OPEN NETWORKING FOUNDATION

Optical Transport Protocol Extensions

Version1.0
March 15, 2015

ONF TS-022

OpenFlow

ONF Document Type: Technical Specification
ONF Document Name: Optical Transport Protocol Extensions

Disclaimer

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Scope and Objectives

This document specifies a set of recommended extensions to the OpenFlow-Switch protocol to support the requirements for control of optical transport networks and equipment. The requirements have been identified in [1].

These extensions have been defined using the experimental extension mechanism supported in OpenFlow-Switch to add messages and attributes to the base protocol specification. The extensions are identified using the experimenter ID assigned to the Optical Transport Working Group. While these extensions are being specified in a standalone Technical Specification, they may in future be merged into the main OpenFlow-Switch specification.

This document further specifies procedures for the use of OpenFlow-Switch base specification features in the context of optical transport networks and equipment, including interpretation of attributes that are described in the base specification using a packet switch framework.

This document does not consider OF-Config extensions at this time; however future versions may also incorporate recommendations for extensions to OF-Config or other ONF protocols.

1.2 Common Terms and Abbreviations

NE	Network Element	FCS	Frame Check Sequence
OAM	Operations, Administration and Maintenance	OCh	Optical Channel
OF	OpenFlow	ODU	Optical Data Unit
OH	Overhead	OMS	Optical Multiplex Section
OTWG	Optical Transport Working Group	OPS	Optical Physical Section
OUI	Organizationally Unique Identifier	OTS	Optical Transmission Section
TCM	Tandem Connection Monitoring	OTU	Optical channel Transport Unit
TDM	Time Division Multiplexing	TTI	Trail Trace Identifier

2 Initial Extensions to OpenFlow-Switch Protocol

This section presents initial extensions to OpenFlow-Switch protocol in support of Optical Transport. Follow-up versions will address additional functions, such as control of OAM/Monitoring, Protection and Multilayer Adaptation.

2.1 Basis in Optical Transport Requirements

Note: Version 1.0 of this document addresses extensions to OF-Switch protocol, but not OF-Config. While some functions addressed by the extensions involve OAM, no conclusion is made as to whether similar functionality may be supported in OF-Config as well.

The recommendations defined in this document are based on requirements for OpenFlow/SDN control of optical transport networks identified in [1]. Some of the key requirements that are addressed are:

- Ability of the controller to support discovery and reporting of adjacent NEs through the OF-Switch protocol. L0 and L1 network elements switching on wavelengths or timeslots do not insert or remove packets from the data and so do not support the Packet_In/Packet_Out functionality by which discovery and reporting of adjacent network elements is possible in a packet network. (See [1] Section 4.3, R-4.3.1).
- Ability to support L0 and L1 circuit technologies, as well as packet technologies, including specification of appropriate match elements and port attributes for L0 and L1 optical transport networks. Detailed attributes for L0 and L1 technologies are captured in the Information Model [IM]. (See [1] Section 4.1, R-4.1.1, R-4.1.2 and R-4.1.4).
- Support of carrier reliability mechanisms for the dataplane such as monitoring and protection mechanisms that can ensure that services meet service level agreements negotiated between a carrier and its customers for transport networks. (See [1] Section 4.4, R-4.4.1).
- Support of network elements handling multiple technology layers according to transport network layering models, for consistency with carrier network operations. Layer relationships are captured in the Information Model [2]. (See [1] Section 4.1, R-4.1.3).

The recommended extensions in this document build on the work done in OpenFlow 1.4, especially the extensions to allow definition of new port types and port attributes. However, the extensions go beyond the optical port type defined in OpenFlow 1.4. The scope of the extensions is intended to be larger to cover many kinds of interfaces, including

- optical/L0, TDM/L1, packet transport, OF 1.4 specifies an OPTICAL port type to model optical characteristics
- line, section, path and client interfaces
- support of multiple layers of switching and signal adaptation with the ability to identify supported clients and available adaptations

2.2 Match/Action for Support of OTN Optical and Electrical Connections

2.2.1 Description

Today's networks support two different traffic modes: Packet and Circuit. Packet switching identifies a signal by fields in the packet header. Circuit switching identifies a signal by its position in time or space, often with the aid of defined header information in the case of digital electrical signals such as OTN ODU.

While existing OpenFlow supports Packet switching technologies, it does not have support for circuit switching. Extensions are necessary to identify a circuit-switched signal and execute appropriate actions.

The proposal for match and action extensions provides the following:

- match extensions that identify a signal using the attributes of an OCh (i.e. Grid, Channel Spacing, center frequency, channel mask) for Layer 0
- match extensions that identify a signal using the attributes of an ODUj/k (i.e. ODU type, ODU Tributary Slot, ODU Tributary Port Number) for Layer 1

Action extensions are not proposed; instead the existing SET_FIELD mechanism is used to determine the attributes of the egress signal.

These extensions have been identified under JIRA ticket EXT-445 and 446 in the Extensibility Working Group and corresponding protocol extensions are defined in this document.

2.3 Port Attributes for Support of OTN Optical and Electrical Connections

2.3.1 Description

As discussed above, existing OpenFlow supports Packet switching technologies, but it does not have support for circuit switching. Extensions are necessary to identify port types suited for L0 and L1 circuit switching control using OTN standard interfaces.

The proposed port attribute extensions for Layer 0 and Layer 1 consist of the following:

- Port capability extensions to identify the capabilities of OTS and OPS ports. This includes understanding the signals supported and granularity of switching available, as well as basic compatibility information for Layer 0 signals based on ITU-T application code standards.
- Port capability extensions to identify the capabilities of OTU ports. This includes understanding the type of signal supported by the port as well as the client signals supported by the port.

These extensions have been identified under JIRA ticket EXT-445 and 446 in the Extensibility Working Group and corresponding protocol extensions are defined in this document.

2.4 Adjacency Discovery

2.4.1 Description

For optical transport networks, it is not possible for the controller to ask for a discovery packet to be added into the datapath alongside other flows. It is also not possible to have discovery packets passed up from the switch isolated for other flows. Instead, the switch will often support a separate associated channel with shared routing and shared fate for exchanging local and remote identification (“adjacency discovery”) across the optical transport interface.

One of the possible mechanisms for adjacency discovery for OTN transport networks is the in-band exchange of identifier information defined in ITU-T Recommendation G.7714.1 [3]. For adjacency discovery, the identifier (known as a “trail trace identifier” or TTI) that is sent out of the interface should be configurable by a controller. The TTI received across the interface should also be retrievable by the controller. Once known by the controller, the TTI can be used to develop or confirm network topology.

As described in G.798.1 [4], Figure 1 below depicts an example of an OTN layer stack with OAM entities. Table 1 that follows describes the abilities of each layer to support TTI overhead as well as the associated discovery, as not all layers provide this functionality. For example, there is ongoing work within the ITU to add discovery capabilities to OCh links.

