action
0xfffa	OFPP_NORMAL	既存の L2/VLAN/L3 に沿ったポート
0xfffb	OFPP_FLOOD	受信ポート以外の全てのポート(スパニングツリー)
0xfffc	OFPP_ALL	受信ポート以外の全てのポート
0xfffd	OFPP_CONTROLLER	OFC 接続ポート
0xfffd	OFPP_LOCAL	ローカルスイッチの接続ポート
0xffff	OFPP_NONE	物理ポート以外

表 12: ofp_port 列挙体

curr、advertised、supported、peer の各値にはリンクモード、リンクタイプ、リンク特徴が設定値になる。なお、設定値は以下の列挙体になる。

値	定数名	内容
1<<0	OFPPF_10MB_HD	10MB 半二重
1<<1	OFPPF_10MB_FD	10MB 全二重
1<<2	OFPPF_100MB_HD	100MB 半二重
1<<3	OFPPF_100MB_FD	100MB 全二重
1<<4	OFPPF_1GB_HD	1GB 半二重
1<<5	OFPPF_1GB_FD	1GB 全二重
1<<6	OFPPF_10GB_FD	10GB 全二重
1<<7	OFPPF_COPPER	ツイストペアケーブル
1<<8	OFPPF_FIBER	ファイバーケーブル
1<<9	OFPPF_AUTONEG	オートネゴシエーション
1<<10	OFPPF_PAUSE	ポーズ機能
1<<11	OFPPF_PAUSE_ASYM	Asymmetric ポーズ機能

表 13: ofp_port_features 列挙体

2.4.4.2.2. Queue Structures

OpenFlow ver1.0 では制限付きで OoS に対応している。複数の Queue を各ポートに割り当てることができる。Queue の設定に基づいてパケットを処理する。Queue の構造体は以下になる。

型	変数名	内容
uint32_t	queue_id	Queue 番号
uint16_t	len	Queue バイト長
uint8_t	pad[2]	64bit alignment
struct ofp_queue_prop_header	properties[0]	Queue プロパティリスト

表 14: ofp_packet_queue 構造体

Queue プロパティには以下の列挙体が設定される。

値	定数名	内容
0	OFPQT_NONE	プロパティなし
1	OFPQT_MIN_RATE	最小データレート保証

表 15: ofp_queue_properties 列挙体

Queue ヘッダ構造体は以下になる。

型	変数名	内容
uint16_t	queue_id	Queue プロパティ(OFPQT_*)
uint16_t	len	ヘッダを含む Queue プロパティ長
uint8_t	pad[4]	64bit alignment

表 16: ofp_queue_prop_header 構造体

OpenFlow ver.1.0 では現在、ofp_queue_prop_min_rate による最小データレート保証のみ対応する。

型	変数名	内容
struct ofp_queue_prop_header	prop_header	Queue ヘッダ
uint16_t	rate	データレート
uint8_t	pad[6]	64bit alignment

表 17: ofp_queue_prop_min_rate 構造体

2.4.4.2.3. Flow Match Structures

フローエントリの際、以下の構造体が使用される。

型	変数名	内容
uint32_t	wildcards	ワイルドカード
uint16_t	in_port	受信ポート
uint8_t	dl_src[OFP_ETH_ALEN]	送信元 Ethernet アドレス
uint8_t	dl_dst[OFP_ETH_ALEN]	宛先 Ethernet アドレス
uint16_t	dl_vlan	VLANID
uint8_t	dl_vlan_pcp	VLAN プライオリティ
uint8_t	pad1[1]	64bit alignment
uint16_t	dl_type	Ethernet フレームタイプ
uint8_t	nw_tos	ToS フィールド(DSCP の 6bit のみ)
uint8_t	nw_proto	IP プロトコル
uint8_t	pad2[2]	64bit alignment
uint32_t	nw_src	送信元 IP アドレス
uint32_t	nw_dst	宛先 IP アドレス
uint16_t	tp_src	送信元 TCP/UDP ポート番号
uint16_t	tp_dst	宛先 TCP/UDP ポート番号

表 18: ofp_match 構造体

ワイルドカードには以下の列挙体の Flag が設定される。

値	定数名	内容
1<<0	OFPFW_IN_PORT	受信ポート
1<<1	OFPFW_DL_VLAN	VLANID
1<<2	OFPFW_DL_SRC	送信元 Ethernet アドレス
1<<3	OFPFW_DL_DST	宛先 Ethernet アドレス
1<<4	OFPFW_DL_TYPE	Ethernet フレームタイプ
1<<5	OFPFW_NW_PROTO	IP プロトコル
1<<6	OFPFW_TP_SRC	送信元 TCP/UDP ポート番号
1<<7	OFPFW_TP_DST	宛先 TCP/UDP ポート番号
8	OFPFW_NW_SRC_SHIFT	送信元 IP アドレスシフト
6	OFPFW_NW_SRC_BITS	送信元 IP アドレスビット
((1 << OFPFW_NW_SRC_BITS) - 1) << OFPFW_NW_SRC_SHIFT	OFPFW_NW_SRC_MASK	送信元 IP アドレスマスク
32 << OFPFW_NW_SRC_SHIFT	OFPFW_NW_SRC_ALL	送信元 IP アドレスビットオール
14	OFPFW_NW_DST_SHIFT	宛先 IP アドレスシフト
6	OFPFW_NW_DST_BITS	宛先 IP アドレスビット
((1 << OFPFW_NW_DST_BITS) - 1) << OFPFW_NW_DST_SHIFT	OFPFW_NW_DST_MASK	宛先 IP アドレスマスク
32 << OFPFW_NW_DST_SHIFT,	OFPFW_NW_DST_ALL	宛先 IP アドレスビットオール
1<<20	OFPFW_DL_VLAN_PCP	VLAN プライオリティ
1<<21	OFPFW_NW_TOS	ToS フィールド
((1<<22) - 1)	OFPFW_ALL	全てのフィールド

表 19: ofp_flow_wildcards 列挙体

2.4.4.2.4. Flow Action Structures

多くの Action はフローかパケットに関連付けることができる。現在定義されている Action タイプは以下の列挙体になる。

値	定数名	内容
0	OFPAT_OUTPUT	指定ポートから送信
1	OFPAT_SET_VLAN_VID	VLANID を設定
2	OFPAT_SET_VLAN_PCP	VLAN プライオリティ設定
3	OFPAT_STRIP_VLAN	VLAN タグを除去
4	OFPAT_SET_DL_SRC	送信元 Ethernet アドレス設定
5	OFPAT_SET_DL_DST	宛先 Ethernet アドレス設定
6	OFPAT_SET_NW_SRC	送信元 IP アドレス設定
7	OFPAT_SET_NW_DST	宛先 IP アドレス設定
8	OFPAT_SET_NW_TOS	ToS フィールド設定
9	OFPAT_SET_TP_SRC	送信元 TCP/UDP ポート番号設定
10	OFPAT_SET_TP_DST	宛先 TCP/UDP ポート番号設定
11	OFPAT_ENQUEUE	Queue に格納
0xffff	OFPAT_VENDOR	ベンダ独自アクション

表 20: ofp_action_type 列挙体

Action ヘッダにはタイプ、長さなどの関連情報が含まれている。

型	変数名	内容
uint16_t	type	Action タイプ
uint16_t	len	Action 長
uint8_t	pad[4]	64bit alignment

表 21: ofp_action_header 構造体

Action_output の構造体は以下になる。

型	変数名	内容
uint16_t	type	OFPAT_OUTPUT
uint16_t	len	8
uint16_t	port	出力ポート
uint16_t	maxlen	OFC に送信する最大バイト長

表 22: ofp_action_output 構造体

Enqueue アクションは ToS や VLAN PCP の設定に関わらず Queue に割り当てる。パケットは Enqueue アクション後、変更してはならない。ToS や VLAN PCP の設定変更が必

要な場合、Enqueue アクション実行前に変更する必要がある。

Enqueue action の構造体は以下になる。

型	変数名	内容
uint16_t	type	OFPAT_ENQUEUE
uint16_t	len	16
uint16_t	port	出力ポート
uint8_t	pad[6]	64bit alignment
uint32_t	queue_id	OFC に送信する最大バイト長

表 23: ofp_action_enqueue 構造体

Action_vlan_vid の構造体は以下になる。
なお vlan_vid=0xffff は VLANID が使用されていないことを示す。

型	変数名	内容
uint16_t	type	OFPAT_SET_VLAN_VID
uint16_t	len	8
uint16_t	vlan_vid	VLANID
uint8_t	pad[2]	64bit alignment

表 24: ofp_action_vlan_vid 構造体

Action_vlan_pcp の構造体は以下になる。

型	変数名	内容
uint16_t	type	OFPAT_SET_VLAN_PCP
uint16_t	len	8
uint8_t	vlan_vid	VLAN プライオリティ
uint8_t	pad[3]	64bit alignment

表 25: ofp_action_vlan_pcp 構造体

Action_strip_vlan は引数がなく、ofp_action_header のみで構成されている。1 が存在する場合、VLAN タグを除去する。

Action_dl_src の構造体は以下になる。

型	変数名	内容
uint16_t	type	OFPAT_SET_DL_SRC/DST
uint16_t	len	16
uint8_t	dl_addr[OFP_ETH_ALEN]	Ethernet アドレス
uint8_t	pad[6	64bit alignment

表 26: ofp_action_dl_addr 構造体

Action_nw_addr の構造体は以下になる。

型	変数名	内容
uint16_t	type	OFPAT_SET_NW_SRC/DST
uint16_t	len	8
uint8_t	nw_addr	IP アドレス

表 27: ofp_action_nw_addr 構造体

Action_nw_tos の構造体は以下になる。

型	変数名	内容
uint16_t	type	OFPAT_SET_NW_TOS
uint16_t	len	8
uint8_t	nw_tos	ToS フィールド(DSCP の 6bit のみ)
uint8_t	pad[3]	64bit alignment

表 28: ofp_action_nw_tos 構造体

Action_tp_port の構造体は以下になる。

型	変数名	内容
uint16_t	type	OFPAT_SET_TP_SRC/DST
uint16_t	len	8
uint8_t	tp_port	TCP/UDP ポート番号
uint8_t	pad[3]	64bit alignment

表 29: ofp_action_tp_port 構造体

Action_vender の構造体は以下になる。

型	変数名	内容
uint16_t	type	OFPAT_VENDOR
uint16_t	len	8
uint32_t	vender	ベンダ ID

表 30: ofp_action_vendor_header 構造体

2.4.4.3. Controller-to-Switch Messages

2.4.4.3.1. Handshake

TLS が確立されると OFC は OFPT_FEATURES_REQUEST を送信する。OFS は OFPT_FEATURES_REPLY メッセージを返信する。

ofp_switch_features の構造体は以下になる。

型	変数名	内容
struct ofp_header	header	OpenFlow プロトコルヘッダ
uint64_t	datapath_id	データパス ID
uint32_t	n_buffers	バッファ領域に格納できるパケット数
uint8_t	n_tables	テーブルサポート数
uint8_t	pad[3]	64bit alignment
uint32_t	capabilities	サポートする ofp_capabilities
uint32_t	actions	サポートする ofp_action_type
struct ofp_phy_port	ports[0]	物理ポート情報

表 31: ofp_switch_features 構造体

データパス ID は OFS を識別するためのユニークな ID が設定される。上位 16bit は実装者に委ねられるが、下位 48bit は OFS の MAC アドレスが格納される。上位 16bit の使用例としてインスタンスを区別するために VLANID が格納されることがある。

サポートする ofp_capabilities の列挙体は以下になる。

値	定数名	内容
1<<0	OFPC_FLOW_STATS	フロー統計情報
1<<1	OFPC_TABLE_STATS	テーブル統計情報
1<<2	OFPC_PORT_STATS	ポート統計情報
1<<3	OFPC_STP	802.1d スパニングツリー
1<<4	OFPC_RESERVED	予約値(必ず 0)
1<<5	OFPC_IP_REASM	フラクメントされた IP のリアセンブル
1<<6	OFPC_QUEUE_STATS	Queue 統計情報
1<<7	OFPC_ARP_MATCH_IP	ARP パケットの IP アドレスマッチング

表 32: ofp_capabilities 列挙体

acitons には OFS がサポートするアクションのビットマップが格納される。ports[0]には物理ポートの詳細情報が格納される。

2.4.4.3.2. Switch Configuration

OFC は OFS に OFPT_SET_CONFIG を送信し、コンフィグを設定する。また、OFC は OFPT_GET_CONFIG_REQUEST を送信すると、OFS から OFPT_GET_CONFIG_REPLY が返信されコンフィグ設定を確認することができる。

ofp_switch_config 構造体は以下になる。

型	変数名	内容
struct ofp_header	header	OpenFlow プロトコルヘッダ
uint16_t	flags	ofp_config_flags
uint16_t	miss_send_len	OFC に送信するパケットの最大バイト数

表 33: ofp_switch_config 構造体

ofp_config_flags の列挙体は以下になる。

値	定数名	内容
0	OFPC_FRAG_NORMAL	なにもしない
1	OFPC_FRAG_DROP	ドロップする
2	OFPC_FRAG_REASM	リアセンブル
3	OFPC_FRAG_MASK	-

表 34: ofp_config_flags 列挙体

2.4.4.3.3. Modify State Messages

OFC が OFPT_FLOW_MOD を送信すると OFS のフローテーブルを変更する。

ofp_flow_mod の構造体は以下になる。

型	変数名	内容
struct ofp_header	header	OpenFlow プロトコルヘッダ
struct ofp_match	match	Match 条件
uint64_t	cookie	OFC が発行した識別子
uint16_t	command	ofp_flow_mod_command
uint16_t	idle_timeout	削除までのアイドル時間
uint16_t	hard_timeout	削除までの最大時間
uint16_t	priority	プライオリティ
uint32_t	buffer_id	バッファリングパケット
uint16_t	out_port	出力ポート
uint16_t	flags	ofp_flow_mod_flags
struct ofp_action_header	actions[0]	フローエントリの action

表 35: ofp_flow_mod 構造体

cookie の値は-1(0xffffffffffffffff)は事前に予約されている。

ofp_flow_mod_command の列挙体は以下になる。

値	定数名	内容
0	OFPFC_ADD	フロー追加
1	OFPFC_MODIFY	マッチした全フローの変更
2	OFPFC_MODIFY_STRICT	ワイルドカードに厳密にマッチしたフローの変更
3	OFPFC_DELETE	マッチした全フローの削除
4	OFPFC_DELETE_STRICT	ワイルドカードに厳密にマッチしたフローの削除

