OPEN NETWORKING
FOUNDATION

OpenFlow Table Type Patterns

Version No. 1.0
15 August 2014

ONF TS-017

NOTE: ONF specification
TS_OpenFlow_NDM_Synchronization_v.1.0_062014
is closely related to this specification.

OpenFlow™

ONF Document Type: OpenFlow Spec
ONF Document Name: OpenFlow Table Type Patterns v1.0

Disclaimer

THIS SPECIFICATION IS PROVIDED “AS IS” WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE.

Without limitation, ONF disclaims all liability, including liability for infringement of any proprietary rights, relating to use of information in this specification and to the implementation of this specification, and ONF disclaims all liability for cost of procurement of substitute goods or services, lost profits, loss of use, loss of data or any incidental, consequential, direct, indirect, or special damages, whether under contract, tort, warranty or otherwise, arising in any way out of use or reliance upon this specification or any information herein.

No license, express or implied, by estoppel or otherwise, to any Open Networking Foundation or Open Networking Foundation member intellectual property rights is granted herein.

Except that a license is hereby granted by ONF to copy and reproduce this specification for internal use only.

Contact the Open Networking Foundation at https://www.opennetworking.org for information on specification licensing through membership agreements.

Any marks and brands contained herein are the property of their respective owners.

WITHOUT LIMITING THE DISCLAIMER ABOVE, THIS SPECIFICATION OF THE OPEN NETWORKING FOUNDATION (“ONF”) IS SUBJECT TO THE ROYALTY FREE, REASONABLE AND NONDISCRIMINATORY (“RANDZ”) LICENSING COMMITMENTS OF THE MEMBERS OF ONF PURSUANT TO THE ONF INTELLECTUAL PROPERTY RIGHTS POLICY. ONF DOES NOT WARRANT THAT ALL NECESSARY CLAIMS OF PATENT WHICH MAY BE IMPLICATED BY THE IMPLEMENTATION OF THIS SPECIFICATION ARE OWNED OR LICENSABLE BY ONF'S MEMBERS AND THEREFORE SUBJECT TO THE RANDZ COMMITMENT OF THE MEMBERS.

© 2014 Open Networking Foundation

List of Figures

List of Tables

© 2014 Open Networking Foundation

1 Introduction

This document describes Table Type Patterns (TTPs) that support the lifecycle of development, deployment and operation of OpenFlow networks. The development of a TTP-based approach has been motivated by several factors including: to better accommodate heterogeneity of existing hardware switches, to enable future innovation in hardware switches, to enable precise communication between app/controller developers and switch vendors, and to enable automated communication between apps/controllers and switches. For a more in depth discussion of these and other motivations see Appendix A.

This document is an ONF Technical Specification and has normative content, primarily in section 3. The normative content includes the keywords defined for identifying attributes of the OpenFlow-Switch protocol and the structure of the information in a Table Type Pattern.

2 Table Type Patterns

A Table Type Pattern (TTP) is an abstract switch model that describes specific switch forwarding behaviors that an OpenFlow controller can program via the OpenFlow-Switch protocol. A TTP represents the flow processing capabilities of an OpenFlow Logical Switch (OFLS). In declaring support for a particular TTP an OFLS indicates support for all the required capabilities specified by the TTP, and a negotiated subset of optional capabilities. In accepting a TTP a controller understands the valid commands for the OFLS are a constrained subset of the standard OpenFlow-switch protocol – that is, the OFLS is not required to accept commands that do not conform to the constraints specified by the TTP. An OpenFlow Capable Switch may support a number of TTPs, and a controller may understand a number of TTPs, but only one TTP can be active for a given OFLS.

The TTP framework is an optional enhancement to the OpenFlow framework. The use of TTPs is optional because there will be situations where they will not be needed. For example, when the forwarding model can be implemented in a single flow table, or when the forwarding device faithfully implements all aspects of the implicit OpenFlow abstract switch, then TTPs may not be needed. By abstract switch model, we mean that each table in a TTP does not necessarily have a one-to-one mapping with a hardware table; mapping a TTP onto hardware is the responsibility of the switch implementer.

When using the TTP framework, the controller and the switch agree on a specific TTP before the controller begins sending OpenFlow messages to the switch. This agreement may be implicit (i.e., each side is configured *a priori*) or negotiated before forwarding is enabled. The TTP framework can simplify the job of implementing an OpenFlow controller or OpenFlow agent for a switch. The TTP provides a description of the required switch behavior so that implementers can perform optimizations or deliver more complex forwarding behaviors (beyond what can be represented in a single OpenFlow table). TTPs are expected to be most useful in situations where multiple flow tables are needed, particularly when the forwarding devices are ASIC based. In the absence of TTPs, those use cases (and others) may face significant interoperability challenges. In such cases, table properties are typically manipulated out-of-band or device-specific coding is needed in the controller.

A TTP specifies a set of flow tables and describes the valid flow table entries and group table entries for an OFLS. As the OpenFlow-Switch 1.x specification evolves and incorporates new objects, the TTP syntax must be updated to support them. Some future changes to OpenFlow may be dramatic, requiring switch models beyond TTPs. Where appropriate, allowance has been made for such future models.

2.1 Future-proofing the TTP Framework

The structure of switch models, such as TTPs, may evolve significantly as the OpenFlow protocol (and its implicit switch model) evolves to future generations. TTPs, which are the focus of this document, are based on the OpenFlow-Switch 1.x protocol. We can anticipate that future generations of switch models will go beyond the table-centric approach of OF-Switch 1.x. Such future models will likely be named something other than TTP. To refer to TTPs together with their future cousins, we use the term "Negotiable Datapath Models" (NDMs) as a category term. That is, TTPs are the first (and as yet, the only) type of NDM that has been described. The term "NDM" appears elsewhere in this document where a particular effort has been made to future-proof of the TTP framework. For example, files that represent TTPs include an "NDM_metadata" field which is intended to make it simpler for future tools to distinguish different generations of models.

2.2 The Basic Lifecycle of TTPs in an OpenFlow Network

The OpenFlow protocol specifications (OpenFlow-Switch, OpenFlow-Config) are necessarily focused on the bits and bytes and detailed behaviors related to elements of these protocols. These specifications do not provide a comprehensive explanation of the framework within which these protocols are employed, i.e., the natural lifecycle of OpenFlow network design, deployment and operation.

Like any other network, a Software Defined Network (SDN) must implement a consistent set of behaviors across a set of distributed elements. Network clients must send and receive information using an encapsulation or encoding required by the network. Each network element must understand this encoding to know how to process/forward packets of information. Therefore the first step in the OpenFlow Network lifecycle is determining what network encapsulation/encoding will be used (e.g., ODU frames, Ethernet MAC frames, IP packets, etc.) and how each device will make use of it. In short, the first step in an OF network lifecycle is to define the network architecture in detail.

Table Type Patterns are a vehicle for fully describing switch behaviors based on the OpenFlow multi-table switch model. To get a feel for TTPs are and how they are used, it's useful to walk through a lifecycle diagram. The Lifecycle shown in Figure 1 and some variations are described more fully in the sections below. Note that many of the steps in the Lifecycle are not new, but existed in much the same form before the introduction of TTPs.

© 2014 Open Networking Foundation

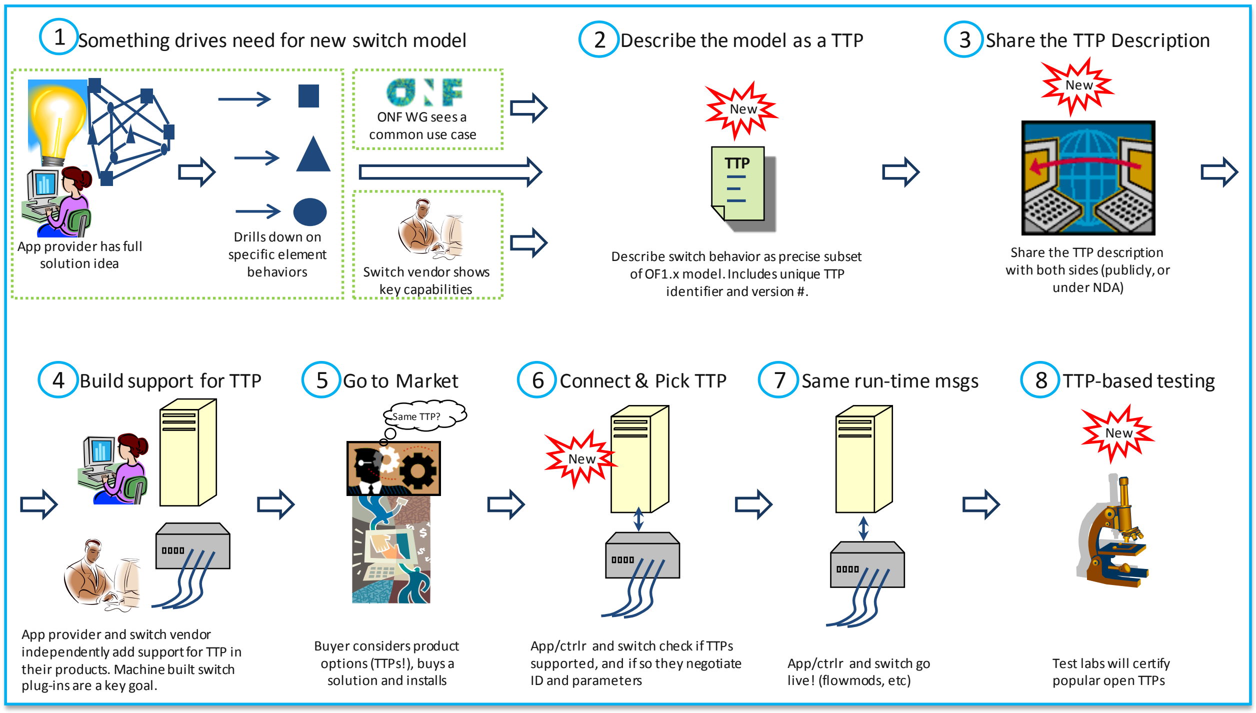
1 Something drives need for new switch model
App provider has full solution idea
Drills down on specific element behaviors
ONF WG sees a common use case
Switch vendor shows key capabilities
2 Describe the model as a TTP
New
TTP
Describe switch behavior as precise subset of OF1.x model. Includes unique TTP identifier and version #.
3 Share the TTP Description
New
Share the TTP description with both sides (publicly, or under NDA)
4 Build support for TTP
App provider and switch vendor independently add support for TTP in their products. Machine built switch plug-ins are a key goal.
5 Go to Market
Same TTP?
Buyer considers product options (TTPs!), buys a solution and installs
6 Connect & Pick TTP
New
App/ctrlr and switch check if TTPs supported, and if so they negotiate ID and parameters
7 Same run-time msgs
App/ctrlr and switch go live! (flowmods, etc)
8 TTP-based testing
New
Test labs will certify popular open TTPs

Figure 1: OpenFlow Network Lifecycle with TTPs

2.2.1 Step 1: Something Prompts a New TTP

Any interested party is able to define a new TTP. The idea for a new TTP can be motivated by a wide variety of circumstances. This step is comparable to the point in "traditional" OpenFlow development where an architect conceives of a new application. It is in the interest of the architect to construct a TTP that is both suitable to the application and supportable on available platforms. Where there is uncertainty on these points, partnerships between application providers and equipment providers may be helpful.

The network architecture identifies the types of network elements ("logical switches" in OpenFlow terms) required to implement the network. See step 1 in the lifecycle in Figure 1. In this example three network element types (shown as different shapes) are identified in the network architecture. For each element type to be controlled using OpenFlow a description of the controllable behaviors, a TTP, can be created (see step 2).

App providers (vendors or network operators) can conceive of various switch-level behaviors (as seen from outside the box) that they would like to use for solution architectures that they develop. Switch vendors (either ASIC-based or NPU-based) may choose to create a new TTP as a way to provide access to differentiated functionality in their platform.

To get things started, the ONF will define a small number of TTPs that it believes will be useful and informative.

2.2.2 Step 2: Describing a TTP

The format of a TTP is described in detail in section 3. A TTP is a first generation detailed abstract switch model built on an existing OpenFlow protocol version (e.g., OpenFlow-Switch 1.x). We describe a TTP as proper subset of an OpenFlow multi-table logical switch. That is, a TTP describes a constrained subset of the OpenFlow logical switch described in the OpenFlow specifications [https://www.opennetworking.org/sdn-resources/onf-specifications/openflow]. The details of TTP Description are covered in Section 3, but we'll provide a summary here.

TTP Descriptions include metadata such as the naming authority, the NDM type (which for this document is always a kind of TTP), name and version. A TTP description also includes a set of flow tables, the flow_mod message types that must be supported in specific flow tables, and the group entries required in the group table. The TTP defines valid flow_mod and group_mod messages for a given logical switch, and so effectively specifies the "table graph" of the TTP.

TTP Descriptions may in future support the inclusion of test information in order to help drive interoperability, and inclusion of such test information is recommended as a best practice. Interoperability is a key goal of TTPs, and tests are important to confirm interoperability. Every TTP implementation will need testing and every TTP definition would benefit from including suitable test information. By including test information, consistent support for TTPs is more straightforward. Providers that offer TTP support can self-test and confirm that their implementation is conformant.

Note: In this first phase of TTP support, TTPs are independent switch models. TTP developers can copy/paste elements of other TTPs, but the current framework does not provide explicit support for importing or combining TTP elements. These ideas are interesting and may be considered as TTPs mature.

2.2.3 Step 3: Sharing a TTP Description

As previously mentioned, the purpose of a TTP is to allow both ends of the OpenFlow control channel to support a shared switch behavioral model. Clearly, the providers of these products (controller and switch) will both require access to the TTP description. There are several scenarios:

1) ONF-developed TTPs
    a. Once stable, these will be fully public, like OpenFlow specs.
        i. Sharing will be accomplished by posting at the ONF website
        ii. Note that TTPs are not intended to be "standards." Instead, TTPs are "common practices", and ONF-developed TTPs will not hold any special distinction with respect to TTPs developed by others.
2) ONF Member-developed open TTPs
    a. Application, controller and switch providers who are ONF members can develop a TTP and share it.
        i. Partnerships of members can also co-develop and share TTPs.
        ii. Sharing of member TTPs will be supported on the ONF website, but can be accomplished in other ways as well.
3) ONF Member-developed private TTPs
    a. Market participants can also develop TTPs and keep them "under wraps", either temporarily during development or for more extended periods. This might be handled in two different ways:
        i. Fully hidden TTPs: the IDs, descriptions and the supporting products are hidden.
            1. The ability of providers to keep TTPs private may be useful during the development phase of a TTP. The alternative would be that vendors would need to share information during their development phase, which might impede adoption of TTPs.
        ii. Partially hidden TTPs: TTPs with Open IDs, but hidden descriptions, where partners can advertise support for the semi-secret TTP. This may be helpful for vendors or users that regard the contents of TTPs as being "trade secrets". Since only implementers really need to see the descriptions, it seems conceivable that descriptions could remain secret for extended periods
    b. Private TTPs may, at the discretion of the controlling parties, be made open over time.

Sharing TTPs openly will help advance interoperability, and will help market participants in determining what products interoperate. This will be true from the beginning and will become more compelling as TTP-based conformance testing gains traction. (Note: TTPs may be useful as "test profiles" even for products that do not explicitly support TTPs. More on this in section 2.1.9.)

