SPARC

ICT-258457

Deliverable 4.2

Description of OpenFlow protocol suite extensions

Editor:	*András Kern, Ericsson Hungary (ETH)*
Deliverable nature:	Report (R)
Dissemination level: (Confidentiality)	Public (PU)
Contractual delivery date:	M18
Actual delivery date:	M19
Version:	1
Total number of pages:	38
Keywords:	OpenFlow 1.0, OpenFlow 1.1, OpenFlow Protocol Extensions, Virtualization, BFD, Pseudo wire, Energy-Efficient Networking, Virtual Port Management

Abstract

This deliverable specifies OpenFlow protocol extensions for implementation of configuration support for carrier-grade functionalities inline with the design choices done in the SPARC Architecture Work Package (WP3). First, it investigates the different interfaces in the context of the SPARC split architecture proposal reported in Deliverable D3.2 and discusses the protocols considered there. It also summarizes the key missing OpenFlow features and the proposed functional extension discussed in D3.2. Then the deliverable specifies protocol extensions for several functions: *virtual port configuration*, *flowspace registration*, *configuration of network virtualization*, *BFD based monitoring and protection*, *pseudo wire (PWE)* support, and support for *energy-efficient networking*. After the detailed specification, it summarizes the implementation status of the proposed extensions as well as it reports the level of integration into the developed prototypes.

Disclaimer

Imprint

[Project title]	***Split Architecture for carrier grade networks***
[short title]	***SPARC***
[Number and title of work package]	***WP4 – Prototyping***
[Document title]	***Description of OpenFlow protocol suite extensions***
[Editor]	***András Kern (ETH)***
[Work package leader]	***Attila Takács (ETH)***
[Task leader]	***András Kern (ETH)***

Copyright notice

Executive summary

This document summarizes the OpenFlow protocol extensions that are specified as part of designing functional extensions required to implement OpenFlow based large-scale wide-area networks. As targeted network scenario, the access/aggregation use case of D2.1 has been selected and the details of the proposed control plane architecture have been described in D3.2. This latter document also proposes functional extensions, which are essential for implementing carrier-grade networks.

Following the architecture design of D3.2, two major open interfaces between the controlling elements have been indentified based on the relation of the controlling elements: horizontal (peering relation) and vertical (client-server relation). According to the architecture design of D3.2, open protocols are relevant at the horizontal interface when an OpenFlow based controller interoperates with a legacy control plane.

D3.2 provides functional extensions to the OpenFlow protocol. Since some of the extensions were immature at the time of writing this deliverable, updates to the OpenFlow protocol referring to such extensions cannot be described here. The considered extensions described in the present deliverable target following topics: *Openness and Extensibility, Virtualization and Isolation, Technology-specific MPLS OAM, Service Creation* and *Energy-Efficient Networking*. This deliverable specifies OpenFlow protocol extensions supporting these areas.

Finally, this document reports the implementation status of the specified extensions as well as their integration to the SPARC prototypes.

List of authors

Organisation/Company	Author
ACREO	**Pontus Sköldström, Viktor Nordell**
EICT	**Andreas Köpsel, Tobias Jungel**
ETH	**András Kern, Dávid Jocha**
IBBT	**Dimitri Staessens, Sachin Sharma**
EAB	**Wolfgang John**

Table of Contents

List of tables

List of figures

Abbreviations

ALR	Adaptive Link Rate
BFD	Bidirectional Forwarding Detection
BM	Burst Mode
CLI	Command Line interface
G-ACh	Generalized Associated Channel
GAL	Generalized Associated Label
GMPLS	Generalized Multiprotocol Label Switching
ISIS-TE	Intermediate System to Intermediate System with Traffic Engineering
MEP-ID	Maintenance Endpoint Identifier
MLTE	Multilayer Traffic Engineering
MP-BGP	Multiprotocol Border Gateway Protocol
MPLS	Multiprotocol Label Switching
MPLS-TP	Multiprotocol Label Switching Transport Profile
OAM	Operation Administration and Maintenance
OFP1.0	OpenFlow Protocol version 1.0
OFP1.1	OpenFlow Protocol version 1.1
OSPF-TE	Open Shortest Path First with Traffic Engineering
PBB-TE	Provider Backbone Bridging - Traffic Engineering
PPP	Point-to-Point Protocol
PPPoE	Point-to-Point Protocol over Ethernet
PWE	Pseudo wire Emulation
RGW	Residential Gateway
SCP	Service Creation Point
SSL	Secure Sub-layer
VN	Virtual Network

1 Introduction

1.1 Project Context

The project SPARC "Split architecture for carrier-grade networks" aims towards implementing a new split in the architecture of Internet components. In order to better support network design and operation in large-scale networks millions of customers, high automation and high reliability, the project will investigate splitting the traditionally monolithic IP router architecture into separable forwarding and control elements. The project will implement a prototype of this architecture based on the OpenFlow concept and demonstrate the functionality at selected international events with high industry awareness, e.g., the MPLS Congress.