表 36: ofp_flow_mod_command 列挙体

idle_timeout と hard_timeout の値が両方”0”の場合、タイムアウトでフローが削除されることはない。

プライオリティ値が高いほど優先される。

ofp_flow_mod_flags の列挙体は以下になる。

値	定数名	内容
1<<0	OFPFF_SEND_FLOW_REM	フローが削除された場合、Flow Removed メッセージを送信する
1<<1	OFPFF_CHECK_OVERLAP	フローがオーバーラップした場合、error を OFC に送信する
1<<2	OFPFF_EMERG	フローを緊急エントリとして設定する

表 37: ofp_flow_mod_flags 列挙体

OFC が OFPT_PORT_MOD を送信すると OFS のポート設定を変更する。
ofp_port_mod の構造体は以下になる。

型	変数名	内容
struct ofp_header	header	OpenFlow プロトコルヘッダ
uint16_t	port_no	物理ポート番号
uint8_t	hw_addr[OFP_ETH_ALEN]	ポートの MAC アドレス
uint32_t	config	ポート詳情報
uint32_t	mask	変更対象のポート詳細情報
uint32_t	advertise	ofp_port_features
uint8_t	pad[4]	64bit alignment

表 38: ofp_port_mod 構造体

2.4.4.3.4. Queue Configuration Messages

Queue コンフィグは OpenFlow プロトコルでは行わない。コマンドラインツールや別のコンフィグプロトコル経由で実行される。

OFC は OFS の各ポートに設定された Queue を参照することができる。

型	変数名	内容
struct ofp_header	header	OpenFlow プロトコルヘッダ
uint16_t	port_no	物理ポート番号
uint8_t	pad[2]	64bit alignment

表 39: ofp_queue_get_config_request 構造体

OFS は ofp_queue_get_config_reply 構造体を OFC に返信する。

型	変数名	内容
struct ofp_header	header	OpenFlow プロトコルヘッダ
uint16_t	port_no	物理ポート番号
uint8_t	pad[6]	64bit alignment
struct ofp_packet_queue	queues[0]	Queue リスト

表 40: ofp_queue_get_config_reply 構造体

2.4.4.3.5. Read State Messages

OFC は OFPT_STATS_REQUEST メッセージを使用し、OFS の現在状況を照会する。

型	変数名	内容
struct ofp_header	header	OpenFlow プロトコルヘッダ
uint16_t	type	ofp_stats_types
uint16_t	flags	フラグ(未定義)
uint8_t	body[0]	リクエストの body 部

表 41: ofp_stats_request 構造体

OFS は OFPT_STATS_REPLY メッセージを使用し、OFC に返信する。

型	変数名	内容
struct ofp_header	header	OpenFlow プロトコルヘッダ
uint16_t	type	ofp_stats_types
uint16_t	flags	フラグ(未定義)
uint8_t	body[0]	リプライの body 部

表 42: ofp_stats_reply 構造体

ofp_stats_types の列挙体は以下になる。

値	定数名	内容
0	OFPST_DESC	OFS の情報
1	OFPST_FLOW	各フローの統計情報
2	OFPST_AGGREGATE	全フローの統計情報
3	OFPST_TABLE	テーブル統計情報
4	OFPST_PORT	ポート統計情報
5	OFPST_QUEUE	Queue 統計情報
0xffff	OFPST_VENDER	ベンダ拡張

表 43: ofp_flow_mod_command 列挙体

OFPST_DESC をリクエストすると、以下の構造体がリプライの body に格納される。

型	変数名	内容
char	mfr_desc[DSEC_STR_LEN]	製造元説明
char	hw_desc[DSEC_STR_LEN]	ハードウェア説明
char	sw_desc[DSEC_STR_LEN]	ソフトウェア説明
char	serial_num[SERIAL_NUM_LEN]	シリアルナンバー
char	dp_desc[DESC_STR_LEN]	ユーザ向け datapath

表 44: ofp_desc_stats 構造体

各エントリは ASCII 形式かつ右側に NULL が埋め込まれる。
DSEC_STR_LEN は”256”、SERIAL_NUM_LEN は”32”である。

OFPST_FLOW をリクエストする際、以下の構造体が body に格納される。

型	変数名	内容
struct ofp_match	match	match 条件
uint8_t	table_id	テーブル ID
uint8_t	pad	32bit alignment
uint16_t	out_port	出力ポート

表 45: ofp_flow_stats_request 構造体

tabile_id は単一のテーブル ID か全てのテーブル(0xff)を設定する。
out_port は全てのポート情報を取得したい際は OFPP_NONE を設定する。

OFPST_FLOW のリプライは以下の構造体が body に格納される。

型	変数名	内容
uint16_t	length	データ長
uint8_t	table_id	テーブル ID
uint8_t	pad	32bit alignment
struct ofp_match	match	match 条件
uint32_t	duration_sec	フローが生成されてからの経過時間
uint32_t	duration_nsec	フローが生成されてからの経過時間(ナノ秒)
uint16_t	priority	プライオリティ
uint16_t	idle_timeout	削除までのアイドル時間
uint16_t	hard_timeout	削除までの最大時間
uint8_t	pad2[6]	64bit alignment
uint64_t	cookie	OFC が発行した識別子
uint64_t	packet_count	フローエントリが処理した総パケット数
uint64_t	byte_count	フローエントリが処理したパケットの総バイト数
struct ofp_action_header	actions[0]	フローエントリの action

表 46: ofp_flow_stats 構造体

OFPST_AGGREGATE をリクエストする際、以下の構造体が body に格納される。

型	変数名	内容
struct ofp_match	match	match 条件
uint8_t	table_id	テーブル ID
uint8_t	pad	32bit alignment
uint16_t	out_port	出力ポート

表 47: ofp_aggregate_stats_request 構造体

OFPST_ AGGREGATE のリプライは以下の構造体が body に格納される。

型	変数名	内容
uint64_t	packet_count	条件にマッチするフローエントリが処理した総パケット数
uint64_t	byte_count	条件にマッチするフローエントリが処理したパケットの総バイト数
uint32_t	flow_count	条件にマッチするフローエントリ数
uint8_t	pad[4]	64bit alignment

表 48: ofp_aggregate_stats_reply 構造体

OFPST_TABLE をリクエストする際、以下の構造体を body に格納する。

型	変数名	内容
uint8_t	table_id	テーブル ID
uint8_t	pad[3]	32bit alignment
char	name[OFP_MAX_TABLE_NAME_LEN]	テーブル名 t
uint32_t	wildcards	ワイルドカード
uint32_t	max_entries	フローエントリ最大数
uint32_t	active_count	格納されているフローエントリ数
uint64_t	lookup_count	マッチング処理を実施したパケット数
uint64_t	matched_count	マッチングしたパケット数

表 49: ofp_table_stats 構造体

OFP_MAX_TABLE_NAME_LEN は”32”である。

OFPST_PORT をリクエストする際、以下の構造体を body に格納する。

型	変数名	内容
uint16_t	port_no	物理ポート番号
uint8_t	pad[6]	64bit alignment

表 50: ofp_port_stats_request 構造体

全てのポート情報を取得する際、port_no は OFPP_NONE を設定する。

OFPST_ PORT のリプライは以下の構造体が body に格納される。

型	変数名	内容
uint16_t	port_no	物理ポート番号
uint8_t	pad[6]	64bit alignment
uint64_t	rx_packets	受信パケット数
uint64_t	tx_packets	送信パケット数
uint64_t	rx_bytes	受信パケットのバイト数
uint64_t	tx_bytes	送信パケットのバイト数
uint64_t	rx_dropped	受信パケットのドロップ数
uint64_t	tx_dropped	送信パケットのドロップ数
uint64_t	rx_errors	受信エラーパケット数
uint64_t	tx_errors	送信エラーパケット数
uint64_t	rx_frame_err	受信パケットのフレーム割り当てエラー回数
uint64_t	rx_over_err	受信パケットのオーバーラン発生回数
uint64_t	rx_cre_err	受信パケットの CRC エラー回数
uint64_t	collisions	64bit alignment

表 51: ofp_port_stats 構造体

OFPST_QUEUE をリクエストする際、以下の構造体が body に格納される。

型	変数名	内容
uint16_t	port_no	物理ポート番号
uint8_t	pad[2]	32bit alignment
uint32_t	queue_id	Queue 番号

表 52: ofp_queue_stats_request 構造体

port_no に OFPP_ALL、queue_id に OFPQ_ALL が設定されている場合、全ポートの全 Queue を参照する。

OFPST_ QUEUE のリプライは以下の構造体が body に格納される。

型	変数名	内容
uint16_t	port_no	物理ポート番号
uint8_t	pad[2]	32bit alignment
uint32_t	queue_id	Queue 番号
uint64_t	tx_bytes	Queue から送信したパケットのバイト数
uint64_t	tx_packets	Queue から送信したパケット数
uint64_t	tx_errors	Queue の送信エラー数

表 53: ofp_queue_stats 構造体

OFPST_VENDOR はベンダ独自の統計情報を取得する。最初の 4 バイトはベンダ識別子である。残りはベンダ定義になる。

2.4.4.3.6. Send Packet Message

OFC は datapath 経由でパケットを送信する場合、OFPT_PACKET_OUT メッセージを使用する。

ofp_packet_out の構造体は以下になる。

型	変数名	内容
struct ofp_header	header	OpenFlow プロトコルヘッダ
uint32_t	buffer_id	バッファ ID
uint16_t	in_port	受信パケット
uint16_t	actions_len	Action の長さ
struct ofp_action_header	actions[0]	Action

表 54: ofp_packet_out 構造体

buffer_id は OFS の ofp_packet_in のバッファ ID と同一になる。buffer_id が”-1”の場合、OFS 上のバッファ領域を使用しない。

2.4.4.3.7. Barrier Message

OFC はメッセージ依存性を確認したい場合や完了したオペレーションについてリプライを受けたい場合に OFPT_BARRIER_REQUEST を送信する。

OFPT_BARRIER_REQUEST を受信した OFS は当該のリクエストに対応する動作をすべて中止し、OFPT_BARRIER_REPLY メッセージを返信する。

2.4.4.4. Asynchronous Messages

2.4.4.4.1. Packet-In Message

OFS でパケット受信後、OFC へ転送する場合、OFPT_PACKET_IN メッセージを使用する。

ofp_packet_in 構造体は以下になる。

型	変数名	内容
struct ofp_header	header	OpenFlow プロトコルヘッダ
uint32_t	buffer_id	バッファ ID
uint16_t	total_len	フレーム最大長
uint16_t	in_port	受信ポート
uint8_t	reason	ofp_packet_in_reason
uint8_t	pad	alignment
uint8_t	data[0]	パケットデータ

表 55: ofp_packet_in 構造体

OFC へ送信する Action の場合、max_len バイト送信される。

テーブルミスフローの場合、OFPT_SET_CONFIG メッセージから miss_send_len バイトのメッセージが送信される。miss_send_len のデフォルト値は”128”である。

ofp_packet_in_reason の列挙体は以下になる。

値	定数名	内容
0	OFPST_DESC	OFS の情報
1	OFPST_FLOW	各フローの統計情報

表 56: ofp_packet_in_reason 列挙体

2.4.4.4.2. Flow Removed Message

フローエントリが削除された場合、OFC は OFPT_FLOW_REMOVED メッセージを受信する。

ofp_flow_removed 構造体は以下になる。

型	変数名	内容
struct ofp_header	header	OpenFlow プロトコルヘッダ
struct ofp_match	match	match 条件
uint64_t	cookie	OFC が発行した識別子
uint16_t	priority	プライオリティ
uint8_t	reason	ofp_flow_removed_reason
uint8_t	pad[1]	32bit alignment
uint32_t	duration_sec	生成されてからの経過時間
uint32_t	duration_nsec	生成されてからの経過時間(ナノ秒)
uint16_t	idle_timeout	削除までのアイドル時間
uint8_t	pad2[2]	64bit alignment
uint64_t	packet_count	処理した総パケット数
uint64_t	byte_count	処理した総バイト数

表 57: ofp_flow_removed 構造体

ofp_flow_removed_reason の列挙体は以下になる。

値	定数名	内容
0	OFPRR_IDLE_TIMEOUT	idle_timeout 時間超過
1	OFPRR_HARD_TIMEOUT	hard_timeout 時間超過
2	OFPRR_DELETE	FlowMod メッセージにより削除

表 58: ofp_flow_removed_reason 列挙体

2.4.4.4.3. Port Status Message

OFS は物理ポートが追加、変更、削除された場、OFC へ OFPT_PORT_STATUS メッセージを送信する

ofp_port_status 構造体は以下になる。

型	変数名	内容
struct ofp_header	header	OpenFlow プロトコルヘッダ
uint8_t	reason	ofp_port_reason
uint8_t	pad[7]	64bit alignment
struct ofp_phy_port	desc	物理ポート情報

表 59: ofp_port_status 構造体

ofp_port_reason の列挙体は以下になる。

値	定数名	内容
0	OFPPR_ADD	ポート追加
1	OFPPR_DELETE	ポート削除
2	OFPPR_MODIFY	ポート属性値変更

表 60: ofp_port_reason 列挙体

2.4.4.4.4. Error Message

OFS は異常が発生した場合、OFC に OFPT_ERROR_MSG を送信する。

ofp_error_msg 構造体は以下になる。

型	変数名	内容
struct ofp_header	header	OpenFlow プロトコルヘッダ
uint16_t	type	エラータイプ
uint16_t	code	エラーコード
uint8_t	data[0]	エラーデータ

表 61: ofp_error_msg 構造体

エラータイプ列挙体は以下になる。

値	定数名	内容
0	OFPET_HELLO_FAILED	HELLO プロトコル失敗
1	OFPET_BAD_REQUEST	リクエスト解析不能
2	OFPET_BAD_ACTION	Action 記述エラー
3	OFPET_FLOW_MOD_FAILED	フローエントリ変更失敗
4	OFPET_PORT_MOD_FAILED	Port-Mod リクエスト失敗
5	OFPET_QUEUE_OP_FAILED	Queue へのオペレーション失敗