2.2.4 Step 4: Adding support for a TTP: The compilation phase

In this step, the information in a TTP Description is used to build support for the abstract switch behavior that the TTP represents. TTP Descriptions have been made both human and machine readable so that support can be provided either by human developers or through machine compilation. (Machine compilation will be limited to switch-based TTP support until well-defined Northbound API's allow for machine-built controller support.) Machine-based compilation of TTPs into switch-based plug-ins is quite feasible, and will eliminate the delays implicit in human plug-in development. However, such compilation will require the development of compilers and device operating systems that support the machine-built plug-ins. Meanwhile, the initial scenario is that, controller and switch implementers (human developers) use the TTP description to code (and test) support for the TTP-defined switch model, so that the devices interoperate properly when connected at run time. The timing of this development effort may vary significantly depending on the scenario:

1) Third party TTPs (including ONF-developed TTPs) may be supported by both controller and switch in similar timeframe
2) For TTPs provided by switch vendors and supported by their products, support can be added later by controller developers wishing to support these products in OpenFlow network applications
3) When hardware suppliers are adding support for a TTP, there are two scenarios:
    a. They may need to release on a typical hardware schedule, or
    b. They may be able to decouple TTP support from their main OS release, and offer shorter timescales for supporting new TTPs.

TTP descriptions are designed to be human readable so implementers can easily understand what is required of the implementation to support the TTP.

In addition to supporting readability for implementers, there is also the potential for machine processing support. For example, TTP compilation to produce software artifacts supporting use of the TTP is conceivable; however, TTP compilation is matter for future investigation. TTP-oriented tools will simplify TTP use and adoption. Some work on tools has begun, and the ONF will encourage further development. Detailed discussion of tools is, however, beyond the scope of this document.

While the TTP description syntax is machine readable (JSON) it is recognized that this syntax may need enhancement to support such goals. Additional protocol mechanisms will also be needed to support exchanges of either TTP descriptions or compiled versions thereof.

2.2.5 Step 5: Going to Market with a TTP

The go-to-market step exists in the absence of TTPs, but a TTP will change the picture in important ways. First, consider the diversity of OpenFlow-enabled devices as well as the range of optional features described in the OpenFlow specifications. In such a context, it is difficult to know which switches fully support a given controller or application. Explicitly declaring support for an applicable TTP gives customers confidence that a controller from one vendor will interoperate with a switch from another vendor. Alternatively, a very large network operator may define their own TTPs (and their own apps or controllers), and may require support for these TTPs from their switch vendors.

Even for products that do not include TTP-based implementations, TTPs may be useful for understanding or clarifying controller/switch interoperability issues.

2.2.6 Step 6: Picking a TTP at Connection Time

For products to perform TTP-based network control, the controller and the switch must have a synchronized TTP context. This context is defined by a specific (identified) TTP and specific parameter values for the TTP's parameters. The TTP framework includes a powerful mechanism, based on the OF-Config Protocol, for synchronizing controller and switch TTP contexts. (A future-proof scheme is preferred for negotiating TTP support between a controller and switch. With that in mind, the negotiation scheme was named using the umbrella term “NDM” rather than “TTP”.)

The negotiation mechanism allows the OpenFlow Configuration Point (OFCP) to perform several important functions in support of various scenarios:

- Discovery of available switch-supported TTPs, based on well-known TTP identifiers
- Simple activation of a desired TTP using default parameters
- An optional mechanism for the OFCP to negotiate with the switch for preferred TTP parameters
    - Switches that support parameter negotiation must support an RPC extension defined within the NETCONF framework

Details of the OF-Config1.2-based negotiation mechanism are provided the OF-Config 1.2 specification and its use is described in a separate document "OpenFlow Controller / Switch NDM Synchronization" available on the ONF website.

To allow adoption of TTPs and OF-Config1.2 to advance in parallel (to avoid the delays of serial adoption), alternative mechanisms can also be used to configure the TTP context of both ends of the OpenFlow control connection. For example, management interfaces in the controller and in the switch (both of which are out-of-scope for this document) can be used to set up the context on the two ends. Because manually configured TTP contexts may inadvertently lead to unsynchronized TTP contexts, an OF-Switch extension[1] has been proposed to confirm that the switch is indeed configured with the same TTP context as the controller.

2.2.7 Step 7: Run-time Messaging with TTPs

A key goal of the TTP description approach has been to minimize the impact on the OpenFlow-Switch control messaging. That goal has been achieved in the sense that OF-Switch control messages, such as flow_mod and group_mod messages, are no different in the context of an active TTP versus a non-TTP context. That is, the flow_mods and group_mods that a controller sends to a switch in context of an active TTP will also work if the controller sends precisely those same messages to a switch (such as a soft switch or NPU-based switch) that offers the same functionality without explicit support for the TTP.

Some messaging semantics are, however, impacted in the context of an active TTP.

1) Table Features messages will not be allowed to alter table attributes.
2) The switch may reject a message using a new error message if the controller attempts to exceed features defined by the TTP.

The switch is not required to provide any particular behavior if the controller configures flow_mods, group_mods, or meter_mods outside the valid set specified by the TTP. It may accept or reject the configuration and, if accepted, may provide implementation dependent behavior. It is recommended that every switch supporting TTPs also support a "debug mode" that strictly checks commands for validity with respect to the negotiated TTP and rejects invalid commands. This mode can be used to diagnose whether or not a controller is conforming to the negotiated TTP in

[1] Implementers are encouraged to check the ONF website for information on OpenFlow extensions.

configuring a logical switch. Since debug mode checking may impact performance the switch should also provide a mechanism for enabling/disabling debug mode.

2.2.8 Step 8: TTP-based Testing and Certification

TTPs offer several ways to improve and clarify interoperability. (See Appendix A for further discussion.) Here is a brief list of the ways in which TTPs aid in establishing interoperability:

- Products (both controllers and switches) with built-in TTP awareness can be independently tested to assess conformance with specific TTPs.
    - TTPs are precise and unambiguous; if both ends conform to a TTP, interoperability is more easily achieved.
- TTPs can be used as test profiles, even when the products lack built-in TTP awareness.
    - The Testing and Interoperability WG has expressed a need for unambiguous test profiles.
    - TTP Descriptions allow various market participants to define TTPs and, thus, test profiles.
- TTP Descriptions may in future include test information, to encourage self-testing of TTP conformance.
    - As a TTP gains market traction, 3rd party testing of that TTP will be increasingly likely.
    - For new TTPs, self-testing provides a mechanism to build adoption until 3rd party testing becomes available.

To make TTPs more interoperable (and thus more attractive in the market), TTP Descriptions support the inclusion of test information. TTP developers are encouraged (in terms of “best practice”) to include information regarding flow_mod and group_mod messages, and additional functionality. This test information will greatly enhance interoperability based on self-testing by vendors, and will serve as a starting point for conformance testing for TTPs that are popular enough to justify that level of investment by the larger market.

3 Creating a Table Type Pattern

A TTP is a text file that describes a set of flow tables and the valid flow_mods, group_mods, and meter_mods to be supported by an OpenFlow logical switch controllable using an OpenFlow-Switch 1.x protocol. Given this association with the OpenFlow-Switch protocol, it is important for the reader to be familiar with that protocol or be able to refer to the OpenFlow-Switch 1.3.x protocol specifications while reading this section.

A TTP does not define new OpenFlow protocol features; however, it can require certain protocol features be supported and can constrain the uses of protocol features to those necessary for a particular network application. A TTP can also include additional information useful in the development and deployment lifecycle, including:

- **parameters** – variables that represent flexibility in the implementation of the TTP. Parameters may relate to table capacity, optional feature support, or others aspects in which implementations may vary.
- **flow paths** – a set of valid paths through the set of flow tables, used to more precisely specify the forwarding behavior supported by a logical switch.

A TTP is intended to describe the behavior of a switch that is controllable by an OpenFlow Controller. An OpenFlow Logical Switch (OFLS) can support the TTP based on an allocation of resources provided by a physical switch. Each OFLS can be allocated distinct resources, and capacity and/or optional feature capabilities may vary from one OFLS to another. The resource allocation and optional feature support can be negotiated at initialization time between the OF Controller and the OFLS. This enables a TTP to be used in a variety of deployment scenarios with varying resource requirements (as long as the required OFLS behavior is the same).

A TTP is used to describe the capabilities required of an OpenFlow Logical Switch in a particular network application. That is, it is not expected that any TTP would describe all the capabilities of a particular OpenFlow Capable Switch. Instead, an OpenFlow Capable Switch would normally support a number of TTPs, and each TTP would describe a set of capabilities required of an OpenFlow Logical Switch playing a particular role in a network architecture. A TTP can include the following members:

1) **metadata**, including a unique **TTP identifier,**
2) **security** considerations and signing**,**
3) **identifier** descriptions
4) required OpenFlow protocol **features**
5) a **table map,**
6) a **meter table**,
7) a list of **flow tables**, each specifying a set of **flow_mod types**,
8) a list of **group table entry types**,
9) a set of **TTP parameters**, and
10) a set of **flow paths**.

The language used to write a TTP description is essentially a meta language for describing a set of valid flow_mods, group_mods, and meter_mods. Therefore the language includes reserved words that correspond to data elements or data values carried in these control messages. In many cases these reserved words are taken from symbols defined in the OpenFlow-Switch specification. The TTP language may evolve over time due to 1) extensions to the TTP description structure, and 2) changes to the OpenFlow-Switch protocol. Therefore the TTP metadata includes an indication of the version of the TTP language used in writing the TTP.

This specification defines version 1 of the TTP description language (TTPv1). The elements of a TTP are described in the following sections and an example TTP is provided in Appendix C.

The essential structure of a TTP description is driven by its purpose as a specification of valid OpenFlow control messages. A TTP description could be encoded in a variety of ways but in light of the possibility of developing TTP toolsets there is a preference for machine parsable encodings. A number of machine parsable encodings may be considered including XML, JSON and YAML. In this document we use JSON encoding (RFC4627) because it is relatively human-friendly (i.e., easier for the author to write and the reader to read) and has a reasonable level of support in the industry. In the JSON encoding numbers are used occasionally (e.g., for priority values); however, since JSON limits number formats to decimal notation many header field and mask values are represented as strings. These strings may employ various formats, including the following:

- C-style hexadecimal format: `"0xhhhh"` where `'h'` is a hexadecimal digit
- IEEE 802 MAC address format – `"hh-hh-hh-hh-hh-hh"` where 'h' is a hexadecimal digit
- IPv4 address dotted decimal format – `"dd.dd.dd.dd"` where `'d'` is a decimal digit
- IPv6 address format – `"hhhh:hhhh::hhhh"` where `'h'` is a hexadecimal digit (as described in RFC 5952)

Note that while a machine readable representation has been specified, not all aspects of a TTP are machine accessible. Some constraints specified by a TTP are left to human readable commentary, i.e. "doc" members, and are thus not in a form suitable for machine consumption. In future revisions of this document additional requirements or constraints may be supported in machine readable form.

Following the JSON style a TTP description is structured as an object comprising a set of members (name:value pairs). Each of the members is itself either an object (a set of name:value pairs) or an array of values. Each value may be an object, array, string, or number. Thus a TTP is structured as a tree of objects and arrays with strings and numbers at the leaf points. The keywords naming each member of a TTP are shown in Table 1.

TTP Member	**Keyword**
Metadata	`"NDM_metadata"`
Identifier descriptions	`"identifiers"`
OpenFlow features required	`"features"`
Table map	`"table_map"`
Meter table	`"meter_table"`
Flow tables	`"flow_tables"`
Group table entry types	`"group_entry_types"`
Required PACKET_OUT actions	`"packet_out"`
TTP parameters	`"parameters"`
Flow paths	`"flow_paths"`
Security considerations and signing	`"security"`

Table 1: TTP member keywords

There are three member names that may be used in any object in the TTP. These are listed in Table 2.

Member	**Keyword**
Object name	`"name"`
Commentary	`"doc"`
Optional functionality tag	`"opt_tag"`

Table 2: General member keywords

The "name" member has a string value that can be used elsewhere in the TTP to refer to the object. The "doc" member is an arbitrary length string that can be used to provide comments on the object or to specify details for which a machine readable format is not provided. To improve readability the `doc` member may be represented as an array of strings to be concatenated, with newlines or spaces inserted between elements, when displayed. The "opt_tag" member is used to indicate optional functionality and is described below.

There are meta-members that may be inserted into a TTP where it is either (a) optional to support a branch of the TTP, or (b) optional to use a branch of the TTP. The use and placement of meta-members is up to the author of the TTP. The keywords for optional support meta-members are shown in Table 3 and the keywords for optional use meta-members are shown in Table 4.

Support Meta-Member	**Keyword**
All elements on the branch must be supported (default)	`"all"`
At least one element on the branch must be supported	`"one_or_more"`
Support for all elements on the branch is optional	`"zero_or_more"`

Table 3: Support meta-member keywords

Use Meta-Member	**Keyword**
One element of the branch must be used	`"exactly_one"`
One element of the branch may be used	`"zero_or_one"`

Table 4: Use meta-member keywords

Typically the meta-members are used to select branches from an array. That is, an array of elements is converted into an object containing meta-members. Each meta-member has as its value an array that is a subset of the original array or one element of the original array. Meta-members can be used to make a TTP more compact; however, they should be used sparingly as they may make interpreting the TTP more complex.

The support meta-members provide a simple optioning capability for cases in which support of the TTP does not requires support of an optional branch. A more powerful form of optioning is also supported for cases in which an optional functionality requires support of objects in distinct branches of the TTP. In these cases objects may be tagged by including an "opt_tag" member with a value representing the optional functionality. The inclusion of an "opt_tag" member is similar to the insertion of a "zero_or_more" support meta-member before the object, but it also provides a tag value that can be associated with other objects in the TTP. All of the objects with a given "opt_tag" value are considered as a unit when negotiating support for the optional feature represented by that value. In other words, when an optional functionality is provided by multiple flow-mod-types in separate flow tables or includes associated group bucket types, that functionality is described in distinct branches of the TTP. Typically, the set of flow-mod-types and group bucket types required for the optional functionality should be enabled or disabled together. The opt_tag member associates those separate flow-mod-types and group bucket types as a single unit, providing a means for the TTP designer to make clear that the optional functionality comes as a single package.

Optional functionality can be negotiated, as part of TTP negotiation, using the "OptFunc" built-in parameter (see section 3.9).

3.1 TTP Metadata

Each TTP is uniquely identified by the following information:

- the organization defining the TTP (a string unique to the organization in reverse-DNS format)
- the NDM type string "TTPv1" indicating the TTP description language and version
- the name of the TTP

- a version number, which may include multiple parts, indicating the version of this TTP description

In a TTP description the `NDM_metadata` member contains this identifying information. NDM metadata also indicates the OpenFlow-Switch protocol version on which the TTP is based. It should also include a general description of the TTP (i.e., a "doc" member). For example:

```
  "NDM_metadata": {
    "authority": "org.opennetworking.fawg",
    "type": "TTPv1",
    "name": "L2-L3-ACLs",
    "version": "1.0.0",
    "OF_protocol_version": "1.3.3",
    "doc": ["Example of a TTP supporting L2 (unicast, multicast, flooding), L3 (unicast only),",
            "and an ACL table."]
  }
```

The keywords naming the members of the NDM_metadata object are shown in Table 5.