The project, if successful, will open the field for new business opportunities by lowering the entry barriers present in current components. It will build on OpenFlow and MPLS technology as starting points, investigating if and how the combination of the two can be extended and study how to integrate IP capabilities into operator networks emerging from the data center with simpler and standardized technologies.

1.2 Relation with Other Work Packages

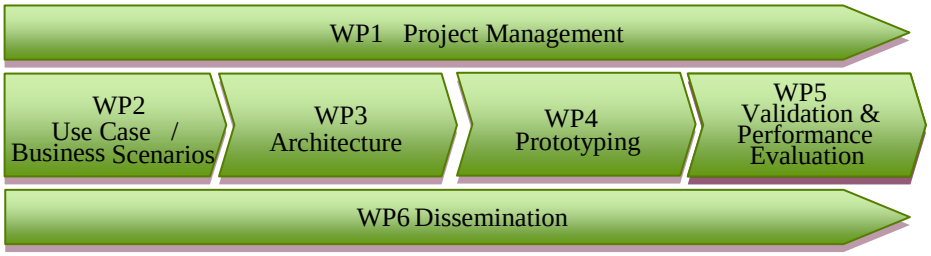

Figure 1: Relation of SPARC work packages

In the "workflow" of the dedicated work packages of SPARC, WP2 sets the stage for the other technical work packages, namely WP3 (Architecture), WP4 (Prototyping) and WP5 (Validation & Performance Evaluation). WP2 defines use cases and requirements. Based on those use cases and additional generic requirements, an initial study for a Split Architecture in relation to an enhanced version of OpenFlow is done by WP3. In addition, this architecture is evaluated against certain architectural trade-offs. WP4 implements a selected sub-set of the resulting architecture, whose feasibility is validated in WP5. WP6 disseminates the result at international conferences and publications. The schematic of the workflow is shown in Figure 1.

1.3 Scope of Deliverable

Both the design and prototyping activities explored that OpenFlow lacks such features, which are essential for implementing OpenFlow based carrier-grade split architectures. To implement such advanced features in OpenFlow context, architectural, functional and protocol design decisions had to be made. This means strong connection between activities run in the architecture and the prototyping work packages. The Architecture work package (WP3) focused on identifying the missing features and on proposing functional extensions. The design work gave a comprehensive set of functional extensions to OpenFlow switch forwarding model.

This deliverable specifies the OpenFlow protocol extensions that are essential to implement the functional extensions documented in Deliverable D3.2. It describes the updates of the configuration procedures and the new or altered OpenFlow messages and structures. The content of this deliverable may serve as a basis of possible OpenFlow standardization contributions.

2 Review of Protocols Associated to Split Architecture Interfaces

Preceding work in this project focused on sketching carrier-grade split architecture through describing the overall control solution as a set of cooperating control entities (see Figure 2). Each control entity is responsible for some parts and/or some aspects of the configuration to be done, and the full configuration is obtained through the joint operation of these entities. Therefore, it is essential for these entities to communicate with each other, for instance, to exchange identifiers, state information and configuration instructions. This means that well-specified interfaces between the control entities must be deployed. Further issues to be solved are the flexibility, extensibility and openness of the overall architecture [3]. To cope with these issues it is crucial to make these interfaces open.

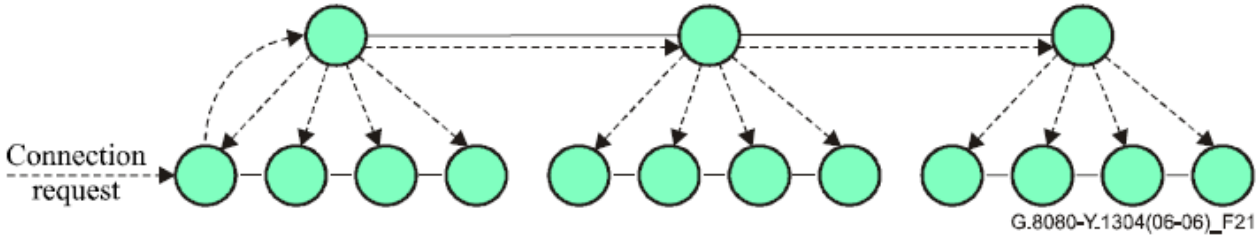

Figure 2: Implementing the control plane as a federation of peering and serving control entities [1].