表 62: ofp_error_type 列挙体

エラータイプが OFPET_HELLO_FAILED の場合、エラーコードは以下の列挙体になる。

値	定数名	内容
0	OFPHFC_INCOMPATIBLE	互換性がないバージョン
1	OFPHFC_EPERM	権限エラー

表 63: ofp_hello_failed_code 列挙体

エラータイプが OFPET_BAD_REQUEST の場合、エラーコードは以下の列挙体になる。

値	定数名	内容
0	OFPBRC_BAD_VERSION	ofp_header.verson 未サポート
1	OFPBRC_BAD_TYPE	ofp_header.type 未サポート
2	OFPBRC_BAD_STAT	ofp_stats_request.type 未サポート
3	OFPBRC_BAD_VENDOR	ベンダ未サポート
4	OFPBRC_BAD_SUBTYPE	ベンダサブタイプ未サポート
5	OFPBRC_EPERM	権限エラー
6	OFPBRC_BAD_LEN	不正なリクエスト長
7	OFPBRC_BUFFER_EMPTY	対象バッファが既に使用済み
8	OFPBRC_BUFFER_UNKNOWN	対象バッファが存在しない

表 64: ofp_bad_request_code 列挙体

エラータイプが OFPET_BAD_ACTION の場合、エラーコードは以下の列挙体になる。

値	定数名	内容
0	OFPBAC_BAD_TYPE	不明な Action
1	OFPBAC_BAD_LEN	Action のバイト長
2	OFPBAC_BAD_VENDOR	不明なベンダ ID
3	OFPBAC_BAD_VENDOR_TYPE	対象ベンダ ID についての不明な Action
4	OFPBAC_BAD_OUT_PORT	Output Action の検証エラー
5	OFPBAC_BAD_ARGUMENT	Action 引数エラー
6	OFPBAC_EPERM	権限エラー
7	OFPBAC_TOO_MANY	アクション数が多すぎる
8	OFPBAC_BAD_QUEUE	Queue の検証エラー

表 65: ofp_bad_action_code 列挙体

エラータイプが OFPET_FLOW_MOD_FAILED の場合、エラーコードは以下の列挙体になる。

値	定数名	内容
0	OFPFMFC_ALL_TABLES_FULL	全テーブルが一杯
1	OFPFMFC_OVERLAP	CHECK_OVERLAP*が設定されている状態でオーバーラップしたフロー追加を試みた
2	OFPFMFC_EPERM	権限エラー
3	OFPFMFC_BAD_EMERG_TIMEOUT	緊急エントリの timeout 設定が 0 以外
4	OFPFMFC_BAD_COMMAND	不明なコマンド
5	OFPFMFC_UNSUPPORTED	未対応 Action

表 66: ofp_flow_mod_failed_code 列挙体

エラータイプが OFPET_PORT_MOD_FAILED の場合、エラーコードは以下の列挙体になる。

値	定数名	内容
0	OFPPMFC_BAD_PORT	対象ポートが存在しない
1	OFPPMFC_BAD_HW_ADDR	対象ハードウェアアドレス不正

表 67: ofp_port_mod_failed_code 列挙体

エラータイプが OFPET_QUEUE_OP_FAILED の場合、エラーコードは以下の列挙体になる。

値	定数名	内容
0	OFPQOFC_BAD_PORT	不正なポート
1	OFPQOFC_BAD_QUEUE	Queue が存在しない
2	OFQQOFC_EPERM	権限エラー

表 68: ofp_queue_op_failed_code 列挙体

2.4.4.5. Synchronous Messages

2.4.4.5.1. Hello

OFPT_HELLO メッセージは OpenFlow ヘッダのみで構成されている。body は持っていない。

2.4.4.5.2. Echo Recuest

Echo リクエストは OpenFlow ヘッダと任意の長さのデータフィールドで構成されている。データフィールドはレイテンシや帯域幅、OFC と OFS の死活監視などに使用する。

2.4.4.5.3. Echo Reply

Echo リプライは OpenFlow ヘッダと Echo Request メッセージからなる。

2.4.4.5.4. Vendor

ベンダメッセージは以下の構造体からなる。

型	変数名	内容
struct ofp_header	header	OpenFlow プロトコルヘッダ
uint32_t	vendor	ベンダ ID

表 69: ofp_vendor_header 構造体

vendor の最上位バイトが"0"の場合、次の 3 バイトはベンダの IEEE OUI を示す。

OFS はベンダ拡張の理解ができない場合、OFC へ OFPT_ERROR を送信する必要がある。

第3章　ホームネットワーク

3.1. ホームネットワークとは

家庭内に構築した通信システムの LAN 環境である。ホームネットワークにはパソコンだけでなく、ビデオや、オーディオといった「AV 機器」、冷蔵庫や電子レンジなどの「白物家電」、さらには自動車なども接続される。

ホームネットワークは小規模ながらも制御系やストリームデータ伝送系など、様々なアプリケーションが混在し、異なる性質を有するトラヒックが発生する。また、ユーザの行動に応じてトラヒックの時間的な変動も大きいという特徴がある。

ホームネットワークでは UTP ケーブルなどによる専用の情報配信線を新規に行うことが望ましくない。電力線や無線といった伝送能力が不安定な媒体を使わざるを得ないことが多い。そのため十分な通信品質が得られない状況に陥ることが珍しくない。

ホームネットワークにおいて、ユーザの利用状況に応じてネットワーク機器の設定をダイナミックに変更し、適切な QoS 制御を行うことにより、限られた伝送能力を最大限に活用できる可能性が取りざたされている。しかし、QoS の設定方法は機器ごとに異なる。さらに、網羅的に対応する必要があるなど、その実現は容易ではなかった。

3.2. インターネット回線種類

現在、ホームネットワークからのインターネット接続は”ブロードバンド”と称される大容量通信ができるサービスが主流である。代表的な接続サービスを述べる。

3.2.1. ADSL(Asymmetric Digital Subscriber Line)

xDSL はメタル回線にデジタル信号を重畳して伝送する技術である。日本の xDSL 技術は上りと下りの速度が異なる ADSL が最も普及している。[6]

1999 年の商用化開発当時では下り 1.5Mbps、上り 512kbps であった。現在では信号処理技術発展に伴い下り 47Mbps、上り 5Mbps の伝送速度となった。

ただし、ADSL の伝送速度は電話局からの距離が離れると低下する。

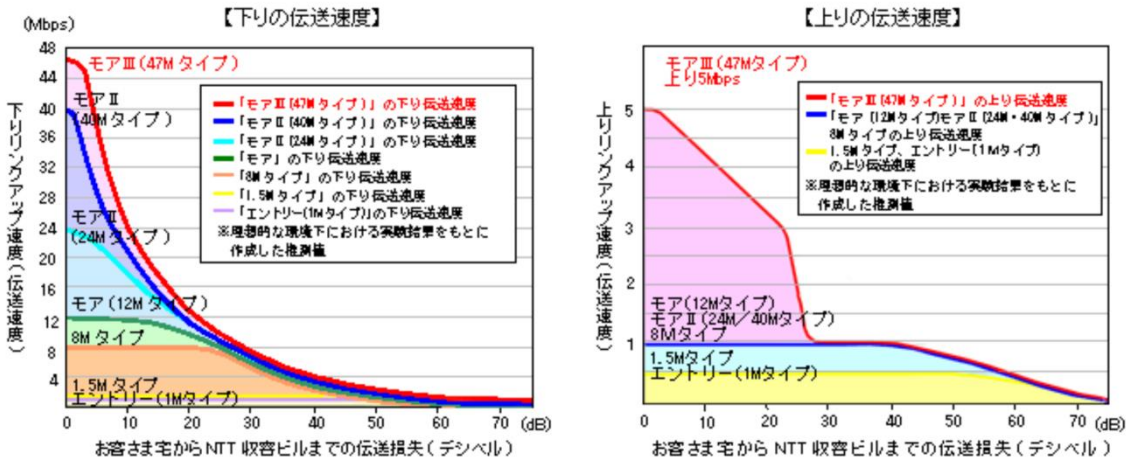

図 4:ADSL 伝送速度と伝送損失図 …NTT 東日本 フレッツ・ADSL 通信速度チェックコーナーより引用

3.2.2. FTTH(Fiber To The Home)

FTTH は光信号を使用するサービスで、ADSL などの電気信号と違い収容局接続からの距離の影響が受けづらく、高速な通信を提供する。

通信速度は 100Mbps～1Gbps が主流となっている。

3.3. ホームネットワークの代表伝送媒体

3.3.1. UTP、STP、FTP ケーブル

現在、主流となっている Ethernet にて標準的に使用されている。UTP(Unshielded Twisted Pair)ケーブルはシールドがないツイストペアケーブルである。STP(Shielded Twisted Pair)ケーブルは電線対にシールドがされているツイストペアケーブルである。FTP(Foilded Twisted Pair)ケーブルは STP ケーブルの一種であり、シールド方式の違いにより名称が変わる。

ケーブルには以下の様な Category に分かれて規格化されている。

Category	最大周波数	最大転送速度	主な用途
Category1	規定なし	20Kbps	電話線
Category2	1MHz	4Mbps	ISDN
Category3	16MHz	100Mbps	Ethernet(10BASE-T)
Category4	20MHz	100Mbps	トークンリング、ATM
Category5	100MHz	100Mbps	100BASE-TX
Category5e	100MHz	1000Mbps	100BASE-TX、1000BASE-T
Category6	250MHz	1000Mbps	1000BASE-TX、10GBASE-T
Category6a	500Mhz	10Gbps	10GBASE-T
Category7	600MHz	10Gbps	10GBASE-T

表 70:ケーブル Category 一覧

IEEE の Ethernet 規格のうちツイストペアケーブル部を以下に記載する。

規格名		別名	通信速度	標準化規格	距離
1BASE5		-	1Mbps	IEEE 802.3e	250m
10BASE-T		-	10Mbps	IEEE 802.3i	100m
100BASE-T	100BASE-TX	Fast Ethernet	100Mbps	IEEE 802.3u	100m
	100BASE-T4	-	100Mbps		100m
	100BASE-T2	-	100Mbps	IEEE 802.3y	100m
1000BASE-T	1000BASE-T	Gigabit Ethernet	1000Mbps	IEEE 802.3ab	100m
10GBASE-T		-	10Gbps	IEEE 802.3an	100m

表 71:Ethernet 規格

3.3.2. 無線

無線通信を利用し、データの送受信を行う構内通信網である。IEEE 802.11 規格に準拠した機器で構成されるネットワークを指すことが多い。

IEEE 802.11 規格の他に IEEE 802.15 規格がある。IEEE 802.15 規格は Bluetooth や ZigBee などの技術仕様を標準化している。[10]

Wi-Fi は IEEE 802.11 規格を利用したデバイス間の相互接続を認められたことを示す名称である。

家庭で利用されている IEEE 802.11 の主な規格を以下に記載する。

規格	周波数帯	最大通信速度
IEEE 802.11a	5GHz 帯	54Mbps
IEEE 802.11b	2.4GHz 帯	11Mbps
IEEE 802.11g	2.4GHz 帯	54Mbps
IEEE 802.11n	2.4G/5GHz 帯	600Mbps
IEEE 802.11ac	5GHz 帯	6.93Gbps

表 72:IEEE 802.11 主な規格

3.3.2.1. 認証方式

現在の家庭用ブロードバンドルータでは WPA(Wi-Fi Protected Access)や WPA2 と呼ばれる認証方式が主流である。

WPA はパーソナルモード(PSK)とエンタープライズモードがある。エンタープライズモードは RADIUS サーバが必要になるため、ホームネットワークではあまり使用されない。

3.3.2.2. 暗号化

暗号化の種類には WEP(Wired Equivalent Privacy)、TKIP(Temporol Key Integirity Protocol)、AES(Advanced Encryption Standard)と呼ばれる暗号化方式がある。暗号化方式の比較は以下の通りである。

名称	WEP	TKIP	AES
種類	RC4	RC4	AES
セキュリティ強度	X	○	◎
暗号キーの交換	X	○	○

表 73:暗号化方式比較

3.3.3. 電力線搬送通信

PLC(Power Line Communication)と呼ばれる電力線を通信回路として利用する技術である。450KHz 以下の周波数を用いるものを低速 PLC、2-30MHz を用いるものを高速 PLC と呼ぶことがある。

高速 PLC は日本が中心となり規格を制定した「HD-PLC」とアメリカが中心となり規格を制定した「Home Plug」が合体した「IEEE1901」という規格が制定された。HD-PLC 規格の通信速度の最大理論値は 210Mbps 、Home Plug 規格の通信速度の最大理論値は 200Mbps となっている。[11]

3.4. Quality of Service

ホームネットワークでは前項で述べた通信媒体を主に利用するがそれぞれに弱点がある。

UTP などの有線ケーブルはパフォーマンス面や安定性の面で有利である。さらに、セキュリティの面でも有利であるが、有線であるため移動には向かず、場合によっては設置工事が必要になるのでホームネットワークには向かない。[12]

無線はパフォーマンス面では有線ケーブルには劣る。さらに、セキュリティを確保する為の技術が必要である。しかし、移動しながらでも電源が確保できれば利用可能である。[12]

PLC は家の 1 階～2 階間通信などの障害物が多い場合、無線よりパフォーマンス面で有利になることが多い。しかし、PLC は通信用途ではない電力線を利用しているので、コンセントに接続された他の電化製品の影響を常に受ける。IH クッキングヒータやドライヤーなどは特に影響が大きい。

QoS 技術はホームネットワークで多く使用される、無線や PLC など、不安定な媒体の通信品質問題を解決する為に注目されている技術のひとつである。

3.4.1. QoS のモデルタイプ

QoS のモデルはベストエフォート、Integrated Service(以下 IntServ)及び、Differentiated Service(以下 DiffServ)の 3 種類が存在する。

ベストエフォートは特別な処理をすることなく、到着順にパケットを転送していく FIFO(First In First Out)が提供される。[3]

IntServ はアプリケーションの通信フローごとに帯域幅を予約、遅延を保証する予約する方式である。RSVP(Resource reSerVation Protocol)を使用する。各ネットワーク機器は通信フローごとのデータを保持しなければいけない。大量にフローが流れる現在ではあまり使用されない。[3][4]

DiffServ はトラヒックを分類、マーキング、キューイング、スケジューリングする。各ネットワーク機器に設定する必要がある。DiffServ は受信したパケット毎に優先度を判断し、パケットを転送する。広く採用されており、QoS といえば DiffServ を指す事が多い。[3][4]

3.4.2. DiffServ

3.4.2.1. 分類

ネットワーク機器は受信したパケットをクラス毎に分類する。代表的なキーワードは送信元・宛先 IP アドレスやポート番号、IP Precedence、DSCP 等で分類する。

3.4.2.2. マーキング

分類されたパケットの優先度をつける。L2、L3 のそれぞれにマーキング方法が存在する。

L2 マーキングは VLAN(IEEE802.11Q)タグの PCP(Priority Code Point)値によって優先度を定義する。3bit のフィールドを持ち「0~7」の値が入る。