Metadata Member	**Keyword**
Organization responsible for the TTP	`"authority"`
NDM type (language version)	`"type"`
TTP name	`"name"`
Version of this TTP	`"version"`
OpenFlow-Switch protocol version on which the TTP is based	`"OF_protocol_version"`

Table 5: Metadata member keywords

The unique TTP identifier comprises the four members `authority`, `type`, `name`, and `version`. Uniqueness is provided by allowing each TTP author to use their own DNS name to form the authority member and within that manage their own TTP name and version identifiers. The TTP's identifying information can also be encoded as a string comprising the authority, NDM type, name, and version separated by forward slashes. For example, the TTP identifying information above is encoded as follows:

```
org.opennetworking.fawg/TTPv1/L2-L3-ACLs/1.0.0
```

The version number contains three subfields: <major>.<minor>.<edit>. The last subfield <edit> is numeric and increasing for newer versions. From an implementation perspective, changes in the <edit> field should make no difference (e.g., there may be changes in variable names, flow table names, doc strings, etc. but nothing that impacts interoperability). The <minor> subfield is numeric and increases for newer versions. When the <minor> subfield is incremented, the newer TTP should be a compatible superset of the previous TTP version, providing the same tables (with the same table numbers) and same parameters (with the same parameter names) as the earlier TTP. The point is, the same flow mod messages and the same parameter negotiations that are acceptable in the earlier TTP version must still be valid in the new TTP version differing only in minor version number. Thus the newer TTP version can add a new flow-mod-type, but it cannot remove an old flow-mod-type, or change a parameter name, or renumber tables. The latter changes would require a increase in the TTP's major version number. This means that a controller (or OFCP) that is looking for TTP version 1.2.0 can use 1.3.0 if it finds that instead (but not necessarily the other way around). The <major> subfield is numeric and increases for newer versions. When a new version of a TTP requires changes to flow-mod-types or parameters, then the new TTP version will not offer backward compatibility to the previous version at the level of OpenFlow Switch messaging. Instead, the new TTP version offers compatible flow paths for all (or at least most) flow paths in the previous TTP version. ("Compatible flow paths" means that the new path contains the same packet modifications and forwarding actions, based on the same match fields, as the earlier flow path.) But the new matches and actions could be located in different tables. In other words, a new table may have been inserted in order to support new flow paths (new packet handling) and so the previous flow-mod messaging and constraints would not work with the new TTP. Still, an application that worked with the previous TTP could be updated to work on the new TTP. When changes to a TTP are so dramatic that there is very little backward compatibility, then the time has come to change the name rather than the version number.

3.2 Identifiers

The `identifiers` member is an array of variable names and/or extension identifiers with descriptions.

Variables convey information about constraints on values that may be used in flow_mods, group_mods, or meter_mods specified by the TTP. A variable name is enclosed in angle brackets as shown in the example below.

```
{"var": "<port_vid>",
 "doc": "A VLAN ID to be assigned to untagged or priority tagged frames received on a port."},
```

Some variables represent elements that must conform to the flow_mod, group_mod, or meter_mod types defined in the TTP. These variables are not included individually in the variables list. Instead the structure of these variables is described, as in the example below for group table entry type variables.

```
{"var": "<<group_entry_types:name>>",
 "doc": ["An OpenFlow group identifier (integer) identifying a group table entry",
         "of the type indicated by the variable name"]}
```

The extra set of angle brackets indicate that the variable name itself is variable and, as the description indicates, the variable name must be the name of a group entry type defined in the TTP.

Variable Element Member	**Keyword**
Variable name	`"var"`
Variable range	`"range"`
Variable description	`"doc"`

Table 6: Variable element member keywords

A variable can specify a range of valid values using the `range` member. This member has a string value and the convention `"x..y"` where `x` and `y` are integers is used to indicate a range of integer values.

```
{"var": "<local_vid>",
 "range": "1..4094",
 "doc": "A VLAN ID valid on the wire at a port."},
```

Extension identifiers allow string names to be assigned to extensions for instructions, actions, match fields, or error codes. This allows these identifier strings to be used in flow_mod type specifications along with strings taken from the OpenFlow standard. When used the identifier strings must be prefixed with "$" to indicate that they are identifiers defined in the TTP. The identifier strings defined must be unique, that is, no two extension ID elements can use the same identifier string.

Extension ID Element Member	**Keyword**
Identifier string	`"id"`
Identifier type	`"type"`
Experimenter ID	`"exp_id"`
Experimenter code	`"exp_code"`
Identifier description	`"doc"`

Table 7: Extension identifier element member keywords

Each identifier element includes the following members:

• `id`:	The identifier string
• `type`:	One of "field", "inst", "action", or "error"
• `exp_id`:	The Experimenter ID for the extension
• `exp_code`:	The code point for the extension
• `doc`:	A description of the identifier

The `exp_code` member can be used to indicate a code point, if one is defined, that distinguishes this extension from others of the same kind using the same Experimenter ID. For example, if an Experimenter ID is used to add multiple actions or instructions and these extensions include a code point for distinguishing the added actions or instructions, that code point can be indicated using the `exp_code` member. If a header field extension uses the `oxm_field` value to distinguish different header fields added under the same Experimenter ID, the exp_code member can indicate the `oxm_field` code point. For an error code extension the `exp_code` should indicate the value for `exp_type` defined for the added error. If the `exp_code` member is not useful it can be omitted.

3.3 Required Protocol Features

If specific protocol features are required for the application(s) the TTP is intended to support, these can be indicated in the `features` member. This member is an array of feature descriptions. A feature description may be machine readable or not, depending on whether or not there is a specific named element in the protocol associated with the feature.

Feature Element Member	**Keyword**
Feature identifier	`"feature"`
Feature description	`"doc"`

Table 8: Extension identifier element member keywords

The `feature` member is a string that identifies the required feature. A string derived from the openflow.h header file can indicate a specific message element is required. Such strings use language element names in the header file to identify specific message fields. For example, the string `"ofp_flow_stats.packet_count"` would indicate that the packet counter for a flow entry is required (a field in the OFPMP_FLOW reply message). The string `"ofp_flow_mod.idle_timeout"` would indicate that flow entry aging must be supported (a field in the

OFPT_FLOW_MOD message). If a feature is required for specific cases but does not have to be universally supported, this can be indicated using a context identifier. A context identifier can be a table name, flow_mod type name, group entry type name, or built-in flow entry name followed by "`::`". For example, if the flow entry packet count is only required for the built-in flow entry used for MAC learning in the TTP example in Appendix C, the string would be "`MAC-Miss::ofp_flow_stats.packet_count`".

Extensions that do not have a simple association with message elements or are associated with too many message elements to be practical can be indicated using a suggestive string. For example, if message bundle support is required the string "Bundle messages" could be used. Alternatively the extension number can be used as a way to refer to the document specifying a particular extension. For message bundles the string "ext230" could be used to refer to the file openflow-switch-extension-ext230-stateless.pdf in which this extension is specified for OpenFlow 1.3. For example, a requirement to support notification of changes in the flow tables, group table, and meter table can be indicated as shown here.

```
"features": [
  {"feature": "ext187",
   "doc": "Flow entry notification Extension - notification of changes in flow entries"},
  {"feature": "ext235",
   "doc": "Group notifications Extension - notification of changes in group or meter entries"}
]
```

When a suggestive string is used the description in the `doc` member must make the feature requirement clear enough for an implementer to understand it.

3.4 Table Map

The `table_map` member is an object in which each member name is a table name associated with a flow table in the TTP and the member value is the table number assigned to that flow table. These tables are not required to be distinct types; however, it may be uncommon to find multiple flow tables of the same type in a TTP. Including the table map allows flow tables to be referenced by name in the remainder of the TTP making the flow table relationships clearer for the reader. This also enhances portability and reuse of chunks of TTP description and eases maintenance when table numbers must change but the relationships between tables (i.e., Goto-Table instructions) do not. For example:

```
"table_map": {
  "ControlFrame": 0,
  "IngressVLAN": 10,
  "MacLearning": 20,
  "ACL": 30,
  "L2": 40,
  "ProtoFilter": 50,
  "IPv4": 60,
  "IPv6": 80
}
```

The table map also governs the assignment of flow table numbers. The first table in the array is assigned table number 0 (zero) and each subsequent table is assigned a higher table number. The numbering may have gaps (e.g., subsequent tables may be numbered 10, 20, 30, etc.) to allow for the addition of new flow tables to a revised TTP without requiring changes to the numbering of existing tables.

© 2014 Open Networking Foundation

3.5 Meter Table

The `meter_table` member is an object containing an array of meter type definitions and an array of built-in meter definitions.

Meter Table Member	**Keyword**
Meter type definitions	`"meter_types"`
Built-in meter definitions	`"built_in_meters"`

Table 9: Meter table member keywords

Each meter type definition includes a name , the band type ("DROP" or "DSCP_REMARK"), and the range of valid values for rate and burst size. The defined meter types may be used in `flow_mod_types` or `built_in_flow_mods` where indicated in the flow tables section of the TTP. For example the member

```
"meter_types": [
  {"name": "ControllerMeterType",
   "bands": [{"type": "DROP", "rate": "1000..10000", "burst": "50..200"}]
  },
  {"name": "TrafficMeter",
   "bands": [{"type": "DSCP_REMARK", "rate": "10000..500000", "burst": "50..500"},
             {"type": "DROP", "rate": "10000..500000", "burst": "50..500"}]
  }
 ]
```

defines two meter types, one for limiting traffic to the controller and one for policing IP traffic flows. The instantiation and detailed configuration of meters (i.e., setting specific rate and burst size) are left to the controller when the TTP is taken into use.

A meter instruction with a meter type name may be included as an option in the instruction set of a flow mod type description to indicate that the controller may configure a meter and include it in a flow mod of that type to meter the associated traffic flow.

Meter Types Element Member	**Keyword**
Meter type name	`"name"`
Meter type description	`"bands"`

Table 10: Meter type element member keywords

Meter Band Element Member	Keyword	Valid Values
Meter band type	`"type"`	`enum ofp_meter_band_type string after "OFPMBT_"`
Meter band rate range	`"rate"`	`string "m..n" indicating range of supported meter rates in KBPS`
Meter burst range	`"burst"`	`string "m..n" indicating range of supported burst sizes`

Table 11: Meter type bands element member keywords

Each built-in meter definition specifies a meter instance that is expected to exist when the TTP is taken into use. The controller may adjust the meter parameters, if desired, within the limits of the meter's associated type. For example, the following member defines two built-in meters to be used to limit traffic sent to the controller.

```
"built_in_meters": [
  {"name": "ControllerMeter", "meter_id": 1,
    "type": "ControllerMeterType", "bands": [{"rate": 2000, "burst": 75}]},
  {"name": "AllArpMeter", "meter_id": 2,
    "type": "ControllerMeterType", "bands": [{"rate": 1000, "burst": 50}]}
 ]
```

Built-In Meter Element Member	Keyword	Valid Values
Meter name	`"name"`	`string`
Meter identifier	`"meter_id"`	`meter instance number`
Meter type	`"type"`	`name of a meter type defined in this TTP`
Meter bands	`"bands"`	`array of meter band objects`

Table 12: Built-in meter element member keywords

Built-In Meter Bands Element Member	Keyword	Valid Values
Meter band rate	`"rate"`	`number within the rate range for the meter type`
Meter burst	`"burst"`	`number within the burst range for the meter type`

Table 13: Built-in meter band element member keywords

3.6 Flow Tables

The `flow_tables` member is an array of flow table descriptions. Each flow table description comprises an array of `flow_mod_types` and, optionally, an array of `built_in_flow_mods` and an array of `table_subsets`. Each flow table is given a name to be included in the table map and used in GOTO_TABLE instructions. A description may be provided to explain the role or function of the flow table in the TTP. The structure of a flow table description is shown in the following example:

```
{
  "name": "IngressVLAN",
  "doc": ["Ingress VID processing table, including:",
          " - accepting or blocking untagged and priority tagged frames",
          " - accepting or blocking VLAN tagged frames",
          " - ingress VID filtering control",
          " - (optional) ingress VID translation" ],
  "flow_mod_types": [
    …
  ],
  "built_in_flow_mods": [
    …
  ],
  "table_subsets": [
    …
  ]
}
```

The `flow_mod_types` member may include optional elements, i.e., flow_mod types that may or may not be supported by a switch supporting the TTP. Optional flow_mod types are indicated by inserting a meta-element in the array as described in 3.6.1.

Flow Table Element Member	**Keyword**
Flow table name	`"name"`
Flow table entry types	`"flow_mod_types"`
Built-in flow table entries	`"built_in_flow_mods"`
Named paths through this flow table	`"table_subsets"`

Table 14: Flow table element member keywords

The following subsections describe the parts of a flow table description.

3.6.1 flow_mod Types

Each flow_mod type element comprises at least a `name`, `match_set`, and `instruction_set`. It may also include a `priority` and `description`. For example, the following flow_mod type element allows priority tagged MAC frames to ingress a switch port.

```
{
  "name": "Allow-Priority-Tagged",
  "priority": "6..7",
  "doc": "Allow priority tagged traffic on a port or all ports.",
  "match_set": [
    {"field": "IN_PORT", "match_type": "all_or_exact"},
    {"field": "VLAN_VID", "mask": "0x1fff", "value": "OFPVID_PRESENT"}
  ],
  "instruction_set": [
    {"instruction": "APPLY_ACTIONS",
      "actions": [
        {"action": "SET_FIELD", "field": "VLAN_VID", "value": "<port_vid>"}]},
    {"instruction": "GOTO_TABLE", "table": "MacLearning"}
  ]
 }
```

The priority of a flow_mod type may be specified to interact appropriately with other flow_mod types defined for the flow table. In the example above, the priority must be higher than the built-in flow_mod instance in the table ("Default-Allow-Priority-Tagged") that sets VLAN_VID to 1 (the default VID) to ensure VLAN_VID is set to the <port_vid> value instead.

Variables of the form `"<variable_name>"` may be used to indicate values to be used in flow_mods. This enables both indicating the role of the value (as with `"<port_vid>"` in the example above) and indicating where the same value is used in different places in the TTP.

Flow_Mod Type Member	**Keyword**
Flow_mod name	`"name"`
Flow_mod priority	`"priority" or "priority_rank"`
Match condition	`"match_set"`
Instructions	`"instruction_set"`

Table 15: Flow_mod type member keywords

The `priority` and `priority_rank` specify the valid priority values for the flow_mod type. The value of a `priority` member is an integer greater than zero, or an integer range of the form `"n..m"` where `n` and `m` are integers with `0 < n < m`. Specifying a priority range allows the controller to differentiate flow_mods of the same type, for example to make a narrower match higher priority than an overlapping broad match. The value of a `priority_rank` member is an integer greater than zero. Priority rank can be used to specify relative priorities for each flow_mod type. Flow_mods having types of different priority rank must be assigned priorities in non-overlapping ranges, selected by the controller, and the range for a lower rank must be lower than the range for a higher rank. Using priority rank allows more flexibility for the controller to assign priorities but may make mapping the TTP to fixed-function hardware platforms more difficult. If no priority member is included in the flow_mod type, any priority greater than zero is valid for flow_mods of that type. When the match type "prefix" (see below) is used the priority for a flow_mod must be set to the length of the prefix mask, so no priority member is allowed in the flow_mod type specification in this case.

