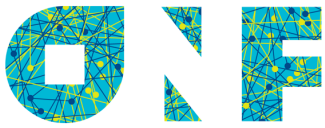

OPEN NETWORKING
FOUNDATION

ONF SDN Evolution

Version 1.0

ONF TR-535

2016-09-08

ONF Document Type: Technical Recommendation
ONF Document Name: ONF SDN Evolution

Disclaimer

Open Networking Foundation
2275 E. Bayshore Road, Suite 103, Palo Alto, CA 94303
www.opennetworking.org

Table of Contents

1 Executive Summary

This document considers the evolution of Software Defined Networking (which includes, but is not limited to, OpenFlow-based networking). The intent is to consider the technologies and standardized interfaces required to support new use cases, and the associated impact on the ecosystem, in order to facilitate ecosystem growth.

The evolution process aims to retain the tenets of SDN, for example logically centralized control and support for the match/action model standardized by OpenFlow. It also aims to support new use cases, e.g. new services and applications like stateful firewalling, as well as new media, domains, and functionality like optical networking and OAM. It furthermore intends to introduce additional flexibility, for example by making provision for Protocol Independent Forwarding and flexibly programmed datapaths (leveraging languages like P4), and supporting mechanisms that enable a broader range of software interaction with datapaths.

The aim is specifically to support backward compatibility with, and migration from, systems based on OpenFlow 1.x. The next generation run-time interface needs to be able to cope with flexibly defined dataplane protocols (e.g. protocols defined by a program) and associated behaviors while retaining the ability to interact with current SDN devices which only support the protocols and behaviors defined by the current OpenFlow 1.x standards.

The OpenFlow specification currently defines the expected behavior of a switch (i.e. an abstract model for it), the dataplane protocols that can be matched and acted on, and an interface to the switch (e.g. to enable a controller to populate tables and retrieve statistics). Combining all of this in a single standard impedes adding support for new dataplane protocols and new use cases because the specification needs to be modified to introduce new protocols / behaviors. This initiative proposes to introduce the concept of a modular specification, whereby the specifics of dataplane protocols and behaviors are moved into extension modules that can be created and evolved independently from the core specification. It furthermore enables support for programmed datapaths where the program itself defines the dataplane protocols and behaviors. The intent is for these two mechanisms to coexist by enabling an abstract model describing how to interface to new protocols or behaviors to be constructed, irrespective of whether the implementation is opaque (with only a control interface being exposed) or whether it is visible as a datapath program (potentially written in a language like P4).

Today OpenFlow makes provision for network traffic processing behavior to be either implemented within switches themselves or implemented in software running on the SDN controller (by leveraging OpenFlow's packet in and packet out messages). The aim is to introduce additional open and standardized interfaces to support the interaction of a wider range of software modules running in additional locations. Software in a switch could perform control and monitoring functions delegated to it by the controller, for example aggregating statistics and updating flow entries to block attacks if the statistics indicate that a denial of service attack is in progress. Software could also participate in the dataplane, performing exception path or even fastpath processing. The software could be located in the enclosure of a physical switch, running on the switch chip itself or on a general purpose CPU attached to the switch chip. The software could also form part of a virtual network function (VNF) running on a server, interacting with a virtual switch running on the server's CPU or on a smart NIC in the server. The interfaces between the software and the datapath need to support the performance required by the use case,

for example, high packet rates and flow update rates might be required, therefore callable interfaces must be considered to avoid the overhead associated with a protocol based interface like the existing OpenFlow. The intent is to create open and standardized interfaces to enable a broader range of software interaction with datapaths and to enable the software to be supplied by new ecosystem players.

One needs to also consider the lifecycle implications of these new approaches. A flexibly programmed datapath for example introduces new concepts like a program, a programmer, a system compiling the program, a mechanism to distribute the program, as well as implications in time and space, for example co-existence of different programs and program versions in a network, and concerns associated with deploying, upgrading and migrating programs over time. The broader range of software interaction with the datapath introduces similar concerns. In essence, the lifecycle of each existing and newly introduced technology element (e.g. program, controller, or switch) needs to be considered, together with the interactions of these elements (e.g. deploying a new program may necessitate upgrading the hardware of the switch hosting the program).

This document is intended to act as a catalyst for discussing the potential evolution of ONF SDN. As it presents a snapshot of an ongoing discussion, it is necessarily incomplete. Some items requiring further study are explicitly called out as such. All points made in the document are presented for discussion. Contributions and other feedback are appreciated.

2 Overview

2.1 Goals, Objectives, and Scope

This document is intended to facilitate the ongoing discussion related to the evolution of Software Defined Networking and related ecosystems in general, and those ecosystems that are emerging around OpenFlow and Protocol Independent Forwarding / flexibly programmed datapaths in particular, by capturing and structuring the results of the discussion so far, and by presenting issues and other items that need to be considered in future.

Especially noteworthy concerns or aspects requiring further discussion are highlighted using a paragraph border.

The overall goal of the evolution process is to improve support for specific emerging technologies and use cases such as protocol independence and Network Functions Virtualisation, but also to facilitate the growth of the ecosystem comprising support for a wide variety of technologies and use cases, with a number of players (vendors, operators etc.).

The document first outlines the motivations underlying the new technologies, use cases, and approaches. It proceeds to consider requirements associated with flexible (i.e. programmed) datapaths, for example, the construction and compilation / initial configuration of datapath programs, and their continued operation. The focus then shifts to the various datapath variants, considering various system and silicon types, as well as profiles and models to document their characteristics. Introducing the time dimension and interrelationships, the dynamics of the

broader ecosystem are then considered, with aspects like lifecycles and deployment considerations being elucidated. Shifting to the meta level, the processes associated with resolving these issues and with standards creation are covered. Finally, the proposals are measured against a specific use case, with conclusions being drawn.

As the domain specific languages and other mechanisms to permit flexibly programming the datapath itself are covered in other forums (for example p4.org and OpenSourceSDN.org's Protocol Independent Forwarding activity), this document focuses on the wider considerations associated with these mechanisms as well as the run-time interface to these datapaths, not the intricacies of the datapaths themselves.

2.2 Motivation for a Next Generation of OpenFlow: Flexibility and Ecosystem Growth

2.2.1 Software Defined Networking and Logically Centralized Control

The principle of logically centralized control of the network underpins many variants of Software Defined Networking. This principle is generally accepted and is expected to remain valid for the foreseeable future. It may be augmented and refined, for example by introducing virtualization, partitioning, selective delegation, federation of responsibility, as well as hybrids of SDN and traditional networking.

2.2.2 OpenFlow, the Controller – Switch Interface (Southbound Interface)

OpenFlow has since its inception been the premier *standardized interface*[1] between SDN controllers and switches or other datapaths. It features support for a number of commonly used dataplane protocols ranging from Layer 2 to Layer 4, with packet classification being performed using stateless match tables, and packet processing operations (called actions or instructions) ranging from header modification, metering, QoS, packet replication (e.g. to implement multicast or link aggregation) and packet encapsulation/decapsulation. Various statistics are defined per port, per table, and per table entry. Information can be retrieved on demand or via notifications.

OpenFlow is however not merely an interface. It also defines the *expected behavior* of the switch (and how the behavior can be customized using the interface). OpenFlow, therefore, represents a *model* and an *architecture blueprint*.

As these aspects have proven useful, it is desirable to retain them. The aim is specifically to support co-existence of the current OpenFlow and the next generation of OpenFlow, as well as migration from the current to the next generation.

2.2.3 Supporting Additional Use Cases

The aim is furthermore to broaden the applicability of SDN by supporting additional use cases. This could involve introducing support for new services and applications (e.g. stateful firewalling), new networking domains or new media (e.g. optical or microwave), and/or legacy

[1] This interface is often referred to as the southbound (from the perspective of the controller) interface.
[2] Multiple presentations at the first P4 workshop (June 2015) confirmed this.

functionality that is not supported yet (e.g. OAM). These, in turn, can require making provision for additions or changes to dataplane protocols as well as supporting new or changed switch and/or controller behavior. The following sections explore various approaches that can facilitate these enhancements.

2.2.3.1 Standardizing Dataplane Protocols and Behavior using Specification Modules

The practice of enshrining the complete set of supported dataplane protocols in a monolithic specification document needlessly impedes the introduction of new dataplane protocols or protocol variants. A monolithic specification document similarly constrains the evolution of behavior specifications.

The concept of a modular specification, with the core describing common concepts and infrastructure, and with required behavior and protocol details being specified in separately developed modules, alleviates these concerns as behaviors and protocols can be defined by multiple parties in parallel in a more streamlined manner.

2.2.3.2 Flexibly Programming Datapath Behavior and Dataplane Protocols

While a few new dataplane protocols are expected[2] to be introduced per year, the behavior of the datapath may need to be modified more often. Changes in behavior may not be associated with supporting new dataplane protocols, for example influencing the statistics retained by datapaths or introducing additional network monitoring and network debugging facilities may be required. The overall behavior of the datapath may furthermore need to change, for example, while the OpenFlow specification had to be amended to introduce the notion of egress tables, such specification changes would no longer be required given the ability to describe arbitrary control flows and arbitrary juxtapositions of primitive elements.

Emerging initiatives permit datapath behavior and dataplane protocols to be defined in a flexible and in some cases quite dynamic manner without these aspects necessarily being defined by a formal standard. Such initiatives include Protocol Oblivious Forwarding (POF), Programming Protocol-Independent Packet Processors (P4), OpenFlow-Protocol Independent (OF-PI) and Protocol Independent Forwarding (PIF). With these initiatives, the required datapath behavior and dataplane protocols are either described in a standardized domain specific language (e.g. P4 or the PIF Intermediate Representation), or are encoded in messages conveyed over a southbound interface.

Although terms like PIF and POF continue to be used for historic reasons, the understanding is that the new mechanisms permit flexibly defining (i.e. “programming”) the datapath, not merely introducing support for new dataplane protocols.

2.2.3.3 Facilitating Interactions Between Software and Datapaths

Some datapaths are implemented using software running on general purpose processors (located for example in datacenter servers or on service blades) or networking focused processors (located for example in line cards in a chassis or on intelligent network interface cards). Other

[2] Multiple presentations at the first P4 workshop (June 2015) confirmed this.

datapaths may be implemented on less expressive devices, for example, switches or gateways based on pipelines. In either case, the need for software to interact with the datapath arises.

Such software may be hosted within the switch device or server platform, running on the control CPU of the switch / the host CPU of the server, or on the switch / NIC fastpath hardware itself. It may also be located near the switch or server, for example in the same chassis or rack. It may also be positioned further afield, for example on the SDN controller. This leads to the notions of the dataplane fastpath, a medium speed path, a slower speed path, etc., with the lower speed paths being located further away from the fastpath hardware, and being used for rarely occurring traffic (for example exception path traffic).

Another perspective arises when considering the deployment of Virtual Network Functions or L4-L7 Services. Either of these may need to be furnished with traffic, leading to the notion of a service chain comprising multiple VNFs or instances of services. The VNFs or L4-L7 Services may be packaged in virtual appliances / virtual machine images, container images, or as regular applications. They may be hosted on the server / gateway / middlebox containing the fastpath hardware (e.g. switch or intelligent NIC), or be hosted elsewhere in the network, leading to the notion of steering traffic over the network to the VNF or L4-L7 Service (often by employing a tunnel).

In all of these cases, the software implementing the specific function needs to be able to interact with the network overall and specifically with the fastpath in the networking device(s) feeding traffic to it. These interactions may be limited to the control plane, with software for example obtaining statistics or influencing the forwarding behavior by populating match-action tables (i.e. downloading so-called rules). Such interactions need to either be routed via the SDN controller or be coordinated with the SDN controller by employing notions like selective delegation and access permissions for the affected data structures in the fastpath. The interactions may also extend to the dataplane, with the software being able to forward, originate, terminate, drop, or modify traffic.