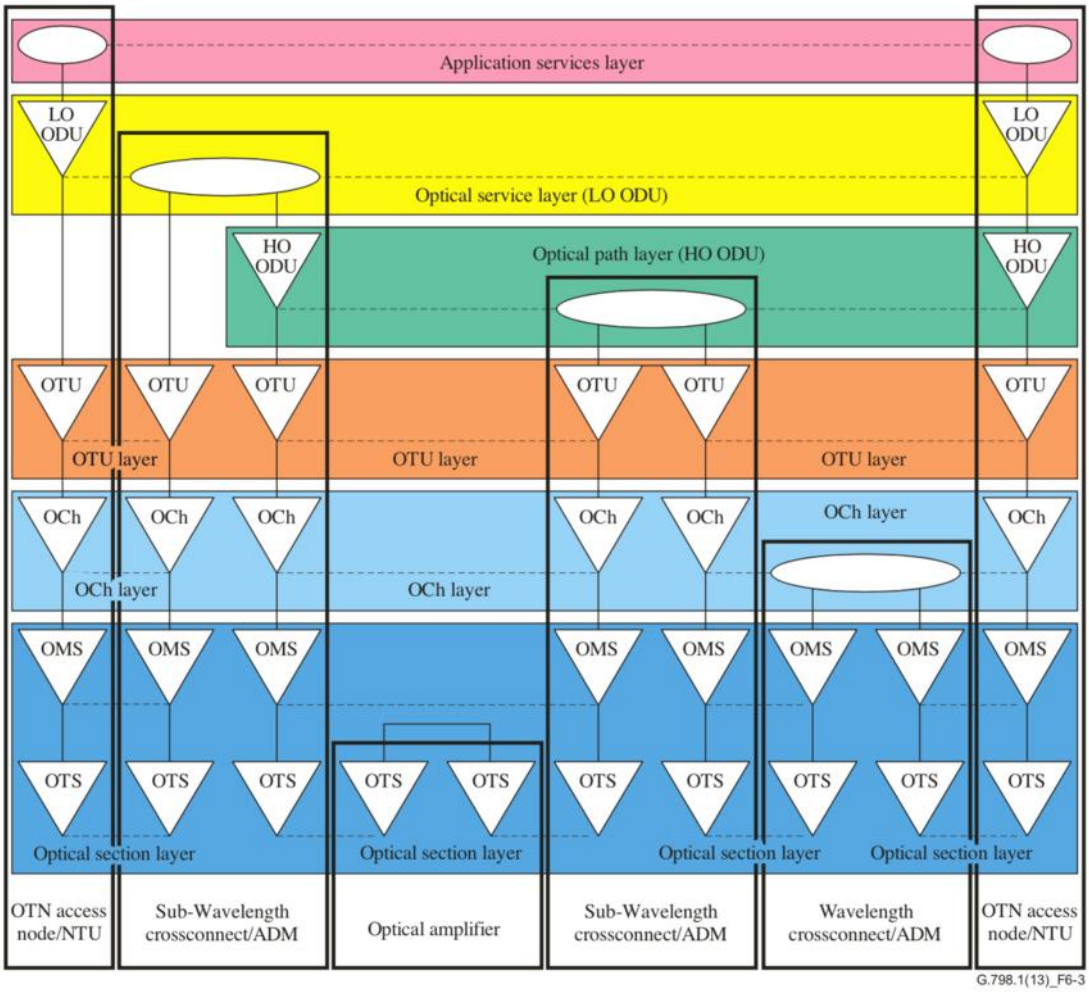


Figure 1: OTN Layer Stack with OAM Entities [4]

Table 1: Trail Message and Trail Trace Identifier (TTI) by Connection

Connection	TTI OH	Discovery Messages
ODUk	Yes	Format Defined
OTUk	Yes	Format Defined
OCh	Pending (Q11/15)	Format Defined
OMSn	Yes (Vendor Specific)	Format Defined

OTSn	Yes	Format Defined
OPSn (Single lambda)	No OH defined, use client	N.A.

The extension proposed for adjacency discovery allows the switch to provide the controller with the discovery information it has received from the remote link end. It also provides the optional ability for the controller to specify the discovery information it sends to the remote end, as well as the information it should expect to receive and the actions to be taken if it receives inconsistent information. This can be used to increase the efficiency of the process.

The protocol extension for adjacency discovery has been defined to be applicable to multiple technologies and discovery mechanisms. These extensions have been identified under JIRA ticket EXT-449 in the Extensibility Working Group and corresponding protocol extensions are defined in this document.

2.5 Future Work

A number of potential future areas of extension or clarification have been identified and are under discussion:

- Extensions to support OAM/Monitoring of optical network links (EXT-450) – OAM/monitoring is a key function in carrier transport networks in order to verify network health and performance.
- Extensions to support Connection Protection (EXT-448) – transport networks typically support rapid protection switching functions that guarantee recovery from link failure within a set time in order to meet carrier Service Level Agreements.
- Extensions to support Multilayer Connections (EXT-447) – transport network connections commonly involve adaptation of the client signal into one or more network sublayers to support aggregation of traffic and efficient packing of optical links.
- CVNI support: OpenFlow can be used in a CVNI (Control Virtual Network Interface) context, where it is exchanged between a parent controller and child controller rather than controller to network infrastructure layer. For the CVNI case, issues have been raised concerning the use of the OpenFlow control channel when controlling multiple logical switches subtending a single child controller (EXT-492) and the use of OpenFlow to convey abstracted network topology from the child controller to the parent controller.

3 OF-S Protocol Extensions for Optical Transport

3.1 Flow Match Fields and Actions

This section specifies extensions to flow match fields and actions for optical transport.

Experimental flow match extensions are identified by setting oxm_class to OFPMC_EXPERIMENTER, the inclusion of experimenter ID in the OXM TLV and allocation of specific code points for the oxm_field. The experimenter ID used for optical transport has

been reserved for Optical Transport Extensions and falls within the ONF-Managed range of IDs. The experimenter ID and code points are defined as follows:

```
#define OFP_OTWG_EXPERIMENTER_ID 0xFF000007

enum ofp_experimenters_field {
    OFPXMT_EXP_ODU_SIGTYPE     = 2,
    OFPXMT_EXP_ODU_SIGID       = 3,
    OFPXMT_EXP_OCH_SIGTYPE     = 4,
    OFPXMT_EXP_OCH_SIGID       = 5,
};
```

For each match field extension, a payload format also needs to be defined. For OFPXMT_EXP_ODU_SIGTYPE , the following payload format will be used:

```
struct ofp_oxm_exp_ODU_sigtype {
    uint32_t oxm_header;   /* oxm_class = OFPXMC_EXPERIMENTER */
    uint32_t experimenter; /* Experimenter ID which takes the same
                            * form as in struct ofp_experimenter_header. */
    uint8_t sigtype;       /* ODU Signal Type */
};
OFP_ASSERT(sizeof(struct ofp_oxm_exp_ODU_sigtype) == 9);
```

For sigtype, one of the values from ofp_odu_signal_type or enumerations defined in the port description section is used.

For OFPXMT_EXP_OCH_SIGTYPE, the following payload format will be used:

```
struct ofp_oxm_exp_OCH_sigtype {
       uint32_t oxm_header;       /* oxm_class = OFPXMC_EXPERIMENTER */
       uint32_t experimenter;     /* Experimenter ID which takes the same
                                  form as in struct ofp_experimenter_header. */
       uint8_t sigtype;           /* OCH Signal Type */
};
OFP_ASSERT(sizeof(struct ofp_oxm_exp_OCH_sigtype) == 9);
```

For sigtype, one of the values from ofp_och_signal_type enumerations defined in the port description section is used.

```
/* OFPXMT_EXP_ODU_SIGID Payload format */
struct ofp_oxm_exp_ODU_sigid {
    uint32_t oxm_header;       /* oxm_class = OFPXMC_EXPERIMENTER */
    uint32_t experimenter;     /* Experimenter ID which takes the same
                                * form as in struct ofp_experimenter_header. */
    uint16_t tpn;           /* Tributary Port Number */
    uint16_t tslen;         /* Number of Tributary Slots included in tsmap,
                                * based on the Server ODU type and TS Granularity. */
    uint8_t tsmap[0];          /* Tributary slot bitmap */
};
OFP_ASSERT(sizeof(struct ofp_oxm_exp_ODU_sigid) == 12);

/* OFPXMT_EXP_OCH_SIGID Payload format */
struct ofp_oxm_exp_OCH_sigid {
    uint32_t oxm_header;       /* oxm_class = OFPXMC_EXPERIMENTER */
    uint32_t experimenter;     /* Experimenter ID which takes the same
                                * form as in struct ofp_experimenter_header. */
    uint8_t grid_type;         /* Grid Type */
    uint8_t chl_spacing;       /* Channel spacing */
    uint16_t n;                /* n is used to calculate the frequency as in
```



```
                                * [ITU G.694.1]
                                * Frequency(THz)= 193.1 THz + n*chl_spacing (THz)*/
    uint16_t m;                /* m is used to identify the slot width as defined in
                                * [ITU G.694.1],
                                * Slot Width (GHz) = m*12.5 (GHz)
                                * For fix grid networks, m=1 */

};
OFP_ASSERT(sizeof(struct ofp_oxm_exp_OCH_sigid) == 14);

enum ofp_grid_type {
    OFPGRIDT_RES    = 0,
    OFPGRIDT_DWDM   = 1,
    OFPGRIDT_CWDM   = 2,
    OFPGRIDT_FLEX   = 3          /* Note: Flex grid pending further discussion */
}

enum ofp_chl_spacing {
    OFPCS_RES       = 0,
    OFPCS_100GHZ    = 1,
    OFPCS_50GHZ     = 2,
    OFPCS_25GHZ     = 3,
    OFPCS_12P5GHZ   = 4,   /* 12.5 GHZ */
    OFPCS_6P25GHZ   = 5    /* 6.25 GHZ */
}
```

Note: Further details on the use of tsmap can be found in [11], Section 6.1.