The `match_set` member is an array of match field descriptions. Each match field description includes a `field` member identifying the header or metadata field to be matched. Some fields have prerequisite field matches that OpenFlow requires be included in the match set. If a match set includes a match field with prerequisites, the transitive closure of these prerequisites is assumed to be included in the match set as well, whether or not they are explicitly listed. In addition to identifying the field, a match field member may include any of the following:

- `match_type`: match type for the field, one of
    - `exact` – exact match only, equivalent to all 1's mask, default if match_type not specified
    - `mask` – arbitrary bit mask allowed

  - `prefix` – mask must be a set of contiguous bits, beginning with the most significant bit of the field
  - `all_or_exact` – mask is either all 0's or all 1's (i.e., field may be omitted in match)

- `mask`: a bit mask identifying bit positions that may be matched to a variable value (must be consistent with match_type)
- `value`: a fixed value for the field or a variable name
- `const_mask`: a bit mask indicating bit positions that must have fixed values in the match (needed only if there are other bits that are to be matched to a variable)
- `const_value`: a value indicating the required settings for the bits identified in const_mask (needed only if there are other bits that are to be matched to a variable)

The match type "prefix" is used to indicate Longest Prefix Match (LPM) and may be associated with at most one field in a match set. There may be other fields in an LPM match set and is up to the TTP author to ensure that an extended match set is unambiguous and implementable.

For example, an IP forwarding flow table could use a prefix match type to implement basic longest prefix match (LPM) for an IPv4 address as shown in the example below.

```
"match_set": [
  {"field": "IPV4_DST", "match_type": "prefix"}
],
```

Matches may use a constant mask and value to ensure a subset of the bits in a masked match field are set or unset and a mask and variable value for the remainder of the field. For example, a flow_mod for flooding in a VLAN (when no higher priority flow_mod matches the ETH_DST) may use a constant mask and value to ensure the VID is valid and a variable match against the (masked) VID value.

```
"match_set": [
  {"field": "VLAN_VID", "const_mask": "0x1000", "const_value": "0x1000",
     "mask": "0x0fff", "value": "<VID>"}
 ]
```

Match Set Element Member	**Keyword**	**Valid Values**
Match field	`"field"`	`enum oxm_ofb_match_fields string after "OFPXMT_OFB_"`
Match type	`"match_type"`	`"exact", "mask", "prefix", "all_or_exact"`
Mask for matching bits	`"mask"`	`bitmap`
Value for matching bits	`"value"`	`bitmap or variable name`
Mask for fixed value bits	`"const_mask"`	`bitmap`
Value for fixed value bits	`"const_value"`	`bitmap`

Table 16: Match set element member keywords

The `instruction_set` member is an array of instructions for the flow_mod. These are encoded following the OpenFlow instruction structure. For example, an instruction set to push a VLAN tag with the port VID onto an untagged frame and continue to the next table is shown below. The OpenFlow-Switch specification constrains the content and execution order of an instruction set. The instruction set may include at most one instruction of each type and, although the instructions in an instruction set may be listed in any order, it is recommended that instructions be listed in the OpenFlow specified order of execution.

```
"instruction_set": [
  {"instruction": "APPLY_ACTIONS",
    "actions": [
      {"action": "PUSH_VLAN"},
      {"action": "SET_FIELD", "field": "VLAN_VID", "value": "<port_vid>"}]},
  {"instruction": "GOTO_TABLE", "table": "MacLearning"}
 ]
```

Instruction Set Element Member	**Keyword**	**Valid Values**
Instruction type	`"instruction"`	`enum ofp_instruction_type string after "OFPIT_"`
Table name	`"table"`	`string name of a table defined in this TTP`
Metadata	`"metadata"`	`64 bit value`
Metadata mask	`"metadata_mask"`	`64 bit bitmap`
Action list	`"actions"`	`array of action objects`

Table 17: Instruction set element member keywords

Action Object Member	Keyword	Valid Values
Action type	"action"	enum ofp_action_type string after "OFPAT_"
Port identifier	"port"	port number or variable name or reserved port name (enum ofp_port_no string after "OFPP_")
Group identifier	"group_id"	group table entry type name defined in this TTP
Queue identifier	"queue_id"	queue number for the switch port or variable name
MPLS TTL value	"ttl"	MPLS TTL value or variable name
IP TTL value	"ttl"	IP TTL value or variable name
EtherType value	"ethertype"	EtherType value
Set field name	"field"	enum oxm_ofb_match_fields string after "OFPXMT_OFB_"
Set field value	"value"	value for the field being set, e.g., bitmap of the appropriate size, or variable name
Meter name	"meter_name"	Name of a meter type or built-in meter

Table 18: Action object member keywords

The match set and instruction set may include optional elements indicating that a controller may optionally select these elements in creating a flow_mod. All the optional elements in a match set or instruction set must be supported by a switch supporting the TTP so the controller is free to select any match set or instruction set option.

Examples of optional instruction_set elements:

Selecting among different instruction sets, without optional instructions:

```
"instruction_set": {"exactly_one": [
 [
   {"instruction": "METER", "meter_name": "ControllerMeter",
      "doc": ["This meter may be used to limit the rate of PACKET_IN frames",
             "sent to the controller"]},
   {"instruction": "APPLY_ACTIONS",
      "actions": [{"action": "OUTPUT", "port": "CONTROLLER"}]
   }],
 [
   {"instruction": "APPLY_ACTIONS",
      "actions": [{"action": "OUTPUT", "port": "CONTROLLER"}]
   }]
]}
```

© 2014 Open Networking Foundation

Optional instructions within an instruction set:

```
"instruction_set": [
  {"zero_or_one": {"instruction": "METER", "type": "TrafficMeter"}},
  {"instruction": "GOTO_TABLE", "table": "IPv4"}
]
```

Optional flow_mod types can be included in a TTP. These are not required to be supported to claim support for the TTP, but if they are supported they provide additional capabilities that may be useful in some applications of the TTP. To indicate optional support for some flow_mod types meta-members are inserted in the flow_mod_types array as follows:

```
"flow_mod_types": [
 {"all": [

  <required flow_mod types>

 ],
 "zero_or_more": [

  <optional flow_mod types>

 ]}
]
```

The flow_mod types preceded by "all" are required. Those preceded by "zero_or_more" are optional. If at least one flow_mod type in a particular set must be supported but support for more than one is optional the meta-member "one_or_more" may be used.

3.6.2 Built-in flow_mods

The `built_in_flow_mods` member of a flow table description is an array of specific flow entries that are required to be installed in the flow table. These are usually flow_mods that implement default behaviors that are expected to be present without any flow programming. For example, a built-in flow_mod to block the set of control addresses that are not permitted to be forwarded by an 802.1Q bridge is shown below.

```
{
  "name": "Control-Frame-Filter",
  "doc": "Mandatory filtering of control frames with C-VLAN Bridge reserved DA.",
  "priority": 1,
  "match_set": [{"field":"ETH_DST","mask":"0xfffffffffff0","value":"0x0180C2000000"}],
  "instruction_set": []
 }
```

A built-in flow_mod includes a `name`, `priority`, `match_set`, and `instruction_set` and optional `description`. The members for a built-in flow_mod object are the same as those for a flow_mod type object (Table 15, Table 16, Table 17, and Table 18) except that the match set and instruction set must not include any variable names, though they may contain identifiers with well-defined values, and meta-members must not be used.

A built-in flow mod may not be modified or deleted by the controller. If the Controller sends a flow_mod matching a built-in flow_mod in both the match set and the instruction set the switch should accept it (i.e., return a successful response) without altering the built-in flow table entry except to set the cookie value (for detailed behavior see B.3). It is possible to override a built-in flow mod using a higher priority flow mod if a compatible flow_mod type is defined for the flow table.

3.6.3 Table Subsets

The table_subsets member is an array of table subset definitions. Each table subset has a name that is an alias for a set of flow_mod types or flow_mod instances. The table subset name can be used in flow path descriptions to indicate progressing through a flow table via any of the flow_mod types or instances in that table subset. For example, a subset for an IngressVID table may be defined as follows:

```
"table_subsets": [
  {"name": "IV-pass",
   "subset": [ "Allow-Untagged", "Allow-Priority-Tagged", "Disable-Ingress-VID-Filter",
               "Ingress-VID-Allow", "Ingress-VID-Filter", "Ingress-VID-Translate",
               "Default-Allow-Untagged", "Default-Allow-Priority-Tagged",
               "Default-Disable-Ingress-VID-Filter" ]
  },
  {"name": "IV-drop",
   "subset": [ "Block-Untagged", "Block-Priority-Tagged", "Enable-Ingress-VID-Filter",
               "MISS" ]
  }
 ]
```

This defines two subsets, one that pass to the next table (IV-pass) and one that drops the packet (IV-drop). Table subsets can be used as aliases for a set of flow_mods in flow paths where finer grained identification of the path through a particular flow table is not needed. The table subset indicates that any one of flow_mods in the table subset may be matched as a part of the flow path.

Table Subsets Element Member	**Keyword**	**Valid Values**
Name of table subset	`"name"`	`string`
Set of entry types	`"subset"`	`array containing names of flow_mod types or built-in flow mods defined for this flow table or "MISS" (representing the table miss condition)`

Table 19: Table subsets member keywords

Defining table subsets is not required; however, these may be helpful in reducing the number of flow paths that must be defined to represent the required set for the TTP. For example, the "IV-pass" table subset in the example above allows one flow path description to include all the paths through the IngressVLAN flow table (as described in 3.10). This technique reduces the number of flow path descriptions by using a single flow path description to represent multiple flow paths, each including a different flow mod type or flow mod instance from a given flow table.

3.7 Group Table Entries

If a TTP uses the group table, the types of group table entries required are described in the `group_entry_types` member. Each group entry type includes a `name`, `group_type`, and an array of `bucket_types`, and optionally a `description`. The group type is one of the OpenFlow group types: ALL, SELECT, INDIRECT, or FF (fast failover). Each bucket type includes a name and an OpenFlow action set. It is recommended that the actions in the action set be listed in the order of execution specified by the OpenFlow protocol.

```
{
  "name": "EgressPort",
  "doc": ["Output to a port, removing VLAN tag if needed.",
          "Entry per port, plus entry per untagged VID per port."],
  "group_type": "INDIRECT",
  "bucket_types": [
    {"name": "OutputTagged",
     "action_set": [{"action": "OUTPUT", "port": "<port_no>"}]
    },
    {"name": "OutputUntagged",
     "action_set": [{"action": "POP_VLAN"},
                    {"action": "OUTPUT", "port": "<port_no>" }]
    },
    {"opt_tag": "VID-X",
     "name": "OutputVIDTranslate",
     "action_set": [{"action": "SET_FIELD", "field": "VLAN_VID", "value": "<local_vid>"},
                    {"action": "OUTPUT", "port": "<port_no>" }]
    }
  ]
}
```

In the example above a group entry of type EgressPort can contain only one bucket (i.e., INDIRECT group type allows only one bucket) but that bucket may either a bucket that executes one output action (OutputTagged) or a bucket that strips the VLAN tag from the packet and then executes one output action (OutputUntagged). A third bucket type that translates the VID on egress (OutputVIDTranslate) is available if the "VID-X" option is supported. Since the INDIRECT group type limits the group entry to one bucket there is no need to use a meta-member in the TTP to impose this constraint.

Group Table Entry Type Element Member	**Keyword**	**Valid Values**
Group table entry type name	`"name"`	`string`
Group type	`"group_type"`	`enum ofp_group_type string after "OFPGT_"`
Bucket types allowed	`"bucket_types"`	`array containing bucket objects or names of bucket type objects defined elsewhere in this TTP`

Table 20: Group table entry member keywords

Bucket Type Member	**Keyword**	**Valid Values**
Bucket type name	`"name"`	`string`
Action set	`"action_set"`	`array of action objects` (Table 18)

Table 21: Bucket type object member keywords

3.8 PACKET_OUT support

A TTP may contain a `packet_out` member specifying the actions that can be associated with a PACKET_OUT message. This member is an array of action lists. Each action list is an array of action objects (Table 18) representing an action list that may accompany a PACKET_OUT message. The port identifier `"TABLE"` may be used in an output action to indicate the packet is to be processed by the OpenFlow pipeline.

© 2014 Open Networking Foundation

3.9 TTP Parameters

A TTP may have a set of parameters that govern aspects of the TTP's instantiation in a logical switch. These may include flow table sizes, indication of support for optional features, or other aspects of a TTP's implementation in which a controller may have interest. Parameters are defined but do not have values in the TTP. The parameter values are negotiated or set when the OFLS is created or when it is connected to a controller. Some parameters represent capacities guaranteed by the switch. In this case the controller may use resources up to the limit of a stated parameter, but not beyond. For example, in the L2-L3-ACLs TTP the parameter set includes the table sizes for tables containing forwarding flow entries.

```
"parameters": [
  {"name": "ACL::TableSize", "type": "integer"},
  {"name": "L2::TableSize", "type": "integer"},
  {"name": "IPv4::TableSize", "type": "integer"},
  {"name": "IPv6::TableSize", "type": "integer"},
  {"name": "Meter::TableSize", "type": "integer",
     "doc": "Number of meters that can be configured in the switch."},
  {"name": "Meter::Accuracy", "type": "integer",
     "doc": "Accuracy of meters on the switch."},
  {"name": "OptFunc", "type": "array of opt_tag values",
     "doc": "Support for optional functions can be negotiated using the OptFunc parameter."}
]
```

Each parameter has a name and a type. A parameter description may, but is not required to, indicate a default value by including a "default" member.

Parameter Element Member	**Keyword**	**Valid Values**
Parameter name	`"name"`	`string`
Parameter type	`"type"`	`string`
Parameter default value	`"default"`	`value of parameter type`

Table 22: Parameter element member keywords

One built-in parameter is defined in this specification – the "`OptFunc`" parameter. This parameter may be used to negotiate support for optional functionality in the TTP. The value of the "OptFunc" parameter is an array of items that represent optional functionality, including names associated with optional flow-mod names, optional group-mod names, optional bucket names and opt_tag values. Optional functionality (in the form of flow_mod_types, group_mod_types and buckets that may be optionally supported in the switch) is expressed using support meta members and the opt_tag member. (See the discussion of support meta members and the `opt_tag` member earlier in Section 3.)

Some parameters are associated with specific flow tables (e.g., the maximum number of flow table entries supported). Parameters associated with a specific flow table use the flow table name as a scope identifier (a prefix followed by '::') in the parameter name. For example, the maximum number of entries in the MAC forwarding table "L2" is indicated by the parameter "L2::TableSize".

3.10 Flow Paths

The `flow_paths` member is an array of flow path descriptions, each representing a required (supported) path through the set of flow tables and group table entries in the TTP. A flow path description includes a `name` and a path. The path is an array of flow_mod type or built-in flow mod names and group entry type names indicating the entries executed for a packet processed by the flow tables in the TTP. For example a flow path for an L2 multicast frame in the L2-L3-ACLs TTP may look like the following:

```
{"name": "L2-4",
 "path": ["Non-Control-Frame","IV-pass","Known-MAC","ACL-skip","VID-flood",
          "VIDflood", [ "EgressPort" ] ]
 }
```

Each flow path specified in a TTP must be consistent with the instructions (e.g., GOTO_TABLE) and actions (e.g., GROUP) specified in the flow_mod types and group entry types to which they correspond. If this is not the case, the flow path is incorrect.

A flow path may use the name of a table_subsets element to represent any one of a set of flow mod types, all of which are valid for the given flow path. For example, the flow path above uses the table subset "IV-pass" that represents several possible paths through the IngressVLAN flow table. The remaining entries are names of specific flow_mod types or built-in flow_mods and group entry types. If multiple instances of a group entry type may be invoked (e.g., by multiple buckets in an ALL group) this is indicated by putting the group entry type name in brackets. For example, in the flow path above the EgressPort group entry type is enclosed in brackets to indicate that multiple entries of this type may be executed in this flow path.