Based on their roles two major control interfaces have been identified in [3]: *vertical interface* between entities in client-server relation and a *horizontal interface* between control entities with similar functionality. Besides, a second vertical interface is defined between the control plane and the forwarding one. To add flexibility to the control architecture the two vertical interfaces, namely the control-to-forwarding and the control-to-control interfaces, should be defined the same way [3]. In other word, the same protocols and functions should be provided at these interfaces.

2.1 Vertical Interface

This interface expresses a client-server relation of the control entities of the adjacent layers, where the client (upper) layer sends configuration instructions to the server (lower) layer and where the server layer notifies the client one on the updates of its internal state. Note, that the server layer has no information on the state of the client layer.

Based on the state-of-the-art (D3.1 [2]) and the targeted control architecture (D3.2 [3]), OpenFlow [4] is considered as the sole control plane protocol running on this interface, although management plane protocols may also run in parallel. However, OpenFlow was originally designed to configure forwarding aspects of a switch only; therefore, it lacks key features, which are essential for implementing carrier grade split architecture or support advanced configuration between adjacent control layers. Along with the architecture design, model extensions have been described in [3], but the detailed protocol specification extensions to OpenFlow will take place in this deliverable.

2.2 Horizontal interface

Based on the types of the control entities between which the horizontal interface is defined, two use cases can be differentiated: (1) both adjacent control entities implement OpenFlow based Split Architecture, and (2) a split controller interoperates with a legacy control plane solution. Regardless which of the two previous cases is considered, the interoperating parties must agree on a common information model as well as the set of protocols to be used, since different control entities may use different internal representations of the managed domain. The only difference is that in the first case it is possible that the two control entities agree on the model and protocols to be used, but in the latter case there is no room for such choices: the type of the legacy control plane determines what protocols and models must be used.

In the first case the interoperating parties are OpenFlow controllers that must agree on the used models and protocols. Several examples on such models and protocol are already known. The ONIX controller uses the Network Information Base (NIB), where a state distribution mechanism, e.g. a distributed hash table or a central database, can provide methods for sharing state information [6]. Another solution for NOX

controllers is reported in [7], which extends the NOX's event handling mechanisms for multiple NOX instances. Since the SPARC split architecture proposed in Section 2 of D3.2 considered such horizontal interface to be implementation specific, no open protocols are required at this interface. During prototyping we have not considered splitting the control function between multiple interoperating control entities either. Therefore, the investigation of such cases is excluded.

According to [2] and [3] the assumed legacy control planes are either IP/MPLS or GMPLS. In the SPARC prototype, only IP/MPLS is considered [5]. Both IP/MPLS and GMPLS are distributed control planes formed by a set of protocols: OSPF-TE or ISIS-TE for synchronizing identifiers and TE attributes, LDP and RSVP-TE to manage connectivity and last but not least MP-BGP for configuring connectivity over multiple domains. As a result, any updates to the protocols of the interface between the two domains will necessitate updates of the code base at all protocol speakers, otherwise full interoperation cannot be guaranteed. Most router implementations issue closed software code that prevents the inventor of a new feature adding it to those implementations. This practically disables the easy spread of a novel feature even if the inventor were able to extend the code bases. Instead, the proposed extensions must be included into the relevant specification through standardization processes, even though the desired feature would have been used only locally.

A possible alternative, which was actually followed during SPARC prototyping as documented in D4.1 [5], is to introduce an intermediate translator element or proxy. It is responsible for mapping the internal information models of the controllers to the one used by the considered legacy control plane. This element also translates the events and triggers between the two elements. As a consequence, the need of making any updates to the legacy protocol stacks has been practically eliminated.

During SPARC prototyping work, we exploited the mentioned advantages of using protocol proxies for interoperating with an IP/MPLS control plane. So far, we have not indentified any limitations of IP/MPLS that would impair the interoperation between the OpenFlow based split domains and the distributed IP/MPLS controlled domains. Due to the introduced proxy entities, no extensions to the assumed horizontal protocols have been considered in the current prototype and consequently no extensions of such horizontal interface protocols are reported in this deliverable.