L3 マーギングは IP Precedence または DSCP によって優先度を定義する。IPv4 ヘッダの ToS フィールドの 8bit のうち、3bit 使用するのが IP Precedence、6bit 使用するのが DSCP である。DSCP は IP Precedence の拡張機能であり、互換性も保たれている。

IP Precedence のクラスは以下の表のように分けられる。

IP Precedence 値	IP Precedence
111(7)	Network Control
110(6)	Internetwork Control
101(5)	Critical
100(4)	Flash Override
011(3)	Flash
010(2)	Immediate
001(1)	Priority
000(0)	Routine

表 74:IP Precedence クラス一覧

IP Precedence 値の 6 及び 7 はルーティングパケットや KeepAlive など、ネットワーク制御用に予約済みである。通常は「0~5」の値を使用する。

DSCP は PHB(Per Hop Behavior)と呼ばれるクラスに分ける。PHB は以下の表のように区別される。

PHB(Per Hop Behavior)		DSCP 値
Default		000 000 (0)
CS (Class Selector)	CS0	000 000 (0)
	CS1	001 000 (8)
	CS2	010 000 (16)
	CS3	011 000 (24)
	CS4	100 000 (32)
	CS5	101 000 (40)
	CS6	110 000 (48)
	CS7	111 000 (56)
AF (Assured Forwarding)	AF11	001 010 (10)
	AF12	001 100 (12)
	AF13	001 110 (14)
	AF21	010 010 (18)
	AF22	010 100 (20)
	AF23	010 110 (22)
	AF31	011 010 (26)
	AF32	011 100 (28)
	AF33	011 110 (30)
	AF41	100 010 (34)
	AF42	100 100 (36)
	AF43	100 110 (38)
EF(Expedited Forwarding)		101 110 (46)

表 75:PHB 一覧

CS(Class Selector)は先頭 3bit だけを使用し、後半 3bit は 0 が設定される。

AF(Assured Forwarding)は先頭 3bit の値が大きいほど優先度が高く、後半 3bit の値が大きいほど破棄(Drop)される可能性が高くなる。

EF(Expedited Forwarding)はユーザデータの中では最優先度になっており、破棄を考慮されていない。通常音声パケットなどのトラヒックが割り当てられることが多い。

3.4.2.3. キューイングとスケジューリング

パケットを送信する前にキューにパケットを格納することをキューイングという。

また、キューに格納されたパケットの送信順を決定することをスケジューリングと呼ぶ。

キューイングやスケジューリングは様々な方式があり、代表的な方式を以下表にまとめる。

方式	説明
FIFO (First In First Out)	受信した順にパケットを送信する方式
PQ (Priority Queuing)	複数のキューの中で優先度の高いキューの特定パケットを最優先に送信する方式
WFQ (Weighted Fair Queuing)	通信フローごとにキューを作成し、優先度に応じてパケットを送信する方式
CBWFQ (Class Based WFQ)	クラス毎に帯域幅を割り当て、各トラヒックが帯域幅を確保して通信する方式。
LLQ (Low Latency Queuing)	帯域幅を割り当て、優先度の高いキューからパケットが送信される方式
WRED (Weighted Random Early Detection)	バッファが溢れる直前に、優先度の低いパケットを Drop する方式。

表 76:代表的なキューイング・スケジューリング方式

3.4.3. ホームネットワークにおける QoS 問題点

ホームネットワークは無線や PLC など様々な媒体が利用され、L2 レイヤの QoS は媒体毎に設定が異なる。有線なら IEEE802.1p 規格、無線なら IEEE802.11e 規格、PLC なら IEEE1901 規格など様々な規格が存在する。このため、L2 レイヤの QoS を設定するためにはすべての機器において、設定を網羅的に対応する必要がある。

また、L3 レイヤの QoS においてもホームネットワークは小規模ながらも制御系やストリームデータ伝送系など、様々に異なる性質を有するトラヒックを発生するアプリケーションが混在し、ユーザの行動に応じてトラヒックの時間的な変動も大きいという特徴がある。そのため、伝送能力を最大限に伝えるためには QoS の設定を利用状況に応じて設定変更していく必要がある。

第4章　OpenFlow ネットワーク構築

4.1. ホームネットワーク問題点抽出

第 3 章のホームネットワークより、OpenFlow 技術にて解決出来そうな問題点を抽出する。3.3.3.ホームネットワークにおける「伝送能力を最大限に伝えるためには QoS の設定を利用状況に応じて設定変更していく必要がある」問題は OpenFlow 技術にて ToS を変更する事により解決できないかを検討する。この問題点の一部例を抜粋し、仮想ネットワークを構築する。

4.2. OFC

OpenFlow 技術を使用し、ネットワークを構築するにあたり、OFC を選定する。
オープンソースで公開されている代表的な OFC は以下の種類がある。

名前	OpenFlow 対応バージョン	言語
NOX	1.0	C++
POX	1.0	Python
Floodlight	1.0、1.3	Java
OpenDayLight	1.0、1.3	Java
Trema	1.0、1.3(Trema-edge)	Ruby、C
Ryu	1.0、1.2、1.3	Python

表 77:代表的な OFC 一覧

4.2.1. NOX

NOX は Nicira Networks(現 VMware)によって開発された OpenFlow ver.1.0 準拠初の OFC フレームワークである。C++に対応しており、GPL v3 ライセンスで提供している。2008 年に研究コミュニティに寄付された。以後、SDN の多くの調査・研究プロジェクトで使用されている。

4.2.2. POX

POX は NOX から派生した OFC フレームワークである。様々な OFC フレームが誕生した中で生産性の高い Python に対応している。NOX と同じく GPL v3 ライセンスにて提供している。また、NOX の GUI、可視化ツールを使用できる。

4.2.3. Floodlight

Floodlight は Big Switch Netwoks 社によって開発された OFC フレームワークである。Java に対応しており、Apache 2.0 ライセンスで提供している。オープンソースではあるが、商用向け OFC として開発された。OpenStack と連携できる API も提供する。

4.2.4. OpenDayLight

OpenDayLight は 2013 年 4 月に Linux Foundation が発表した OFC フレームワークである。Cisco、Brocade などのネットワーク機器ベンダからマイクロソフト、Red Hat などのソフトウェアベンダなど幅広いベンダによって発足された。Java に対応しており、EPL 1.0 ライセンスで提供している。

4.2.5. Trema

Trema は開発、テスト、デバッグツールを備えた OFC フレームワークである。NEC が中心となり開発されている。Ruby、C に対応しており、GPL v2 ライセンスで提供している。TremaEdge は OpenFlow ver.1.3 に対応しており、将来は統合される予定である。

4.2.6. Ryu

Ryu は NTT 研究所により開発された OpenFlow ver1.0、1.2、1.3 対応の OFC フレームワークである。Python に対応しており、Apatch 2.0 ライセンスにて提供している。

今回の課題研究では様々な OpenFlow のバージョンに対応し、生産性の高い Python 言語を使用する Ryu を OFC として採用し、開発する。

4.3. OFS

OpenFlow 技術を使用し、ネットワークを構築するにあたり、OFS を選定する。OFS は Mininet や Ubuntu 標準パッケージが用意されている Open vSwitch を使用し、ネットワークを構築する。

4.3.1. Open vSwitch

Open vSwitch(以下 OVS)は Nicira Networks(現 VMware)によって開発された OFS である。Apatch 2.0 ライセンスで提供している。多くの Linux ディストリビューションにて採用されており、Xen や KVM などの仮想化プラットフォームにも対応している。また、Openstack などといったクラウド基盤にも連携している。

4.4. Mininet

Mininet は仮想ホスト、スイッチ、コントローラおよびネットワークのリンクを作成できるネットワークエミュレータである。Mininet は OpenFlow 技術をサポートし、Mininet 上で OFC や OFS のテストや性能評価などができる。

Mininet は公開されている VM を使用するか、Linux にインストールして使用することができる。公開元であるスタンフォード大学は VM を使用することを推奨している。

本研究では VM を使用し、OpenFlow ネットワークを構築する。

4.5. ネットワーク構築

4.5.1. 環境構築

1. Mininet の GitHub(https://github.com/mininet/mininet/wiki/Mininet-VM-Images)から VM をダウンロードする。
2. VirtulBox(https://www.virtualbox.org/wiki/Downloads)をダウンロードし、インストールを行う。
3. VirtulBox に 1 でダウンロードした VM をインポートする。
4. Mininet の VM を起動し、以下コマンドより、必要パッケージを取得する。

```
$ sudo apt-get update
$ sudo apt-get install git tmux python
$ sudo apt-get install python-dev python-setuptools python-pip python-lxml
$ git clone git://github.com/osrg/ryu.git
$ cd ryu
$ sudo python ./setup.py install
```

4.5.2. ユニキャスト

ユニキャスト通信のプログラムを作成した。

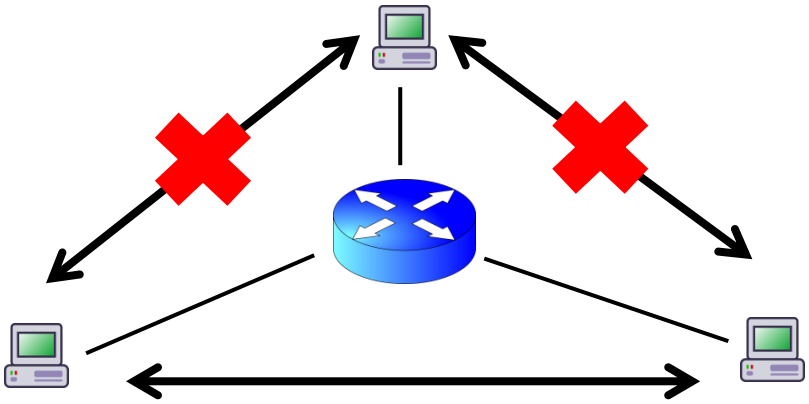

図 5:ユニキャスト通信トポロジ図

今回のユニキャスト通信では OFS は以下の命令をコントローラから指示される。

1. ポート 1 から受信したパケットをポート 2 へ転送
2. ポート 2 から受信したパケットをポート 1 へ転送

上記の通信を実現するために以下のような Ryu アプリケーションを作成した。

1. クラスの定義と初期化

```
class Unicast(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_0.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(Unicast, self).__init__(*args, **kwargs)
```

Ryu アプリケーションとして実装するために、ryu.case.app_manager.RyuApp を継承する。

また、OpenFlow バージョン 1.0 を指定する。

2. Packet-In イベントハンドラの作成

```
    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def _packet_in_handler(self, ev):
        msg = ev.msg
        datapath = msg.datapath

        self.unicast_flow(datapath)
```

フローテーブルに登録されていないパケットを受信した際の処理を作成する。

3. フロー登録

```
    def unicast_flow(self, datapath):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        match = parser.OFPMatch(in_port=HOST1_PORT)
        actions = [parser.OFPActionOutput(HOST2_PORT)]

        mod = parser.OFPFlowMod(datapath=datapath, match=match, cookie=0,
                                command=ofproto.OFPFC_ADD,actions=actions)

        datapath.send_msg(mod)
…
```

ポート１番から受信したらポート２番に転送し、ポート２番から受信したらポート１番に転送する。このフローを OFS に登録することによりユニキャスト通信を実現する。

動作確認を行うために Mininet を下記コマンドにて起動する。

```
$ sudo mn --topo single,3 --mac --switch ovsk --controller remote
```

パラメータ	値	設定
topo	Single,3	スイッチが 1 台、ホストが 3 台
mac	なし	自動的にホストの macアドレスを指定する
switch	ovsk	Open vSwitch を使用する。
controller	remote	OFC は外部コントローラを使用する

表 78:Mininet パラメータの説明

GitHub からダウンロードしたフォルダに移動し、以下コマンドで Ryu アプリケーションを実行する。

```
$ ./bin/ryu-manager app/unicast.py
loading app app/unicast.py
loading app ryu.controller.ofp_handler
instantiating app app/unicast.py of Unicast
instantiating app ryu.controller.ofp_handler of OFPHandler
```

Ryu アプリケーション起動後、Mininet にて ping を実行し導通確認を行う。

```
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=1000 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=2.28 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.084 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.090 ms
^C
--- 10.0.0.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3001ms
rtt min/avg/max/mdev = 0.084/250.784/1000.674/432.950 ms
mininet> h1 ping h3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
From 10.0.0.1 icmp_seq=1 Destination Host Unreachable
From 10.0.0.1 icmp_seq=2 Destination Host Unreachable
From 10.0.0.1 icmp_seq=3 Destination Host Unreachable
^C
--- 10.0.0.3 ping statistics ---
4 packets transmitted, 0 received, +3 errors, 100% packet loss, time 3004ms
pipe 3
```

```
mininet> h2 ping h3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
From 10.0.0.2 icmp_seq=1 Destination Host Unreachable
From 10.0.0.2 icmp_seq=2 Destination Host Unreachable
From 10.0.0.2 icmp_seq=3 Destination Host Unreachable
^C
--- 10.0.0.3 ping statistics ---
5 packets transmitted, 0 received, +3 errors, 100% packet loss, time 4025ms
pipe 3
```

上記のログにより”h1-h2”間通信可能であり、”h1-h3”間および”h2-h3”間通信は不可能なことが確認できる。

なお、OFS 上のフローは以下の通りである。

```
$ sudo ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=5047.587s, table=0, n_packets=12, n_bytes=728,
idle_age=5035, in_port=1 actions=output:2
 cookie=0x0, duration=5047.586s, table=0, n_packets=12, n_bytes=728,
idle_age=5025, in_port=2 actions=output:1
```

4.5.3. L2 スイッチ

L2 スイッチのプログラムは Ryu のサンプルアプリを参考に作成した。[5]