Specific action extensions are not necessary as the Match fields defined use OXM TLVs. The OpenFlow SET_FIELD action enables use of the OXM TLV format to specify the overwrite data to be applied.

3.2 Port Description Extensions for OTN Ports

Port attribute extensions for optical transport are defined here for both OF 1.3 implementation using the experimenter extension mechanism and experimenter messages, and for OF 1.4 implementation using the port attribute extensibility defined in OF 1.4.

3.2.1 Port Description Extensions Using Experimenter Messages

The extensions specified in this section allow the use of OF1.3 as the base protocol, taking into account that OF1.3 does not support port attribute extensibility. The format of extensions defined in this section is designed to be consistent with extensions for OF1.4 and later, taking into account the need to use experimenter messages.

The OF1.3 does not support a mechanism to implement the OTWG extensions for port description directly. Instead, experimenter multipart message is used to provide the information in the OTWG extended port description object.

The OTWG Optical Transport port extensions to Openflow protocol version 1.3 will use an experimenter ID assigned to OTWG and code points as defined as follows:

```
#define OFP_OTWG_EXPERIMENTER_ID 0xFF000007
```



```
/* Port description property types.*/
enum ofp_experimenter_multipart-type_exp {
    OFPEMPTE = 1,   /* Extended port description. */
};
```

The OTWG extended port description object should be supported in addition to (not in replacement of) the standard OF1.3 port description object.

The controller will request an OFPT_MULTIPART_REQUEST for an experimenter multipart message. The body of the message will contain:

```
struct ofp_multipart_request {
     struct ofp_header header;
     uint16_t type;       /* OFPMP_EXPERIMENTER = 0xffff */
     uint16_t flags;      /* OFPMPF_REQ_* flags. */
     uint8_t pad[4];
     uint8_t body[0];     /* Body of the request. 0 or more bytes. */
|;
OFP_ASSERT(sizeof(struct ofp_multipart_request) == 16);


/* Body for ofp_multipart_request of type OFPMP_EXPERIMENTER. */
struct ofp_experimenter_multipart_header {
     uint32_t experimenter;      /* Experimenter ID which takes the same form
                                    as in struct ofp_experimenter_header. */
     uint32_t exp_type;          /* OFPEMPTE = 1 */
     /* Experimenter-defined arbitrary additional data. */
};
OFP_ASSERT(sizeof(struct ofp_experimenter_multipart_header) == 8);
```

No additional data is provided in the body for the request message.

The multipart reply will use the extensions for optical transport encoded in the experimenter multipart reply message.

```
struct ofp_multipart_reply {
     struct ofp_header header;
     uint16_t type;        /* OFPMP_EXPERIMENTER = 0xffff*/
     uint16_t flags;       /* OFPMPF_REPLY_* flags. */
     uint8_t pad[4];
     uint8_t body[0]; /* Body of the reply. 0 or more bytes. */
};
OFP_ASSERT(sizeof(struct ofp_multipart_reply) == 16);


/* Body for ofp_multipart_reply of type OFPMP_EXPERIMENTER. */
struct ofp_experimenter_multipart_header {
     uint32_t experimenter;      /* Experimenter ID which takes the same form
                                    as in struct ofp_experimenter_header. */
     uint32_t exp_type;          /* OFPEMPTE = 1 */
     /* Experimenter-defined arbitrary additional data. */
};
OFP_ASSERT(sizeof(struct ofp_experimenter_multipart_header) == 8);
```


The data contained in the additional data field is the normal OF1.4 ofp_port with OTWG OF1.5 OTN extensions:

```
/* Description of a port */
struct ofp_experimenter_port {
     uint32_t experimenter;      /* Experimenter ID which takes the same form
                                    as in struct ofp_experimenter_header. */
     uint32_t exp_type;    /* OFPEMPTE = 1 */
     uint32_t port_no;
     uint16_t length;
     uint8_t pad[2];
     uint8_t hw_addr[OFP_ETH_ALEN];
     uint8_t pad2[2];                    /* Align to 64 bits. */
     char name[OFP_MAX_PORT_NAME_LEN];     /* Null-terminated */

     uint32_t config;          /* Bitmap of OFPPC_* flags. */
     uint32_t state;           /* Bitmap of OFPPS_* flags. */

     /* Port description property list - 0 or more properties */
     struct ofp_port_desc_prop_header properties[0];
};
OFP_ASSERT(sizeof(struct ofp_port) == 48);

struct ofp_port_desc_prop_header {
     uint16_t type;         /* OFPPDPT_OPTICAL_TRANSPORT = 2 */
     uint16_t length;       /* Length in bytes of this property. */
};
OFP_ASSERT(sizeof(struct ofp_port_desc_prop_header) == 4);
```