3 OpenFlow Protocol Extensions

In D3.2, four stages of how OpenFlow can be integrated into carrier grade-networks have been specified:

1. *Basic emulation of transport service*: In this stage an OpenFlow data and control plane emulates and replaces legacy transport technologies (e.g., Ethernet, MPLS).
2. *Enhanced emulation of transport services:* OpenFlow is again used to provide transport services. However, a number of features and functions are added to both the data and control plane in order to comply with carrier-grade requirements such as OAM and resiliency aspects.
3. *Service node virtualization*: In this stage, besides transport services, OpenFlow also takes control of (distributed) service node functionalities, including service creation, authentication and authorization.
4. *All-OpenFlow-network*: In this integration stage OpenFlow also controls other network domains, e.g., customer premises equipment such as RGWs and the operator's core domain.

Considering that currently published OpenFlow 1.0 and OpenFlow 1.1 are sufficient to provide stage 1, the prototyping activities placed focus on integration step 2, i.e., studying possible extensions to OpenFlow in order to fulfill carrier-grade requirements. That work concluded several key features that are missing from the current OpenFlow specifications, which are summarized at Table 1.

Study topic	Level of the proposed improvements in D3.2	OpenFlow extensions studied in D4.2
Openness and Extensibility	Detailed	Yes
Virtualization and Isolation	Detailed	Yes
OAM: technology-specific MPLS OAM	Detailed (BFD)	Yes
OAM: technology-agnostic Flow OAM	Conceptual	No
Resiliency	Conceptual	No
Topology Discovery	Conceptual	No
Service Creation	Detailed (PWE, PPP)	Yes
Energy-Efficient Networking	Conceptual	Yes
QoS	Conceptual	No
Multilayer Aspects	Conceptual	No

Table 1: Summary of OpenFlow extensions studied in D3.2 [3].

In the above table the level of detail of the resulting study in D3.2 is also shown. Some topics have been discussed at the high level resulting in concepts and guidelines, but all details have not been described yet. Other topics have been thoughtfully studied and some parts of these topics have been prototyped. In such cases OpenFlow protocol extensions have been already provided.

Openness and Extensibility (section 3.1 of D3.2): Extended and more advanced processing functionalities at the data plane are desirable in many situations. In order to avoid polluting the main OpenFlow protocol with the configuration of specific features we discussed the concept of virtual ports as a means of hiding the details of advanced processing functionalities. Beside virtual ports, similar processing entities are considered, which allow keeping state on datapath elements. A processing entity in this sense is a generalized OpenFlow action that encapsulates some processing logic. Nevertheless at this state only the virtual port extensions have been specified.

Virtualization (section 3.2 of D3.2): A crucial feature of future carrier-grade networks is network virtualization, enabling multi-service and multi-operator scenarios on a single set of physical network infrastructures. D3.2 describes a limited model on virtualization implemented in OpenFlow switches.

OAM: As an example of a *technology-specific OAM* (section 3.3.3 of D3.2), we propose the configuration of MPLS-TP BFD in OpenFlow through utilizing the extended processing functionalities in the datapath elements by means of virtual ports.

Service Creation (section 3.6 of D3.2): In our context, we define service creation as the configuration process of network functions at service creation points (SCP). We discussed integration of residential customer services in the form of PPP and business customer services in the form of pseudo-wires (PWE) in an OpenFlow-controlled operator network. The two extensions have been implemented in different ways. The PPP processing has been encapsulated in separate processing entities outside of OpenFlow pipeline and therefore no OpenFlow protocol extensions related to PPP has been needed. The PWE implementation followed the other options, where the related procedures are implemented as generalized OpenFlow actions carrying state information. The configuration of these stateful OpenFlow actions has been provided in this deliverable.

Energy-Efficient Networking (section 3.7 of D3.2): Centralized control software like OpenFlow offers additional option for reducing network energy consumption. We discussed possible energy saving approaches (e.g., network topology optimization, burst mode operation, adaptive link rate) and the requirements they place on OpenFlow.

3.1 Common SPARC vendor extension messages

Since its early versions, the OpenFlow protocol defines a simple mechanism for implementing not standardized protocol messages. For this purpose the protocol specification allocates a common message type, the OFPT_VENDOR, and attaches a vendor identifier to the OpenFlow message header. The resulting vendor extension header format of OpenFlow 1.0 is shown in Figure 3. Any vendor specific messages must include this updated header. For OpenFlow 1.1, a similar extension mechanism under different name (experimenter) is also defined.

```
struct ofp_vendor_header {
        struct ofp_header header; // type == OFPT_VENDOR
        uint32_t vendor; // vendor identifier
};
```

Figure 3: OpenFlow Protocol 1.0 (OFP1.0) vendor extension header format.