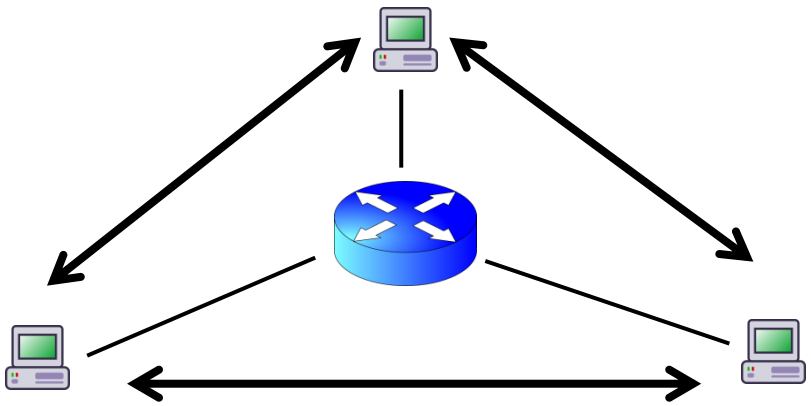

図 6:L2 スイッチトポロジ図

今回の L2 スイッチでは OFS は以下の命令を OFC から指示される。

1. パケットを受信したら Mac アドレスを確認する
2. 登録済み Mac アドレスかつ、受信ポートが登録済みポートと異なる場合、関連する Mac アドレスのフローをフローテーブルから変更・削除する
3. 登録済み Mac アドレスなら送信ポートを指定してフローを登録する
4. 未登録の Mac アドレスならば、受信ポート以外に最小のスパニングツリーに沿ったパケットを送信する。

上記の通信を実現するために以下のような Ryu アプリケーションを作成した。Ryu のサンプルアプリ(simple_switch)を参考に作成し、主な追加箇所を記載する。

1. 受信ポートを確認し、登録されているポートと異なる場合はフローを削除する

```
        self.mac_to_port[dpid].setdefault(src, msg.in_port)

        if self.mac_to_port[dpid][src] != msg.in_port:
            self.del_flow(datapath, haddr_to_bin(src))
            self.modify_flow((datapath, haddr_to_bin(src), msg.in_port)
            self.mac_to_port[dpid][src] = msg.in_port
```

2. フロー削除

```
    def del_flow(self, datapath, mac):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        match = parser.OFPMatch(dl_src=mac)

        mod = parser.OFPFlowMod(datapath=datapath, match=match, cookie=0,
                                command=ofproto.OFPFC_DELETE)

        datapath.send_msg(mod)
```

3. フロー修正

```
def modify_flow(self, datapath, mac, port):
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser

    match = parser.OFPMatch(dl_dst=mac)

    actions = [datapath.ofproto_parser.OFPActionOutput(port)]

    mod = parser.OFPFlowMod(datapath=datapath, match=match,
                            cookie=0, idle_timeout=MAC_IDLE_TIME,
                            command=ofproto.OFPFC_MODIFY,actions=actions)

    datapath.send_msg(mod)
```

4. フロー登録

```
def add_flow(self, datapath, in_port, src, dst, actions):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        match = parser.OFPMatch(in_port=in_port, dl_src=src, dl_dst=dst)

        mod = parser.OFPFlowMod(datapath=datapath, match=match,
                                cookie=0, idle_timeout=MAC_IDLE_TIME,
                                command=ofproto.OFPFC_ADD,actions=actions)

        datapath.send_msg(mod)
```

動作確認を行うために Mininet、Ryu アプリケーションを実行する。Mininet の設定はユニキャスト通信と同一である。

Ryu アプリケーション起動後、Mininet にて ping を実行し導通確認を行う。

```
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=9.82 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.586 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.129 ms
^C
--- 10.0.0.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2000ms
rtt min/avg/max/mdev = 0.129/3.513/9.826/4.467 ms
mininet> h1 ping h3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 bytes from 10.0.0.3: icmp_seq=1 ttl=64 time=19.1 ms
64 bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=0.639 ms
64 bytes from 10.0.0.3: icmp_seq=3 ttl=64 time=0.169 ms
^C
--- 10.0.0.3 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2004ms
rtt min/avg/max/mdev = 0.169/6.664/19.186/8.856 ms
mininet> h2 ping h3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 bytes from 10.0.0.3: icmp_seq=1 ttl=64 time=17.7 ms
64 bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=0.431 ms
64 bytes from 10.0.0.3: icmp_seq=3 ttl=64 time=0.140 ms
^C
--- 10.0.0.3 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 0.140/6.120/17.791/8.253 ms
```

上記ログより、h1-h2-h3 間で通信可能であることが確認できる。

以下のコマンドを OFS インターフェースに実行し、パケットキャプチャすることにより L2 スイッチとして動作することが確認できる。

```
$ sudo tcpdump -n -vvv -i [interface] -s 0 -w [filename].cap
```

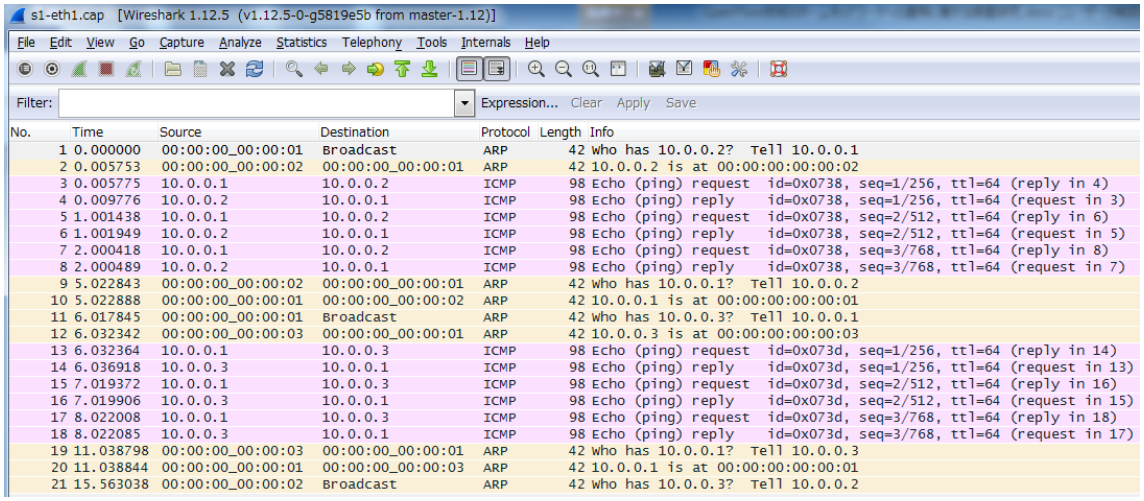

図 7:OFS(s1-eth1)パケットキャプチャ

なお、OFS 上のフローは以下の通りである。

```
$ sudo ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=13.044s, table=0, n_packets=3, n_bytes=238, idle_timeout=300,
idle_age=8, in_port=2,dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:03
actions=output:3
 cookie=0x0, duration=27.341s, table=0, n_packets=4, n_bytes=336, idle_timeout=300,
idle_age=22, in_port=3,dl_src=00:00:00:00:00:03,dl_dst=00:00:00:00:00:01
actions=output:1
 cookie=0x0, duration=13.05s, table=0, n_packets=4, n_bytes=336, idle_timeout=300,
idle_age=8, in_port=3,dl_src=00:00:00:00:00:03,dl_dst=00:00:00:00:00:02
actions=output:2
 cookie=0x0, duration=27.335s, table=0, n_packets=3, n_bytes=238, idle_timeout=300,
idle_age=22, in_port=1,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:03
actions=output:3
 cookie=0x0, duration=41.072s, table=0, n_packets=8, n_bytes=728, idle_timeout=300,
idle_age=34, in_port=1,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02
actions=output:2
 cookie=0x0, duration=41.077s, table=0, n_packets=9, n_bytes=826, idle_timeout=300,
idle_age=34, in_port=2,dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:01
actions=output:1
```

4.5.4. トラヒックモニタ

OpenFlow 技術を使用し、トラヒックモニタのプログラムを作成した。なお、トラヒックモニタは L2 スイッチのサブクラスとして作成した。

1. クラスの定義と初期化

```
class Monitor(switch_hub.SwitchHub):
    def __init__(self, *args, **kwargs):
        super(Monitor, self).__init__(*args, **kwargs)
        self.datapath = {}
        self.monitor_thread = hub.spawn(self._monitor)
```

モニタクラスを L2 スイッチで作成したクラスのサブクラスとして作成する。定期的に統計情報取得のリクエストを OFS へ送信するため、スレッドを生成する。

2. StateChange イベントハンドラの作成

```
    @set_ev_cls(ofp_event.EventOFPStateChange,MAIN_DISPATCHER)
    def _state_change_handler(self, ev):
        self.datapath = ev.datapath
```

接続中のスイッチを監視対象とするため、スイッチの接続および切断の検出に EventOFPStateChange イベントを利用する。このイベントは Ryu フレームワークが発行するもので、Datapath のステートが変わったときに発行される。今回のプログラムではスイッチ 1 台を対象としてスイッチの離脱は考慮していない。

3. モニタスレッドの作成

```
    def _monitor(self):
        while True:
            if self.datapath:
                self.stats_request(self.datapath)
            hub.sleep(POLLING_TIME)
```

10 秒間隔でポートに関する統計情報をスイッチに要求する。

4.　統計情報要求

```
    def stats_request(self, datapath):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        req = parser.OFPPortStatsRequest(datapath, 0, ofproto.OFPP_NONE)
        datapath.send_msg(req)
```

OFS に対してポートに関する統計情報を要求する。OFPP_NONE を指定することにより全ての OFS ポートに関する統計情報を取得できる。

5.　PortStatsReply イベントハンドラの作成

```
    @set_ev_cls(ofp_event.EventOFPPortStatsReply, MAIN_DISPATCHER)
    def _port_stats_reply_handler(self, ev):
        body = ev.msg.body

        for stat in sorted(body, key=attrgetter('port_no')):
            bytes = stat.rx_bytes + stat.tx_bytes
            print 'port:%x bytes:%d' % (stat.port_no, bytes)
        print '---------------------------------'
```

PortStatsReply メッセージを受信したら、各ポートの送受信バイト数を表示する。

動作確認を行うために Mininet、Ryu アプリケーションを実行する。Mininet の設定はユニキャスト通信、L2 スイッチと同一である。

```
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=19.5 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.496 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.106 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.479 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.142 ms
^C
--- 10.0.0.2 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4002ms
rtt min/avg/max/mdev = 0.106/4.151/19.536/7.694 ms
```

```
$ vi app/monitor.py
mininet@mininet-vm:~/ryu$ bin/ryu-manager app/monitor.py
loading app app/monitor.py
loading app ryu.controller.ofp_handler
instantiating app app/monitor.py of Monitor
instantiating app ryu.controller.ofp_handler of OFPHandler
port:1 bytes:0
port:2 bytes:0
port:3 bytes:0
port:fffe bytes:0
------------------------------
port:1 bytes:1148
port:2 bytes:1148
port:3 bytes:42
port:fffe bytes:42
------------------------------
```

各ポートの送受信バイト数が確認できる。

4.5.5. QoS スイッチ

これまでの経緯を踏まえ、ホームネットワークをイメージした ToS フィールドを書き換えるスイッチを作成した。このプログラムも L2 スイッチのサブクラスでありトラヒックモニタを一部参考にした。

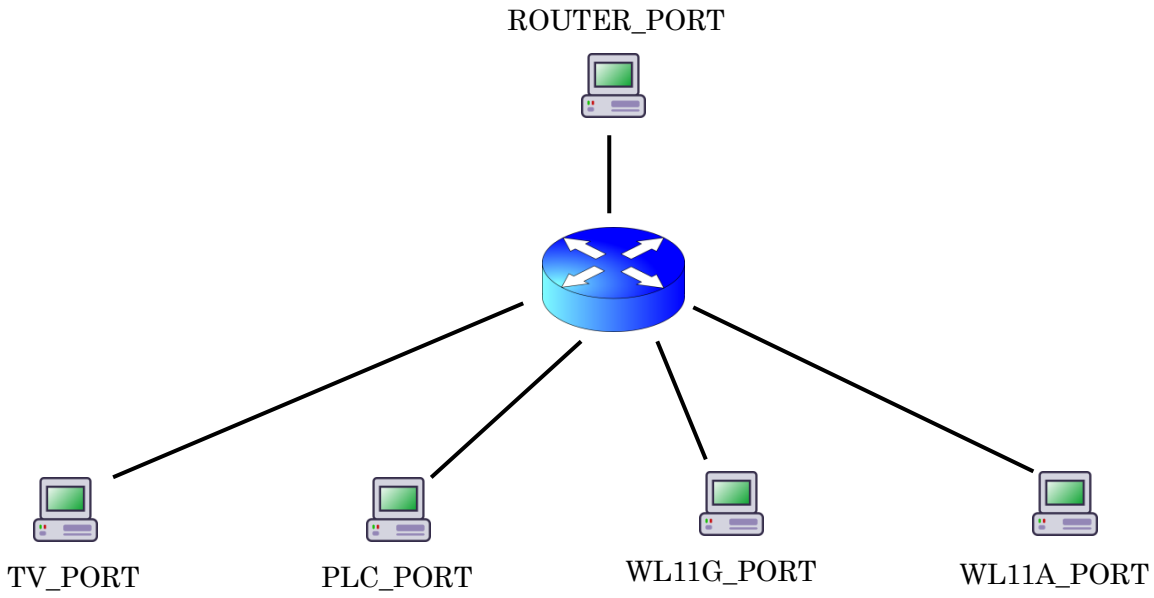

図 8:QoS スイッチトポロジ図

1.　クラスの定義と初期化

```
class Qos(switch_hub.SwitchHub):
    def __init__(self, *args, **kwargs):
        super(Qos, self).__init__(*args, **kwargs)

        self.datapath = {}

        self.base = {TV_PORT:TV_TRAFFIC,
                     PLC_PORT:PLC_TRAFFIC,
                     WL11A_PORT:WL_TRAFFIC,
                     WL11G_PORT:WL_TRAFFIC}

        self.traffic = {TV_PORT:0,
                        PLC_PORT:0,
                        WL11A_PORT:0,
                        WL11G_PORT:0}

        self.qos = {TV_PORT:{TRAFFIC:0, QOS_FLAG:QOS_OFF, TOS:TV_TOS},
                    PLC_PORT:{TRAFFIC:0, QOS_FLAG:QOS_OFF, TOS:PLC_TOS},
                    WL11A_PORT:{TRAFFIC:0, QOS_FLAG:QOS_OFF, TOS:WL_TOS},
                    WL11G_PORT:{TRAFFIC:0, QOS_FLAG:QOS_OFF, TOS:WL_TOS}}

        self.monitor_thread = hub.spawn(self._qos_monitor)
```

base 辞書に各ポートが ToS フィールド書き換えフローを登録判断するための通信量を規定する。

traffic 辞書に取得した各ポートの通信量を保存する。

qos 辞書に前回取得した通信量、QoS 設定フラグ、および ToS フィールドの設定値を保存する。

2. モニタスレッドの作成

```
def _qos_monitor(self):
    # Initialize Setting
    while True:
        if self.datapath:
            self.stats_request(self.datapath,
                               self.datapath.ofproto.OFPP_NONE)
            self.renew_traffic()
            break
        hub.sleep(INIT_TIME)

    hub.sleep(POLLING_TIME)
    while True:
        self.stats_request(self.datapath,
                           self.datapath.ofproto.OFPP_NONE)
        hub.sleep(REPLY_TIME)
        self.qos_setting()
        self.renew_traffic()
        hub.sleep(POLLING_TIME)
```