The optical transport port feature structures are defined as follows:

```
/* Port description property types.*/
enum ofp_port_desc_prop_type_exp {
    OFPPDPT_OPTICAL_TRANSPORT = 2,      /* Optical Transport port property. */
};

/* Optical Transport port description property. */
struct ofp_port_desc_prop_optical_transport {
     uint16_t type;                /* OFPPDPT_OPTICAL_TRANSPORT */
     uint16_t length;              /* Length  in  bytes of this property. */
     uint8_t port_signal_type;     /* Base port layer signal type - OFPPOTST_* */
     uint8_t reserved;
     unit8_t pad[ 2];
     struct ofp_port_optical_transport_feature_header features[  0];
};
OFP_ASSERT(sizeof(struct  ofp_port_desc_prop_optical_transport)  == 8);

/* Supported signal types for layer class OFPPOTL_PORT */
enum ofp_port_optical_transport_signal_type {
    OFPPOTST_OTSn           = 1,
    OFPPOTST_OMSn           = 2,
    OFPPOTST_OPSn           = 3,
    OFPPOTST_OPSM           = 4,
    OFPPOTST_OCH            = 5,
    OFPPOTST_OTU1           = 11,
    OFPPOTST_OTU2           = 12,
    OFPPOTST_OTU3           = 13,
    OFPPOTST_OTU4           = 14,
};

/*Optical Transport port feature property. */
```



```
struct ofp_port_optical_transport_feature_header {
    uint16_t feature_type; /* OFPOTPF_. */
    uint16_t length;       /* length of feature excluding padding*/

};
OFP_ASSERT(sizeof(struct ofp_port_optical_transport_feature_header) == 4);

/* Features of optical transport ports available in switch. */
enum ofp_port_optical_transport_feature_type {
    OFPPOTFT_OPT_INTERFACE_CLASS = 1,  /* Application code/ID encoding */
    OFPPOTFT_LAYER_STACK         = 2,  /* Supported signal types and adaptations */
};

/* Optical Interface Class Feature Encoding */
struct ofp_port_optical_transport_application_code {
    uint16_t feature_type;   /* Set to OFPOTPF_OPT_INTERFACE_CLASS */
    uint16_t length;        /* length of feature excluding padding*/

    uint8_t oic_type;        /* OFPOICT_* identifies the relevant standard reference
                              * specifying the application code */
    char app_code[15];       /* Null-terminated - Valid format/content for this field
                              * is as per the relevant ITU-T standard indicated by
                              * oic_type field or proprietary */
};
OFP_ASSERT(sizeof(struct ofp_port_optical_transport_application_code) == 20);

/* Supported optical interface class types. */
enum ofp_optical_interface_class_type {
    OFPOICT_PROPRIETARY    = 1 << 7,   /* Proprietary application code */

    OFPOICT_ITUT_G698_1    = 1,        /* [ITU-TG.698.1] application code */
    OFPOICT_ITUT_G698_2    = 2,        /* [ITU-TG.698.2] application code */
    OFPOICT_ITUT_G959_1    = 3,        /* [ITU-TG.959.1] application code */
    OFPOICT_ITUT_G695      = 4,        /* [ITU-TG.695] application code */
};

/* OTN Layer Stack Feature Encoding */
struct ofp_port_optical_transport_layer_stack {
    uint16_t feature_type; /* Set to OFPOTPF_LAYER_STACK */
    uint16_t length;       /* length of feature excluding padding*/
    uint8_t  pad[4];       /* Zero bytes - see above for sizing */

    struct ofp_optical_transport_port_layer_entry value[0]; /* 0 or more fields */
};
OFP_ASSERT(sizeof(struct ofp_port_optical_transport_layer_stack) == 8);

/* OTN Layer Stack Entry Encoding */
struct ofp_port_optical_transport_layer_entry {
    uint8_t layer_class;    /* OFPOTPL_* */
    uint8_t signal_type;    /* OFP(OTP/OCH/ODU/ODUCL)T_*/
    uint8_t adaptation;     /* OFPADAPT_*/
    uint8_t pad[5];         /* Align to 64-bit boundary */
};
OFP_ASSERT(sizeof(struct ofp_port_optical_transport_layer_entry) == 8);

/* Layer classes (families) supported for optical transport port. */
enum ofp_port_optical_transport_layer_class {
    OFPPOTL _PORT   = 1,           /* Class of base port layer signal types */
    OFPPOTL _OCH    = 2,           /* Class of OCH layer signal types*/
    OFPPOTL _ODU    = 3,           /* Class of ODU layer signal types*/
    OFPPOTL _ODUCLT = 4,           /* Class of ODU client layer signal types*/
```



```
};

/* Supported signal types for layer class OFPOTPL_OCH */
enum ofp_och_signal_type {
    OFPOCHT_FIX_GRID       = 1,
    OFPOCHT_FLEX_GRID      = 2,
};

/* Supported signal types for layer class OFPOTPL_ODU */
enum ofp_odu_signal_type {
    OFPODUT_ODU1           = 1,
    OFPODUT_ODU2           = 2,
    OFPODUT_ODU3           = 3,
    OFPODUT_ODU4           = 4,
    OFPODUT_ODU0           = 10,
    OFPODUT_ODU2E          = 11,
    OFPODUT_ODUfCBR        = 20,
    OFPODUT_ODUfGFPfHAO    = 21,
    OFPODUT_ODUfGFPf       = 22,
};

/* Supported signal types for layer class OFPOTPL_ODUCLT */
enum ofp_oduclt_signal_type {
    OFPODUCLT_STM16        = 1,
    OFPODUCLT_STM64        = 2,
    OFPODUCLT_STM256       = 3,
    OFPODUCLT_STM1         = 4,
    OFPODUCLT_STM4         = 5,
    OFPODUCLT_1GBE         = 6,
    OFPODUCLT_10GBE        = 7,
    OFPODUCLT_40GBE        = 8,
    OFPODUCLT_100GBE       = 9,
    OFPODUCLT_FC100        = 10,
    OFPODUCLT_FC200        = 11,
    OFPODUCLT_FC400        = 12,
    OFPODUCLT_FC800        = 13,
    OFPODUCLT_FC1200       = 14,
    OFPODUCLT_GPON         = 15,
    OFPODUCLT_XGPON        = 16,
    OFPODUCLT_IB_SDR       = 17,
    OFPODUCLT_IB_DDR       = 18,
    OFPODUCLT_IB_QDR       = 19,
    OFPODUCLT_SBCON_ESCON  = 20,
    OFPODUCLT_DVB-ASI      = 21,
    OFPODUCLT_SDI          = 22,
    OFPODUCLT_SDI1G5       = 23,
    OFPODUCLT_SDI3G        = 24,
    OFPODUCLT_ATM          = 25,
    OFPODUCLT_ETH          = 26,
    OFPODUCLT_MPLS         = 27,
    OFPODUCLT_IP           = 28,
};

/* Supported adaptations for optical transport port layer stack*/
enum ofp_adaptations_type {
    OFPADAPT_OTS_OMS       = 1,
    OFPADAPT_OMS_OCH       = 2,
    OFPADAPT_OPS_OCHr      = 3,
    OFPADAPT_OPSM_OTUk     = 4,
    OFPADAPT_OCH_OTUk      = 5,
    OFPADAPT_ODUk_ODUij    = 6,
    OFPADAPT_ODUk_ODUj21   = 7,
```



```
    OFPADAPT_ODUkh_ODUj21  = 8,
    OFPADAPT_ODU0_CBRx     = 9,
    OFPADAPT_ODUk_CBRx     = 10,
    OFPADAPT_ODUk_CBRxg    = 11,
    OFPADAPT_ODUk_RSn      = 12,
    OFPADAPT_ODUk_ATM      = 13,
    OFPADAPT_ODUk_ETH      = 14,
    OFPADAPT_ODUkh_ETH     = 15,
    OFPADAPT_ODUk_ETHPPOS  = 16,
};
```

3.2.2 Port Description Extensions Using Port Attribute Extensibility

The following section specifies port attribute extensions using port attribute extensibility supported in OpenFlow-Switch versions 1.4 and later.

The Openflow protocol version 1.4 allows for extending the port description properties via the OFPPDPT_EXPERIMENTER property that uses the following structure and fields:

```
/* Experimenter port description property. */
struct ofp_port_desc_prop_experimenter {
    uint16_t  type;              /* OFPPDPT_EXPERIMENTER. */
    uint16_t  length;            /* Length in bytes of this property. */
    uint32_t  experimenter;      /* Experimenter ID which takes the same
                                    form as in struct ofp_experimenter_header. */
    uint32_t  exp_type;          /* Experimenter defined. */

    /*  Followed by:
     * - Exactly (length - 12) bytes containing the experimenter data, then
     * - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *   bytes of all-zero bytes */
    uint32_t  experimenter_data[0];
};
OFP_ASSERT(sizeof(struct ofp_port_desc_prop_experimenter) == 12);
```

The OTWG Optical Transport port extensions to Openflow protocol version 1.4 will use an experimenter ID assigned to OTWG and code points as defined as follows:

```
#define OFP_OTWG_EXPERIMENTER_ID 0xFF000007
```

The Optical Transport port extensions will assign enumeration value OFPPDPT_OPTICAL_TRANSPORT to the *exp_type* field of the *ofp_port_desc_prop_experimenter* structure.

To allow for future extensions, the proposed optical transport port extensions themselves follow a sub-TLV structure classified into feature property groups.

```
/*Optical Transport port experimenter property. */
struct ofp_port_desc_prop_exp_optical_transport {
    uint16_t  type;              /* Set to OFPPDPT_EXPERIMENTER. */
    uint16_t  length;            /* Length in bytes of this property. */
    uint32_t  experimenter;      /* OTWG ID */
    uint32_t  exp_type;          /* Set to OFPPDPT_OPTICAL_TRANSPORT */

    uint8_t   port_signal_type;  /* Base port layer signal type - OFPOTPT_* */
    uint8_t   reserved;
```



```
    uint8_t   pad[2];

    struct ofp_port_optical_transport_feature_header features[0];
};
OFP_ASSERT(sizeof (struct ofp_port_desc_prop_exp_optical_transport) == 16);
```

3.3 Port Extensions for Adjacency Discovery

Adjacency Discovery support in OpenFlow is accomplished through a new Port Description Type. The codepoint is defined as follows:

```
enum ofp_port_desc_prop_type_exp {
      …
      OFPPDPT_ADJACENCY_DISCOVERY = 3, /* Adjacency Discovery property. */
      …
};
```

The Port Description object associated with this Type will contain SubTLVs used to:

- specify/retrieve the identity string being sent
- specify/retrieve the identity string(s) expected,
- retrieve the peer identity string(s) received, and

The format is as follows:

```
/* Common header for all port description properties. */
struct ofp_port_desc_prop_adjacency_discovery {

       uint16_t type;              /* OFPPDPT_ADJACENCY_DISCOVERY */
       uint16_t length;    /* Length  in  bytes of this property. */
       struct ofp_exp_ext_ad_id ad_id[0];
};
```