In the SPARC project, a vendor identifier with value `0x551bcccc` has been selected for prototyping purposes. Note that this extension type may be registered with the OpenFlow consortium for permanent assignment in the case when the project would publish any of the below defined extensions. However, the above header just indicates that the message is SPARC specific. Therefore, a common SPARC vendor extension header has been defined including the common OFP1.0 vendor header:

```
struct ofp_vendor_ext_sparc {
        struct ofp_vendor_header header; // common vendor ext. header
        uint32_t exttype; // SPARC vendor extension type
};
```

Figure 4: Structure of SPARC vendor extension messages.

The SPARC vendor extension message types have been defined for the purpose of flow space registration, and virtual port management. These extension messages are identified through message types whose values form a continuous domain started with index 3221225473 (0xC0000001).

```
enum ofp_vendor_ext_sparc_type {
        OFPSET_FSP_OPEN_REQUEST = ((0xC000 << 16) | (0x0001),
        OFPSET_FSP_OPEN_REPLY,
        OFPSET_FSP_CLOSE_REQUEST,
        OFPSET_FSP_CLOSE_REPLY,
        OFPSET_FSP_IOCTL_REQUEST,
        OFPSET_FSP_IOCTL_REPLY,
        OPFSET_VPORT_OPEN,
        OFPSET_VPORT_CLOSE,
        OFPSET_VPORT_IOCTL,
        OFPSET_VPORT_ATTACH,
        OFPSET_VPORT_DETACH,
};
```

Figure 5: Core SPARC vendor extension message types.

All SPARC vendor extension messages are symmetric in nature, as the basic vendor extension mechanism of OpenFlow also assumes a symmetric packet exchange. A datapath element must send an appropriate reply upon reception of a request message.

3.2 Dynamic Configuration of Virtual Ports

3.2.1 Overview

According to the OpenFlow specifications a port represents a construct where packets enter and exit the OpenFlow processing pipeline (or forwarding engine). One can differentiate physical and virtual ports. The packets enter or exit the switch through physical ports, while virtual ports are interfaces between a datapath element's OpenFlow managed forwarding engine and additional processing or filtering instances. A specific type of virtual ports is called reserved ports: such ports denote specific behavior defined by the OpenFlow specification. A virtual port is a bidirectional element with two input/output port pairs at each side. The left side is connected to the forwarding engine. The right side is freely configurable and may be connected to other functional elements of the node (for instance to a processing instance, a physical port or another virtual port).

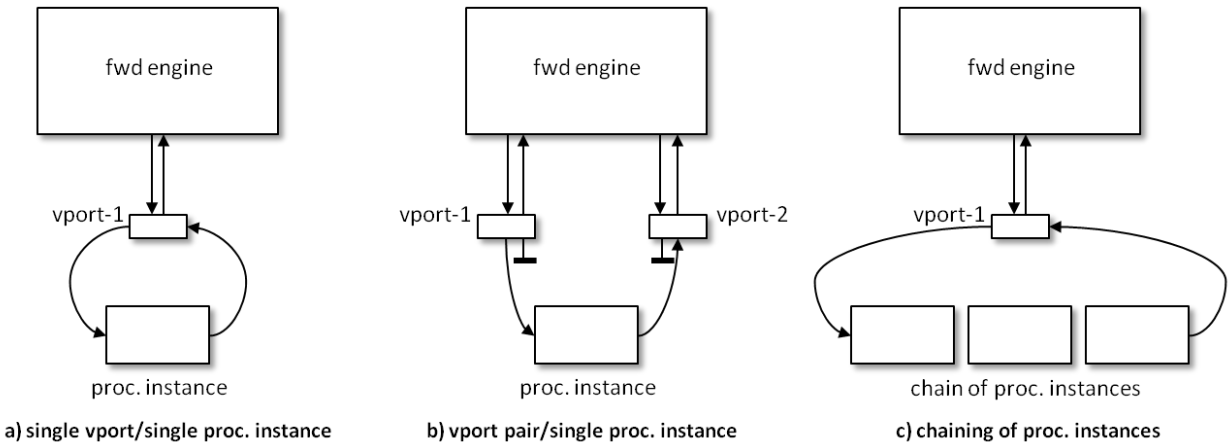

Figure 6: Virtual port configuration options

Figure 6 depicts several example setups on connecting the OpenFlow controlled forwarding engine to processing instances. All examples assume bidirectional virtual ports and one or more processing instances. Each processing instance acts as a simple filter entity and defines a single input port and a single output port Input and output port may be arbitrarily connected to either a single virtual port or a pair of virtual ports (see as an example Figure 6b). A chaining of processing instances defines the final option (see Figure 6c).