The interactions may be restricted to certain phases associated with the datapath, for example some interactions may only be supported shortly after system (re-)initialization (e.g. datapath instantiation or the downloading of programs may need to occur at this time), while other interactions (e.g. querying of statistics) may be supported subsequently or at any time. This leads to the notions of configuration time vs. run time respectively. Further delineations are conceivable.

The interfaces used by software to accomplish the aforegoing may be defined as protocols, in which case the invoking software needs to implement the interfaces directly or (more typically) by employing protocol stacks implemented in libraries. It is significantly more convenient for software authors however if the interfaces are available directly as standardized callable APIs with various language bindings.

The overall motivation for considering all of these types of software interacting with the datapath, as well as the mechanisms to support such interaction, is to support new use cases while simplifying the development process and permitting the emergence of new ecosystem players for existing and new use cases.

2.2.4 Deployment and Lifecycle Considerations

In addition to considering the technologies that enable packets to be matched and acted on ("day in the life of a packet"), and the technologies that enable a switch to be programmed, configured and operated ("day in the life of a switch / controller / program" etc.), there are important considerations related to deployment and the full lifecycle of these technologies. These need to be considered in order to ensure that the technologies make provision for all the scenarios that may arise. Lifecycle considerations include understanding the preparation of the technology (pre-deployment), for example the construction of a datapath program and corresponding SDN controller and application software or at least data to enable the program to be used, the deployment of the technology (in greenfield or brownfield environments – also considering upgrading and migration), as well as the re-use of technology. These need to be considered for each technology element in isolation as well as for a certain permutation of technologies, for example, upgrading a program to a new version may require a concomitant upgrade to the hardware of a programmed switch.

2.2.5 Facilitating Ecosystem Growth

Facilitating ecosystem growth is a key motivating factor underlying the entire SDN Evolution effort as well as the PIF effort. This includes expanding the addressable market and applicability for existing technologies and ecosystem players as well as enabling new ecosystem players / new categories of players and new technologies to emerge.

Understanding all the deployment and lifecycle related considerations enables the technologies and recommended processes to be optimized in order to promote ecosystem growth. The growth of the ecosystem also requires understanding and documenting all the actors and roles in the ecosystem as well as their interactions. This enables the interactions to be optimized and provision to be made for additional ecosystem players.

2.3 Terminology

Term / Acronym	Definition
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
ASSP	Application-Specific Standard Product
CPU	Central Processing Unit
DSL	Domain-Specific Language
FPGA	Field-Programmable Gate Array
HLL	High-Level Language
ID	Identifier
IP	Internet Protocol
IR	Intermediate Representation
ISPU	In-Service Pipeline Upgrade
LAN	Local Area Network
LOM	LAN On Motherboard
MAC	Media Access Control
NDM	Negotiable Datapath Model
NIC	Network Interface Card
OAM	Operations, Administration, and Maintenance
OF-PI	OpenFlow-Protocol Independent
ONF	Open Networking Foundation
OS	Operating System
OXM	OpenFlow eXtensible Match
OpenFlow, OF-Switch	A controller to switch (“southbound”) interface being standardized by the ONF

Term / Acronym	Definition
P4	Programming Protocol-Independent Packet Processors, a domain specific language to enable datapaths to be programmed
PIF	Protocol Independent Forwarding
POF	Protocol Oblivious Forwarding
QoS	Quality of Service
RPC	Remote Procedure Call
SBI	SouthBound Interface
SDN	Software Defined Networking
ToR	Top of Rack
TR	Technical Report
TTP	Table Type Pattern
UDP	User Datagram Protocol
VLAN	Virtual LAN
VXLAN	Virtual eXtensible LAN

3 Flexible Datapaths: Configuration and Run Time

This section describes the two main phases associated with programming datapaths:

1. configuration time, involving the preparation, translation/compilation, downloading and starting of a datapath program; and
2. run time, involving ongoing interaction with the program.

Note that although these are described as distinct perspectives in the interest of separation of concerns, they may share technology, for example it is conceivable for the program to be downloaded using the same protocol (e.g. OF-Switch) that is used to interact at run time with the datapath program, and it is even conceivable to modify the entire program or parts of the program at run time.

As unifying these perspectives is potentially problematic and controversial, further discussion of this aspect is required.

3.1 Configuration Time: Programming the Datapath

3.1.1 Datapath Programs

Datapath programs, in essence, describe the expected behavior of the datapath. This is mostly done from the perspective of a “day in the life of a packet”, for example, the program covers the aspects depicted in Figure 2-1 below:

1. parsing the packet (implying declaring the header fields per supported protocol as well as the parse tree) - orange in the diagram;
2. matching the packet (implying some declaration of the supported tables or other matching data structures and the control flow between them) - green in the diagram;
3. performing actions on the packet according to the result of matching (implying declaration of the actions) - green in the diagram;
4. performing traffic management / QoS operations - purple in the diagram; and
5. performing monitoring related operations, for example maintaining counters.

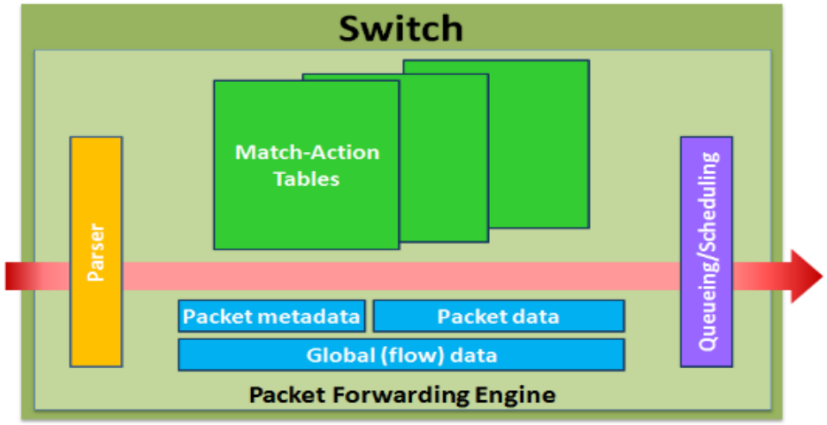

Figure 2-1: Switch Executing a Datapath Program

Datapath programs can also define behavior unrelated to packets, for example, tasks that need to be executed periodically such as aggregation of statistics.

In order to enable this to be achieved by the program, underlying infrastructure objects can also be declared and used. These include for example per-packet metadata and per-table / global state objects – light blue in Figure 2-1.

Although the diagram depicts parsing and match/action table processing as distinct operations, parsing need not be a distinct operation performed at ingress. It can be combined with match/action table processing, and it can (and, for some protocols, must) be performed incrementally, e.g. just before the parsing results are required.

In OpenFlow 1.x, the required behavior is in effect specified as a combination of the standard behavior expressed in the OpenFlow specification, and the information downloaded via the OpenFlow protocol. The specification, for example, defines the supported protocols / header fields. Actions and instructions expressed using the OpenFlow protocol conversely define the actions to be performed when table entries are matched, and the flow of control between tables.

With datapath programs, the notion of pre-defining some behavior in the program is added. The sequence of actions to be performed when a table entry is matched could, for example, be predefined in the program as in effect a subroutine / macro, with matching operations invoking this subroutine / macro with appropriate parameters (e.g. packet header fields or packet metadata). Control flow between tables can be influenced by the matching process, but the permissible paths can also be explicitly defined in the program.

Required standard behavior can continue to be documented in a specification (expressed using human comprehensible natural languages), but it can also be documented in standard libraries (using machine and human readable domain specific languages).

The required behavior can in principle continue to be defined using the run-time interface (e.g. a successor of OpenFlow) as well. Permitting extensive flexibility w.r.t. run-time specification of behavior enables controllers to dynamically change behavior without introducing downtime by forcing programs to be recompiled. It may, however, increase the burden on implementers to achieve high performance.

The extent to which behavior needs to be dynamically defined by the run-time interface vs. to which it can be predefined in the datapath program needs to be discussed / determined.

3.1.2 Compilation

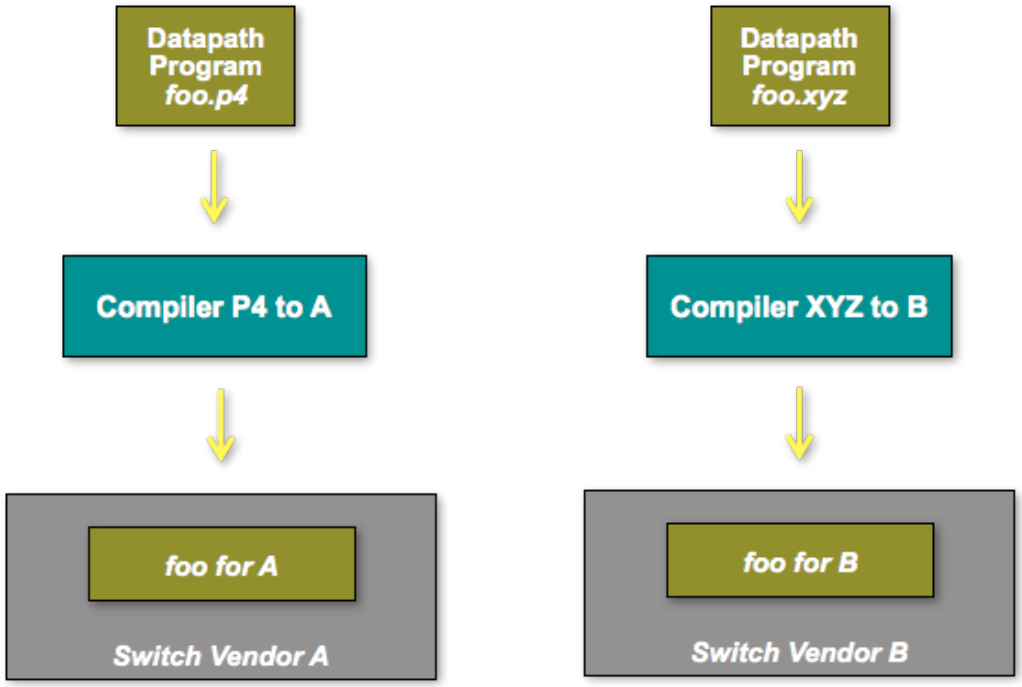

Figure 2-2: Language / Target Specific Compilation

Figure 2-2 depicts the process of compilation using a few examples. The P4 to A compiler translates the P4 language to whatever object code is required to execute it on the platform from vendor A, whereas the XYZ to B compiler performs this task for the XYZ language and the platform from vendor B.

In Figure 2-3 below, an Intermediate Representation is introduced to avoid an individual compiler being required for every high-level language – target platform combination. The various high-level language front-end compilers target the common Intermediate Representation, with target specific back-end compilers from the IR to the object code for the target being created. This is feasible if the IR is abstracted at the appropriate level to enable it to be targeted by multiple high-level languages, while remaining vendor independent.