QoS スイッチのモニタスレッドでは、始めにポートの統計情報を取得し、qos 辞書に通信量を保存する。これは OFC と接続前に OFS が通信していた場合でも、正しく通信量を判断するために必要な処理である。その後、10 分間隔でポートの統計情報を取得し、QoS の設定判断を行う。

3. PortStatsReply イベントハンドラの作成

```
@set_ev_cls(ofp_event.EventOFPPortStatsReply, MAIN_DISPATCHER)
def _port_stats_reply_handler(self, ev):
    body = ev.msg.body
    for stat in body:
        if stat.port_no in self.traffic.keys():
            self.traffic[stat.port_no] = stat.rx_bytes + stat.tx_bytes
```

PortStatsReply メッセージを受信したら、各ポートの通信量を更新する。

4. QoS 設定

```
def qos_setting(self):
    for port in self.traffic.keys():
        if self.traffic[port] - self.qos[port][TRAFFIC] > self.base[port]:
            if self.qos[port][QOS_FLAG] == QOS_OFF:
                self.add_qos(port)
        elif self.qos[port][QOS_FLAG] == QOS_ON:
            self.del_qos(port)
```

QoS 設定判断を行う。各ポートの通信量が規定値を超えた場合は ToS フィールドを書き換えるフローを登録する。また、通信量が規定値以下の場合はフローを削除する。

5. フロー登録

```
def add_qos(self, port):
    ofproto = self.datapath.ofproto
    parser = self.datapath.ofproto_parser

    match = parser.OFPMatch(in_port=port, dl_type=types.ETH_TYPE_IP,
                            dl_dst=haddr_to_bin(ROUTER_MAC))

    actions = [parser.OFPActionSetNwTos(self.qos[port][TOS]),
               parser.OFPActionOutput(ROUTER_PORT)]

    mod = parser.OFPFlowMod(datapath=self.datapath,
                            match=match, cookie=0,
                            command=ofproto.OFPFC_ADD,
                            idle_timeout=QOS_IDLE_TIME,
                            priority=QOS_PRIORITY,
                            flags=ofproto.OFPFF_SEND_FLOW_REM,
                            actions=actions)

    self.datapath.send_msg(mod)

    self.qos[port][QOS_FLAG] = QOS_ON
```

idle_timeout を設定し、フローを登録する。QoS 設定フラグを ON に更新する。

6. フロー削除

```
def del_qos(self, port):
    ofproto = self.datapath.ofproto
    parser = self.datapath.ofproto_parser

    match = parser.OFPMatch(in_port=port, dl_type=types.ETH_TYPE_IP,
                            dl_dst=haddr_to_bin(ROUTER_MAC))

    mod = parser.OFPFlowMod(datapath=self.datapath,
                            match=match, cookie=0,
                            command=ofproto.OFPFC_DELETE,
                            idle_timeout=QOS_IDLE_TIME,
                            priority=QOS_PRIORITY)

    self.datapath.send_msg(mod)
```

通信量が規定値以下のポートに対して、ToS フィールドを書き換えるフローを削除する。

7. FlowRemoved イベントハンドラの作成

```
@set_ev_cls(ofp_event.EventOFPFlowRemoved, MAIN_DISPATCHER)
def _flow_removed_handler(self, ev):
    msg = ev.msg
    port = msg.match.in_port
    self.qos[port][QOS_FLAG] = QOS_OFF
    if msg.reason == self.datapath.ofproto.OFPRR_IDLE_TIMEOUT:
        self.stats_request(self.datapath, port)
        hub.sleep(REPLY_TIME)
        self.qos[port][TRAFFIC] = self.traffic[port]
```

FlowRemoved イベントが発生したら、QoS 設定フラグを OFF にする。また、フロー削除の要因が idle_timeout ならば、対象ポートの統計情報を取得し、通信量を更新する。

8.　通信量更新

```
def renew_traffic(self):
    for port in self.traffic.keys():
        self.qos[port][TRAFFIC] = self.traffic[port]
```

各ポートの通信量を更新する。

動作確認を行うために Mininet、Ryu アプリケーションを実行する。Mininet の設定はホストが 5 台に増える。ポート 1 番を Router、ポート 2 番を TV、ポート 3 番を PLC、ポート 4 番を Wireless11G、ポート 5 番を Wireless11A と想定する。

```
mininet> h2 ping h1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
64 bytes from 10.0.0.1: icmp_seq=1 ttl=64 time=1.26 ms
64 bytes from 10.0.0.1: icmp_seq=2 ttl=64 time=0.134 ms
<中略>
64 bytes from 10.0.0.1: icmp_seq=26 ttl=64 time=0.104 ms
64 bytes from 10.0.0.1: icmp_seq=27 ttl=64 time=0.110 ms
^C
--- 10.0.0.1 ping statistics ---
27 packets transmitted, 27 received, 0% packet loss, time 26004ms
rtt min/avg/max/mdev = 0.104/0.200/1.260/0.223 ms
```

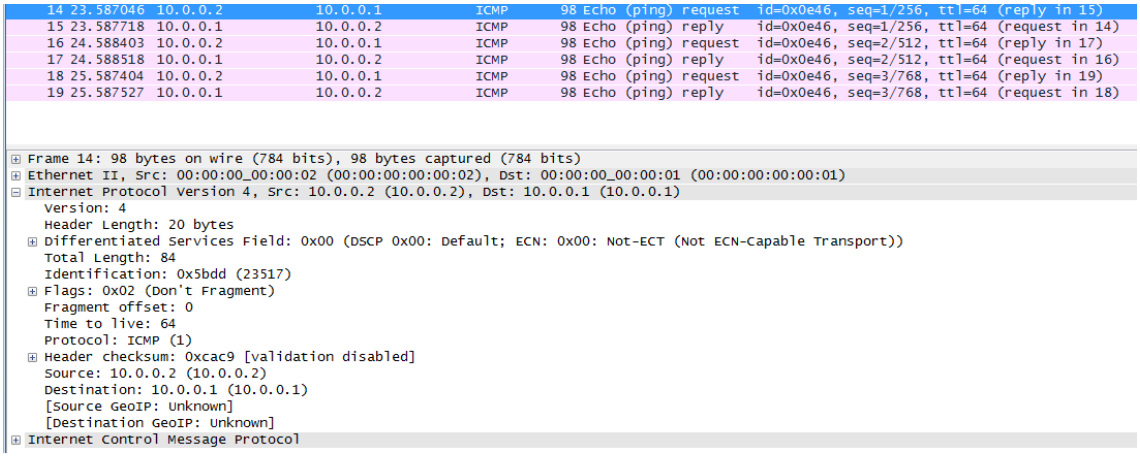

図 9:OFS(s1-eth2)パケットキャプチャ

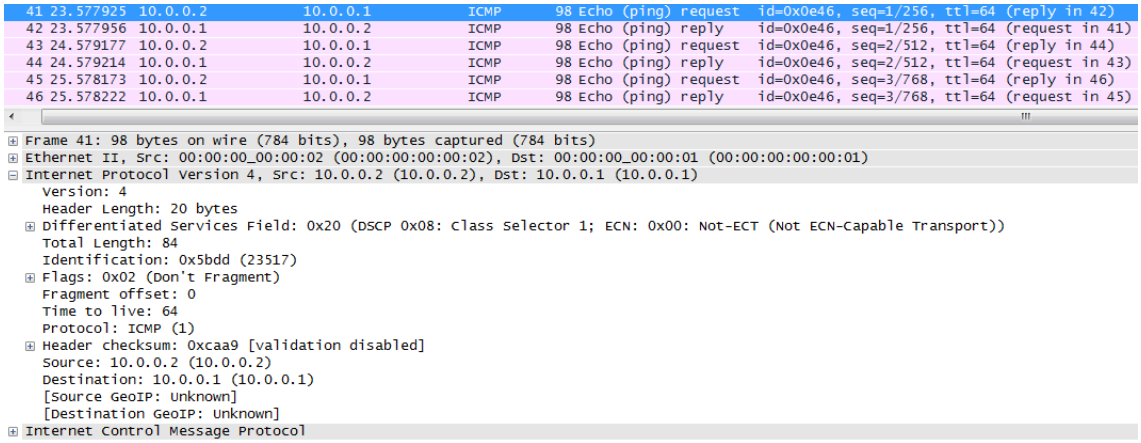

図 10:OFS(s1-eth1)パケットキャプチャ

図 9、図 10 のパケットキャプチャより”h2-h1”間の ping で ToS フィールド書き換えが行われていることが確認できる。

なお、ToS フィールド書き換えフローがある状態の OFS のフローは以下の通りである。

```
$ sudo ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=23.885s, table=0, n_packets=3, n_bytes=238, idle_timeout=300,
idle_age=18, in_port=3,dl_src=00:00:00:00:00:03,dl_dst=00:00:00:00:00:01
actions=output:1
 cookie=0x0, duration=18.926s, table=0, n_packets=4, n_bytes=336, idle_timeout=300,
idle_age=13, in_port=1,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:04
actions=output:4
 cookie=0x0, duration=13.372s, table=0, n_packets=4, n_bytes=336, idle_timeout=300,
idle_age=8, in_port=1, dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:05
actions=output:5
 cookie=0x0, duration=23.889s, table=0, n_packets=4, n_bytes=336, idle_timeout=300,
idle_age=18, in_port=1,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:03
actions=output:3
 cookie=0x0, duration=18.924s, table=0, n_packets=3, n_bytes=238, idle_timeout=300,
idle_age=13, in_port=4,dl_src=00:00:00:00:00:04,dl_dst=00:00:00:00:00:01
actions=output:1
 cookie=0x0, duration=13.37s, table=0, n_packets=3, n_bytes=238, idle_timeout=300,
idle_age=8, in_port=5,dl_src=00:00:00:00:00:05,dl_dst=00:00:00:00:00:01
actions=output:1
 cookie=0x0, duration=29.029s, table=0, n_packets=3, n_bytes=238, idle_timeout=300,
idle_age=24, in_port=2,dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:01
actions=output:1
 cookie=0x0, duration=29.033s, table=0, n_packets=4, n_bytes=336, idle_timeout=300,
idle_age=24, in_port=1, dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02
actions=output:2
 cookie=0x0, duration=12.352s, table=0, n_packets=0, n_bytes=0, idle_timeout=120,
idle_age=12, priority=65535,ip,in_port=2,dl_dst=00:00:00:00:00:01
actions=mod_nw_tos:32,output:1
```

第5章　OpenFlow 技術を用いたホームネットワーク構築

5.1. 家庭用 OpenFlow スイッチ作成

株式会社バッファロー社(以下バッファロー社)の WZR-HP-AG300H を利用し、OFS を作成する。@SRCHACK.ORG 氏のサイト(http://openflow.inthebox.info/)からファームウェアをダウンロードする。このファームウェアを使った OFS は Interop Tokyo 2012 にて開催されたオープンルータ・コンペティションにて NEC 賞を受賞している。[8]

5.1.1. 環境構築

WZR-HP-AG300H の OFS 化は通常のファームウェア更新と同様に、ダウンロードしたファームウェア(squashfs-factory.bin)をファームウェア更新画面から更新する。正常に更新された場合、WAN 側ポート(青色)に 192.168.1.1/24 という IP アドレスが設定される。また、OFC の IP アドレスは 192.168.1.10/24 が設定される。

eth0.4　eth0.3　eth0.2　eth0.1　eth1

192.168.1.1/24

図 11:WZR-HP-AG300H ポート写真

ファームウェア更新後、無線インターフェースが無効になっているので有効にする必要がある。telnet で接続し、以下ファイルを修正する。

```
root@OpenWrt:/# vi /etc/config/wireless

nfig wifi-device   radio0
        <中 略>
        # REMOVE THIS LINE TO ENABLE WIFI:
        # option disabled 1
        <中 略>
config wifi-device   radio1
        <中 略>
        # REMOVE THIS LINE TO ENABLE WIFI:
        # option disabled 1
                                                    <以下略>
```

また、OFS の設定変更、無線インターフェースを追加するため、以下のファイルを修正する

```
root@OpenWrt:/# vi /etc/config/openflow

config 'ofswitch'
        option 'dp' 'dp0'
        # option 'ofports' 'eth0.1 eth0.2 eth0.3 eth0.4'
        option 'ofports' 'eth0.1 eth0.2 eth0.3 eth0.4 wlan0 wlan1'
        option 'ofctl' 'tcp:192.168.1.10:6633'
        # option 'mode'   'outofband'
        option 'mode'   'inband'
```

ファイル修正後、再起動を実施すると無線インターフェースも OFS のポートとして使用できる。OFS のポート番号と対応するインターフェースは以下の通りである。

ポート番号	インターフェース	QoSスイッチポート
1	eth0.1	ROUTER_PORT
2	eth0.2	TV_PORT
3	eth0.3	PLC_PORT
4	eth0.4	-
5	wlan0	WL11G_PORT
6	wlan1	WL11A_PORT

表 79:OFS ポート番号と対応インターフェース

無線インターフェースに接続するための SSID、パスワードは以下の通りである。

設定項目	wlan0	wlan1
SSID	jaist_research_11g	jaist_research_11a
Password	jaist_research_11g	jaist_research_11a
認証方式	WPA2-PSK	WPA2-PSK
暗号化	AES	AES

表 80:無線インターインターフェース設定項目

5.2. 速度検証

バッファロー社 WZR-HP-AG300H の正期ファーム(ver.1.73)と OFS 化した WZR-HP-AG300H との速度比較を行う。OFS 化した WZR-HP-AG300H は第 4 章で作成した L2 スイッチと QoS スイッチを使用して速度測定する。なお、回線速度は NURO オリジナル通信速度測定システム(http://www.nuro.jp/speedup/nuroCheck.html)を使用し、速度測定する。なお、ホームネットワークは以下のように構成した。

用途	機器名称
インターネット回線	So-net 社 NURO 光 G2V
ONU 兼 Router	HUAWEI 社 HG8045D
OFC	NEC 社 PC-LL570JG3E
OFS or L2 スイッチ	バッファロー社 WZR-HP-AG300H
PLC	IO-DATA 社 PLC-HP240EA-S
無線子機	PLANEX 社 FFP-900D