The dynamic virtual port configuration protocol extensions comprise two message categories:

- Messages for the dynamic creation, deletion, and reconfiguration of virtual ports
- Messages for attaching virtual ports to other physical/virtual port or processing instances

A virtual port does not host processing functionality on its own, i.e. it must not change a packet's header or payload. A virtual port that has no processing unit attached drops any packet received from the forwarding engine.

The PPPoE/PPP termination prototype is an example using a single virtual port and a single processing instance according to Figure 6/a. A controller detecting a new PPPoE connection is able to set up a new virtual port connected to a PPPoE/PPP processing instance in the datapath element. In order that PPPoE/PPP can be processed, the controller has to install a flow-mod matching the users PPPoE connection after the new created virtual port shows up in the controller via a port status message. The implementation foresees to handle user data (IPv4 datagram in PPP) in the datapath. All other contents of the PPP connection (e.g. link control related data like LCP, IPCP) are currently processed in the controller.

3.2.2 Virtual Port extension messages

This group of messages controls basic creation, deletion, and updating of virtual ports:

- OFPSET_VPORT_OPEN
- OFPSET_VPORT_CLOSE
- OFPSET_VPORT_IOCTL

This group of messages controls the attachment of a virtual port's right side to another virtual port or processing instance:

- OFPSET_VPORT_ATTACH
- OFPSET_VPORT_DETACH

All virtual port related vendor extension messages adopt the following format:

```
struct ofp_vendor_ext_sparc_vport {
        struct ofp_vendor_ext_sparc header; // common headers
        uint64_t controller_id; // controller ID that generated this vport
        uint32_t result; // enum ofp_vendor_ext_sparc_vport_result_type
        uint16_t portno; // 0 = dynamic assignment of port number
        uint32_t config; // initial port configuration
        uint64_t procid; // identifier of processing instance to attach to
};
```

Figure 7: SPARC virtual port provisioning vendor extension structure.

All request message types (open, close, ioctl) set the result field to 0. The index of the virtual port is defined by field "`portno`" and selects value from the port identifier domain, which also comprises physical and protocol reserved port identifiers. The "`portno`" field may be set to 0 which is an illegal port number in OFP1.0 indicating dynamic assignment of a port number by the datapath element. Any non-null value indicates a fixed port number and must be assigned to the virtual port by the datapath element. If this port number has been assigned already, the datapath element must send an error message indication to the controller entity. The processing identifier defines an attached processing instance or may be set to 0 indicating that this port is not attached to a processing instance. Last but not least, the `controller_id` gives a reference to the configuring controller entity that can be used by the processing element associated to the virtual port, for instance supporting the synchronization of state machines run in the data plane and in the controllers.

The processing identifier namespace comprises all port instances (physical and virtual ones) and all processing instances. The port number namespace is mapped into the processing identifier namespace by setting all higher bytes to null. In OFP1.0 the lowest 2 bytes are used for the port number namespace, in OFP1.1 the lowest 4 bytes.

The following result codes have been defined:

```
enum ofp_vendor_ext_sparc_vport_result_type {
        OFPSET_VPORT_RESULT_OK,  // request was accepted, vport has been created
with specified config
        OFPSET_VPORT_RESULT_NAK,  // request was rejected
        OFPSET_VPORT_RESULT_NOT_FOUND,  //  close request failed, vport not found
};
```

Figure 8: SPARC flowspace vendor extension result types

3.3 Flowspace registration

3.3.1 Overview

Contrary to the 1.0 and 1.1 versions of the OpenFlow specification, the OpenFlow extension proposed by SPARC allows several controller entities to control a single datapath element. A controller may register for controlling a single or multiple flowspaces (or parts thereof).

A data path element maintains a table of flowspace entries. Each flowspace entry maps specific parts of the overall flowspace to individual controllers. It consists of a struct `ofp_match` depicting the flowspace the controller claims responsibility for and a data structure pointing to the controller device and its associated control channel.