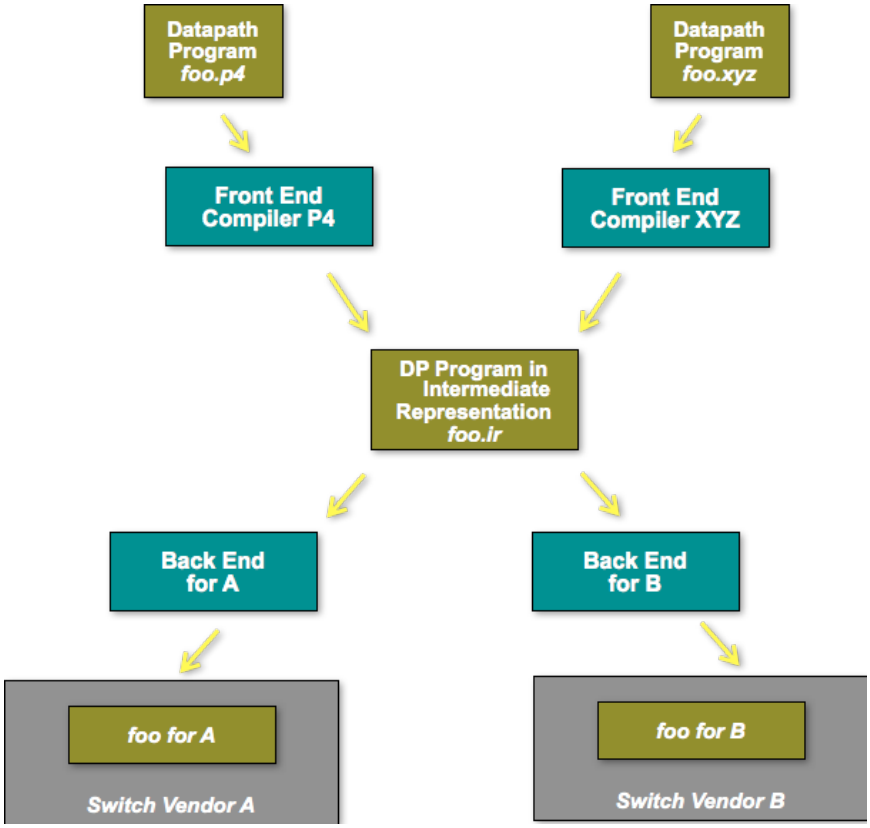

Figure 2-3: Compilation to Intermediate Representation and Vendor Specific Representation

3.1.3 Ecosystem

For each of the new components (datapath program – high-level / low-level language, front-end / back-end compilers), we need to consider:

- What is standardized or not, and where / how, e.g. the OpenSourceSDN PIF project evolves the Intermediate Representation via an open source project, leading to a specification.
- Who supplies what and who consumes it, e.g. one option would be for the industry to adopt a common open/closed source HLL to IR compiler, with individual vendors supplying device / target specific back-end compilers.

3.1.4 Existing Technologies

The following existing projects, languages, and technologies are relevant to this effort. Which in each category ends up being adopted remains to be seen. Details w.r.t. each of them should be considered to determine the broader implications and other considerations associated with the concepts and their usage and evolution.

- P4, a domain specific high-level language being standardized by the P4 Consortium (p4.org)
- PacketC, a domain specific high-level language resembling C, originated by CloudShield, acquired by LookingGlass (www.apress.com/9781430241584)

- Protocol Independent Forwarding (PIF) open source project, standardizing an intermediate representation (OpenSourceSDN.org)
- Protocol Oblivious Forwarding (POF) intermediate representation and configuration/runtime interface, defined by Huawei (poforwarding.org)
- eBPF (extended Berkeley Packet Filter) Virtual Machine, present in the Linux kernel (www.kernel.org/doc/Documentation/networking/filter.txt and iovisor.org)
- PX, a domain specific high-level language defined by Xilinx (www.xilinx.com/sdnet)
- NetASM, an assembler / intermediate representation level language (www.cs.princeton.edu/~mshahbaz/sites/netasm)

3.2 Run Time: Interacting with the Datapath (Southbound Interface)

This section describes the considerations that arise during operation of a (logical) switch, i.e. after it has been instantiated and (where applicable) furnished with a suitable datapath program.

3.2.1 Run-Time Interaction via Southbound Interface

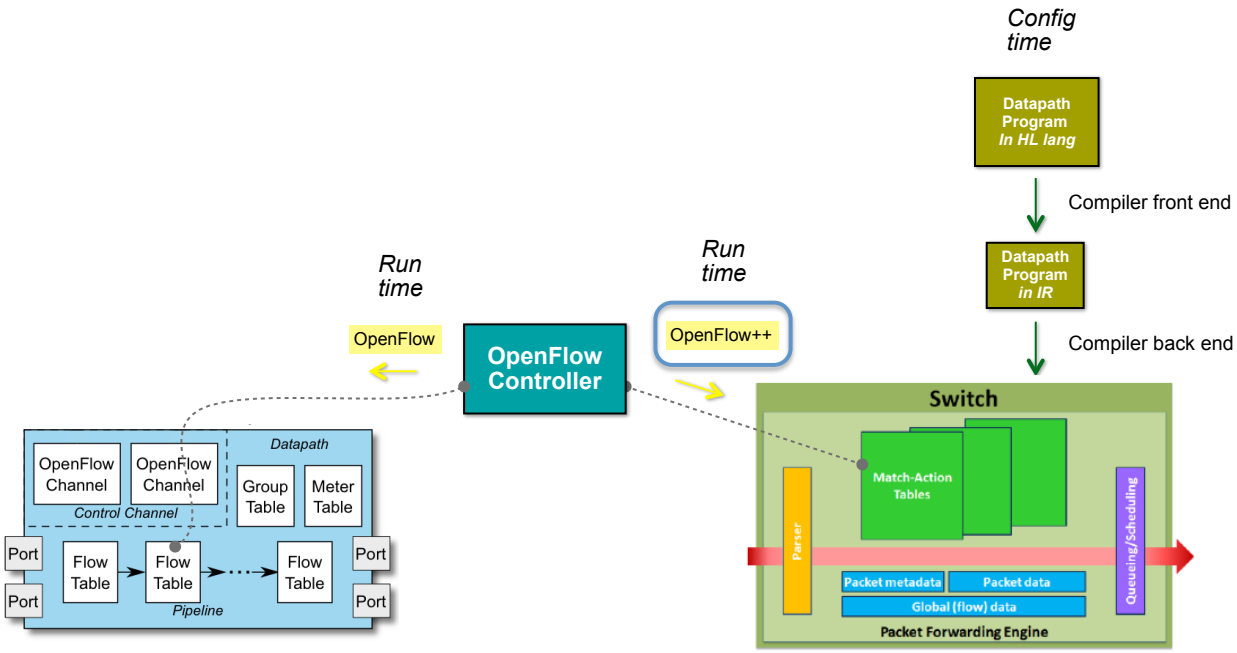

Figure 2-4: Run-time – Southbound Interface from Controller to Switch

Figure 2-4 depicts the run-time interaction between an SDN controller (in this case an OpenFlow controller) and various types of switches.

The switch on the left is a pre-programmed (non-protocol-independent) OpenFlow switch, with the controller interacting with it using the existing OpenFlow 1.x (OF-Switch 1.x) protocol. In this case, the program compilation aspect of configuration time does not apply, however, configuration time as a broader concept is still relevant. A logical switch needs to, for example,

be instantiated on the hardware (i.e. a capable switch) prior to the logical switch being accessible using OpenFlow, and this is performed during configuration time.

The switch on the right is furnished with a datapath program during the configuration and compilation process. Once the datapath program has been started, the run-time interface is used to interact with it.

In either case, the run-time interface enables interaction between the controller and aspects like the following on the switch:

- Table entries or other data structures consulted during the classification process.
- Statistics and other elements reporting status or monitored information, for example, event queues.
- Sending exception path packets to the controller, or enabling it to inject packets into the network, for example for network topology discovery.

These interactions are similar at a high level in both cases. Where custom protocols are defined by the program, the details may differ, because the packet header fields cannot be predefined by a standard. The southbound protocol may need to be extended to accommodate this (hence referring to it as "OpenFlow++" in the diagram). One may, for example, need to dynamically allocate OpenFlow Extensible Match (OXM) IDs to each newly defined protocol header field.

In cases where custom protocol fields are not created, and where the behavior of the switch (expressed in the datapath program) conforms to the OpenFlow specification, the southbound protocol would not need to be extended, as depicted in Figure 2-5 below.

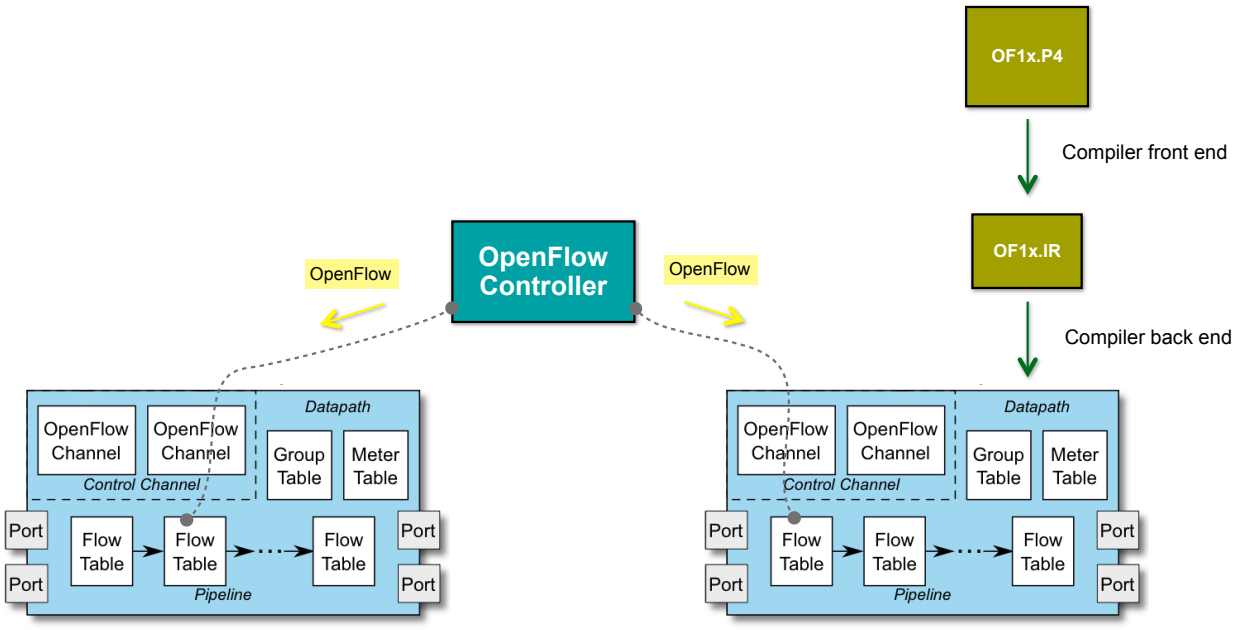

Figure 2-5: Datapath Program Implementing OpenFlow 1.x Switch

It could, however, be nontrivial to express the full functionality of OpenFlow in such a datapath program. OpenFlow, for example, permits the control flow, i.e. the path taken from table to table,

as well as the specific actions to be taken when entries match to be defined at arbitrary times using flow modification messages. Datapath programming languages do not necessarily permit these to be changed during the run-time phase, as they may need to be pre-defined at configuration time. Similarly, the fields matched by each table may also need to be defined at configuration time. The implementation may therefore need to detect that a new control flow path, match table pattern, action list pattern etc. are being used, and dynamically regenerate and recompile the program to take this into account. The compilation time and the difficulty to restart the program with state being preserved may impose significant burdens on the implementers and users of such a system.

It is therefore advisable to either restrict the flexibility supported by the switch using a Table Typing Pattern (TTP) which is locked in during configuration time and/or to amend the OpenFlow specification to require predefining the problematic aspects during configuration time. Although this would arguably offer less dynamic flexibility than the existing OpenFlow, note that this would for most practical scenarios not result in overall functionality regressing when considering the combination of configuration time and run time. One would merely need to pre-declare the relevant aspects at configuration time, with references (bindings) to them being introduced at run-time. The additional constraints w.r.t. dynamic behavior may actually facilitate performance optimizations.

Further investigation is required to ascertain whether this reduction in dynamic capability proves to be a problem in practice for actual use cases.