The SubTLVs used for sent, expected and received identity strings all have the same format. The types are as follows:

```
enum ofp_exp_ext_port_tlv_types {
      ...
      OFP_EXP_EXT_PORT_TLV_AD_ID_SENT = 2, /* TTI to be sent by the port */
      OFP_EXP_EXT_PORT_TLV_AD_ID_EXPECTED = 3,  /* TTI expected to be rcvd on port */
      OFP_EXP_EXT_PORT_TLV_AD_ID_RECEIVED = 4,  /* TTI expected to be rcvd on port */
};
```


The format for Adjacency Discovery Identity TLVs are as follows:

```
struct ofp_exp_ext_ad_id {
       uint16_t type;                   /* OFP_EXP_EXT_AD_ID_SENT,
                                           OFP_EXP_EXT_AD_ID_EXPECTED, or
                                           OFP_EXP_EXT_AD_ID_RECEIVED. */
       uint16_t length;                 /* The TLV value field length, defined as */
                                        /* 8+ the length of the id[] field */
       uint16_t namespace;              /* One of OFPHTN_*. */
       uint16_t ns_type;                /* Type within namespace */
       uint8_t id[0];                   /* ID */
       /* Pad to 64 bits. */
};
```

NOTE: The id[] field contains id formatted for the technology identified by the namespace/ns_type fields. As a result, this field will be different lengths depending on the technology of the port. See section 3.3.1 for technology specific encodings of the id[] field.

In addition to these TLVs, the controller needs to be able to specify behaviors for the logical switch and be notified of conditions. These are specified using the config and state attribute in the ofp_port.

The state attribute is extended to indicate when a mismatch of the expected AD_ID and received AD_ID has occurred. The state bit is defined as follows:

```
enum ofp_port_state {
       OFPPS_AD_ID_MISMATCH = 1 << 16    /* Indicates AD_ID mismatch condition */
};
```

When this bit changes state (i.e. 0->1 or 1->0), a notification will be sent to the controller based on the asynchronous message configuration for the controller-OpenFlow Logical Switch session.

The config attribute is extended to configure adjacency discovery behavior for the port. The bitfield is used to enable/disable adjacency discovery as well as control the insertion of AIS when the received AD_ID information does not match the Expected identifier. These bits are defined as follows:

```
enum ofp_port_config {
       OFPPF_AD_ENABLE = 1 << 16,          /* Enable Adjacency Discovery exchange */
       OFPPF_AIS_AD_ID_MISMATCH = 1 << 17  /* Cause AIS insertion on AD_ID mismatch */
```

3.3.1 Technology Specific Encodings

3.3.1.1 OTN and OCh

For OTN (ODU, OTU, OCh and OTS layers), the Adjacency Discovery ID is exchanged using a discovery message within the TTI field. This field is defined to carry a 64-byte message with 16 bytes of SAPI, 16 bytes of DAPI and 32 bytes of carrier specific value. When the namespace and ns_type fields specify one of the OTN layers, the id field is encoded as follows:

```
struct ofp_port_adid_otn {
       uint8_t sapi[16];               /* ASCII SAPI. Note: no null termination
                                       character is included in the sapi. */
       uint8_t dapi[16];               /* ASCII DAPI. Note: no null termination
                                       character is included in the dapi. */
       uint8_t opspec[32];             /* ASCII Operator specific value. Note: no
                                       null termination character is included. */
};
```

3.3.1.2 SONET/SDH

For SONET/SDH, the Adjacency Discovery ID is exchanged using the SONET/SDH J0 or J1 TTI fields. These fields are defined to carry a 16-byte message with one byte of FCS and 15 bytes of Trail Trace Identification value. In the case of J1 Path Trace the message will be limited to 16 bytes. When the namespace and ns_type fields specify SONET/SDH, the id field is encoded as follows:

```
struct ofp_port_adid_sdh {
       uint8_t id[15];                 /* ASCII TTI. Note: no null termination
                                       character is included in the id. */
};
```

The FCS will be automatically computed by the NE.

4 Use of OpenFlow for Optical Transport

4.1 Initial Session Establishment

It is assumed that there is reachability between the switch and the OpenFlow Controller suitable for establishment of the control channel. The switch is configured with the IP address of its OpenFlow Controller and establishes a connection to the Controller using the procedures in the OpenFlow specification.

4.2 Port Capability Discovery

Once connectivity to the Controller has been established, the switch provides information to the Controller about the features and ports it supports using switch configuration messages as specified in the OpenFlow specification. Port attribute extensions as defined in Section 3.2 above are used to identify OTN electrical or optical characteristics of the port.

4.3 Adjacency Discovery

Two scenarios for adjacency discovery that may be typical are the unprompted and prompted scenarios.

4.3.1 Unprompted Adjacency Discovery

In this scenario, a discovery mechanism has been configured on the network element, and discovery information has been exchanged prior to the establishment of the OpenFlow session between network element and controller.

OpenFlow is used to provide discovery information from the network element to the controller through the port attributes.

4.3.2 Prompted Adjacency Discovery

In this scenario, the network element is configured to wait for establishment of the OpenFlow session before initiating the discovery mechanism.

The controller uses port attributes to specify to the network element what identifier to send to the remote endpoint of the port and what identifier it should expect to receive, as well as the action to be taken if the received identifier is different from expected.

4.4 Setup of a Transport Connection

The flow mod commands are given below (in this case to support OTN ODU connectivity). The flow mod command OFPFC_ADD is used to establish OTN connections. For bidirectional circuits, two flow mod OFPFC_ADD commands are required – one in each direction. Barrier or Bundle (for OF1.4) mechanisms may be used to ensure that flows in both directions together with any associated actions are instantiated together.

4.4.1 Settings for Flow Entry Establishment

The following are general settings on the flow mod command to establish a flow entry:

- Cookie/Cookie_Mask - Discussed below.
- Table ID - Table ID is implementation dependent. It is recommended that a default value of 0 be used for single layer optical switches.
- Command - Add
- Idle_Timeout – Set to 0 for L0/L1 as entries must be explicitly removed.
- Hard_Timeout – Set to 0 for L0/L1 as entries must be explicitly removed.
- Priority - Use of Priority is for further study. It is recommended that this be set to 0 at this time.
- Buffer_ID – Set to OFP_NO_BUFFER – not applicable to transport networks.
- Output_Port/Group – OFPP_ANY/OFPG_ANY
- Flags
    - OFPFF_SEND_FLOW_REM – Recommended to be set to 0
    - OFPFF_CHECK_OVERLAP – Recommended to be set to 1 to support detection of duplicate entries or overlaps
    - OFPFF_RESET_COUNTS – Recommended to be set to 0
    - OFPFF_NO_PKT_COUNTS – Recommended to be set to 0
    - OFPFF_NO_BYT_COUNTS – Recommended to be set to 0

4.5 Flow Retrieval

The multipart message with type=OFPMP_FLOW is used to retrieve the entries in the flow table.

4.5.1 Retrieve all Entries

To retrieve all entries in the flow table, the following parameters shall be used:

- out_port = OFPP_ANY
- out_group = OFPG_ANY
- cookie = 0
- cookie_mask = 0
- match type = OFPMT_OXM, payload = null

Flow retrieval responses should support:

- duration_sec = 0 (not supported)
- duration_nsec; =0 (not supported)
- priority = 0 (see above)
- idle_timeout = 0 (see above)
- hard_timeout =0 (see above)
- cookie = 0
- packet_count = 0 (see above)
- byte_count = 0 (see above)

4.5.2 Retrieve Specific Entry

To retrieve a specific entry in the flow table, the following parameters should be used:

- out_port = OFPP_ANY
- out_group = OFPG_ANY
- cookie/mask = match original cookie with mask = ~~0xFFFFFFFFFFFFFFFF or any/0~~ 0/0
- match = should be the same as the match field in the original flow mod command

Flow retrieval responses should support:

- duration_sec = 0 (not supported)
- duration_nsec; =0 (not supported)
- priority = 0 (see above)
- idle_timeout = 0 (see above)
- hard_timeout =0 (see above)
- cookie = 0
- packet_count = 0 (see above)
- byte_count = 0 (see above)

4.6 Flow Deletion

The flow mod command with command=OFPFC_DELETE is used to delete an OTN connection. For bidirectional circuits, two flow mod OFPFC_DELETE commands are required – one in each direction.