A controller may register an arbitrary number of flowspaces at a data path element in order to control non-contagious flowspaces. The number of allowed flowspace entries per data path is implementation dependent and may be limited. A strategy for limiting the number of flowspace entries per controller device is out of scope of this extended specification and may be defined by the implementer.

Flowspace registrations of different controller entities may overlap: the datapath element compares a *Packet-In* event against all flowspace registration entries and its `ofp_match` structures. The matching process is identical to the process adopted by the datapath element's forwarding engine for determining a FlowMod entry for an incoming packet: the datapath calculates the number of exact and wildcard hits of each flowspace entry for the *Packet-In* event. In case of a missed field, the flowspace entry is ignored and the search continues on the next flowspace entry. The flowspace entry with the highest number of exact hits wins the matching comparison; if multiple entries with the same number of exact hits exist, the flowspace entry with the higher number of wildcard hits is selected. If both exact and wildcard hits are identical, the flowspace entry priority field is used to determine the appropriate controller entity. If multiple flowspace entries match also with the priority fields, all controllers, which registered with the matching flowspace entries, will receive the *Packet-In* event in parallel.

A datapath element must verify all *Packet-Out* and *Flow-Mod* messages against the registered flowspace entries, i.e. a controller must have registered for the specific flowspace before it can actually generate or handle packets that are part of the defined flowspace. The matching process is again identical to the one used for *Packet-In* messages. For *Packet-Out* messages, the buffered packet (or the packet appended to the *Packet-Out* message) is matched against all flowspace entries. The winning flowspace entry must point to the *Packet-Out/Flow-Mod* message emitting controller entity; otherwise the message is dropped and an error notification is sent to the generating controller entity.

For each Flow-Mod entry created the datapath element stores the controller entity that created this entry. Upon expiration of a Flow-Mod entry, the corresponding *Flow-Removed* message is sent to the generating controller entity (when requested according to the flags in the Flow-Mod entry). In case of an aborted or closed controller connection, all Flow-Mod entries set by this controller entity may be removed from the datapath element's flow table or may be kept. The preferred behavior is the removal of the flows.

All configuration and device management related messages (Port-Status, Features-Request/Reply, etc.) are duplicated and sent to all control entities, so that all controllers share the same data path model. A datapath may restrict the number of Flow-Mod entries per controlling entity, thus resulting in Features-Reply messages with an e.g. reduced number of flow table entries. Such behavior is out of scope of this extended specification.

3.3.2 SPARC flowspace vendor extension messages

All SPARC flowspace related vendor extension messages adopt the following format:

```
struct ofp_vendor_ext_sparc_fsp {
        struct ofp_vendor_ext_sparc header; // common headers
        uint64_t controller_id;
        uint32_t result; // = enum ofp_vendor_ext_sparc_fsp_result_type
        struct ofp_match match;
};
```

Figure 9: SPARC flowspace manipulating vendor extension structure

The "`controller_id`" provides a reference to the controller requesting the flow space. The match field must contain the flow space as requested by the controller. In request message, the result field must be set to null by the controller must be ignored by the switch. Reply messages must contain a result value according to `ofp_vendor_ext_sparc_fsp_result` enumeration and copy the match and `controller_id` fields from the associated request message.

The following result codes have been defined:

```
enum ofp_vendor_ext_sparc_fsp_result_type {
        OFPSET_FSP_RESULT_OK,  // request was accepted, flowspace is registered now
        OFPSET_FSP_RESULT_OVERLAP,  // request was accepted,
        // but overlaps with another registration
        OFPSET_FSP_RESULT_NAK,  // request was rejected
        OFPSET_FSP_RESULT_NOT_FOUND,  //  close request failed, flowspace not found
};
```

Figure 10: SPARC flowspace vendor extension result types

3.3.3 SPARC flowspace related error messages

OpenFlow 1.0 defines an error message to indicate various issues during operation. Upon reception of a Packet-Out/Flow-Mod message the datapath element may determine that the controlling entity is not allowed to handle the specified packet (*Packet-Out*) or create a flow table entry (*Flow-Mod*). In such a case an OF1.0 compliant error message is created reporting permission error to the controller with the following error type/code pairs:

- OFPET_BAD_REQUEST/OFPBRC_EPERM for not allowed *Packet-Out* messages, and
- OFPET_FLOW_MOD_FAILED/OFPFMFC_EPERM for *Flow-Mod* messages.