3.2.2 Southbound Interface Concerns

3.2.2.1 Alignment between OF-Config and OF-Switch

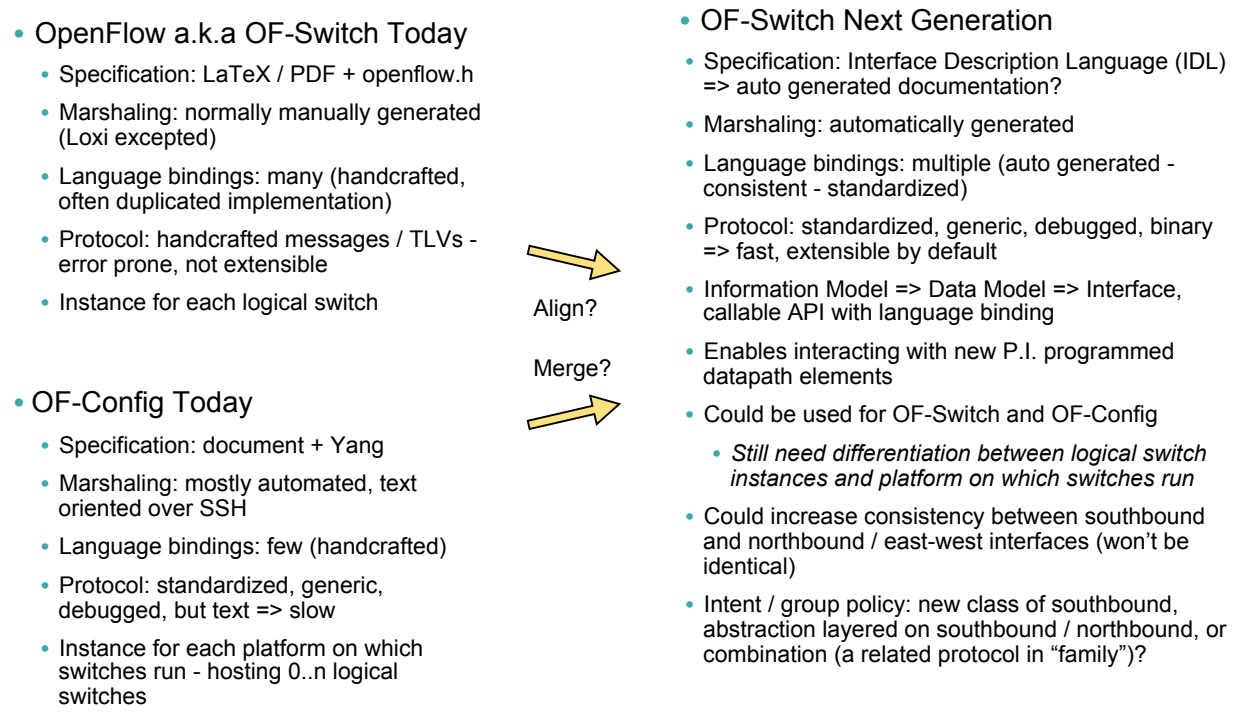

Figure 2-6: OF-Switch vs. OF-Config

Several distinct protocols are currently employed between SDN controllers and SDN switches. Logical switches are for example instantiated using OF-Config or OVSDB, whereas table entries in the logical switches are populated using OF-Switch (a.k.a. OpenFlow).

When considering the evolution of OpenFlow, the extent to which these protocols can and should be aligned needs to be considered. As a starting point for this discussion, Figure 2-6 summarizes various attributes of the existing OF-Switch and OF-Config protocols, as well as the corresponding attributes of a hypothetical merged protocol, called OF-Switch Next Generation here. The next sections explore specific considerations in more detail.

3.2.2.2 Automatic Interface Generation and Callable APIs

The current OF-Switch specification defines the southbound interface as a protocol, with the protocol details being comprehensively defined from first principles, i.e. all details of the messages on the underlying stream transport (TCP or TLS) are specified. This imposes a burden on implementers of the specification, as each implementation needs to include in effect an OpenFlow protocol stack.

Another approach would be to take advantage of existing distributed system infrastructure technologies. In this case, the controller and the switch would communicate via Remote Procedure Calls (RPCs), with the "remoting" of the calls being handled by the existing infrastructure, or they would at least use existing marshaling technologies to generate and parse

the messages that constitute the protocol. This approach would also enable various language bindings to be automatically generated by the infrastructure.

Figure 2-7 explores an extension of this concept, whereby the callable run-time API is standardized, not the wire protocol itself. This enables the wire protocol to be evolved without affecting the remaining controller or switch software components.

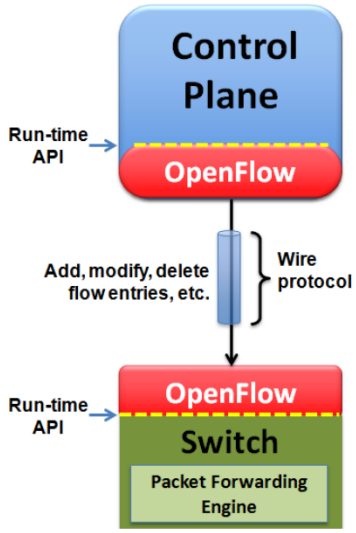

Figure 2-7: Callable Run-Time API vs. Run-Time Interface as Protocol

The callable interface could be fairly generic (independent of the program being executed in the switch). In this case, specific API calls would need to be supplied to detect the data structures (e.g. match tables and protocol fields) exposed by the datapath program in the switch. This process is referred to as introspection.

A unique callable interface could also be generated from the datapath program itself. While this would enable the interface to conveniently permit access to the program's data structures, it would also necessitate recompiling the client (e.g. the controller) invoking the run-time interface whenever the program's interface changes. This may be impractical in controllers that need to support many switch types, programs, and program versions.

When datapath program specific generated run-time interfaces are appropriate vs. when generic run-time interfaces are appropriate needs to be discussed. This is related to whether or not controllers need to be datapath program agnostic.

3.2.2.3 Alignment with Northbound Interface

The interface northbound of the controller is necessarily different to the interface southbound from the controller. If this were not the case, the controller would only fulfill a partitioning and virtualization role, whereas it actually translates a higher level network-wide intent into a lower level set of instructions (e.g. table entries) targeted to a specific logical switch.

It is nevertheless worth considering to which degree the underlying infrastructure for the northbound interface can be aligned with the southbound interface. Similar RPC and automated

marshaling technologies could, for example, be employed. Furthermore, there are data elements that are common between the interfaces, e.g. ports, speeds, links, node addresses, etc.

Aligning the interfaces as far as possible would facilitate developing software that can be repositioned to execute either on the controller or on a switch. This is non-trivial, as it requires provision to be made for delegation of responsibility from the controller to the switch.

Further study is required to explore the feasibility of alignment of the northbound and southbound interfaces, as well as the implications of repositionable software modules and the infrastructure required to support delegation.

3.2.2.4 Evolution of the Southbound Interface

The current OpenFlow southbound interface expresses desired behavior at the level of specific match/action table entries. It is theoretically possible to define the southbound interface to be a higher-level interface, where the responsibility of translating the high-level intent (communicated to the controller via a northbound interface) to details is partially delegated to the switch. This needs to be considered with due care, as burdening the switches with network-wide considerations may be a regression vs. the simplicity of logically centralized control. Furthermore relying on extensive data and complex algorithms being present in switches may reduce flexibility vs. locating this intelligence in the controller.

Further consideration of the appropriate level of abstraction of the southbound interface is required.

3.2.2.5 Ecosystem Considerations

For each component or interface, we need to consider what is standardized vs. what is defined via open source code, and what remains vendor-specific.

4 Target Platform Considerations

4.1 Platform Types and Capabilities

This section illustrates the wide variation in the capabilities offered by the various target platforms by considering the various datapath architecture types, silicon device types, and system types. The sections do not contain exhaustive lists, as the intent is to illustrate the spectrum that needs to be considered by presenting examples.

4.1.1 Datapath Architecture and Degree of Programmability

For devices that perform packet processing (e.g. packet parsing, header matching, and actions involving header modification), the following main datapath architectures and corresponding degrees of programmability can be identified.

- Control flow centric stereotype:
  - Random packet access

- Run to completion
- Generalized execution units and datapaths
- High programmability through instructions

- Data flow centric stereotype:
  - Linear packet access
  - Pipelined
  - Specialized function units and datapaths
  - Varied programmability through configurations
- Hybrids of the above are also possible.

Devices that do not perform packet processing (for example optical transport devices) do not fall into these categories.

4.1.2 Silicon Device Types

Typical examples of silicon types are listed in this section, in increasing order of programmability and decreasing order of performance.

What some refer to as the "switching chip" can be a virtual concept. It describes the abstract forwarding model being employed. It may be implemented using a combination of physical devices (i.e. silicon device types). This section focuses on the actual silicon device types.

4.1.2.1 Traditional ASIC/ASSP Switch

The traditional switch has a set of built-in protocols as well as switching capabilities based on these protocols' packet fields. There is some degree of configuration of switch instances, but this does not extend to adding or subtracting protocols or known packet fields.

4.1.2.2 Protocol Independent Flexible ASIC/ASSP Switch

A new generation of switch offers programmability that allows the addition or subtraction of protocols under runtime control. This includes programmable parsing of packets and updating of packets at various stages in a fixed packet processing datapath architecture.

4.1.2.3 Programmable Logic (FPGA)

The FPGA allows custom packet processing datapaths to be defined using programmable hardware. Furthermore, these datapaths may have runtime programmable features. Together, these capabilities allow programmability of the datapath architecture itself, and of datapath architecture instances.

4.1.2.4 Networking Focused Processor (Network Processor, Flow Processor)

The networking-focused processor, which may have a multicore processor architecture or a more specialized pipelined processor architecture, is programmed using a proprietary instruction set. It may be able to accommodate custom packet datapaths, and it is capable of supporting programmed protocol packet handling in the datapath.

4.1.2.5 General Purpose Processor (Software Switch)

The general-purpose processor can emulate any switch datapath, with features programmed as desired. This allows protocols to be added and subtracted as software updates, or as software configuration updates.

4.1.2.6 Finer Grained Classification / Supporting Devices

In some cases, additional silicon devices need to be deployed to enable the silicon devices of each aforementioned type to perform their functions. In other cases, the aforementioned types are implemented using individual devices with the following subtypes.

- Forwarding related silicon, performing packet parsing and forwarding, e.g. fabric devices.
- Processing related silicon, enabling packet modification (header editing, encryption etc.).
- Classification related silicon, performing lookups or matching.
- Storage related silicon, storing table entries or packets.

A further distinction can be drawn between silicon with predefined functionality / logic and programmable logic.

In all cases, at least a small general purpose processor is present, to enable firmware to be downloaded / updated, or to implement the control plane and perform exception path processing.

4.1.3 System Type

One or more silicon devices (potentially belonging to several of the categories listed in the previous section) are assembled on one or more printed circuit boards to form systems. This section describes the frequently encountered types of systems. Each type can furthermore occur in various sizes and footprints, for example, small or large pizza box, modular chassis with few or many blades, fixed-configuration units that can be stacked, etc. Some of these may be more or less appropriate to be deployed at certain locations, e.g. smaller integrated devices are frequently preferred at the customer premises for cost / power consumption reasons, while larger and more flexibly configurable units are deployed in access and core networks / datacenters.

4.1.3.1 Switch / Router

A traditional switch / router is furnished with multiple dataplane networking ports (at least two - but typically 4 to 96 or more) and optionally out of band control / management networking ports as well. These may occur in various form factors, for example, pizza box Top of Rack (ToR) switch, modular chassis, stackable units, etc. The difference between a switch / router and an appliance / middlebox is that the general purpose processor and software is typically only used for the control plane in a switch / router, with forwarding being delegated to purpose-designed silicon, whereas in the appliance / middlebox case, substantial capabilities to inspect and process (not merely forward) traffic exists, either as specialized hardware or as general purpose / networking focused processors with sufficient capabilities to be used for the dataplane.

4.1.3.2 Appliance / Middlebox

A networking appliance, sometimes called a middlebox, is a device comprising networking hardware / software which interacts with the dataplane in more complex ways than by merely

forwarding the traffic. It often fulfills security functions or hosts L4-L7 services. It is often physically a standard server or a server augmented with network acceleration hardware or other alterations to improve support for networking (e.g. an integrated switch, front-facing ports, modular network interfaces supporting various media types or speeds, fail-to-wire hardware, etc.). The distinction is not absolute, as switch / routers can also enforce security policies (e.g. implement Access Control Lists).