表 81:仕様機器一覧

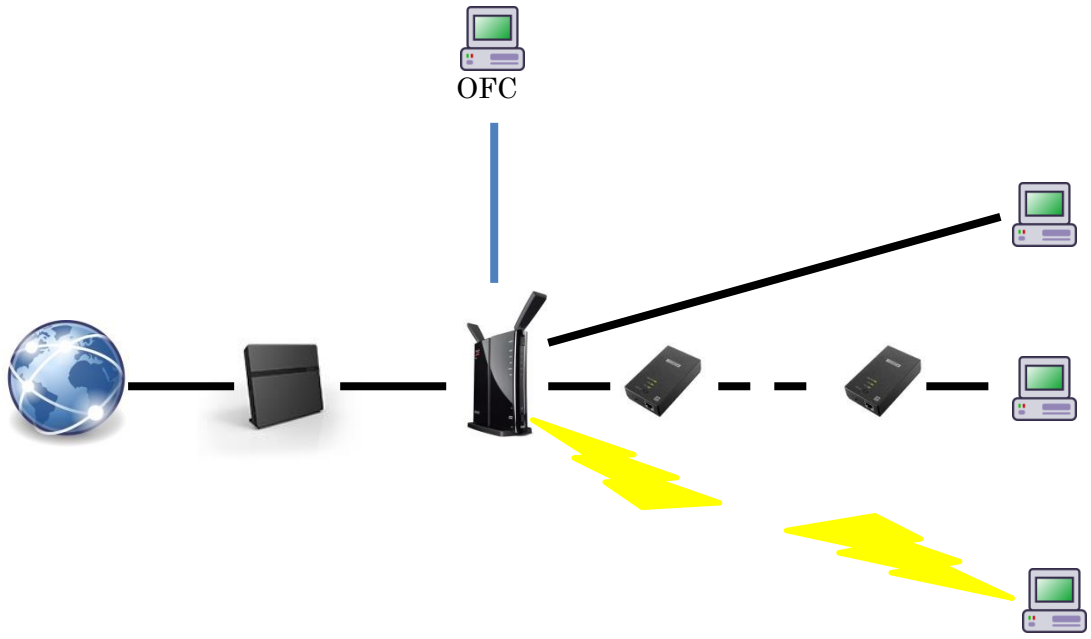

図 12:ホームネットワーク図

5.2.1. 正規ファームウェア検証結果

バッファロー社 WZR-HP-AG300H 正規ファームウェア(ver.1.73)の検証結果は下記表の通りである。

回数	有線(上り)	有線(下り)	無線(上り)	無線(下り)	PLC(上り)	PLC(下り)
1	335.5Mbps	790.6Mbps	87.81Mbps	162.0Mbps	20.66Mbps	19.40Mbps
2	313.0Mbps	675.1Mbps	80.29Mbps	134.3Mbps	26.15Mbps	22.45Mbps
3	271.3Mbps	797.2Mbps	79.82Mbps	160.9Mbps	26.65Mbps	23.18Mbps
4	304.3Mbps	797.3Mbps	86.49Mbps	146.8Mbps	26.68Mbps	24.31Mbps
5	324.6Mbps	756.5Mbps	85.91Mbps	138.3Mbps	17.98Mbps	24.22Mbps
平均	309.7Mbps	763.3Mbps	84.06Mbps	148.5Mbps	23.62Mbps	22.71Mbps

表 82:正期ファームウェア速度検証結果

5.2.2. L2 スイッチ検証結果

第 4 章にて作成した L2 スイッチプログラムの検証結果は下記表の通りである。

回数	有線(上り)	有線(下り)	無線(上り)	無線(下り)	PLC(上り)	PLC(下り)
1	50.86Mbps	48.30Mbps	31.58Mbps	30.87Mbps	26.19Mbps	21.77Mbps
2	50.63Mbps	48.58Mbps	31.62Mbps	31.41Mbps	28.75Mbps	24.32Mbps
3	50.53Mbps	48.27Mbps	31.76Mbps	32.01Mbps	29.56Mbps	25.04Mbps
4	50.65Mbps	48.51Mbps	31.97Mbps	31.98Mbps	26.46Mbps	25.86Mbps
5	50.68Mbps	48.47Mbps	31.57Mbps	31.95Mbps	26.37Mbps	24.88Mbps
平均	50.67Mbps	48.43Mbps	31.70Mbps	31.64Mbps	27.47Mbps	24.37Mbps

表 83:L2 スイッチ速度検証結果

5.2.3. QoS スイッチ検証結果

第 4 章にて作成した QoS スイッチプログラムの検証結果は下記表の通りである。

回数	有線(上り)	有線(下り)	無線(上り)	無線(下り)	PLC(上り)	PLC(下り)
1	50.40Mbps	48.00Mbps	32.44Mbps	30.85Mbps	29.73Mbps	27.60Mbps
2	50.41Mbps	47.94Mbps	32.83Mbps	30.09Mbps	29.18Mbps	25.83Mbps
3	50.43Mbps	48.07Mbps	32.28Mbps	30.73Mbps	28.51Mbps	28.60Mbps
4	50.77Mbps	48.31Mbps	31.33Mbps	31.03Mbps	28.60Mbps	29.19Mbps
5	50.46Mbps	48.16Mbps	31.08Mbps	30.09Mbps	28.61Mbps	26.65Mbps
平均	50.49Mbps	48.10Mbps	31.99Mbps	30.56Mbps	28.93Mbps	27.57Mbps

表 84:QoS スイッチ速度検証結果

第6章　研究結果と今後の課題

6.1. 研究結果

第 4 章にて抽出した「伝送能力を最大限に伝えるためには QoS の設定を利用状況に応じて設定変更していく必要がある」問題において、家庭用ブロードバンドルータを OFS として利用するならば、第 5 章の検証結果より 30Mbps 未満の回線速度であれば有効であると考えられる。何故なら、PLC での速度検証において OpenFlow 技術を利用した場合、通信速度が約 25%も改善されたからである。

しかし、有線などの高速通信が可能な伝送媒体では大幅に速度低下している。OpenFlow 技術は google 社などの様々な企業で使用されている。よって、原因は現在発売されている一般的な家庭用ブロードバンドルータでは OFS としてのスペックが足りずパフォーマンス不足であることが考えられる。

本研究により家庭用ブロードバンドルータでも 30Mbps 未満の通信速度ならば、充分に OpenFlow 技術の恩恵を得られることがわかった。

6.2. 今後の課題

本研究では家庭用ブロードバンドルータを OFS として利用することにより、30Mbps 未満の通信速度では有効であることがわかった。しかし、それ以上の回線速度においては有効であるか判明していない。近年、スマートフォンを始めとした小型機器の性能向上が著しいことから家庭用ブロードバンドルータも性能向上することが期待できる。したがって、高速通信での検証は家庭用ブロードバンドルータの性能向上された後に検証できると考える。

さらに、OpenFlow 技術を利用した場合、設計者と管理者がいないホームネットワーク内でメカニズムが動き続けるための問題点が 2 つあると考える。

1 点目の問題は OFC の設置についてである。ホームネットワークでは情報配信線を新規に行うことが望ましくないが、OFC 設置に伴い配信線を 1 本増設する必要がある。さらに、OFC の設置スペースが必要になる。この問題点の解決方法を 2 点提案する。

1 点目は OFC のサイズを出来るだけ小さくし、専有するスペースを減らす方法である。本研究はノートパソコンを OFC としたが Raspberry Pi(https://www.raspberrypi.org/)などのシングルポートコンピュータを使用することにより、専有スペースを大幅に減らすことが可能である。OFC と OFS を 1 台のコンピュータとして動かすには、現在発売されているシングルポートコンピュータではポート数が少ないので、OFS とするのは難しい。

2 点目は IaaS(Infrastructure as a Service)を使用し、クラウド上に OFC を構築する方法である。この場合、OFC の設置スペースはなくなるがクラウドサービスのレンタル料金が

必要になる。さらに、OpenFlow セキュアチャンネルの解析は容易なことから、セキュリティ面を考慮すると VPN 接続が望ましいと考える。その場合、OFS も VPN 接続に対応している必要がある。IPsec に対応しているクラウドサービスとして NTT コミュニケーションズ株式会社の Cloudn(http://www.ntt.com/cloudn/index.html)などがある。Cloudn の場合、月額料金は最大 3672 円となる。(2015 年 7 月 20 日現在)

2 点目の問題は本研究のプログラムでは OFS の配線接続が変更された場合、QoS が正しく動作しなくなる点である。この問題の解決方法も 2 点提案する。

1 点目は設定画面を GUI で作成する方法である。ポートの配線接続を変更した場合、GUI 画面上より QoS 設定を変更する。各ポートに対して優先度を選択する形式にすれば、ユーザにも QoS 設定の変更が可能だと考える。

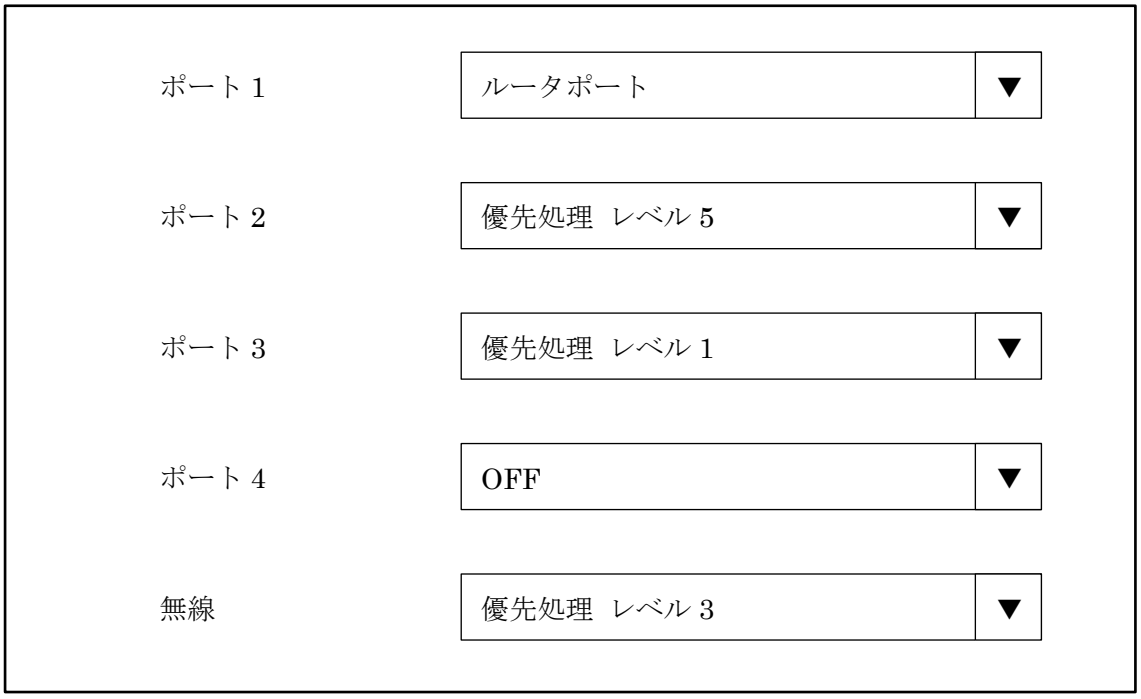

図 13:GUI 設定画面例

2 点目はプログラムを変更し、MAC アドレスを基に各機器の QoS を設定する方法である。この場合は新規に機器が接続された場合、MAC アドレスを登録する必要があり、ユーザには難しいと考える。

また、本研究では OFS は 1 台だが、OFS を複数台組み合わせたホームネットワーク構築はまだ出来ていない。これらの問題を今後の課題としたい。

第7章　謝辞

本研究を進めるにあたり、終始適切な助言を賜り、また熱心なご指導を頂きました丹康雄教授に深く感謝致します。

参考文献

[1] 株式会社インプレス, “OpenFlow ～今までの概念を覆す新しいネットワークの実現～,” 02 02 2012. [オンライン]. Available: http://thinkit.co.jp/book/2012/02/01/3150.

[2] 株式会社技術評論社, “こんな夜中に OpenFlow でネットワークをプログラミング！,” 07 09 2011. [オンライン]. Available: http://gihyo.jp/dev/serial/01/openflow_sd.

[3] “QoS (Quality of Service), ” [オンライン]. Available: http://www.infraexpert.com/study/telephony6.html.

[4] aki, “ITBOOKS,” [オンライン]. Available: http://www.itbook.info/.

[5] RYU project team, “Ryubook 1.0 ドキュメント,” [オンライン]. Available: http://osrg.github.io/ryu-book/ja/html/#.

[6] 丹康雄, ユビキタス技術 ホームネットワークと情報家電, オーム社, 2004.

[7] 馬場達也、大上貴充、関山宣考、高畑知也, OpenFlow 徹底入門 SDN を実現する技術と知識, 翔泳社, 2013.

[8] 高宮安仁、鈴木一哉, クラウド時代のネットワーク技術 OpenFlow 実践入門, 技術評論社, 2013.

[9] あきみち、宮永直樹、岩田淳, マスタリング TCP/IP OpenFlow 編, オーム社, 2013.

[10] 一般社団法人 情報通信技術委員会, “ホームネットワーク通信インターフェース実装ガイドライン,” 2013.

[11] 児玉泰伸, “ホームネットワーク環境における異種データリンク接続の QoS 保証に関する研究,” 2012.

[12] 丹康雄, “ECHONET Lite の最新技術解説と国際動向,” 2013.

[13] Open Networking Foundation, “OpenFlow Switch Specification 1.0.0,” 2009.

[14] Open Networking Foundation, “OpenFlow Switch Specification 1.1.0,” 2011.

[15] Open Networking Foundation, “OpenFlow Switch Specification 1.2,” 2011.

[16] Open Networking Foundation, “OpenFlow Switch Specification 1.3.0,” 2012.