4.6.1 Delete Specific Entry

To delete a specific entry from the flow table, the following parameters should be used:

- Cookie/Cookie_Mask - Discussed in Cookie Support section below.
- Table ID - Table ID is implementation dependent. It is recommended that a default value of 0 be used for single layer optical switches.
- Command – OFPFC_DELETE
- Idle_Timeout – Set to 0 – Transport network flows must be explicitly removed.
- Hard_Timeout – Set to 0 – Transport network flows must be explicitly removed.
- Priority - Use of Priority is for further study. It is recommended that this be set to 0 at this time.
- Buffer_ID – Set to OFP_NO_BUFFER – not applicable for transport networks.
- Output_Port/Group – OFPP_ANY/OFPG_ANY
- Flags
    - OFPFF_SEND_FLOW_REM – Recommended to be set to 0
    - OFPFF_CHECK_OVERLAP – Recommended to be set to 1 to support detection of duplicate entries or overlaps
    - OFPFF_RESET_COUNTS – Recommended to be set to 0
    - OFPFF_NO_PKT_COUNTS – Recommended to be set to 0
    - OFPFF_NO_BYT_COUNTS – Recommended to be set to 0

4.7 Echo Request/Reply

The Echo Request/reply can be used to verify liveliness. The echo request can be initiated by either the controller or the NE.

Echo Request. Arbitrary length data field.

Echo Reply: returns the unmodified data field from the echo request.

4.8 Cookie Support

The use of cookies is optional and determined by the controller.

The OpenFlow agent should support the following capabilities using cookies when performing a flow retrieval or flow deletion.

4.8.1 Retrieve all Entries that Match the Cookie and Cookie Mask Fields

To retrieve all entries in the flow table that matches a specific cookie/cookie_mask, the following parameters shall be used:

- out_port = OFPP_ANY
- out_group = OFPG_ANY
- cookie = variable
- cookie_mask = variable
- match type = OFPMT_OXM, payload = null

The agent shall provide all flow entries that match the cookie/cookie_mask: (*flow entry*:*cookie*&*flow mod*:*cookie mask*) == (*flow mod*:*cookie*&*flow mod*:*cookie mask*)

4.8.2 Delete all Entries that Match the Cookie and Cookie_mask Fields

To delete all entries in the flow table that matches a specific cookie/cookie_mask, the following parameters should be used:

- out_port = OFPP_ANY
- out_group = OFPG_ANY
- cookie = variable
- cookie_mask = variable
- match type = OFPMT_OXM, payload = null

The agent shall delete all flow entries that match the cookie/cookie_mask: (*flow entry*:*cookie*&*flow mod*:*cookie mask*) == (*flow mod*:*cookie*&*flow mod*:*cookie mask*)

4.9 Barrier Message Support

The controller may use a barrier message to ensure that previous messages have been fully processed.

To do this, the controller will send an OFPT_BARRIER_REQUEST message. The network element will finish processing all previous messages including sending replies and/or error messages before executing any additional commands. When it completes processing all previous messages, it will send the OFPT_BARRIER_REPLY message.

The controller may use this capability to ensure that both directions of a bidirectional flow have been fully established.

5 REFERENCES

[1] TR_Requirements Analysis for Transport OpenFlow/SDN_v.1.0, Aug. 20, 2014.
[2] Optical Transport Information Model, work in progress.
[3] ITU-T Recommendation G.7714.1/Y.1705.1, “Protocol for automatic discovery in SDH and OTN networks”, 9/2010.
[4] of-notifications-framework-1.0, Oct. 15, 2013.
[5] OpenFlow Switch Specification, Version 1.3.0 (Wire Protocol 0x04), June 25, 2012.
[6] OpenFlow Switch Specification, Version 1.3.1 (Wire Protocol 0x04), September 6, 2012.
[7] OpenFlow Switch Specification, Version 1.3.2 (Wire Protocol 0x04), April 25, 2013.
[8] OpenFlow Switch Specification, Version 1.3.3 (Wire Protocol 0x04), December 18, 2013.
[9] OpenFlow Switch Specification, Version 1.3.4 (Wire Protocol 0x04), under ratification.
[10] Onf2014.313.01, EXT-445-446 Merged OTWG Specification, May 30, 2014.
[11] IETF RFC 7139, “GMPLS signaling extensions for control of Evolving G.709 OTN”, March 2014, F. Zhang, ed.

A. Appendix I: OF1.3-based Prototype Implementation for Optical Transport

The following Appendix provides historical information about prototype testing done to validate OF1.3-based implementation of optical transport extensions. It is intended only as a record of an implementation of the OpenFlow extensions that may help future implementors – the specification text itself takes precedence over the text in this Appendix. Some formats and settings used in prototyping have been updated and modified in the specification.

A.1. Simplifying Assumptions on the Basic Interoperability OF1.3 proposal

Assumptions (OF1.3):

- OpenFlow 1.3.0 will be used
- OpenFlow will run directly on TCP (not TLS)
- Controller is located at TCP port 6633
- NE initiates the session with controller
- The controller and NE will exchange a Hello message once TCP connection is established. Both ends send the Hello message independently.
- Hello may contain a body (version bitmap element). Implementations may ignore the body.
- Hello protocol indicates OF1.3 version (0x04). This avoids the need to negotiate downwards.
- NE supports a single OpenFlow table with Table ID = 0
- Controller will send the Feature Request command
- Controller will send the Set Config command.
- Controller will send the Port Description command
- Controller will not send the Table Feature command
- Controller will send Flow Mod commands (OFPFC_ADD) to establish connections
- Controller will send Flow Mod commands (OFPFC_DELETE) to remove connections
- Controller will send multipart request/response command with type=OFPMP_FLOW to retrieve the flow entries in a table. Packet In/Out will not be used.
- Echo request/reply can be used to verify liveliness. Echo request/reply do not contain a body.

OTN transport plane assumptions.

- OTN links will be OTU2
- HO-ODU2 will support the payload type of 21 when multiplexing LO-ODUs. In this case, the ODU2 will have 8 tributary slots.
- Service rates supported: ODU0, ODU2 and ODUflex(GFP)
- Non-OTN edge interfaces: 1GE, 10GE

A.2. OpenFlow Header

The version number for OF1.3 is 0x04.

A.3. Session Establishment

The basic interoperability protocol flow is shown in Figure.2 for the establishment of the OF1.3 (1.3.0) session, including discovering the available ports and port descriptions.