4.1.3.3 (Endpoint) Server with Network Interface Card (NIC) or LAN on Motherboard (LOM)

This system type comprises the following sub-types:

- Switch or virtual switch software running on the server CPU using basic (unaccelerated) network interface hardware.
- Embedded (virtual) switch running entirely on the network interface card hardware, without the host server system needing to participate in switching traffic.
- Accelerated (virtual) switch, with the host server performing switching of a subset of the traffic, but with some functionality delegated to intelligent network interface hardware (either all processing of another subset of the traffic, or some / all match processing, or some / all action processing).

Here the term "virtual switch" is used to connote that software is implementing the switching, i.e. that physical switch hardware is not present. The simpler term "switch" can also be used, as some hardware always needs to be involved to host software, and as the use of acceleration hardware blurs the lines between "virtual" switches implemented purely in software and "physical" switches implemented with the assistance of some switching / networking specific hardware.

4.1.4 Non-Packet-Switching Devices, e.g. Optical and Microwave Transport

While OpenFlow tends to have a packet switch focus, it is applicable to other data network systems such as optical switching. Extensions to support Optical Transport networks were prepared by the Open Transport Working Group[3]. Some features were introduced in OpenFlow version 1.4 to enable control and status monitoring of optical devices[4]. These features, combined with the concept of logical ports and OpenFlow's provisions for extensibility, have been proven to function in systems other than packet switches. The new concepts outlined in this Technical Recommendation should further enhance capabilities for such systems.

[3] Refer to http://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/Optical_Transport_Protocol_Extensions_V1.0.pdf
[4] Refer to https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf

4.2 Profiles and Models

4.2.1 Modeling Run-Time Capabilities in Detail

Developing a Software Defined Network requires an understanding of the run-time capabilities of the switches to be used to build the network. This understanding can be derived from the capabilities of available switches or it can represent required capabilities to be provided by (programmed into) programmable switches. In either case, the switches must deliver the flow controls and other behaviors required by the services the network is designed to support. The run-time capabilities of a switch can be expressed as a datapath model (or control profile) that defines switch behaviors and the associated controls offered to the SDN Controller.

One tool for defining SDN switch capabilities is an OpenFlow-based datapath model called a Table Type Pattern (TTP)[5]. The core of a TTP is a set of match-action table definitions. Each table definition specifies the type(s) of entries that can be put into the table. A table entry type is defined as a match type and an associated action to be taken if the entry is selected at run-time. Network flow processing is thus defined by the order in which tables may be visited and the entry types that may be selected for execution at each table. These models are formal and rigorous to enable them to be machine and human readable.

4.2.2 Expressing Device Categories and Capability Categories Using Profiles

A higher-level definition of capabilities would permit classification of devices into specific categories based on the device behavior and/or the market segment addressed by the device. Examples of categories would be traditional product / market segments like L2/L3 switch, firewall, and load balancer, or newly defined segments like high touch NFV platform vs. forwarding platform. Here the capabilities would be described at a higher level than the datapath model / TTP level - potentially using natural language descriptions that are not machine readable. The higher-level categorization could prove useful as an initial filter of products to be considered when procurement decisions are made.

Conformance certification necessitates specifying requirements with sufficient formality and in sufficient detail to permit unambiguously determining whether or not candidate devices comply. A high-level description of requirements expressed in natural language is therefore by itself not sufficient to support conformance certification. Instead of inventing a new formalism to support rigorous definition of requirements, it is advisable to map each higher-level capability category to a number of individual detailed datapath models / TTPs that need to be supported. The datapath models may need to be extended using for example specific test vectors to further reduce ambiguity and promote automated conformance testing.

Certain devices with programmable hardware can fulfill the requirements of multiple higher-level categories, with the downloaded program determining the platform's behavior. In this case, the downloaded program would need to be included in the conformance test suite (if it is expressed in a vendor-independent language) or supplied by the vendor (if it contains vendor / device specific details).

[5] Refer to https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/OpenFlow%20Table%20Type%20Patterns%20v1.0.pdf

4.2.3 Device Type and Application Area Profiles

OpenFlow TTPs can be used in several contexts to define controllable datapath behavior. These include defining the controllable behaviors:

- of a particular switch platform (box-level product or its underlying ASSP chip),
- associated with a particular networking protocol/technology, or
- required of a switch playing a particular role in a network design.

Each of these can be useful in an appropriate context.

The first type of TTP (a “Platform” or “Device” TTP) can be used to define a flexible (OpenFlow) interface to a family of products. This can provide a stable model for run-time control of these products.

In the context of SDN Controller development, the last two are particularly useful. Controller developers can maximize software reuse by implementing libraries for specific networking technologies and implementing subsystems for particular network application areas. These models are (or should be) independent of any particular switch platform and thus define an open and portable interface for run-time control.

Technology specific TTPs are most useful in the form of reusable TTP fragments (since few network applications employ only one network technology). A TTP for a particular network role is called an “Application Area” TTP. An Application Area TTP is expected to be useful across a range of related network applications that rely on a common set of capabilities.

4.2.4 Table Type Patterns (TTPs)

When establishing a control relationship between an SDN Controller and a Logical Switch, there must be a common understanding of the datapath controls that are provided by the switch and thus available to the controller. This understanding may substantially precede the establishment of the control relationship (e.g., the controller may have been designed assuming a datapath model and the switch selected to conform to that assumption). Alternatively, the understanding may be developed (or refined) at the time the control relationship is established via a negotiation process. Either party to the negotiation can be flexible (or not). The controller may be able to take advantage of a range of available behaviors that might be offered by the switch. The switch may be able to support a range of datapath models and adopt a model requested by the controller.

Mechanisms have been defined for negotiating a TTP between an OpenFlow Configuration Point and OpenFlow Capable Switch (using) as well as between an OpenFlow Controller and OpenFlow Logical Switch (using an). These mechanisms provide some flexibility in the selection of the run-time control model in varied operational scenarios.

4.2.4.1 Defining the Datapath Model

While some flexibility in datapath model selection at run-time can be useful, defining the datapath model in advance allows implementations to be stabilized to enhance interoperability, ensure security and optimized to improve performance. Datapath models can define the run-time controls for:

- a switch platform (box-level product or its underlying ASSP chip),
- a datapath program/configuration (e.g., a P4 program), or
- an application area (network role).

In the first two cases, a Device TTP is derived from the functions provided by the datapath in question. The model may hide some implementation details to define run-time controls in OpenFlow terms (i.e., match-action tables); however, some platform or implementation related aspects[6] may remain in the resulting model. The goal is to define an OpenFlow control model for the specific platform or program that can be used across the range of applications it can support.

In the last case, an Application Area TTP (AA TTP) is developed without regard to platform or implementation details. The goal is to define a model that is portable across platforms and captures the controls required by the application area.

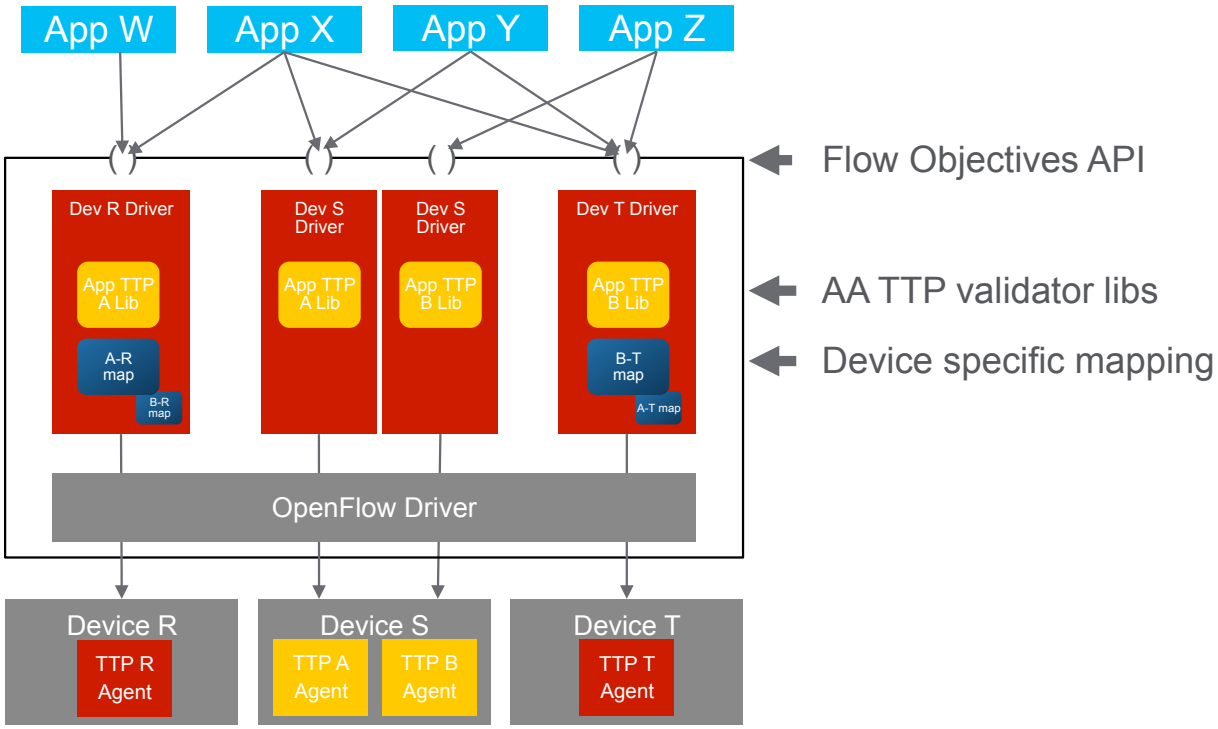

Figure 3-1: Application Area (Use Case) vs. Device TTPs

Both Device TTPs and Application Area TTPs play a role in SDN development. Controller subsystems may be written against Application Area models to enable portability across switch platforms. OpenFlow switches may be delivered with built-in Device models that enable their use across a spectrum of applications supported by the switch. A mapping driver can bridge the gap between controller subsystems written to Application Area models and OpenFlow switches with built-in Device models. The various mapping layers that are required to achieve this are

[6] The capabilities exposed by the switch platform (box) are almost always a subset of the capabilities of the underlying ASIC/ASSP (chip), therefore the capabilities of the platform represent the intersection of the capabilities of the underlying silicon and what the switch platform vendor has elected to expose.

depicted in Figure 3-1 above. This figure shows a software architecture that uses code generated from an AA TTP to validate flow objective requests from controller applications. This ensures the flow objective requests conform to the AA TTP (otherwise the request will be rejected). For devices that do not directly support the AA TTP datapath model (e.g., devices whose datapath model is based on a Device TTP) additional mapping code is required to translate the AA TTP flow entry types into the correct Device TTP table entries.

Generally, mapping between an Application Area model and a Device model that is capable of supporting the application area is straightforward. Since the required behaviors and controls are functionally aligned the mapping mainly involves adjusting syntax and structure. If the scopes of the two models differ, there may be a need to create default configuration for areas that cannot be mapped but require some configuration for the datapath to operate correctly.

As tools are developed to write and manipulate TTPs it may be feasible to generate the required mapping code between a given AA TTP and a particular Device TTP that is capable of supporting the required application and relevant constraints of the particular application. In the meantime, a switch vendor or other motivated party can easily write mapping code to support an application without changing the Device TTP supported in the switch.

To support programmable dataplanes, the OpenFlow TTP language should provide convenient mechanisms for associating a datapath model (run-time control) with a corresponding datapath program (configuration). For example, a mechanism should be provided for associating OpenFlow code points with match fields, pipeline fields, and actions defined in a datapath program. Since a datapath program is different from a run-time control model, the relationship between these two constructs must be well understood to enable automated code generation. This code generation could operate in either direction: generating run-time control model (Device TTP) from a datapath program, or generating a datapath program skeleton from a run-time control model (AA TTP).