Flow Path Element Member	**Keyword**	**Valid Values**
Flow path name	"name"	string
Supported flow path	"path"	array of object names from flow types, built-in flow_mods, table_subsets, or group entry types or "PACKET_OUT"
Non-supported flow path	"no-path"	array of object names from flow types, built-in flow_mods, table_subsets, or group entry types or "PACKET_OUT"

Table 23: Flow path element member keywords

Flow paths may be specified to reduce the set of paths that must be supported to a number less than the full set of possible paths through the TTP. They are also useful as a foundation for a set of test cases that can be used to check compliance with the TTP.

A flow path that results in a packet being sent to the controller may have the element "CONTROLLER" appended to the list as a hint to the reader. A flow path that is initiated by a PACKET_OUT message being output to "TABLE" may begin with "PACKET_OUT" as the first element.

The flow_paths member is not required to be included in a TTP. If no flow_paths member is included in a TTP it is assumed that all possible paths through the TTP must be supported. If a flow_paths member contains only "no-path" elements this indicates that all flow paths except those indicated by "no-path" must be supported.

3.11 Security

In current networks, attackers may need to spend significant effort to find the vulnerabilities of network devices which can be exploited in attacks. A TTP may provide attackers a chance to obtain detailed information about the behavior and capabilities of a logical switch. Using a TTP, an attacker may be able to analyze packet forwarding policies (e.g., firewall capabilities) and gain other useful information. Therefore, when writing a TTP it is important to carefully assess the design and try to identify and remove vulnerabilities which could be taken advantage of by attackers and result in potential security risks. As part of this work, for each TTP the designer should provide guidelines about how to set the values of adjustable parameters to prevent or limit security breaches.

Because a TTP is used to instruct the implementation of capable switches, a tampered or forged TTP may result in unacceptable loss to device vendors. For example, a TTP might be modified to change aspects of the datapath model that impact hardware design and result in incorrect or insufficient hardware implementations. A TTP might also be modified to automatically install flow entries that support traffic snooping and these may go undetected if the TTP implementation process is fully automated. Therefore, to enhance security when distributing TTPs additional measures can be taken to secure the TTP. For instance, when an organization publishes its TTPs on a website, the website can provide TLS for subscribers to authenticate the website and generate a security channel protecting the TTP documents from being tampered with during transfer. Sometimes subscribers also need to be authenticated, especially when they try to download private TTPs. The sensitive contents of private TTPs need to be encrypted during transfer.

Note that it is possible to publish public TTPs on third party websites which may not have sufficient security protection. Therefore, there should be a means to associate a TTP with sufficient information to prove it validity. First of all, in order to avoid confusion, each TTP must have a globally unique ID. The `NDM_metadata` member fulfills this unique identity requirement. In addition, the publisher should generate a signature for the TTP with its private key and provide necessary instructions for subscribers to find out the proper public key for verifying the signature. Existing certificate management mechanisms such as PKI, PGP, and DKIM can all be candidates. The valid period of the signature should be provided. Any TTP with expired signatures must not be trusted any more. Existing techniques for signing documents can be used for TTPs. The signature members of the security object provide these capabilities.

To address these security considerations a TTP can include a `security` member that provides guidance on secure use of the TTP and secures the TTP itself.

```
"security": {
  "doc": ["This TTP is not published for use by ONF. It is an example and for",
          "illustrative purposes only.",
          "If this TTP were published for use it would include",
          "guidance as to any security considerations in this doc member."]
}
```

Security Member	Keyword	Valid Values
Security guidelines	`"doc"`	`string providing guidance on secure use of the TTP`

Table 24: Security member keywords

The `doc` member contains guidance related to the secure use of the TTP. The `sig` member is a digital signature, securing the content of the TTP.

As TTPs become more widely used in SDN development the ability to secure their content will become more important. A TTP signing mechanism will be specified in a future revision of this document by including additional properties in the security object. The signature mechanism for a TTP expressed in JSON is likely to be based on JSON Web Signatures (JWS) being developed in the IETF JOSE working group.

Revocation of TTPs needs to be supported. If the signature of a TTP has expired, the publisher needs to update the TTP with a new signature. If a TTP is found to have drawbacks after it has been published, the document should be revised or revoked. The announcement of TTP revision or revocation can make use of the experience of certification revocation (e.g., certificate transparency, CRL, and etc.). An organization that publishes TTPs should maintain a list of TTPs that are no longer supported and provide a means for TTP users to determine whether a TTP is on that list.

TTP negotiation does not introduce additional security requirements to the OF-Config or OF-Switch protocols. Mutual authentication must be performed between Controller and Logical Switch whether or not TTP negotiation is used. Only an authorized OFCP can configure the TTP for a Logical Switch. Integrity and replay protection for OF-Config must be provided, just as other configuration operations. The messages used in the TTP negotiation process should be encrypted (e.g., using security mechanisms specified for the negotiation protocol) so that attackers cannot easily find out the TTPs supported by an OF capable switch and the negotiation results (e.g., the size of a flow table).

© 2014 Open Networking Foundation

Appendix A Benefits of TTPs and the TTP Lifecycle

TTPs and the TTP Lifecycle were conceived to advance adoption of complex OpenFlow (that is, OpenFlow architectures that leverage multiple flow tables), particularly on legacy ASIC-based platforms. TTPs and the TTP Lifecycle are expected to be helpful in these ways:

- Improving and clarifying product interoperability
- Simplifying implementation of multi-table OpenFlow
- Improving switch resource utilization
- Reducing the chicken-and-egg OpenFlow adoption challenge

Each of the above benefits will be discussed in the sections below.

First, let's revisit what a TTP is. A TTP, or "Table Type Pattern," is a structured and detailed description of a full set of logical switch behavior. The logical switch behavior is described in terms of familiar OpenFlow switch pipeline elements defined in terms of the OpenFlow Switch protocol specifications. We say that the defined switch behavior is "logical" because it is decoupled from any particular (physical) switch platform. Rather, it is defined in terms of the OpenFlow protocol, and specifies a set of flow tables and the OpenFlow messages (flow-mods, group-mods, etc.) that must be supported for each flow table. When products are developed in the context of a TTP, controller developers can be confident about what a switch will support, and switch developers know which messages must be supported. TTP switch models are useful in two distinct contexts:

- TTPs can be used as "switch profiles," useful for developing, testing and marketing OpenFlow products
    - TTP test profiles will be useful even for products with no built-in TTP awareness
- TTP awareness can be added to products for better support of multiple flow tables, e.g., strict checking of message validity against the TTP

A.1 Improving and Clarifying Interoperability

Recent versions of the OpenFlow Switch protocol define many interesting features that can be used to develop compelling SDN applications. But many of those features are optional, which means that applications using such features may not work with switches even if those switches are certified as conformant to the OpenFlow switch spec. In such a context, several difficulties arise.

- How can app architects predict which optional features will be supported in the marketplace?
    - How can switch implementers prioritize which optional features to invest in?
- How will buyers determine which products can interoperate when optional features are in play?

Test profiles have frequently been discussed within the Testing and Interoperability WG as an approach to mitigate the interoperability challenge. This interesting approach has also raised many questions, such as what the profiles should be and who should define them, etc. The development of the TTP concept supports the profile approach by providing a detailed method for various parties (from ONF workgroups or market participants) to precisely define switch behavior in OpenFlow terms, including optional OpenFlow switch functionality. The precision can be used to describe a range of breadth and depth, from individual flow-mod actions in multiple tables, to collections of flow-mods in a single table. The upshot is that, using TTPs,

- App developers can use TTPs to describe precisely what an app needs from its switches
- Switch developers can represent, with precision, key functionality that their products offer

- Testing and Interoperability WG or plugfest coordinators can define broad profiles so that interop events go smoothly
- Product datasheets can list TTP support to clarify app/switch interoperability for buyers

A.1.1 An Interoperability Thought Experiment

Let's imagine 50 products, 25 OpenFlow applications and 25 switch platforms. These numbers seem fairly realistic, perhaps a bit low. Without TTPs, how would market participants know what app works with what switch? Would all 25 switches need to be tested with all 25 switches? Would the application and switch vendors be constantly wrangling with each other who will do the testing and under what conditions? Would conformance testing play any role in such a scenario? How would adoption proceed in this picture, where the promise of interoperability is so elusive?

On the other hand, if the 25 applications can be characterized as operating within 5 common profiles, represented by well-known TTP descriptions, then each switch vendor can assess his own switches based on those well-known TTPs. Conformance testing can use TTPs to verify that applications and switches operate within the "rules" of various TTPs. Vendors can publish the TTPs they support, and customers can readily predict, with some confidence, what works with what. Interoperability has meaning. Network operators are able to justify SDN purchases. Adoption is not far-fetched in this picture.

A.2 Simplifying implementation of multi-table OpenFlow

Supporting a single OpenFlow flow table in a switch is fairly straightforward. In the presence of a single flow table, flow-mod messages convey all the relevant information describing the properties of a flow, as well as all the switch-based packet processing required for that flow. In such a situation, it is a mainstream engineering problem for a switch software designer to analyze each flow-mod message and determine whether it can map the combination of match fields and packet handling onto the underlying platform. This is true even when the underlying platform is a classic ASIC or merchant silicon device. One reason this is a manageable problem is that there is no uncertainty about the "OpenFlow pipeline" when a single flow table is used. This means that the flow-mod messages are only conveying control information, not pipeline structure information. But in the context of multiple flow tables, this changes.

In the traditional OpenFlow framework, mapping multiple flow tables onto classic chips (ASICs and merchant silicon) becomes unwieldy. Interoperability testing has provided some real-world evidence of this, for example as recorded in the Testing and Interoperability WG's June 2013 Plugfest Technical Document. A flow-mod message no longer fully defines a flow, nor fully defines flow handling. In addition, the switch lacks information about the effective switch pipeline that the controller's messages will be built on. Even when the application wants only those functions that are within the capabilities of the switch platform, the multi-table mapping challenge is unworkable due to the complexity of writing a general OpenFlow agent capable of handling arbitrary OpenFlow configurations. In addition, there is no "try it and see" option; production network operators require high confidence, before deployment, that applications (from multiple vendors) and switches (with differing platforms) will interoperate and reliably work "as advertised". In short, the complexity of mapping (either in the controller, or in the switch) diverse application requirements to diverse switch capabilities is too unwieldy and non-deterministic to be practical at run time in operational networks. But this does not need to spell doom for richly functional OpenFlow networking solutions or make such solutions solely dependent on highly programmable devices which have yet to gain the needed traction.

The remedy is the introduction of a pre-run-time “TTP compile phase” as described in the TTP Life Cycle. The TTP Description File defines the required switch pipeline details in terms that are useful for both the application and the switch adaptation layer. This clarity eliminates that uncertainty from the on-the-fly run-time challenge of mapping multiple flow tables onto classic chips, so that the implementation of run-time adaptation layers is again comparable to that of single table OpenFlow.

Initially, TTP compilation will be done by humans, but soon (say, 12 months after launching TTPs), leading vendors can be expected to offer a fully machine-based TTP compile phase. TTP Descriptions are both human and machine readable in order to allow both options. Compiler-generated “TTP plug-in” will be built, installed in the switch and operational in a matter of minutes, posing no real obstacle to research or development. (Compile time is less relevant for production networks, since they will require a “validation phase” that is orders of magnitude longer than the compile phase.)

For correct controller / switch operation in the context of TTPs, the two endpoints must be operating in the same TTP context. Synchronization mechanisms are defined to ensure that this is the case. For the details, see “OpenFlow Controller / Switch NDM Synchronization” (available on the ONF website).

A.3 Improving switch resource utilization

Programmable forwarding platforms, such as servers and NPUs and “highly programmable ASICs” (similar to the Metamorphosis abstract chip proposal) can offer flexible pipelines that are closely aligned to the OpenFlow pipeline model. Achieving on-the-fly run-time mapping of flow-mod messages onto such platforms may be more practical than doing so on fixed-function platforms. For these OpenFlow-like platforms, the TTP awareness is not essential for operation, but it still offers value by improving resource utilization, enabling better performance and scalability.

Highly flexible platforms achieve their flexibility by attempting to provide resources for every legal request. But real apps only issue some subset of what’s legal. The upshot is that for any specific real application, a highly flexible platform is overprovisioned in one way or another, which will either impact scalability or performance or both. Because TTPs describe the actual required switch functionality subset as well as providing for negotiation of key scaling parameters, such as flow table sizing, they enable flexible platforms to correctly allocate resources in advance of run time.

Consider, for example, the aforementioned Metamorphosis chip. As conceived, the device would have a large number of smallish on-chip TCAMs that can be reconfigured in myriad ways to deliver optimal performance and scalability. Flexible low cost devices will be very beneficial, but as discussed above, they will require optimal configuration to deliver full value. The traditional OpenFlow framework does not provide clear mechanisms to settle the optimal configuration issue, and the Metamorphosis paper does not address such questions. TTPs and the TTP Lifecycle can provide the answers, and enable flexible forwarding platforms to deliver high scalability and high performance by enabling optimizations to be made based on an analysis of the subset of functions required by a TTP.

A.4 Reducing the "Chicken-and-Egg" OpenFlow adoption challenge

OpenFlow seeks to deliver on the SDN promise of decoupling the control and data planes, and it is often said that OpenFlow is defining an “x86 instruction set” for networking. This analogy references the Intel®-based architecture’s role in the mainstream compute ecosystem, which allows buyers to pick best-of-breed applications, operating systems, and system platforms from a variety of independent vendors. The analogy highlights the idea that the networking market might resemble the more competitive compute ecosystem if networking platforms could converge on a single common underlying forwarding architecture.

Unfortunately, the networking industry is in a much different context than that of the compute industry when IBM established the *de facto* leading chip and operating system vendors. For various historical reasons, the networking space is dominated by vertically integrated software and hardware; networking platforms, even from specific vendors, are based on a wide variety of silicon architectures. Many vendors view diverse silicon as helpful for differentiation in the market place and see no advantage in pushing for a common silicon architecture. A set of compelling SDN "killer apps" that leverage OpenFlow-enabled hardware would alter the equation, enticing vendors to deliver platforms that can support those apps. But how do the app developers make progress in the absence of predictable platform capabilities?

In the early phase of open SDN adoption, most customers and vendors prefer taking advantage of hybrid switching platforms. Although hybrid platforms are often denigrated by SDN purists, they support an important transition process that allows both sides to explore SDN with incremental investment and modest risk. Hybrid platforms only require incremental investment because they are based on legacy silicon, but they come with the "diverse silicon" baggage. Unfortunately, as described above, traditional OpenFlow only works well on legacy fixed-function silicon when single flow tables are used. Complex OpenFlow applications can be delivered on flexible silicon, but adoption of such silicon is limited, in part because of cost and performance trade-offs.

At first glance, it would appear that open SDN advocates are facing a classic chicken-and-egg situation. App development is waiting for open, standard and (ideally) flexible platforms. Platform providers seem hesitant to invest in such platforms (and the current interoperability challenges of OpenFlow may be contributing to that hesitation). In the face of hesitant system vendors, silicon vendors are under-investing in potentially compelling chip architectures. How do we change this chicken-and-egg dynamic?

TTPs can be part of the answer. A key motivator for the development of TTPs was the recognition that the SDN hardware market faced a "bootstrap" challenge, since open SDN adoption requires big investment, but faces big uncertainty when complex forwarding meets diverse platform architecture. TTPs break the cycle by allowing market participants (large network operators, system vendors, testing houses and ONF workgroups) to define shared "switch models" (TTPs) that provide a pathway around the "complex forwarding / diverse platform" obstacle.