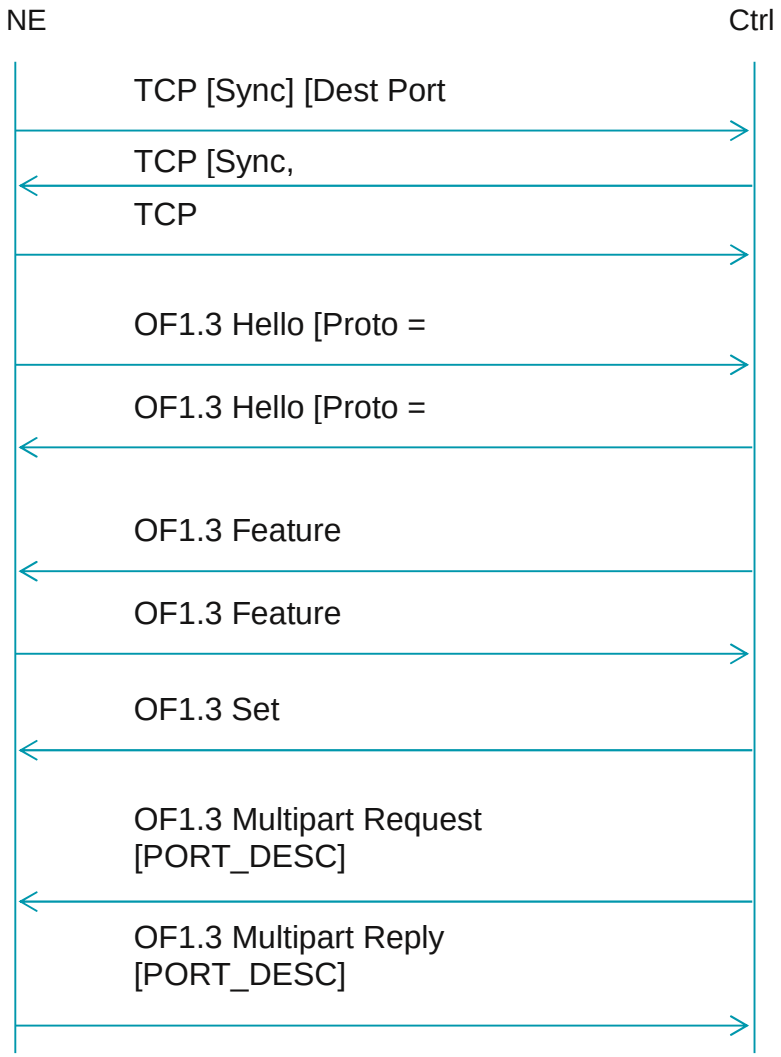


Figure.2: OF1.3 Session Establishment

The network element shall initiate a session with the controller. It is assumed that the network element is provisioned with the IP address of the controller. The mechanism for provisioning the controller's IP address is outside the scope of this document.

The network element shall establish a TCP connection with the controller using port 6633.

> Note: OF1.3.3 switches from port 6633 to the IANA registered TCP port 6653.
>
> Note: It is recommended that OpenFlow run directly over TCP for this demo to allow for protocol capture.

Upon establishing the TCP connection, a Hello exchange will take place. Hellos must be sent by both the controller and the network element. Hellos from each end are sent independently.

In OF1.3.0, the Hello message has no body. It only has the OF header. However, implementations should support the ability to receive (and ignore) a body.

> Note: In OF1.3.1, an optional body was added to the Hello message. The body may contain zero or more elements. The only element defined in OF1.3.1 is the version bitmap element.

In the Hello message, the protocol version number should be set to the highest version supported by the implementation (should be >= 0x04). The negotiated protocol version is the lesser of the sent and received protocol version in the Hello message.

Since it is assumed that all implementations will support OF1.3, protocol version notification should not be necessary.

> Note: If the negotiated protocol version is not supported by an implementation, then that implementation shall reply with an OFPT_ERROR message with a type field of OFPET_HELLO_FAILED, code OFPHFC_COMPATIBLE. The TCP connection shall then be terminated.
>
> Note: a more sophisticated negotiation process is available with OF1.3.1 using the version bitmap element.

After the controller has sent its Hello and received the Hello from the NE, the controller sends an OFPT_FEATURES_REQUEST message. This message does not contain a body. The NE responds with an OFPT_FEATURES_REPLY message.

> OFPT_FEATURES_REPLY.
>
> - The datapath ID should be unique. The lower 48-bits are for a MAC address while the upper 16-bites are implementation defined.
> - n_buffer: <ignored. No plans to support packet_in/out>
> - n_tables: 1 <only one table is supported>
> - auxiliary_id: set to 0x00 (main connection)
> - capabilities: may be set to 0x00. Switch capabilities are not planned to be tested.

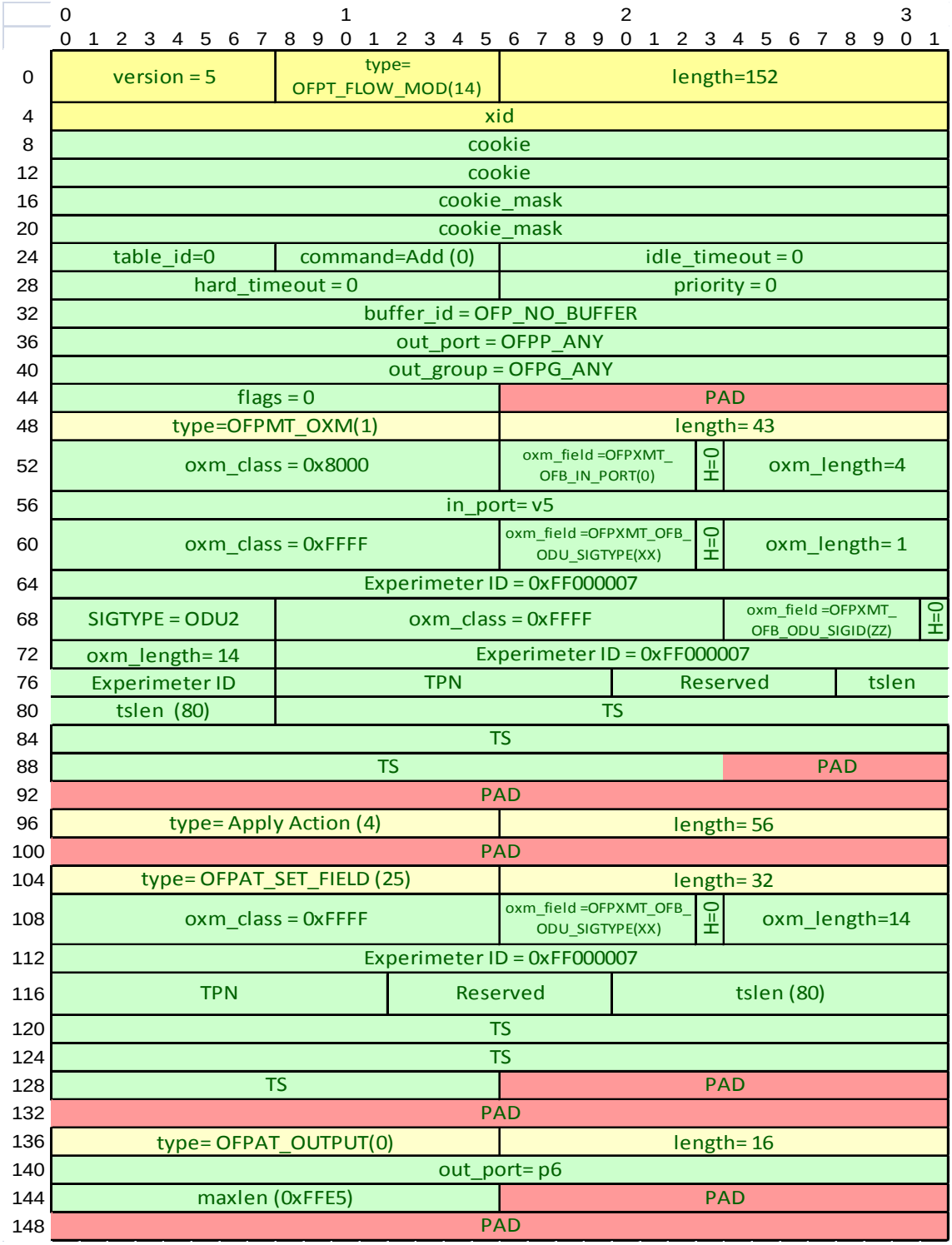
The controller sends an OFP_SET_CONFIG command.

> OFPT_SET_CONFIG.
>
> - Flags = 0x00
> - Miss send length: any value as Packet In is not planned to be tested.

A.3.1. Port Description

The controller sends an OFPMP_PORT_DESC multipart request to learn about the ports on an NE. The body of the request is empty. The NE responds with a multipart reply that provides a list of the ports (plus description) supported by the NE.