4.2.5 Conformance Testing and Certification Profiles

Another class of profile can be defined to facilitate conformance testing and conformance certification. Products would be certified to conform to requirements associated with one or more of these profiles.

These conformance testing and certification profiles map to the profiles and models mentioned in the previous sections in potentially a many to many fashion. High-level device capability profiles (e.g. product / market segment category profiles) may map to one or a few conformance testing profiles. Many individual application type profiles (application TTPs) could be required by a specific conformance testing and certification profile. Multiple conformance testing and certification profiles could invoke a specific device capability or application type profile.

The mapping between these can be even more complex when either the conformance testing and certification profiles become modular, or the high-level device capability / application profiles become modular, as a certain combination of the one would then need to map to a set of combinations of the other.

When highly flexible devices are considered, conformance testing becomes challenging, as one needs to certify that the device is able to run a defined class of program, versus it being equipped

with specific pre-defined functionality. A number of individual programs will need to be developed to explore the space of functionality that needs to be supported. This would roughly resemble a compiler test suite, but it will need to address aspects not typically addressed in suites covering individual functions, for example, program complexity and data structure capacity limits need to also be considered. As this is quite complex and as many other issues associated with programmable devices need to be addressed first, investigating the implications of this scenario is deferred at this stage.

4.2.6 Profiles and Next Generation SDN

TTPs and related profiles that this section describes were developed relative to OpenFlow 1.x constructs. Nevertheless, they foreshadow concepts that will be needed for the next generation of SDN. Pipeline description languages will have control profiles associated with them and a common understanding of the control mechanisms will be needed in the next-gen networks just as they are needed now. In addition, as individual SDN networks are transitioning from the current generation to the next generation, adoption of next-gen architecture will be simplified if it is possible to deploy a network architecture that supports both old-gen and next-gen elements simultaneously. This is especially true since one of the promises of SDN is scale; upgrades of very large networks will be much simpler (and lower risk) if they can be done piecemeal.

A network architecture that supports old-gen and next-gen also allows next-gen efforts to focus on specific features or network elements that especially benefit from next-gen capabilities. If old-gen SDN networks have to be replaced wholesale, then more products and features will need to be developed, and riskier upgrades will be required.

In summary, it will behoove the next-gen SDN toolchain architects and implementers to consider migration and/or compatibility mechanisms as they develop the next-gen SDN ecosystem.

5 Lifecycles and Deployment

5.1 Lifecycles

5.1.1 Actors and their Interaction

The following actors / roles (amongst others) are involved in the use of programmable packet forwarding:

- Switch providers: supply of programmable capabilities;
- Application providers: use of programmable capabilities;
- Controller providers: configuration and control of programmed capabilities;
- Compiler providers: tool flow for expressing and mapping programs;
- Library providers: libraries of pre-programmed components;
- Operators: deploying and managing programmed capabilities;
- Researchers: experimentation with new programmable capabilities;
- Test specifiers and test bodies: conformance testing of programming flows and programmed capabilities;
- Standardization bodies: standard mechanisms for programming flows and control of programmed capabilities.

Notable interactions between actors / roles include:

- Switch providers and operators
- Application providers and operators
- Controller providers and operators
- Application providers and switch providers
- Controller providers and switch providers

(In some contexts, it is helpful to make the distinction between the actor, i.e. the actual entity - organization or person - performing actions, and the role played by that actor, i.e. the entity's behavior and responsibilities in the specific situation. This section does not need to draw these distinctions.)

Some of the actors / roles that can be identified and the interactions between them are depicted in Figure 4-1 below.

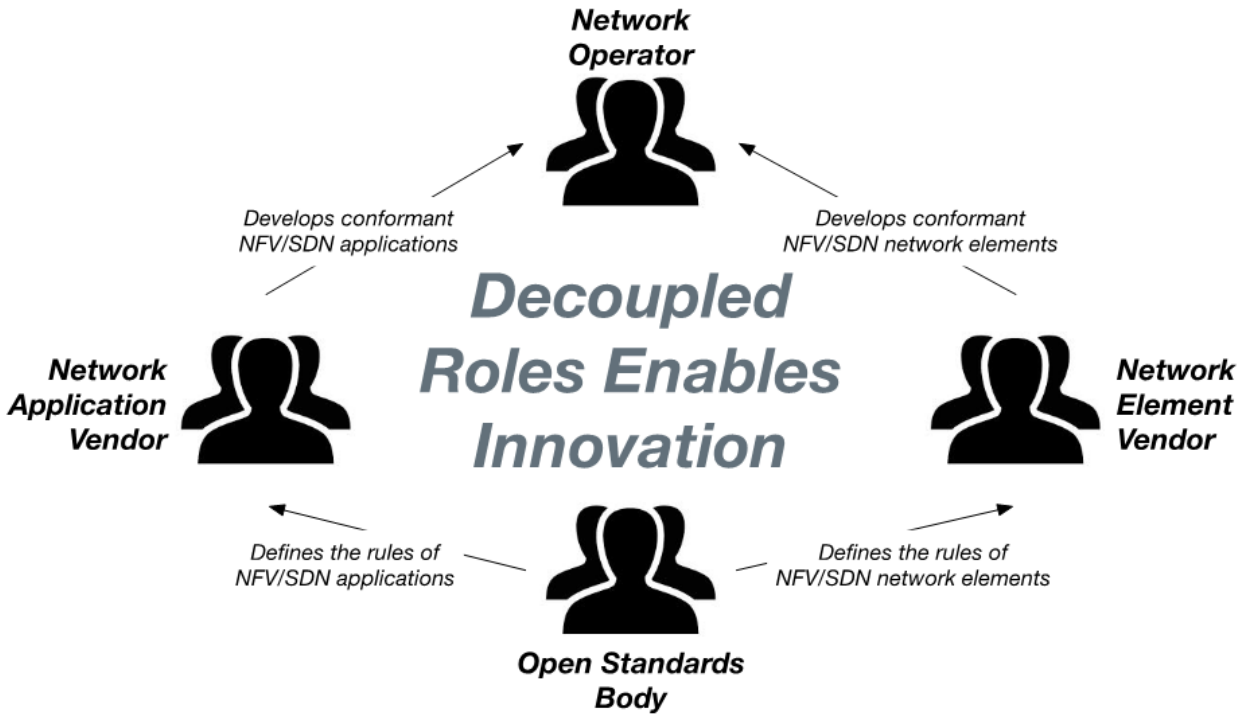

Figure 4-1: Roles and Interactions

5.1.2 Lifecycle Types and Constraints

Next Generation Network Assumptions

An implementation network consists of a number of forwarding devices, potentially in a variety of roles, e.g. Top-of-Rack switch, core switch, server-resident virtual switch, gateways, etc. For each distinct role, a high-level datapath program is written (e.g. in P4). (This is somewhat redundant: the different program suggests a different role, and vice versa.) The datapath program describes a pipeline with control points (flow tables are one example) that are manipulated via some control protocol, potentially a version of OpenFlow Switch Protocol.

Architecturally, the control protocol is expected to be driven from one "logically centralized" SDN controller. In addition, there may also be applications northbound of SDN controller(s).

The Pre-deployment Period

For a production deployment, most or all of the activities below will happen pre-deployment. For lab environments, or demos, or proofs-of-concept, many of the steps may be skipped since agility will rank higher than stability and predictability.

- Datapath program(s) written in high-level language (e.g. P4)
  - Probably generically compiled. Possibly compiled for specific target devices.
  - Simulation performed, code revised, etc.
  - Header Space Analysis performed.

- Signing / authentication of configuration files.

- Specific program support is written for SDN Controller(s)
  - Abstraction code maps the controller APIs into southbound control messages
    - These may be new or previously existing APIs
    - Potentially, model driven abstraction layers may produce the API-to-SBI map code
  - Testing of driver/abstraction code with devices configured with datapath pipeline
- Network applications will be (or were previously) developed on the APIs
- Vendor Integrated Testing, debug
- Conformance testing on various components may make sense (if tests have been defined)

5.1.2.1 Production Programmable Datapath Greenfield Deployment

This would be regarded the most straightforward mainstream use case. It is more complicated than experimental or demo or Proof-of-Concept scenarios, but simpler than upgrades or migration (from non-next-generation situation) deployments, which are operational (are running traffic), and also have variability in their "initial states". The following activities will occur in this case.

- Installation of equipment
- Physical Configuration of equipment and power-up
  - Racking, cabling, power, cooling, etc.
  - Preliminary validation: Stuff runs, seems connected
  - Bare metal configuration etc.
- Soft Configuration of equipment
  - Policy configuration, etc.
  - Self-configuration (topology discovery, etc.)
  - Validation of image compatibility with hardware and operating system versions
- Preliminary testing
  - Pre-production execution of workloads
- Go live!

5.1.2.2 Upgrading a Next Generation Network to Use New Datapath Program(s)

OpenFlow Next Generation has a concept of a "configuration phase" that precedes the "operation phase". In the greenfield launch, we don't think about whether the configuration phase will introduce downtime. In the upgrade case, when hardware pipelines must be reconfigured, we need to think about this question. There are two approaches, and a derivative approach that is something of a hybrid.

1) Long-term Traffic Diversion around upgrading device (simplest to implement)

   a) Through redundancy or another mechanism (such as moving workloads to other parts of the data center and shutting down all local VMs), traffic is steered away from the device being upgraded.

   b) Configure as in greenfield case

      i) Includes validation of image compatibility with hardware and OS versions

   c) Validate functionality

   d) Steer traffic back to the device

2) "In-Service Pipeline Upgrade" (ISPU)[7] means that the traffic running through the device is not noticeably interrupted. Changing a pipeline while traffic is running is not quite "science fiction" (similar notions exist today in some FPGA support), but is quite difficult and likely subject to certain constraints.

   a) Pipeline Configuration downloaded while traffic is active

   b) Validation of pipeline configuration signature with platform versions

   c) Likely halting or restricting of control table changes

      i) This would simplify pre-population of new tables

      ii) May also be needed for translation

   d) Pre-population of control tables directly from controller

   e) Translation of control table changes in prep for populating new tables

      i) Translation removes need for pre-population (which might be slow)

      ii) May need validation of translation (some translations may not be possible)

   f) Then toggle the pipeline to new configuration (non-disruptively)

      i) It should be noted that this is particularly tricky in the sense that any packet must fully transit in one configuration or the other. A non-disruptive toggle would need to be nearly instantaneous to not disrupt traffic. Interpacket gaps might be long enough, but interpacket gaps do not align across all ports. The point of mentioning this is to point out the likely need for supporting the other approaches.

   g) Control traffic is turned back on if it was paused

3) A combination of the above that uses some of the mechanisms in #2 to minimize the traffic disruption.

[7] This resembles In-Service Software Upgrade, but we prefer avoiding that term as "network pipeline configuration" is not yet regarded as "software").

a) Potentially pause control entry changes for pre-population / translation

b) Before steering traffic around the device, download configuration and prepare new control entries

c) Validate everything is compatible (new pipeline is compatible with hardware and software)

d) Redirect traffic around this device

e) Toggle to new pipeline configuration (and new control entries)

f) Direct traffic through the device again

Whatever mechanism is used for the upgrade, it must be recognized that any large network will have a large number of devices. Realistically, the devices will need to be upgraded mostly piecemeal (device-by-device, or rack-by-rack or possibly "pod-by-pod") serially, as taking down a full data center is extremely disruptive and high risk. (If a device upgrade fails, your alternatives are less painful than if the datacenter upgrade fails.)

Because of the piecemeal serial approach, during the upgrade cycle there will be both old and new configurations active in the network. Of course this means that the controller must be able to support both configurations in parallel, though perhaps for a (slightly?) constrained feature / traffic set. New features in the new configuration may not be available until after the full upgrade has completed. Potentially some features in the new configuration are implemented incompatibly with the old configuration, and those features may need to be temporarily disabled or avoided during the upgrade process.