The advent of TTPs will enable rich OpenFlow functionality on legacy silicon. As useful TTPs become available and open SDN vendors recognize the opportunity to bridge the complexity/interoperability gap, the chicken-and-egg problem will be resolved. App developers will be able to develop and deliver interesting applications with being stuck waiting for new platforms. As a diversity of new SDN applications gains traction, system and silicon vendors will have more clarity about where the future opportunities will lie, and will seek to offer more flexible products that can support more TTPs. This market pressure will help drive the adoption of the flexible networking silicon which is currently stalled in the absence of a transition story.

A.5 In Summary

TTPs can help us move from our current restrictive ecosystem to a richer, more competitive ecosystem. They improve interoperability in the face of diverse platforms and diverse applications. They help buyers and other market participants understand what will interoperate with what. They allow for complex OpenFlow applications to run on legacy fixed-function platforms. They also allow for optimization of performance and scaling when applications run on flexible platforms.

Appendix B TTP Implementation Considerations

B.1 Introduction

When a TTP is used to define the datapath model for an OpenFlow Logical Switch there are some adjustments that should be made in the processing of OpenFlow protocol messages. These adjustments derive from the fact that the TTP defines the set of flow tables and the types of entries these tables may contain, and may include built-in table entries that must exist when the Logical Switch is initialized. The adjustments include:

- Changes in support for the TABLE_FEATURES message (B.2)
- Changes in handling FLOW_MOD messages (B.3)
- Changes in handling GROUP_MOD messages (B.4)

These adjustments are described in the following sections. Finally section B.5 presents some ideas about OpenFlow extensions that can enable better TTP support.

B.2 Support for TABLE_FEATURES message

The TABLE_FEATURES message response provides less detail than a TTP. Under normal circumstances, when a switch has an active TTP its controller should support that TTP and thus be aware of the TTP details. In that context, the value of TABLE_FEATURES query support is dubious. However, there may be situations where passive "read-only" controllers seek to gather information from the switch, and in such use cases, TABLE_FEATURES might be useful. The specific TABLE_FEATURES response can be developed by hand or by machine based on a TTP and its parameters. There has been some discussion about developing a tool for producing TTP-related content such as TABLE_FEATURES information. Check with the ONF to see if such a tool is available. The following modifications are made to OpenFlow TABLE_FEATURES message processing:

- If a TTP has been agreed with the controller, writing TABLE-FEATURES is not permitted and the OpenFlow Logical Switch (OFLS) should return an error of type OFPET_TABLE_FEATURES_FAILED and code OFPTFFC_EPERM.
- If a TTP has been configured in an OFLS, reading TABLE-FEATURES should be supported, but support is not required. If reading table features is not supported a TABLE_FEATURES query should result in an error response of type OFPET_BAD_REQUEST and code OFPBRC_BAD_MULTIPART or a new code defined specifically for this case (e.g., Table features defined by TTP).

B.3 Support for FLOW_MOD messages

Since a TTP defines valid flow table entry types and may define built-in flow table entries, the following adjustments to OFPT_FLOW_MOD message processing are made:

- If an OFPFC_ADD or other OFPFC_* command indicates a table number that is not specified in the TTP an error of type OFPET_FLOW_MOD_FAILED must be returned with code OFPFMFC_BAD_TABLE_ID or a new code defined specifically for this case (i.e., Bad table number for TTP).

- If an OFPFC_ADD command specifies a flow table entry that does not conform to one of the flow_mod entry types defined in the TTP for the indicated flow table, an error may be returned of type OFPET_FLOW_MOD_FAILED and code OFPFMFC_EPERM or a new code defined specifically for this case (i.e., Flow entry invalid for TTP specified table).
    - If the TTP defines a flow_mod type containing a match field with "const_mask" and "const_value" members, checking conformance includes verifying that the mask supplied for the field is the logical 'or' of the "const_mask" and "mask" and the match value includes the corresponding "const_value" bit values.
    - If a flow_mod type has a field with match type "prefix" the mask must be a set of contiguous bits beginning at the most significant bit and the priority must be the length of the prefix mask. If the priority is zero (not specified) then the priority can be adjusted to the correct value.
- An OFPFC_MODIFY should check all matching entries to ensure that the resulting entries are all legal before changing any of the entries. That is, if the last entry modified is illegal, the command must fail without any side effects, so the earlier entries should not have been changed. If an OFPFC_MODIFY or OFPFC_MODIFY_STRICT command would result in a non-conforming flow table entry, an error may be returned of type OFPET_FLOW_MOD_FAILED and code OFPFMFC_EPERM or a new code defined specifically for this case (i.e., Flow entry invalid for specified table). In this case it may be helpful to return an indication of which flow table entry caused the error, but this may require a protocol extension.
    - Note: Since OFPFC_MODIFY is “loose” it differs from other commands in that it cannot be easily checked for TTP compliance except by looking at the resulting entries. That makes it data dependent. Controller/app implementers should consequently use the loose version of MODIFY with some caution.
- If an OFPFC_ADD, OFPFC_MODIFY_STRICT or OFPFC_DELETE_STRICT command matches a built-in flow table entry defined in the TTP, an error must be returned of type OFPET_FLOW_MOD_FAILED and code OFPFMFC_EPERM or a new code defined specifically for this case (i.e., Cannot modify or delete a TTP built-in flow entry).
    - There is one exception to this rule: If an OFPFC_ADD or OFPFC_MODIFY_STRICT request matches a built-in entry and either contains no instructions or exactly matches the built-in, the cookie value is used to replace the cookie value in the built-in flow table entry. No other changes are made and a success response is returned.
- If an OFPFC_MODIFY or OFPFC_DELETE command matches a built-in flow table entry the request is ignored for that entry, but this does not affect the response returned. That is, the response returned is determined based only on the other (non-built-in) flow table entries matched.

B.4 Support for GROUP_MOD messages

Since a TTP defines valid group table entry types, the following adjustments to OFPT_GROUP_MOD message processing are made:

- Group entries should match one of the group entry types defined in the TTP, i.e., have matching group type and each bucket matching a bucket type defined for the group entry type. If an OFPGC_ADD command specifies a group table entry that does not conform to one of the group entry types defined in the TTP an error may be returned of type OFPET_GROUP_MOD_FAILED and code OFPGMFC_INVALID_GROUP or a new code defined specifically for this case (i.e., Group entry invalid for TTP).
- If an OFPGC_ADD command specifies a group table entry that conforms to one of the group entry types but one of the buckets does not conform to an associated bucket type defined for that group entry type in the

TTP, an error may be returned of type OFPET_GROUP_MOD_FAILED and code OFPGMFC_BAD_BUCKET or a new code defined specifically for this case (i.e., Group bucket invalid for TTP).

- If an OFPGC_MODIFY command specifies a bucket that does not conform to an associated bucket type defined for an associated group entry type in the TTP, an error may be returned of type OFPET_GROUP_MOD_FAILED and code OFPGMFC_BAD_BUCKET or a new code defined specifically for this case (i.e., Group bucket invalid for TTP).

B.5 Potentially useful OF-Switch extensions for TTPs

A TTP can constrain the types of flow_mod and group_mod messages that are valid for an OpenFlow Logical Switch . In addition, message validity can be verified in debug mode, for example as indicated by the adjustments to message processing cited above. Debug mode (strict checking for message validity according to the TTP) can require pattern matching to determine whether or not a given message matches a type specified in the TTP. This process could be made simpler (more efficient) for the Logical Switch by including the flow_mod type name, built-in flow_mod name, or group entry type name and bucket type name in the message, reducing the range of possibilities that must be considered in validating the message.

Adding error codes specific to TTP validation checks can provide more accurate indications of the source of some errors (e.g., as noted in sections above).

Appendix C Example TTP Description: L2-L3-ACLs

The following is an example TTP (from which the examples in the TTP description in section 3 have been drawn). This TTP describes OpenFlow controllable behavior, based on the OF-Switch 1.3 protocol, for a switch that provides both MAC forwarding (both unicast and multicast) and IP forwarding (unicast only). The example conforms to common standards and practices for MAC Bridging and IP forwarding so that the behavior described is well-known, allowing the reader to focus on how this behavior can be expressed in a TTP. Figure 2 is a diagram of the TTP showing the flow tables and group entry types and their relationships.