A.3.2. Example of Flow_Mod Format (Note: see Section 3.1 for updated format)



A.3.3. ODU2 HO-ODU to HO-ODU

With OTWG extensions:

- Match: ofp_match_type = OFPMT_OXM

  ofp_oxm_class = OFPXMC_OPENFLOW_BASIC
  ofp_oxm_field = OFPXMT_OFB_IN_PORT
  data = input port number

  ofp_oxm_class = OFPXMC_EXPERIMENTER
  ofp-oxm_field = OFPXMT_EXP_ODU_SIGTYPE
  experimenter = 0xFF000007
  sigtype = OFPODUT_ODU2
- Instruction: type = OFPIT_APPLY_ACTIONS
  Actions:
  type = OFPAT_OUTPUT
  port = output port number
  max_len = 65509

A.3.4. ODU0/ODUFlex LO-ODU to LO-ODU

With OTWG extensions:

- Match: ofp_match_type = OFPMT_OXM

  ofp_oxm_class = OFPXMC_OPENFLOW_BASIC
  ofp_oxm_field = OFPXMT_OFB_IN_PORT
  data = input port number

  ofp_oxm_class = OFPXMC_EXPERIMENTER
  ofp-oxm_field = OFPXMT_EXP_ODU_SIGTYPE
  experimenter = 0xFF000007
  sigtype = OFPODUT_ODU0, OFPODUT_ODUfGFPf or OFPODUT_ODUfGFPfHAO

  ofp_oxm_class = OFPXMC_EXPERIMENTER
  ofp-oxm_field = OFPXMT_EXP_ODU_SIGID
  experimenter = 0xFF000007
  tpn = tributary port number
  tslen = 8
  tsmap = tributary slot bitmap

- Instruction: type = OFPIT_APPLY_ACTIONS
  Actions:
  type = OFPAT_SET_FIELD
  ofp_oxm_class = OFPXMC_EXPERIMENTER
  ofp-oxm_field = OFPXMT_EXP_ODU_SIGID
  experimenter = 0xFF000007

tpn = tributary port number
tslen = 8
tsmap = tributary slot bitmap

type = OFPAT_OUTPUT
port = output port number
max_len = 65509

A.3.5. 10GE to ODU2 HO-ODU

With OTWG extensions:

- Match: ofp_match_type = OFPMT_OXM

  ofp_oxm_class = OFPXMC_OPENFLOW_BASIC
  ofp_oxm_field = OFPXMT_OFB_IN_PORT
  data = input port number (10GE port)

  ofp_oxm_class = OFPXMC_EXPERIMENTER
  ofp-oxm_field = OFPXMT_EXP_ODU_SIGTYPE
  experimenter = 0xFF000007
  sigtype = OFPODUT_ODU2
- Instruction: type = OFPIT_APPLY_ACTIONS
  Actions:
  type = OFPAT_OUTPUT
  port = output port number (OTU2 port)
  max_len = 65509

A.3.6. ODU2 HO-ODU to 10GE

With OTWG extensions:

- Match: ofp_match_type = OFPMT_OXM

  ofp_oxm_class = OFPXMC_OPENFLOW_BASIC
  ofp_oxm_field = OFPXMT_OFB_IN_PORT
  data = input port number (OTU2 port)

  ofp_oxm_class = OFPXMC_EXPERIMENTER
  ofp-oxm_field = OFPXMT_EXP_ODU_SIGTYPE
  experimenter = 0xFF000007
  sigtype = OFPODUT_ODU2
- Instruction: type = OFPIT_APPLY_ACTIONS
  Actions:
  type = OFPAT_OUTPUT
  port = output port number (10GE port)
  max_len = 65509

A.3.7. 1GE to ODU0 LO-ODU

With OTWG extensions:

- Match: ofp_match_type = OFPMT_OXM

  ofp_oxm_class = OFPXMC_OPENFLOW_BASIC
  ofp_oxm_field = OFPXMT_OFB_IN_PORT
  data = input port number (1GE port)

  ofp_oxm_class = OFPXMC_EXPERIMENTER
  ofp-oxm_field = OFPXMT_EXP_ODU_SIGTYPE
  experimenter = 0xFF000007
  sigtype = OFPODUT_ODU0
- Instruction: type = OFPIT_APPLY_ACTIONS

  Actions:

  type = OFPAT_SET_FIELD
  ofp_oxm_class = OFPXMC_EXPERIMENTER
  ofp-oxm_field = OFPXMT_EXP_ODU_SIGID
  experimenter = 0xFF000007
  tpn = tributary port number
  tslen = 8
  tsmap = tributary slot bitmap

  type = OFPAT_OUTPUT
  port = output port number (OTU2 port)
  max_len = 65509

A.3.8. ODU0 LO-ODU to 1GE

With OTWG extensions:

- Match: ofp_match_type = OFPMT_OXM

  ofp_oxm_class = OFPXMC_OPENFLOW_BASIC
  ofp_oxm_field = OFPXMT_OFB_IN_PORT
  data = input port number (OTU2 port)

  ofp_oxm_class = OFPXMC_EXPERIMENTER
  ofp-oxm_field = OFPXMT_EXP_ODU_SIGTYPE
  experimenter = 0xFF000007
  sigtype = OFPODUT_ODU0

  ofp_oxm_class = OFPXMC_EXPERIMENTER
  ofp-oxm_field = OFPXMT_EXP_ODU_SIGID
  experimenter = 0xFF000007

tpn = tributary port number
tslen = 8
tsmap = tributary slot bitmap

- Instruction: type = OFPIT_APPLY_ACTIONS
  Actions:
  type = OFPAT_OUTPUT
  port = output port number (1GE port)
  max_len = 65509

A.3.9. Subrate 10GE to ODUflex(GFP) LO-ODU

With OTWG extensions:

- Match: ofp_match_type = OFPMT_OXM

  ofp_oxm_class = OFPXMC_OPENFLOW_BASIC
  ofp_oxm_field = OFPXMT_OFB_IN_PORT
  data = input port number (10GE port)

  ofp_oxm_class = OFPXMC_EXPERIMENTER
  ofp-oxm_field = OFPXMT_EXP_ODU_SIGTYPE
  experimenter = 0xFF000007
  sigtype = OFPODUT_ODUfGFPf or OFPODUT_ODUfGFPfHAO
- Instruction: type = OFPIT_APPLY_ACTIONS
  Actions:
  type = OFPAT_SET_FIELD
  ofp_oxm_class = OFPXMC_EXPERIMENTER
  ofp-oxm_field = OFPXMT_EXP_ODU_SIGID
  experimenter = 0xFF000007
  tpn = tributary port number
  tslen = 8
  tsmap = tributary slot bitmap

  type = OFPAT_OUTPUT
  port = output port number (OTU2 port)
  max_len = 65509

A.3.10. ODUflex(GFP) LO-ODU to Subrate 10GE

With OTWG extensions:

- Match: ofp_match_type = OFPMT_OXM

  ofp_oxm_class = OFPXMC_OPENFLOW_BASIC
  ofp_oxm_field = OFPXMT_OFB_IN_PORT
  data = input port number (OTU2 port)

ofp_oxm_class = OFPXMC_EXPERIMENTER
ofp-oxm_field = OFPXMT_EXP_ODU_SIGTYPE
experimenter = 0xFF000007
sigtype = OFPODUT_ODUfGFPf or OFPODUT_ODUfGFPfHAO

ofp_oxm_class = OFPXMC_EXPERIMENTER
ofp-oxm_field = OFPXMT_EXP_ODU_SIGID
experimenter = 0xFF000007
tpn = tributary port number
tslen = 8
tsmap = tributary slot bitmap

- Instruction: type = OFPIT_APPLY_ACTIONS
  Actions:
  type = OFPAT_OUTPUT
  port = output port number (10GE port)
  max_len = 65509

LIST OF CONTRIBUTORS

The DOC3-4 Design team was responsible for the writing of this document.
List of names in alphabetical order:

- Sergio Bellotti
- Fred Gruman
- Jia He
- Peter Landon
- Young Lee
- Lyndon Ong (ed.)
- Jonathan Sadler
- Meral Sherazipour
- Maarten Vissers
- Shinji Yamashita
- Karthik Sethuraman

This document follows on work by the 2014 Joint OIF/ONF Prototype Demonstration design team looking at prototyping of OpenFlow extensions, especially Fred Gruman, Pawel Kaczmarek, Sergio Belotti and Junjie Li.

The OIF/ONF demo specification was in turn drawn from the OTWG contributions onf2014.313.01 and onf2014.33.01, which were proposed to the Extensibility WG (and approved for prototyping) as solutions to proposals: EXT-445 & EXT-446. The liaisons from the Optical Transport WG to the Extensibility WG, responsible for EXT-445 and 446, were:

- Jonathan Sadler
- Karthik Sethuraman