In short, each of the stages in the upgrade cycle, during which features may be disabled or constrained, represent a different "behavior mode" which will require separate validation and testing. Tools will be required to support this kind of validation / testing methodology for smooth production transitions. The larger SDN networks become, the more important it will be to pre-validate and automate transition methodologies.

5.1.2.3 Upgrading Network to Use New Physical Devices

Physical upgrades will be similar to software upgrades in the sense that they will necessarily be piecemeal serial in some fashion, as in-service upgrade is not possible. Also, physical upgrades will no doubt require more time and potentially more testing as there will be more opportunities for cabling and other physical issues to arise than could occur with soft upgrades. The duration of the upgrade cycle will last longer and thus the ability to tolerate constrained features during the upgrade will be diminished. Predictable, physical upgrades will be more painful and thus more desirable to avoid.

5.1.2.4 Upgrading Legacy Brownfield to Include Fully Programmable Datapaths

In many discussions of OpenFlow Next Generation technologies, there is a tendency to ignore support for legacy (ASIC-based) devices and associated migration concerns. Doing so would mean that deployment would only work in greenfield situations, or in a model where OpenFlow Next Generation coexists with non-SDN-enabled devices (operating "over-the-top" for example). These scenarios are possible, but more likely the adopters of OpenFlow Next Generation were also adopters of SDN on ASIC-based devices. Indeed, operators that use such SDN will be more

prepared for SDN on OpenFlow Next Generation (familiar with controller-based network architectures) than other operators.

At a minimum, upgrade scenarios might include replacing (in serial piecemeal fashion) all legacy SDN fixed-pipeline devices with new programmable pipeline devices. During that upgrade cycle, both legacy and flexible pipeline devices will be operational in parallel.

Consequently, we recommend that OpenFlow Next Generation plan to include mechanisms that offer simultaneous support for OpenFlow Next Generation (programmable pipeline) devices as well as for legacy (fixed or largely fixed) pipelines.

Mechanisms for supporting both OpenFlow Next Generation devices and current OpenFlow devices in parallel need to be discussed.

5.1.3 Mechanisms to Facilitate Re-Use

5.1.3.1 Libraries

In this context, the term “libraries” refers to organized collections of constructs built from something lower level. Just as C libraries are themselves written in C, the OpenFlow Next Generation libraries should be somehow built using some OpenFlow Next Generation language.

Here the library may contain the *implementation* itself, but it may actually only represent the *interface* to the functionality, not the implementation. The analogy in the case of C would be a header file (.h file) that enables code written in languages other than C (e.g. assembler or Pascal) to be invoked. Similarly, an interface written in the OpenFlow Next Generation language may represent the capabilities implemented in otherwise unspecified software or hardware components.

Libraries offer the following benefits:

- Avoid duplication: solve a problem once, in a common or standard way, then invoke that solution by reference;
- Hide complexity; and
- Provide well-defined interfaces.

Layering is an important concept that can leverage libraries. Though many OpenFlow conversations seem to regard layering as a problem, if done correctly, layering provides leverage while also allowing mechanisms to operate at various layers. Layering can, for example, benefit the SDN developer as follows. If one were to create a new Layer 3 protocol, say IPv9 (an alternative to IPv4 or IPv6), it should be able to operate on top of Layer 2 protocols, for example, Ethernet. But there are many varieties of Ethernet, including untagged (no VLAN tag), tagged, dual tagged, etc. New Layer 3 protocols should not need to separately decode and implement all the Layer 2 functionality.

Libraries also help address the “protocol dependence/independence” challenge. We want a protocol independent instruction set, but we want to be able to use that instruction to implement

protocol aware functions. Using layering libraries, we can use the protocol independent instruction set to insert a variable length Layer 3 header at the appropriate offset, whether the packet has one or two VLAN headers.

Similarly, a protocol independent networking language should offer us mechanisms (potentially based on libraries) that allow us to refer to a Layer 3 header field when decrementing a time-to-live field. Of course, the lower level primitives should be capable of implementing the decrement function (or the lower level primitives are inadequate). The high-level coder should not be required to recode that function each time. Also, there may be many different ways to code the "decrement field" function. Compilers for legacy "protocol aware" chips should be able to support a protocol-aware reference like "decrement ipv4ttl", and not be forced to recognize all possible primitive sequences that are equivalent to it. If the networking language offers some mechanism for "decrement ipv4ttl", then a compiler can recognize the "ipv4ttl" keyword and map the request to some equivalent function. Of course, the networking language should also allow other constructs like "decrement xyz" where "xyz" is some newly defined field.

5.1.3.2 Abstractions for Extensibility and Inheritance

Getting abstractions right can be difficult. One simplifying approach is to allow for derivative abstractions and subclassing schemes. Whether that makes sense in this context is not clear, but some mechanism for extensibility and inheritance is probably important.

5.1.4 Conformance Testing

OpenFlow has struggled with the challenges of conformance testing. The challenges are multidimensional. Part of the problem is that OpenFlow features are currently incorporated into a single all-encompassing OpenFlow. This is in contrast to Internet functionality that is described in RFCs without altering the RFCs that define, for example, IPv4. Occasionally a new RFC will replace an old one, but many other RFCs do not affect earlier RFCs. It is very desirable that a new networking language and intermediate representation be defined in a super extensible way such that new features can be added without changing the base language spec. (Indeed, it is desirable that OpenFlow 1.x might be recast/refactored to achieve this also.) That is, new functionality can be defined in separate specs that leverage (potentially through the use of libraries) the base spec.

Once the language and functionality specifications are correctly decoupled, then products that conform to the language specification will remain conformant even after some new feature is defined. (These language-conformant products may not necessarily conform to the new feature specification, however.)

Presumably, one objective of OpenFlow had been to define a single specification to which all products would conform, in so doing creating a homogeneous product space. This objective might seem desirable, but it has turned out to be problematic. The OpenFlow specification has evolved, resulting in multiple versions. In practice, products have furthermore proven to not be homogeneous. Differences are even observed in software switches running on general purpose servers.

Noting this diversity, how shall conformance testing work? Let's imagine that at some point in the future we have a language like P4, and an intermediate representation IR, each with stable

specifications, and "generic" compilers to take P4 and generate IR. We also have libraries, potentially certified by related interest groups (perhaps blessed by the ONF, for example), and a variety of devices, with some device specific compilers. Some devices may be highly (but not infinitely) configurable, and others may mostly be fixed in configuration.

The generic compiler should be fully compliant with the P4 language and IR. A complete language test is conceivable for a generic compiler. But there may be different ways to translate a given P4 program into IR, and testing should allow for that. How will that be done?

Other products may be tested in concert. For example, a network device and the device-specific compiler would need to be tested together somehow. The device specific compilers may not be fully compliant with P4, but may be compliant in some constrained way, such as their ability to support known libraries, and also to generate appropriate error messages when unsupported structures are used. A library might include test cases (or "meta" test cases). The device and device-specific compiler should be testable in conformance to, for example, VXLAN or various IPv6 RFCs or 802.1q libraries.

Further discussion is required to ponder the issues associated with conformance testing that relate to different compilers, intermediate representation (IR) models, and programmed switches.

5.2 Deployment

5.2.1 Configuration Possibilities for Datapath Elements

Figure 4-2 below shows a typical packet forwarding path, annotated to show possibilities for configuration and/or programmability.

It gives examples of different datapath configuration times, which may be possible depending on the underlying target technology:

- … at installation;
- … at initialization;
- … during a pause in operation; and
- … during operation, between packets.

The technical focus of PIF is enabling more features to be configured during configuration and operation of equipment, rather than at the time of deployment of the equipment hardware.

A more dynamic approach raises the issue of ordering dependencies related to safety and soundness of semantics.

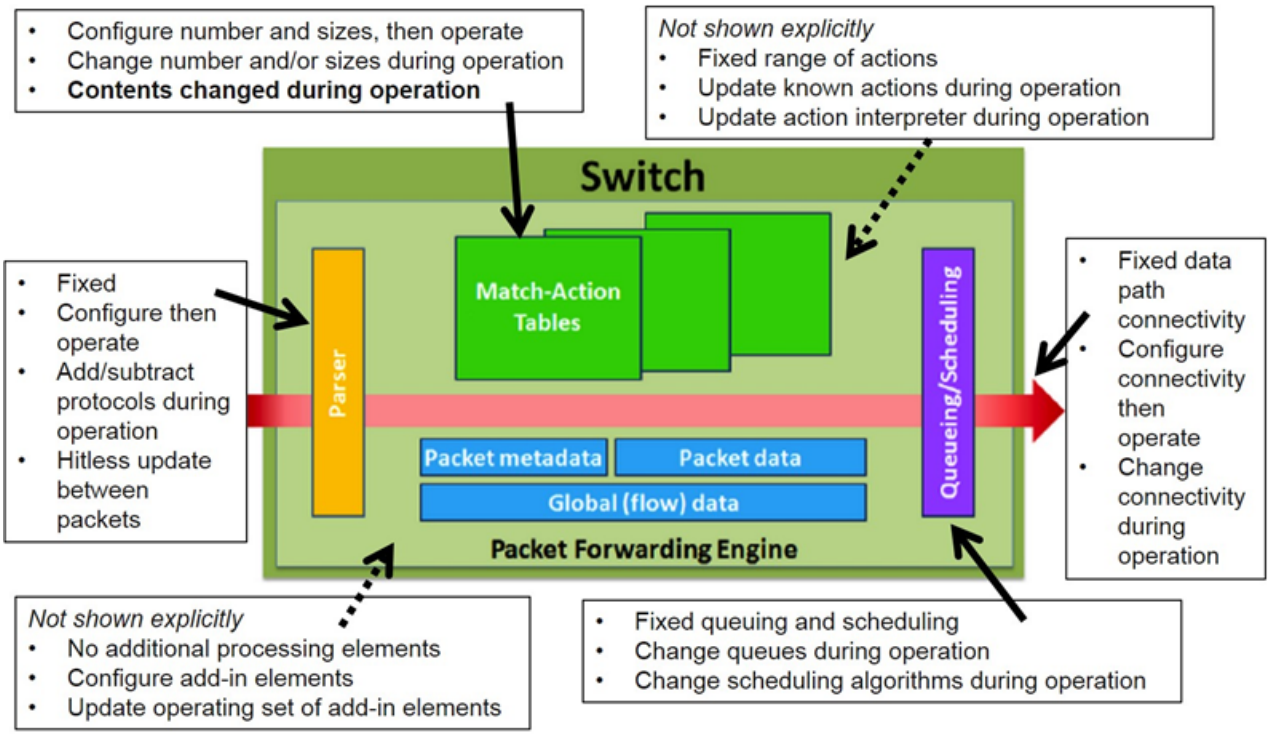

Figure 4-2: Opportunities for Datapath Configuration and Programming

5.2.2 Constraints

The following constraints need to be considered.

- Platform related technical constraints, including:
  - Technical constraints on datapath element configuration possibilities
  - Technical constraints on datapath configuration times
- Network and operational constraints, including:
  - Operational constraints on datapath element configuration possibilities
  - Operational constraints on datapath configuration times
- Business and adoption related constraints, including:
  - Business constraints on datapath element configuration possibilities
  - Business constraints on datapath configuration times

Example: An operator may develop PIF-based network in a greenfield environment, or deployment may involve some legacy, such as an OpenFlow-controlled, ASIC-based device "brownfield" environment.

The implications of these constraints need to be discussed and considered further. Addressing adoption constraints early will be helpful.

6 Standards Creation Process

6.1 Standards vs. (Open Source) Software

Expanding and improving the SDN ecosystem to include programmable dataplanes and more comprehensive support for SDN software development requires standards in key areas. These standards provide a common foundation and structure within which product development can proceed more effectively. In particular, products in different areas should be able to proceed relatively independently, allowing for concurrent innovation in many areas. For example, development of datapath models for optical transport networks and packet service networks can proceed in parallel. Development of new controller products and new dataplane products can proceed in parallel.