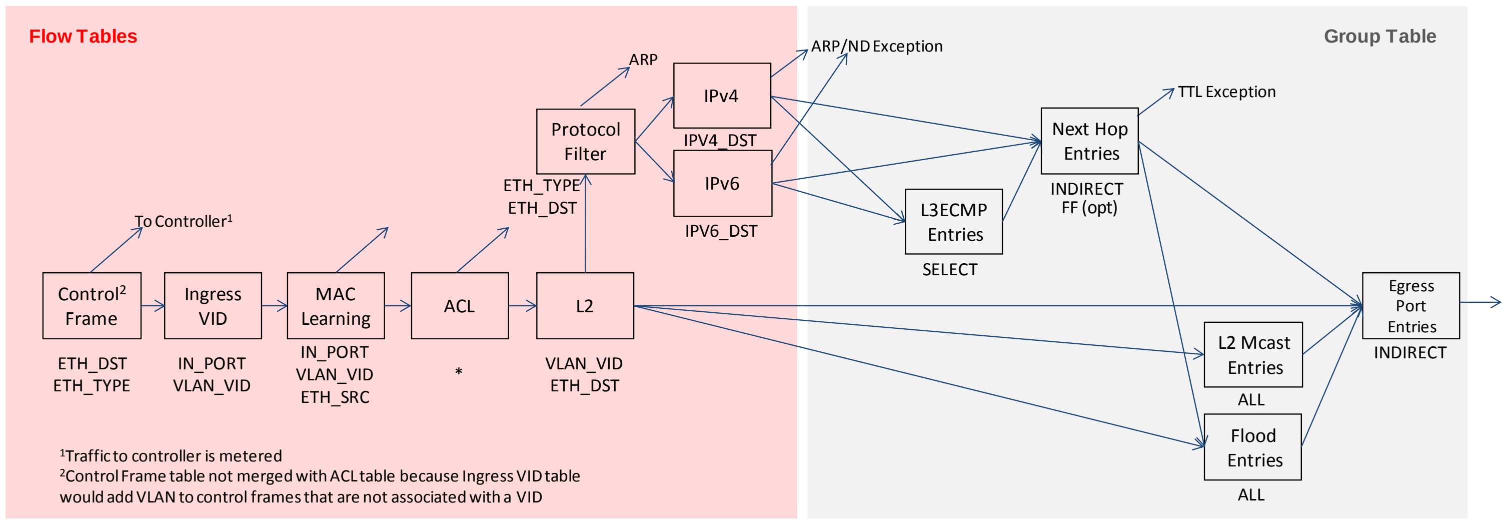

Figure 2: L2-L3-ACLs TTP Diagram

```
{
  "NDM_metadata": {
    "authority": "org.opennetworking.fawg",
    "type": "TTPv1",
    "name": "L2-L3-ACLs",
    "version": "1.0.0",
    "OF_protocol_version": "1.3.3",
    "doc": ["Example of a TTP supporting L2 (unicast, multicast, flooding), L3 (unicast only),",
            "and an ACL table."]
  },

  "security": {
    "doc": ["This TTP is not published for use by ONF. It is an example and for",
            "illustrative purposes only.",
            "If this TTP were published for use it would include",
            "guidance as to any security considerations in this doc member."]
  },

  "table_map": {
    "ControlFrame": 0,
    "IngressVLAN": 10,
    "MacLearning": 20,
    "ACL": 30,
    "L2": 40,
    "ProtoFilter": 50,
    "IPv4": 60,
    "IPv6": 80
  },
"identifiers": [
  {"var": "<port_vid>",
   "doc": "A VLAN ID to be assigned to untagged or priority tagged frames received on a port."},
  {"var": "<local_vid>",
   "range": "1..4094",
   "doc": "A VLAN ID valid on the wire at a port."},
  {"var": "<relay_vid>",
   "doc": "A VLAN ID valid internal to the switch."},
  {"var": "<VID>",
   "doc": "A VLAN ID"},
  {"var": "<Router_MAC_DA>",
   "doc": "A unicast MAC address used to reach the L3 flow tables"},
  {"var": "<Group_MAC>",
   "doc": "A group (multicast) MAC address."},
  {"var": "<Router_IP>",
   "doc": ["An IP address used to reach L3 control functions,",
           "e.g. a loopback address in the Router."]},    {"var": "<LocalSubnet>",
   "doc": "An IP subnet (address prefix) allocated to a directly attached L2 network or link."},
  {"var": "<port_no>",
   "doc": "A valid port number on the logical switch."},
  {"var": "<local_MAC>",
   "doc": "The unicast MAC address of a Router port on which a new L2 frame is transmitted."},
  {"var": "<dest_MAC>",
   "doc": "The destination MAC address for a new L2 frame."},
  {"var": "<subnet_VID>",
   "doc": "The VLAN ID of a locally attached L2 subnet on a Router."},
  {"var": "<<group_entry_types:name>>",
   "doc": ["An OpenFlow group identifier (integer) identifying a group table entry",
           "of the type indicated by the variable name"]}
],

"features": [
  {"feature": "ext187",
   "doc": "Flow entry notification Extension – notification of changes in flow entries"},
  {"feature": "ext235",
   "doc": "Group notifications Extension – notification of changes in group or meter entries"}
],

"meter_table": {
  "meter_types": [
    {"name": "ControllerMeterType",
     "bands": [{"type": "DROP", "rate": "1000..10000", "burst": "50..200"}]
    },
    {"name": "TrafficMeter",
     "bands": [{"type": "DSCP_REMARK", "rate": "10000..500000", "burst": "50..500"},
               {"type": "DROP", "rate": "10000..500000", "burst": "50..500"}]
    }
  ],
  "built_in_meters": [
    {"name": "ControllerMeter", "meter_id": 1,
      "type": "ControllerMeterType", "bands": [{"rate": 2000, "burst": 75}]},
    {"name": "AllArpMeter", "meter_id": 2,
      "type": "ControllerMeterType", "bands": [{"rate": 1000, "burst": 50}]}
  ]
},

"flow_tables": [
  {
    "name": "ControlFrame",
    "doc": ["Filters L2 control reserved destination addresses and",
            "may forward control packets to the controller.",
            "Directs all other packets to the Ingress VLAN table."],
    "flow_mod_types": [
      {
        "name": "Frame-To-Controller",
        "doc": ["This match/action pair allows for flow_mods that match on either",
                "ETH_TYPE or ETH_DST (or both) and send the packet to the",
                "controller, subject to metering."],
        "match_set": [
          {"field": "ETH_TYPE", "match_type": "all_or_exact"},
          {"field": "ETH_DST",  "match_type": "exact"}
        ],
        "instruction_set": {"exactly_one": [
         [
           {"instruction": "METER", "meter_name": "ControllerMeter",
              "doc": ["This meter may be used to limit the rate of PACKET_IN frames",
                     "sent to the controller"]},
           {"instruction": "APPLY_ACTIONS",
              "actions": [{"action": "OUTPUT", "port": "CONTROLLER"}]
           }],
         [
           {"instruction": "APPLY_ACTIONS",
              "actions": [{"action": "OUTPUT", "port": "CONTROLLER"}]
           }]
        ]}
      }
    ],
    "built_in_flow_mods": [
      {
        "name": "Control-Frame-Filter",
        "doc": "Mandatory filtering of control frames with C-VLAN Bridge reserved DA.",
        "priority": 1,
        "match_set": [{"field":"ETH_DST","mask":"0xfffffffffff0","value":"0x0180C2000000"}],
        "instruction_set": []
      },
      {
        "name": "Non-Control-Frame",
        "doc": "Mandatory miss flow_mod, sends packets to IngressVLAN table.",
        "priority": 0,
        "match_set": [],
        "instruction_set": [{"instruction": "GOTO_TABLE", "table": "IngressVLAN"}]
      }
    ]
  },
  {
    "name": "IngressVLAN",
    "doc": ["Ingress VID processing table, including:",
            " - accepting or blocking untagged and priority tagged frames",
            " - accepting or blocking VLAN tagged frames",
            " - ingress VID filtering control",
            " - (optional) ingress VID translation" ],
    "flow_mod_types": [
     {"all": [
      {
        "name": "Block-Untagged",
        "priority": "2..3",
        "doc": "Block untagged traffic on a port or all ports.",
        "match_set": [
          {"field": "IN_PORT", "match_type": "all_or_exact"},
          {"field": "VLAN_VID", "mask": "0x1fff", "value": "OFPVID_NONE"}
        ],
        "instruction_set": [
          {"instruction": "CLEAR_ACTIONS"}
        ]
      },
      {
        "name": "Allow-Untagged",
        "priority": 3,
        "doc": "Allow untagged traffic.",
        "match_set": [
          {"field": "IN_PORT", "match_type": "exact"},
          {"field": "VLAN_VID", "mask": "0x1fff", "value": "OFPVID_NONE"}
        ],
        "instruction_set": [
          {"instruction": "APPLY_ACTIONS",
            "actions": [
              {"action": "PUSH_VLAN"},
              {"action": "SET_FIELD", "field": "VLAN_VID", "value": "<port_vid>"}]},
          {"instruction": "GOTO_TABLE", "table": "MacLearning"}
        ]
      },
      {
        "name": "Block-Priority-Tagged",
        "priority": "5..7",
        "doc": "Block priority tagged traffic on a port or all ports.",
        "match_set": [
          {"field": "IN_PORT", "match_type": "all_or_exact"},
          {"field": "VLAN_VID", "mask": "0x1fff", "value": "OFPVID_PRESENT"}
        ],
        "instruction_set": [
          {"instruction": "CLEAR_ACTIONS"}
        ]
      },
      {
        "name": "Allow-Priority-Tagged",
        "priority": "6..7",
        "doc": "Allow priority tagged traffic on a port or all ports.",
        "match_set": [
          {"field": "IN_PORT", "match_type": "all_or_exact"},
          {"field": "VLAN_VID", "mask": "0x1fff", "value": "OFPVID_PRESENT"}
        ],
        "instruction_set": [
          {"instruction": "APPLY_ACTIONS",
            "actions": [
              {"action": "SET_FIELD", "field": "VLAN_VID", "value": "<port_vid>"}]},
          {"instruction": "GOTO_TABLE", "table": "MacLearning"}
        ]
      }

     ],
     "zero_or_more": [
      {
        "name": "Enable-Ingress-VID-Filter",
        "priority": "2..3",
        "doc": "Used to enable ingress VID filtering on all ports or a specific port.",
        "match_set": [
          {"field": "IN_PORT", "match_type": "all_or_exact"},
          {"field": "VLAN_VID", "mask": "0x1000", "value": "OFPVID_PRESENT"}
        ],
        "instruction_set": [
          {"instruction": "CLEAR_ACTIONS"}
        ]
      },
      {
        "name": "Disable-Ingress-VID-Filter",
        "priority": 3,
        "doc": "Used to disable ingress VID filtering on a specific port.",
        "match_set": [
          {"field": "IN_PORT", "match_type": "exact"},
          {"field": "VLAN_VID", "mask": "0x1000", "value": "OFPVID_PRESENT"}
        ],
        "instruction_set": [
          {"instruction": "GOTO_TABLE", "table": "MacLearning"}
        ]
      },
      {
        "name": "Ingress-VID-Allow",
        "priority": 4,
        "doc": "Used to allow a specific VID to ingress at a port or all ports.",
        "match_set": [
          {"field": "IN_PORT", "match_type": "all_or_exact"},
          {"field": "VLAN_VID", "const_mask": "0xf000", "const_value": "0x1000",
             "mask": "0x0fff", "value": "<local_vid>"}
        ],
        "instruction_set": [
          {"instruction": "GOTO_TABLE", "table": "MacLearning"}
        ]
      },
      {
        "opt_tag": "VID-X",
        "name": "Ingress-VID-Translate",
        "priority": "4..5",
        "doc": "Used to translate specific VIDs at ingress at a port or all ports.",
        "match_set": [
          {"field": "IN_PORT", "match_type": "all_or_exact"},
          {"field": "VLAN_VID", "const_mask": "0xf000", "const_value": "0x1000",
           "mask": "0x0fff", "value": "<local_vid>"}
        ],
        "instruction_set": [
          {"instruction": "APPLY_ACTIONS",
            "actions": [
              {"action": "SET_FIELD", "field": "VLAN_VID", "value": "<relay_vid>"}]},
          {"instruction": "GOTO_TABLE", "table": "MacLearning"}
        ]
      }
     ]}
    ],
    "built_in_flow_mods": [
      {
        "name": "Default-Allow-Untagged",
        "priority": 1,
        "doc": "Default to allow untagged traffic on all ports, default port VID is 1.",
        "match_set": [
          {"field": "VLAN_VID", "mask": "0x1fff", "value": "OFPVID_NONE"}
        ],
        "instruction_set": [
          {"instruction": "APPLY_ACTIONS",
            "actions": [
              {"action": "PUSH_VLAN"},
              {"action": "SET_FIELD", "field": "VLAN_VID", "value": 1}]},
          {"instruction": "GOTO_TABLE", "table": "MacLearning"}
        ]
      },
      {
        "name": "Default-Allow-Priority-Tagged",
        "priority": 4,
        "doc": ["Default flow_mod to allow priority tagged traffic on all ports,",
                "default port VID is 1."],
        "match_set": [
          {"field": "VLAN_VID", "mask": "0x1fff", "value": "OFPVID_PRESENT"}
        ],
        "instruction_set": [
          {"instruction": "APPLY_ACTIONS",
            "actions": [
              {"action": "SET_FIELD", "field": "VLAN_VID", "value": 1}]},
          {"instruction": "GOTO_TABLE", "table": "MacLearning"}
        ]
      },
      {
        "name": "Default-Disable-Ingress-VID-Filter",
        "priority": 1,
        "doc": "Default to disable ingress VID filtering on all ports.",
        "match_set": [
          {"field": "VLAN_VID", "mask": "0x1000", "value": "OFPVID_PRESENT"}
        ],
        "instruction_set": [
          {"instruction": "GOTO_TABLE", "table": "MacLearning"}
        ]
      }
    ],
    "table_subsets": [
      {"name": "IV-pass",
       "subset": [ "Allow-Untagged", "Allow-Priority-Tagged", "Disable-Ingress-VID-Filter",
                   "Ingress-VID-Allow", "Ingress-VID-Filter", "Ingress-VID-Translate",
                   "Default-Allow-Untagged", "Default-Allow-Priority-Tagged",
                   "Default-Disable-Ingress-VID-Filter" ]
      },
      {"name": "IV-drop",
       "subset": [ "Block-Untagged", "Block-Priority-Tagged", "Enable-Ingress-VID-Filter",
                   "MISS" ]
      }
    ]
  },
  {
    "name": "MacLearning",
    "doc": ["By default sends packets whose Source MAC address is ",
            "received on a new IN_PORT to controller for learning.",
            "The controller is expected to install flow_mods for learned",
            "addresses, and remove stale entries when required.",
            "The controller may also disable MAC learning for a VLAN ",
            "by installing a flow_mod to simply go to the next table."],
    "flow_mod_types": [
      {
        "name": "Known-MAC",
        "priority": 2,
        "doc": "Type used to create an entry for a learned MAC",
        "match_set": [
          {"field": "IN_PORT"},
          {"field": "VLAN_VID"},
          {"field": "ETH_SRC"}
        ],
        "instruction_set": [
          {"instruction": "GOTO_TABLE", "table": "ACL"}
        ]
      },
      {
        "name": "Disable-MAC-Learning",
        "priority": 2,
        "doc": "Type used to disable MAC learning on a VLAN",
        "match_set": [{"field": "VLAN_VID"}],
        "instruction_set": [
          {"instruction": "GOTO_TABLE", "table": "ACL"}
        ]
      },
      {
        "name": "MAC-Miss-limit",
        "doc": "Send unknown MACs to the controller, subject to metering.",
        "priority": 1,
        "match_set": [],
        "instruction_set": [
          {"instruction": "METER", "meter_name": "ControllerMeter",
           "doc": ["This meter may be used to limit the rate of PACKET_IN frames",
                   "sent to the controller"]},
          {"instruction": "APPLY_ACTIONS",
            "actions": [
              {"action": "OUTPUT", "port": "CONTROLLER"}
            ]
          },
          {"instruction": "GOTO_TABLE", "table": "ACL"}
        ]
      }
    ],
    "built_in_flow_mods": [
      {
        "name": "MAC-Miss",
        "doc": "Send unknown MACs to the controller.",
        "priority": 0,
        "match_set": [],
        "instruction_set": [
          {"instruction": "APPLY_ACTIONS",
            "actions": [
              {"action": "OUTPUT", "port": "CONTROLLER"}
            ]
          },
          {"instruction": "GOTO_TABLE", "table": "ACL"}
        ]
      }
    ]
  },
  {
    "name": "ACL",
    "doc": "Simple 5-tuple firewalling ACL table.",
    "flow_mod_types": [
      {
        "name": "IP5-Tuple-Block",
        "doc": ["This type allows matching on an IP 5-tuple and",
                "dropping packets."],
        "match_set": [{
          "exactly_one": [
            [
              {"field": "ETH_TYPE", "value": 2048},
              {"field": "IP_PROTO", "value": 6},
              {"field": "IPV4_SRC", "match_type": "mask"},
              {"field": "IPV4_DST", "match_type": "mask"},
              {"field": "TCP_SRC",  "match_type": "mask"},
              {"field": "TCP_DST",  "match_type": "mask"}
            ],
            [
              {"field": "ETH_TYPE", "value": 2048},
              {"field": "IP_PROTO", "value": 17},
              {"field": "IPV4_SRC", "match_type": "mask"},
              {"field": "IPV4_DST", "match_type": "mask"},
              {"field": "UDP_SRC",  "match_type": "mask"},
              {"field": "UDP_DST",  "match_type": "mask"}
            ],
            [
              {"field": "ETH_TYPE", "value": 34525},
              {"field": "IP_PROTO", "value": 6},
              {"field": "IPV6_SRC", "match_type": "mask"},
              {"field": "IPV6_DST", "match_type": "mask"},
              {"field": "TCP_SRC",  "match_type": "mask"},
              {"field": "TCP_DST",  "match_type": "mask"}
            ],
            [
              {"field": "ETH_TYPE", "value": 34525},
              {"field": "IP_PROTO", "value": 17},
              {"field": "IPV6_SRC", "match_type": "mask"},
              {"field": "IPV6_DST", "match_type": "mask"},
              {"field": "UDP_SRC",  "match_type": "mask"},
              {"field": "UDP_DST",  "match_type": "mask"}
            ]
          ]
        }],
        "instruction_set": [
          {"instruction": "CLEAR_ACTIONS"}
        ]
      },
      {
        "name": "IP-5Tuple-Intercept",
        "doc": ["This type allows matching on an IP 5-tuple and",
                "forwarding to the controller."],
        "match_set": [{
          "exactly_one": [
            [
              {"field": "ETH_TYPE", "value": 2048},
              {"field": "IP_PROTO", "value": 6},
              {"field": "IPV4_SRC", "match_type": "mask"},
              {"field": "IPV4_DST", "match_type": "mask"},
              {"field": "TCP_SRC",  "match_type": "mask"},
              {"field": "TCP_DST",  "match_type": "mask"}
            ],
            [
              {"field": "ETH_TYPE", "value": 2048},
              {"field": "IP_PROTO", "value": 17},
              {"field": "IPV4_SRC", "match_type": "mask"},
              {"field": "IPV4_DST", "match_type": "mask"},
              {"field": "UDP_SRC",  "match_type": "mask"},
              {"field": "UDP_DST",  "match_type": "mask"}
            ],
            [
              {"field": "ETH_TYPE", "value": 34525},
              {"field": "IP_PROTO", "value": 6},
              {"field": "IPV6_SRC", "match_type": "mask"},
              {"field": "IPV6_DST", "match_type": "mask"},
              {"field": "TCP_SRC",  "match_type": "mask"},
              {"field": "TCP_DST",  "match_type": "mask"}
            ],
            [
              {"field": "ETH_TYPE", "value": 34525},
              {"field": "IP_PROTO", "value": 17},
              {"field": "IPV6_SRC", "match_type": "mask"},
              {"field": "IPV6_DST", "match_type": "mask"},
              {"field": "UDP_SRC",  "match_type": "mask"},
              {"field": "UDP_DST",  "match_type": "mask"}
            ]
          ]
        }],
        "instruction_set": [