付録

ユニキャストプログラム

```
1  from ryu.base import app_manager
2  from ryu.controller import ofp_event
3  from ryu.controller.handler import MAIN_DISPATCHER
4  from ryu.controller.handler import set_ev_cls
5  from ryu.ofproto import ofproto_v1_0
6  from ryu.lib.packet import packet
7
8  HOST1_PORT = 1
9  HOST2_PORT = 2
10
11 class Unicast(app_manager.RyuApp):
12     OFP_VERSIONS = [ofproto_v1_0.OFP_VERSION]
13
14     def __init__(self, *args, **kwargs):
15         super(Unicast, self).__init__(*args, **kwargs)
16
17     def unicast_flow(self, datapath):
18         ofproto = datapath.ofproto
19         parser = datapath.ofproto_parser
20
21         match = parser.OFPMatch(in_port=HOST1_PORT)
22         actions = [parser.OFPActionOutput(HOST2_PORT)]
23
24         mod = parser.OFPFlowMod(datapath=datapath, match=match, cookie=0,
25                                 command=ofproto.OFPFC_ADD,actions=actions)
26
27         datapath.send_msg(mod)
28
29         match = parser.OFPMatch(in_port=HOST2_PORT)
30         actions = [parser.OFPActionOutput(HOST1_PORT)]
31
32         mod = parser.OFPFlowMod(datapath=datapath, match=match, cookie=0,
```

```
33                                          command=ofproto.OFPFC_ADD, actions=actions)
34
35              datapath.send_msg(mod)
36
37      @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
38      def _packet_in_handler(self, ev):
39              msg = ev.msg
40              datapath = msg.datapath
41
42              self.unicast_flow(datapath)
```

L2 スイッチプログラム

```
1  from ryu.base import app_manager
2  from ryu.controller import mac_to_port
3  from ryu.controller import ofp_event
4  from ryu.controller.handler import MAIN_DISPATCHER
5  from ryu.controller.handler import set_ev_cls
6  from ryu.ofproto import ofproto_v1_0
7  from ryu.lib.mac import haddr_to_bin
8  from ryu.lib.packet import packet
9  from ryu.lib.packet import ethernet
10
11 MAC_IDLE_TIME = 300
12
13
14 class SwitchHub(app_manager.RyuApp):
15      OFP_VERSIONS = [ofproto_v1_0.OFP_VERSION]
16
17      def __init__(self, *args, **kwargs):
18              super(SwitchHub, self).__init__(*args, **kwargs)
19              self.mac_to_port = {}
20
21      def add_flow(self, datapath, in_port, src, dst, actions):
22              ofproto = datapath.ofproto
23              parser = datapath.ofproto_parser
```

```
24
25              match = parser.OFPMatch(in_port=in_port, dl_src=src, dl_dst=dst)
26
27              mod = parser.OFPFlowMod(datapath=datapath, match=match,
28                                      cookie=0, idle_timeout=MAC_IDLE_TIME,
29                                      command=ofproto.OFPFC_ADD,actions=actions)
30
31              datapath.send_msg(mod)
32
33      def del_flow(self, datapath, mac):
34              ofproto = datapath.ofproto
35              parser = datapath.ofproto_parser
36
37              match = parser.OFPMatch(dl_src=mac)
38
39              mod = parser.OFPFlowMod(datapath=datapath, match=match, cookie=0,
40                                      command=ofproto.OFPFC_DELETE)
41
42              datapath.send_msg(mod)
43
44      def modify_flow(self, datapath, mac, port):
45              ofproto = datapath.ofproto
46              parser = datapath.ofproto_parser
47
48              match = parser.OFPMatch(dl_dst=mac)
49
50              actions = [datapath.ofproto_parser.OFPActionOutput(port)]
51
52              mod = parser.OFPFlowMod(datapath=datapath, match=match,
53                                      cookie=0, idle_timeout=MAC_IDLE_TIME,
54                                      command=ofproto.OFPFC_MODIFY,actions=actions)
55
56              datapath.send_msg(mod)
57
58      @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
59      def _packet_in_handler(self, ev):
```

```
60              msg = ev.msg
61              datapath = msg.datapath
62              ofproto = datapath.ofproto
63
64              pkt = packet.Packet(msg.data)
65              eth = pkt.get_protocol(ethernet.ethernet)
66
67              dst = eth.dst
68              src = eth.src
69
70              dpid = datapath.id
71              self.mac_to_port.setdefault(dpid, {})
72
73              self.mac_to_port[dpid].setdefault(src, msg.in_port)
74
75              if self.mac_to_port[dpid][src] != msg.in_port:
76                      self.del_flow(datapath, haddr_to_bin(src))
77                      self.modify_flow((datapath, haddr_to_bin(src), msg.in_port)
78                      self.mac_to_port[dpid][src] = msg.in_port
79
80              if dst in self.mac_to_port[dpid]:
81                      out_port = self.mac_to_port[dpid][dst]
82              else:
83                      out_port = ofproto.OFPP_FLOOD
84
85              actions = [datapath.ofproto_parser.OFPActionOutput(out_port)]
86
87              if out_port != ofproto.OFPP_FLOOD:
88                      self.add_flow(datapath, msg.in_port, haddr_to_bin(src),
89                                              haddr_to_bin(dst), actions)
90
91              data = None
92              if msg.buffer_id == ofproto.OFP_NO_BUFFER:
93                      data = msg.data
94
95              out = datapath.ofproto_parser.OFPPacketOut(
```

```
96                  datapath=datapath, buffer_id=msg.buffer_id, in_port=msg.in_port,
97                  actions=actions, data=data)
98              datapath.send_msg(out)
```

トラヒックモニタプログラム

```
 1  import switch_hub
 2
 3  from operator import attrgetter
 4
 5  from ryu.controller import ofp_event
 6  from ryu.controller.handler import MAIN_DISPATCHER
 7  from ryu.controller.handler import set_ev_cls
 8  from ryu.lib import hub
 9
10  POLLING_TIME = 10
11
12  class Monitor(switch_hub.SwitchHub):
13      def __init__(self, *args, **kwargs):
14          super(Monitor, self).__init__(*args, **kwargs)
15          self.datapath = {}
16          self.monitor_thread = hub.spawn(self._monitor)
17
18      @set_ev_cls(ofp_event.EventOFPStateChange,MAIN_DISPATCHER)
19      def _state_change_handler(self, ev):
20          self.datapath = ev.datapath
21
22      def _monitor(self):
23          while True:
24              if self.datapath:
25                  self.stats_request(self.datapath)
26              hub.sleep(POLLING_TIME)
27
28      def stats_request(self, datapath):
29          ofproto = datapath.ofproto
30          parser = datapath.ofproto_parser
```

```
31
32              req = parser.OFPPortStatsRequest(datapath, 0, ofproto.OFPP_NONE)
33              datapath.send_msg(req)
34
35      @set_ev_cls(ofp_event.EventOFPPortStatsReply, MAIN_DISPATCHER)
36      def _port_stats_reply_handler(self, ev):
37              body = ev.msg.body
38
39              for stat in sorted(body, key=attrgetter('port_no')):
40                      bytes = stat.rx_bytes + stat.tx_bytes
41                      print 'port:%x bytes:%d' % (stat.port_no, bytes)
42              print '----------------------------------'
```

QoS スイッチプログラム

```
1  import switch_hub
2
3  from ryu.controller import ofp_event
4  from ryu.controller.handler import MAIN_DISPATCHER
5  from ryu.controller.handler import set_ev_cls
6  from ryu.lib import hub
7  from ryu.lib.mac import haddr_to_bin
8  from ryu.lib.packet import ether_types as types
9
10 INIT_TIME       = 10
11 POLLING_TIME = 600
12 REPLY_TIME     = 3
13
14 QOS_FLAG          = 'flag'
15 QOS_OFF           = 0
16 QOS_ON            = 1
17 QOS_IDLE_TIME = 120
18 QOS_PRIORITY    = 65535
19
20 TRAFFIC         = 'traffic'
21 TV_TRAFFIC    = 300 * (10 ** 6)
```

```
22 PLC_TRAFFIC = 200 * (10 ** 6)
23 WL_TRAFFIC   = 100 * (10 ** 6)
24
25 TOS        = 'tos'
26 TV_TOS    = 32
27 PLC_TOS = 16
28 WL_TOS    = 8
29
30 ROUTER_MAC    = '00:00:00:00:00:01'
31 ROUTER_PORT = 1
32
33 TV_PORT        = 2
34 PLC_PORT       = 3
35 WL11A_PORT    = 6
36 WL11G_PORT    = 5
37
38 class Qos(switch_hub.SwitchHub):
39     def __init__(self, *args, **kwargs):
40         super(Qos, self).__init__(*args, **kwargs)
41
42         self.datapath = {}
43
44         self.base = {TV_PORT:TV_TRAFFIC,
45                      PLC_PORT:PLC_TRAFFIC,
46                      WL11A_PORT:WL_TRAFFIC,
47                      WL11G_PORT:WL_TRAFFIC}
48
49         self.traffic = {TV_PORT:0,
50                         PLC_PORT:0,
51                         WL11A_PORT:0,
52                         WL11G_PORT:0}
53
54         self.qos = {TV_PORT:{TRAFFIC:0, QOS_FLAG:QOS_OFF, TOS:TV_TOS},
55                     PLC_PORT:{TRAFFIC:0, QOS_FLAG:QOS_OFF, TOS:PLC_TOS},
56                     WL11A_PORT:{TRAFFIC:0, QOS_FLAG:QOS_OFF, TOS:WL_TOS},
57                     WL11G_PORT:{TRAFFIC:0, QOS_FLAG:QOS_OFF, TOS:WL_TOS}}
```

```
58
59          self.monitor_thread = hub.spawn(self._qos_monitor)
60
61      @set_ev_cls(ofp_event.EventOFPStateChange,MAIN_DISPATCHER)
62      def _state_change_handler(self, ev):
63          self.datapath = ev.datapath
64
65      def _qos_monitor(self):
66          # Initialize Setting
67          while True:
68              if self.datapath:
69                  self.stats_request(self.datapath,
70                                     self.datapath.ofproto.OFPP_NONE)
71                  self.renew_traffic()
72                  break
73              hub.sleep(INIT_TIME)
74
75          hub.sleep(POLLING_TIME)
76          while True:
77              self.stats_request(self.datapath,
78                                 self.datapath.ofproto.OFPP_NONE)
79              hub.sleep(REPLY_TIME)
80              self.qos_setting()
81              self.renew_traffic()
82              hub.sleep(POLLING_TIME)
83
84      def stats_request(self, datapath, port):
85          req = datapath.ofproto_parser.OFPPortStatsRequest(datapath, 0, port)
86          datapath.send_msg(req)
87
88      @set_ev_cls(ofp_event.EventOFPPortStatsReply, MAIN_DISPATCHER)
89      def _port_stats_reply_handler(self, ev):
90          body = ev.msg.body
91          for stat in body:
92              if stat.port_no in self.traffic.keys():
93                  self.traffic[stat.port_no] = stat.rx_bytes + stat.tx_bytes
```

```
94
95      def qos_setting(self):
96          for port in self.traffic.keys():
97              if self.traffic[port] - self.qos[port][TRAFFIC] > self.base[port]:
98                  if self.qos[port][QOS_FLAG] == QOS_OFF:
99                      self.add_qos(port)
100             elif self.qos[port][QOS_FLAG] == QOS_ON:
101                 self.del_qos(port)
102
103     def add_qos(self, port):
104         ofproto = self.datapath.ofproto
105         parser = self.datapath.ofproto_parser
106
107         match = parser.OFPMatch(in_port=port, dl_type=types.ETH_TYPE_IP,
108                                 dl_dst=haddr_to_bin(ROUTER_MAC))
109
110         actions = [parser.OFPActionSetNwTos(self.qos[port][TOS]),
111                    parser.OFPActionOutput(ROUTER_PORT)]
112
113         mod = parser.OFPFlowMod(datapath=self.datapath,
114                                 match=match, cookie=0,
115                                 command=ofproto.OFPFC_ADD,
116                                 idle_timeout=QOS_IDLE_TIME,
117                                 priority=QOS_PRIORITY,
118                                 flags=ofproto.OFPFF_SEND_FLOW_REM,
119                                 actions=actions)
120
121         self.datapath.send_msg(mod)
122
123         self.qos[port][QOS_FLAG] = QOS_ON
124
125     def del_qos(self, port):
126         ofproto = self.datapath.ofproto
127         parser = self.datapath.ofproto_parser
128
129         match = parser.OFPMatch(in_port=port, dl_type=types.ETH_TYPE_IP,
```

```
130                                         dl_dst=haddr_to_bin(ROUTER_MAC))
131
132             mod = parser.OFPFlowMod(datapath=self.datapath,
133                                         match=match, cookie=0,
134                                         command=ofproto.OFPFC_DELETE,
135                                         idle_timeout=QOS_IDLE_TIME,
136                                         priority=QOS_PRIORITY)
137
138             self.datapath.send_msg(mod)
139
140         @set_ev_cls(ofp_event.EventOFPFlowRemoved, MAIN_DISPATCHER)
141         def _flow_removed_handler(self, ev):
142             msg = ev.msg
143             port = msg.match.in_port
144             self.qos[port][QOS_FLAG] = QOS_OFF
145             if msg.reason == self.datapath.ofproto.OFPRR_IDLE_TIMEOUT:
146                 self.stats_request(self.datapath, port)
147                 hub.sleep(REPLY_TIME)
148                 self.qos[port][TRAFFIC] = self.traffic[port]
149
150         def renew_traffic(self):
151             for port in self.traffic.keys():
152                 self.qos[port][TRAFFIC] = self.traffic[port]
```

家庭用ホームネットワーク 無線設定コンフィグ

```
1  config wifi-device   radio0
2          option type        mac80211
3          option channel   auto
4          option macaddr   4c:e6:76:c3:1e:7e
5          option hwmode     11ng
6          option htmode     HT40+
7          list ht_capab     SHORT-GI-40
8          list ht_capab     TX-STBC
9          list ht_capab     RX-STBC1
10         list ht_capab     DSSS_CCK-40
```

```
11              # REMOVE THIS LINE TO ENABLE WIFI:
12              # option disabled 1
13
14  config wifi-iface
15              option device     radio0
16              option network    lan
17              option mode        ap
18              option ssid       jaist_research_11g
19              option hidden     1
20              option encryption psk2+aes
21              option key         jaist_research_11g
22
23  config wifi-device    radio1
24              option type       mac80211
25              option channel    auto
26              option macaddr    4c:e6:76:c3:1e:7f
27              option hwmode      11na
28              option htmode     HT40+
29              list ht_capab     SHORT-GI-40
30              list ht_capab     TX-STBC
31              list ht_capab     RX-STBC1
32              list ht_capab     DSSS_CCK-40
33              # REMOVE THIS LINE TO ENABLE WIFI:
34              # option disabled 1
35
36  config wifi-iface
37              option device     radio1
38              option network    lan
39              option mode        ap
40              option ssid       jaist_research_11a
41              option hidden     1
42              option encryption psk2+aes
43              option key         jaist_research_11a
```