The standards development process in ONF initially focused on the OpenFlow protocol. An open and flexible wire protocol standard was needed to enable the first SDN products and applications to be developed. Going forward the ONF is beginning to focus more on the development of an open source software ecosystem that provides tools for more rapid development of SDN products. This shift toward producing open source implementations before writing standards documents is intended to accelerate the path to products and deployment. It is also aimed at developing deeper experience with the challenges of SDN development and only then converting this experience into standards.

6.2 Modular OpenFlow Specification

One aspect of the evolution of ONF standards is the restructuring of the OpenFlow protocol specification to make it modular. The base specification should be datapath protocol agnostic. That is, it should not define or refer (except in examples) to specific datapath traffic formats or technology specific actions. Instead, it should retain only the features that define a run-time control protocol capable of carrying information related to a wide variety of datapath technologies. Additional specification modules can contain technology specific details. This structure will stabilize the base protocol specification and allow independent development of OpenFlow support for additional datapath technologies, carried out by groups with specific interests.

6.3 Arriving at Standards for OpenFlow Next Generation and PIF

A specific area of interest in the evolution of the SDN ecosystem is support for programmable dataplanes. One project already underway under the auspices of OpenSourceSDN.org is Protocol Independent Forwarding: an open source Intermediate Representation (IR) for datapath programs. This project is aimed at developing an intermediate representation for datapath configuration that can serve as a common representation between various high-level datapath programming languages and backend compilers that target specific programmable dataplanes. A standard datapath IR will enable independent development of front-end and back-end compilers for translating datapath programs into loadable images for a variety of dataplanes.

The work on PIF is being done as an open source project to prove the IR works before committing to writing a standard. This approach also provides real tools for SDN software developers to use, albeit somewhat experimental in nature for a while.

The next generation of OpenFlow development may focus on an API (rather than just the wire protocol specification) and an open source implementation of an OpenFlow protocol driver. Moving the focus to a programming interface (API) will accelerate the development of SDN products using the OpenFlow run-time control model by providing a convenient API and associated software to handle the mundane details of marshaling the wire protocol.

Similarly, the evolution the OpenFlow TTP language is expected to happen in the open source space. The next version of the TTP language will first be developed as a formal schema that can be used directly by authoring and validation tools. An English language specification will follow when the schema is deemed stable and useful.

Finally, Application Area TTPs will be written for selected areas of interest and offered with the Flow Objectives API as a run-time control interface for controller subsystems to use. Switch vendors will be invited to implement mapping drivers for their OpenFlow switches that support these Application Area TTPs. Controller and switch vendors will be invited to validate and demonstrate interoperability at ONF AppFests. If an Application Area TTP gains sufficient interest and support the ONF will develop a conformance test suite for that application area based on the TTP and SDN test laboratories will be able to offer ONF certification for the application area.

As these open source products mature and become stable they will be used as the basis for writing standards documents if these are needed.

6.4 OF1.x to PIF Interworking and Transition

The expectation is that OpenFlow 1.x will co-exist and need to interwork with flexibly programmed datapaths, i.e. PIF datapaths. It is understood that not all capabilities will be available on all datapaths (i.e. switches), for example until and unless OpenFlow 1.x is expanded to include support for protocol independence, the use of dynamically defined protocols will only be possible on PIF datapaths.

It is expected that functionality available on traditional OpenFlow SDN switches and PIF SDN switches will need to be accessible from SDN controllers that are shared among these. This functionality may not necessarily be exposed via the same protocol or variant of the southbound interface (i.e. API or protocol between the controller and the switch), for example, communication with OpenFlow 1.x switches would probably use the OpenFlow protocol, while communication with PIF switches may be effected using a callable API with an off-the-shelf remote procedure call mechanism (e.g. Apache Thrift or Google Protocol Buffers) being employed to make it accessible over the network.

TTPs currently enable the required and available subset of OpenFlow 1.x to be encoded. For flexible datapaths, an analog to a TTP could be developed, to enable the interface and overall model representing the PIF program (inherently, and once capacity data is included, as deployed on a switch) to be similarly encoded. Controllers and applications on them could then use these

TTPs as models to describe the available capabilities, irrespective of whether the capabilities are actually implemented using traditional OpenFlow 1.x capable datapaths or PIF capable flexible datapaths. This needs to be considered when driving the evolution of TTP related standards (and, in general, Negotiable Datapath Model standards).

7 Conclusions

This document is intended to provoke discussion and record the results of the ensuing debate. This version of the document represents a snapshot of work in progress.

8 Acknowledgements

The following individuals contributed to this document: André Béliveau, Ben Mack-Crane, Curt Beckmann, Dan Malek, Edwin Peer, Gordon Brebner, Haoyu Song, Jasson Casey, Johann Tönsing, Michael Orr.

9 Revision History

Date	Revision	Description	Editor
2016-09-08	1.0	Initial version for publication	J H Tönsing

10 Appendix: Use Cases

10.1 Custom Tunnels

10.1.1 Scenario

A datacenter operator wants to define a custom tunnel format, e.g. Ethernet in a custom protocol in UDP in Ethernet, and program equipment to support this.

10.1.2 Actors / Roles

Developer (of software and of table entries to express custom tunnel), controller, switch, network nodes (sending/receiving traffic).

10.1.3 Behavior / Lifecycle

Developer produces “code / table entries” => compilation (if needed) => configuring switches => downloading flow entries => enable dataplane traffic to start (across network).

10.1.4 Gaps / Issues

- OpenFlow 1.5 lacks support for custom header fields + nesting headers
  - PIF IR based system could support this (once implemented)
- Need run-time protocol (southbound protocol) with support for field nesting and new header fields (OpenFlow 1.x derived vs. new)
  - Represent tunnel as logical port (opaque) vs. match / action structure (explicit)
  - Need library defining well-known header fields e.g. Ethernet, IP, UDP, with ability to include and nest them (include more than one instance of Ethernet) => could use templating to enable composing program
- Switch and controller need to support PIF and e.g. required packet processing depth (TBD where to run front-end / back-end compiler)
- Issue: deploy new / updated tunnel format
  - Easier if outer protocol is already supported (Ethernet-IP-UDP) - can then deploy customization at tunnel origination / termination point
  - Update all network elements before starting traffic
  - To upgrade tunnel format, either atomically upgrade network (version numbers in protocol detect errors), or let old and new tunnel formats coexist (introduce support for new version while retaining old version => encode which network node supports which version)

10.1.5 PIF Impact

- Reminder: we are assuming that the custom tunnel is layered on UDP / IPv4
- Q: How is IPv4 / UDP support represented in a TTP?
  - By referencing a well-known library name, possibly also referencing field names defined by it? (This assumes a common, standardized library model.)
  - By referencing IETF RFCs?

    - Both? One could, for example, invoke the RFC for the protocol overall, but also include references to UDP field names for layering.
    - => Standardized libraries should not be hard to write. Libraries can refer to RFCs. This avoids the need for different coordination mechanisms. However we need everyone including e.g. controllers to then understand at least the interfaces to such libraries.
    - Interim step: move protocol fields currently in a single monolithic OpenFlow 1.x specification to separate documents (i.e. create a modular specification). This can then be used by predefined protocol implementations (referring to a human readable description of protocol fields in the specification text).
    - Eventually one would have developed corresponding standard libraries for PIF (with machine readable description of protocol fields).
- Q: How are custom tunnel fields represented in a TTP?
    - By referencing a well-known library name and/or fields in libraries? (Easier, but need agreement on naming.)
        - This is the preferred approach.
    - Use a canonical representation that contains the full path leading to a field as well as field widths etc.? (Harder – is this needed though?)
- Q: How are custom tunnel actions (push/pop/set etc.) represented in a TTP?
    - By referencing a well-known library name and/or action "subroutine" names in libraries? (Easier, but need agreement on naming in interfaces.)
    - By invoking a canonical representation of the entire implementation? (Very difficult, without repeating implementation and being implementation dependent.)
- Q: Representing tables referring to well defined and custom defined fields / actions
    - Similar to existing TTP, with new expressiveness (overall control flow including conditions) represented somehow (focusing on interface again)?
    - Derive from expressiveness in P4 language / PIF IR etc.?

10.2 External Datapath Function

10.2.1 Scenario

A "black box" (or C program) with embedded state / unspecified behavior is linked into pipeline somehow (as a custom action or a virtual port etc.), and processes packets or events.

Examples: OAM handler, embedded L4-L7 service like function, ICMP protocol handler.

10.2.2 Actors / Roles

Developer (of software and of table entries to feed traffic to software), controller, switch, network nodes (sending/receiving traffic).

10.2.3 Behavior / Lifecycle

Develop external function code => compile => distribute to switches => start (if separate process) or otherwise ensure ready for use.

Developer produces “code / table entries” feeding traffic to external function => compile => configure switches => install / start software => download flow entries => enable dataplane traffic to start (across network).

10.2.4 Gaps / Issues

- OpenFlow 1.5 lacks mechanism to send traffic to an external function running on switch (but can send to/from controller via packet in/out).
  - Various approaches have been proposed e.g. custom action (packet processor / task etc.), logical port representing software.
  - Interim: represent software as logical port tunnel (service chaining to software running locally) => low performance, no metadata yet, limited interaction with switch.
  - Interim: represent software as additional controller (use packet in / out) => terrible performance, some metadata, permits interaction with switch.
- Supporting external functions (actions execution blocks etc.) is being discussed in PIF IR project.
- For both - need standardized callable API to:
  - Hand traffic to / from software;
  - Enable software to interact with network (originate other traffic), flow / meter / group tables, QoS configuration etc.;
  - Permit software portability across operating systems (initially assume Linux or POSIX - covers most prevalent case).

10.2.5 PIF Impact

- Q: How can a TTP encode whether or not this is supported in principle?
  - Use well-known names - e.g. URL / DNS name based - to create a distributed “registry”?
  - The controller itself can be aware that invoking such a name will turn on certain behavior, or the controller can simply pass through the request to invoke it from a higher level app / OSS etc.
- Q: How does the run-time protocol interact with it?
  - General RPC?
  - Specific messages to set/get parameters, entries in data structures e.g. tables?
- Q: How does TTP encode run time protocol supported sub-features?
  - Well-known sub-names for behavior variations?
  - => Referring to entities (at interface level) is tractable, whereas machines comparing behaviors is not tractable.

10.3 Network Data Analytics

10.3.1 Scenario

Having the capability of defining new forwarding protocols is an important feature of the next generation of open programmable data plane. Another equally important consideration is how the data plane can be programmed to enhance its visibility to control plane and management plane. Network data analytics has been proved an invaluable weapon for network monitoring and fault diagnosis, performance optimization, and security. But it must rely on the data plane to provide relevant and enough data in a timely fashion. This not only requires the dynamic reprogramming and reconfiguration capabilities to the next generation data plane, but also needs more programmable features that are not generally available in the current generation of data plane.

10.3.2 Actors/Roles

Network operator, controller, network nodes.

10.3.3 Behavior/Lifecycle

Operator (or application automatically) generates new data path configurations => compile => re-configure network devices => application starts to consume data and events injected from network devices.

10.3.4 Gaps/Issues

First, data path partial reprogramming and reconfiguration should be able to be triggered by CLI/GUI or direct API called embedded in runtime applications. Due to the limited resource in data plane devices and the volatile application requirements, it is impossible to program the data path for just one time and satisfy all future data analytics requirements. Frequent changes may be needed and the changes may have strict time constraint.

Second, the match-action table forwarding abstraction makes the flow filtering and aggregation relatively easy tasks, but to meet the data analytics and network optimization requirements, some other features are required as follows:

- The data path needs to be able to be programmed to do some custom computing and algorithm implementations.
- Data path needs to provide programmable timers and be able to generate time-based events.
- Data path needs to be able to timestamp packets and events. The network wide synchronization mechanism needs to be defined.
- Packet sampling and digest generation algorithms need to be defined.

10.3.5 PIF Impact

This use case strongly requires the near real-time and dynamic reconfiguration and reprogramming capability. It also involves important features that were neglected before.