          {"instruction": "METER", "meter_name": "ControllerMeter",
              "doc": ["This meter may be used to limit the rate of PACKET_IN frames",
                      "sent to the controller"]},
          {"instruction": "APPLY_ACTIONS",
            "actions": [
              {"action": "OUTPUT", "port": "CONTROLLER"}]
          }
        ]
      },
      {
        "name": "IP-5Tuple-Allow",
        "doc": ["This type allows matching on an IP 5-tuple and",
                "sending on to the L2 table, overriding a lower",
                "priority block or intercept."],
        "match_set": [{
          "exactly_one": [
            [
              {"field": "ETH_TYPE", "value": 2048},
              {"field": "IP_PROTO", "value": 6},
              {"field": "IPV4_SRC", "match_type": "mask"},
              {"field": "IPV4_DST", "match_type": "mask"},
              {"field": "TCP_SRC",  "match_type": "mask"},
              {"field": "TCP_DST",  "match_type": "mask"}
            ],
            [
              {"field": "ETH_TYPE", "value": 2048},
              {"field": "IP_PROTO", "value": 17},
              {"field": "IPV4_SRC", "match_type": "mask"},
              {"field": "IPV4_DST", "match_type": "mask"},
              {"field": "UDP_SRC",  "match_type": "mask"},
              {"field": "UDP_DST",  "match_type": "mask"}
            ],
            [
              {"field": "ETH_TYPE", "value": 34525},
              {"field": "IP_PROTO", "value": 6},
              {"field": "IPV6_SRC", "match_type": "mask"},
              {"field": "IPV6_DST", "match_type": "mask"},
              {"field": "TCP_SRC",  "match_type": "mask"},
              {"field": "TCP_DST",  "match_type": "mask"}
            ],
            [
              {"field": "ETH_TYPE", "value": 34525},
              {"field": "IP_PROTO", "value": 17},
              {"field": "IPV6_SRC", "match_type": "mask"},
              {"field": "IPV6_DST", "match_type": "mask"},
              {"field": "UDP_SRC",  "match_type": "mask"},
              {"field": "UDP_DST",  "match_type": "mask"}
            ]
          ]
        }],
        "instruction_set": [
            {"instruction": "GOTO_TABLE", "table": "L2"}
        ]
      }
    ],
    "built_in_flow_mods": [
      {
        "name": "ACL-skip",
        "doc": "Mandatory miss flow mod, sends packets to L2 table.",
        "priority": 0,
        "match_set": [],
        "instruction_set": [{"instruction": "GOTO_TABLE", "table": "L2"}]
      }
    ]
  },
  {
    "name": "L2",
    "doc": ["MAC forwarding table"],
    "flow_mod_types": [
      {
        "name": "VID-flood",
        "priority": 1,
        "doc": "Flood frames with unknown DA.",
        "match_set": [
          {"field": "VLAN_VID", "const_mask": "0x1000", "const_value": "0x1000",
             "mask": "0x0fff", "value": "<VID>"}
        ],
        "instruction_set": [
          {"instruction": "APPLY_ACTIONS",
            "actions": [
              {"action": "GROUP", "group_id": "<VIDflood>"}
            ]
          },
          {"zero_or_one": {"instruction": "GOTO_TABLE", "table": "ProtoFilter",
           "doc": "Include this instruction of the VID is registered on the Router port."}}
        ]
      },
      {
        "name": "L2-Unicast",
        "priority": 2,
        "doc": "Unicast forwarding entry.",
        "match_set": [
          {"field": "VLAN_VID", "const_mask": "0x1000", "const_value": "0x1000",
             "mask": "0x0fff", "value": "<VID>"},
          {"field": "ETH_DST"}
        ],
        "instruction_set": [
          {"instruction": "APPLY_ACTIONS",
            "actions": [
              {"action": "GROUP", "group_id": "<EgressPort>"}
            ]
          }
        ]
      },
      {
        "name": "L2-Router-MAC",
        "priority": 2,
        "doc": "Router MAC address, so send toward IP flow tables.",
        "match_set": [
          {"field": "VLAN_VID", "const_mask": "0x1000", "const_value": "0x1000",
             "mask": "0x0fff", "value": "<VID>"},
          {"field": "ETH_DST", "value": "<Router_MAC_DA>"}
        ],
        "instruction_set": [
          {"instruction": "GOTO_TABLE", "table": "ProtoFilter"}
        ]
      },
      {
        "name": "L2-Multicast",
        "priority": 2,
        "doc": "L2 Multicast forwarding entry.",
        "match_set": [
          {"field": "VLAN_VID", "const_mask": "0x1000", "const_value": "0x1000",
             "mask": "0x0fff", "value": "<VID>"},
          {"field": "ETH_DST", "value": "<Group_MAC>"}
        ],
        "instruction_set": [
          {"instruction": "APPLY_ACTIONS",
            "actions": [
              {"action": "GROUP", "group_id": "<L2Mcast>"}
            ]
          }
        ]
      }
    ],
    "built_in_flow_mods": [
      {
        "name": "L2-Drop",
        "priority": 0,
        "doc": ["Discard frames with no VID registration,",
                "i.e., VID without a <VIDflood> group and",
                "corresponding VIDflood flow table entry."],
        "match_set": [],
        "instruction_set": [
          {"instruction": "CLEAR_ACTIONS"}
        ]
      }
    ],
    "table_subsets": [
      {"name": "L2-Forward",
       "subsets": ["VIDflood", "L2Unicast", "L2Multicast"]
      }
    ]
  },
  {
    "name": "ProtoFilter",
    "doc": ["Selects IP version flow table and forwards ARPs to controller."],
    "built_in_flow_mods": [
      {
        "name": "IPv4",
        "priority": 1,
        "doc": "Direct IPv4 packets to IPv4 flow table.",
        "match_set": [
          {"field": "ETH_TYPE", "value": 2048},
          {"field": "ETH_DST", "value": "<Router_MAC_DA>"}
        ],
        "instruction_set": [
          {"zero_or_one": {"instruction": "METER", "type": "TrafficMeter"}},
          {"instruction": "GOTO_TABLE", "table": "IPv4"}
        ]
      },
      {
        "opt_tag": "IPv6",
        "name": "IPv6",
        "priority": 1,
        "doc": "Direct IPv6 packets to IPv6 flow table.",
        "match_set": [
          {"field": "ETH_TYPE", "value": 34525},
          {"field": "ETH_DST", "value": "<Router_MAC_DA>"}
        ],
        "instruction_set": [
          {"zero_or_one": {"instruction": "METER", "type": "TrafficMeter"}},
          {"instruction": "GOTO_TABLE", "table": "IPv6"}
        ]
      },
      {
        "name": "Router-ARP",
        "priority": 2,
        "doc": "Direct targeted ARP packets to controller.",
        "match_set": [
          {"field": "ETH_TYPE", "value": 2054},
          {"field": "ARP_TPA", "value": "<Router_IP>"}
        ],
        "instruction_set": [
          {"instruction": "APPLY_ACTIONS",
            "actions": [
              {"action": "OUTPUT", "port": "CONTROLLER"}]
          }
        ]
      },
      {
        "name": "All-ARP",
        "priority": 1,
        "doc": "Direct ARP packets to controller.",
        "match_set": [
          {"field": "ETH_TYPE", "value": 2054}
        ],
        "instruction_set": [
          {"instruction": "METER", "meter_name": "AllArpMeter",
              "doc": ["This meter may be used to limit the rate of PACKET_IN frames",
                      "sent to the controller.  A separate controller meter is used",
                      "here, with a lower rate than main controller meter, to limit ARPs",
                      "before limiting other packets to the controller."]},
          {"instruction": "APPLY_ACTIONS",
            "actions": [
              {"action": "OUTPUT", "port": "CONTROLLER"}]
          }
        ]
      }
    ]
  },
  {
    "name": "IPv4",
    "doc": ["IPv4 unicast forwarding table.  To achieve LPM the flow_mod",
            "priority must be the length of the prefix mask."],
    "flow_mod_types": [
      {
        "name": "v4-Unicast",
        "doc": ["LPM forwarding entry. Valid only if the priority value",
                "matches the length of the prefix mask."],
        "match_set": [
          {"field": "IPV4_DST", "match_type": "prefix"}
        ],
        "instruction_set": [
          {"zero_or_one": {"instruction": "METER", "type": "TrafficMeter"}},
          {"instruction": "APPLY_ACTIONS",
            "actions": [
              {"action": "GROUP", "group_id": "<NextHop>"}]
          }
        ]
      },
      {
        "name": "v4-Unicast-ECMP",
        "doc": ["LPM forwarding entry with ECMP. Valid only if the priority value",
                "matches the length of the prefix mask."],
        "match_set": [
          {"field": "IPV4_DST", "match_type": "prefix"}
        ],
        "instruction_set": [
          {"zero_or_one": {"instruction": "METER", "type": "TrafficMeter"}},
          {"instruction": "APPLY_ACTIONS",
            "actions": [
              {"action": "GROUP", "group_id": "<L3ECMP>"}]
          }
        ]
      },
      {
        "name": "Local-ARP",
        "doc": ["Local subnet address needing ARP. Valid only if the priority value",
                "matches the length of the prefix mask."],
        "match_set": [
          {"field": "IPV4_DST", "value": "<LocalSubnet>", "match_type": "prefix"}
        ],
        "instruction_set": [
          {"instruction": "APPLY_ACTIONS",
            "actions": [
              {"action": "OUTPUT", "port": "CONTROLLER"}]
          }
        ]
      }
    ]
  },
  {
    "opt_tag": "IPv6",
    "name": "IPv6",
    "doc": ["IPv6 unicast forwarding table.  To achieve LPM the flow_mod",
                    "priority must be the length of the prefix mask."],
    "flow_mod_types": [
      {
        "name": "v6-Unicast",
        "doc": ["LPM forwarding entry. Valid only if the priority value",
                "matches the length of the prefix mask."],
        "match_set": [
          {"field": "IPV6_DST", "match_type": "prefix"}
        ],
        "instruction_set": [
          {"zero_or_one": {"instruction": "METER", "type": "TrafficMeter"}},
          {"instruction": "APPLY_ACTIONS",
            "actions": [
              {"action": "GROUP", "group_id": "<NextHop>"}]
          }
        ]
      },
      {
        "name": "v6-Unicast-ECMP",
        "doc": ["LPM forwarding entry with ECMP. Valid only if the priority value",
                "matches the length of the prefix mask."],
        "match_set": [
          {"field": "IPV6_DST", "match_type": "prefix"}
        ],
        "instruction_set": [
          {"zero_or_one": {"instruction": "METER", "type": "TrafficMeter"}},
          {"instruction": "APPLY_ACTIONS",
            "actions": [
              {"action": "GROUP", "group_id": "<L3ECMP>"}]
          }
        ]
      },
      {
        "name": "Local-ND",
        "doc": ["Local subnet address needing Neighbor Discovery. Valid only",
                "if the priority value matches the length of the prefix mask."],
        "match_set": [
          {"field": "IPV6_DST", "value": "<LocalSubnet>", "match_type": "prefix"}
        ],
        "instruction_set": [
          {"instruction": "APPLY_ACTIONS",
            "actions": [
              {"action": "OUTPUT", "port": "CONTROLLER"}]
          }          ]
      }
    ]
  }
],

"group_entry_types": [
  {
    "name": "EgressPort",
    "doc": ["Output to a port, removing VLAN tag if needed.",
            "Entry per port, plus entry per untagged VID per port."],
    "group_type": "INDIRECT",
    "bucket_types": [
      {"name": "OutputTagged",
       "action_set": [{"action": "OUTPUT", "port": "<port_no>"}]
      },
      {"name": "OutputUntagged",
       "action_set": [{"action": "POP_VLAN"},
                       {"action": "OUTPUT", "port": "<port_no>" }]
      },
      {"opt_tag": "VID-X",
       "name": "OutputVIDTranslate",
       "action_set": [{"action": "SET_FIELD", "field": "VLAN_VID", "value": "<local_vid>"},
                       {"action": "OUTPUT", "port": "<port_no>" }]
      }
    ]
  },
  {
    "name": "VIDflood",
    "doc": ["Output to all ports registered for a VID (except IN_PORT).",
            "Entry per VID."],
    "group_type": "ALL",
    "bucket_types": [
      {"name": "VIDport",
       "action_set": [{"action": "GROUP", "group_id": "<EgressPort>"}]
      }
    ]
  },
  {
    "name": "L2Mcast",
    "doc": ["Output to all ports in a multicast tree (except IN_PORT).",
            "Entry per L2 group address."],
    "group_type": "ALL",
    "bucket_types": [
      {"name": "MCASTport",
       "action_set": [{"action": "GROUP", "group_id": "<EgressPort>"}]
      }
    ]
  },
  {
    "name": "NextHop",
    "doc": ["Decrement IP TTL and add L2 header for next hop.",
            "Entry per next hop IP address."],
    "group_type": "INDIRECT",
    "bucket_types": [
      {"name": "KnownMAC",
       "action_set": [
         {"action": "DEC_NW_TTL"},
         {"action": "SET_FIELD", "type": "ETH_SRC", "value": "<local_MAC>"},
         {"action": "SET_FIELD", "type": "ETH_DST", "value": "<dest_MAC>"},
         {"action": "SET_FIELD", "type": "VLAN_VID", "value": "<subnet_VID>"},
         {"action": "GROUP", "group_id": "<EgressPort>"}]
      },
      {"name": "UnknownMAC",
       "action_set": [
         {"action": "DEC_NW_TTL"},
         {"action": "SET_FIELD", "type": "ETH_SRC", "value": "<local_MAC>"},
         {"action": "SET_FIELD", "type": "ETH_DST", "value": "<dest_MAC>"},
         {"action": "SET_FIELD", "type": "VLAN_VID", "value": "<subnet_VID>"},
         {"action": "GROUP", "group_id": "<Flood>"}]
      }
    ]
  },
  {
    "name": "L3ECMP",
    "doc": ["Output to one port in an ECMP set.",
            "Entry per destination border node."],
    "group_type": "SELECT",
    "bucket_types": [
      {"name": "nextHopOption",
       "action_set": [{"action": "GROUP", "group_id": "<NextHop>"}]
      }
    ]
  },
  {"zero_or_more": {
    "name": "NextHopFF",
    "doc": ["Decrement IP TTL and add L2 header for next hop.",
            "Entry per next hop IP address.",
            "Fast Failover allows multiple buckets, picks first operational."],
    "group_type": "FF",
    "bucket_types": [
      {"name": "KnownMAC",
       "action_set": [
         {"action": "DEC_NW_TTL"},
          {"action": "SET_FIELD", "type": "ETH_SRC", "value": "<local_MAC>"},
          {"action": "SET_FIELD", "type": "ETH_DST", "value": "<dest_MAC>"},
          {"action": "SET_FIELD", "type": "VLAN_VID", "value": "<subnet_VID>"},
          {"action": "GROUP", "group_id": "<EgressPort>"}]
       },
       {"name": "UnknownMAC",
        "action_set": [
          {"action": "DEC_NW_TTL"},
          {"action": "SET_FIELD", "type": "ETH_SRC", "value": "<local_MAC>"},
          {"action": "SET_FIELD", "type": "ETH_DST", "value": "<dest_MAC>"},
          {"action": "SET_FIELD", "type": "VLAN_VID", "value": "<subnet_VID>"},
          {"action": "GROUP", "group_id": "<Flood>"}]
       }
     ]
   }}
  ],
  "parameters": [
    {"name": "ACL::TableSize", "type": "integer"},
    {"name": "L2::TableSize", "type": "integer"},
    {"name": "IPv4::TableSize", "type": "integer"},
    {"name": "IPv6::TableSize", "type": "integer"},
    {"name": "Meter::TableSize", "type": "integer",
       "doc": "Number of meters that can be configured in the switch."},
    {"name": "Meter::Accuracy", "type": "integer",
       "doc": "Accuracy of meters on the switch."},
    {"name": "OptFunc", "type": "array of opt_tag values",
       "doc": "Support for optional functions can be negotiated using the OptFunc parameter."}
  ],
  "flow_paths": [
    {"doc": ["This object contains just a few examples of flow paths, it is not",
            "a comprehensive list of the flow paths required for this TTP.  It is",
            "intended that the flow paths array could include either a list of",
            "required flow paths or a list of specific flow paths that are not",
            "required (whichever is more concise or more useful."],
     "name": "L2-2",
     "path": ["Non-Control-Frame","IV-pass","Known-MAC","ACLskip","L2-Unicast",
              "EgressPort"]
    },
    {"name": "L2-3",
     "path": ["Non-Control-Frame","IV-pass","Known-MAC","ACLskip","L2-Multicast",
              "L2Mcast", [ "EgressPort" ]]
    },
    {"name": "L2-4",
     "path": ["Non-Control-Frame","IV-pass","Known-MAC","ACL-skip","VID-flood",
              "VIDflood", [ "EgressPort" ] ]
    },
    {"name": "L2-5",
     "path": ["Non-Control-Frame","IV-pass","Known-MAC","ACLskip","L2-Drop"]
    },
    {"name": "v4-1",
     "path": ["Non-Control-Frame","IV-pass","Known-MAC","ACLskip","L2-Router-MAC",
              "IPv4","v4-Unicast",
              "NextHop", "EgressPort"]
    },
    {"name": "v4-2",
     "path": ["Non-Control-Frame","IV-pass","Known-MAC","ACLskip","L2-Router-MAC",
              "IPv4","v4-Unicast-ECMP",
              "L3ECMP", "NextHop", "EgressPort"]
    }
  ]
}
```

Appendix D Credits

Contributors (in alphabetical order):

Abhijit Kumbhare, Albert Fishman, Alon Harel, Ben Mack-Crane, Ben Pfaff, Colin Dixon, Curt Beckmann, Dacheng Zhang, Dave Hood, David Meyer, Haoyu Song, Jean Tourrilhes, Joel Halpern, Johann Tonsing, John E (Ed) Rathke, Joseph Tardo, Ken Yi, Linda Dunbar, Rakesh Saha, Robert Raszuk, Shaun Crampton, Srini Addepalli, Srinivas Veereshwara, Tsahi Daniel, Vytautas Valancius, Wesley M Felter, Xuewei Wang, Yatish Kumar, Zoltán Lajos Kis

© 2014 Open Networking Foundation