3.4 Configuration of network virtualization

In order to implement the virtualization ideas for OpenFlow 1.1 as proposed in SPARC deliverable D3.2 (section 3.2.5 - the models in which the "translation unit" is moved to the datapath element), several things need to be extended, both in the protocol and in the datapath elements itself. In the datapath, this mainly includes a translation unit that restricts certain commands and translates others. While a large part of the datapath configuration can be handled through the existing protocol (such as configuring the master flow tables) or through existing third party extensions (e.g. for creating queues) the translation unit itself needs to be configured. For this two extensions are necessary, one to authenticate the different controllers and assign roles, and another to define the different virtual networks. Authentication and role assignment could be handled by external means e.g. coupling the role to SSL certificates used when a controller connects. The second extension creates the definitions of the different virtual networks in the translation unit. Again this is something that could be handled by external means, e.g. through manual configuration via a CLI. In the OpenFlow community, there has been a discussion about creating (or adopting) a separate configuration protocol [11]. While a dedicated configuration protocol would be a good place to put this kind of transactional, non-volatile configuration functionality, we currently still lack such a protocol. Thus, we

decided to integrate this functionality into the existing OpenFlow protocol. Keep in mind that the design work is still ongoing so these extensions here are probably to change in the future.

3.4.1 Controller role assignment

This is a very simple implementation of pre-shared key based controller identification that is used primarily to assign roles to different connected controllers. As part of the connection setup, a secret key/identifier is sent by the connecting controller. If the key does not match a preconfigured key on the datapath (or FlowVisor depending on the virtualization model used), the connection is dropped. The preconfigured key determines which privileges the connected controller should have; currently there are two levels in the implementation, *Master* and *Virtual*. A controller given the Master role (authenticated with the Master key) has full privileges while a controller with given the Virtual role has limited access based on the configured virtual networks. This extension is not intended as a security measure but simply as a means of identifying the connecting controller. For certain situations, other means may be more suitable. For example, if SSL/TLS is used, a certificate id could be used, in other situations the controllers IP address and TCP port may be more suitable.

It is important that the role assignment is handled early during connection setup, before the controller and switch starts to exchange details about the switch configuration (i.e. before the exchange of OFPT_FEATURES_REQUEST / OFPT_FEATURES_REPLY messages), since it should present a different configuration depending on the virtual network configuration.

The pre-shared key is sent by the connecting controller as a single unsigned 32-bit integer using this structure:

```
struct ofl_exp_sparc_controller_id {
        struct ofl_exp_sparc_msg_header header;
        uint32_t controller_id;  // Pre-configured ID
};
```

Figure 11: OpenFlow controller ID message, expected during connection setup.

3.4.2 Virtual network configuration

Depending on the virtualization model (see section 3.2.4 of SPARC deliverable D3.2), the options of virtual network configuration are slightly different. In cases where the translation unit is placed outside of the datapath elements, the virtual network configuration can describe the entire virtual network, which is composed of multiple datapaths. In the other cases, where there is one translation unit per datapath element, the configuration has to be datapath-local (although the per-datapath configuration could be generated from a network-wide description). Here we suggest a rather simple datapath-local protocol extension for virtual network configuration, based on the improvement suggestion presented in section 3.2.5 of D3.2. The current design of this extension is by no means final; rather it reflects the current state of the implementation and should be further extended to allow more flexibility in the virtual network configuration.

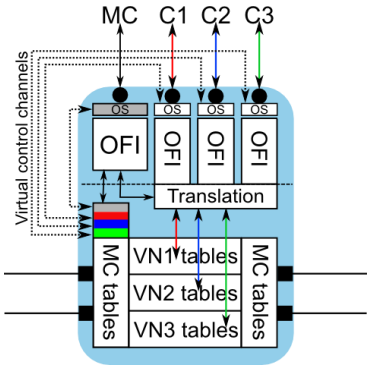

Figure 12: Suggested extended virtualization model, from SPARC deliverable D3.2.

In the suggested model, (i.e. see deliverable D3.2 or Figure 12 above), we need to configure the MPLS labels (and their corresponding EtherType) which should be used for the different virtual networks. With the current state of the extension this is limited to a per-datapath range which could be extended to allow more flexible per-link label set. We also need to assign a number of flow tables to each Virtual Network (VN) as

well as per-VN output queues. Again, in the current state this is limited to a range of tables but could be extended to support a set of flow tables (i.e. assigning e.g. table 1, 5, 6 and 7 to a VN rather than table from 1 to 5). Each VN also has a number of *system* ports (similar to VLAN *trunk* ports) as well as *customer* ports (similar to VLANs *untagged* ports); these restrict which ports are shown to the controller assigned to this particular VN. A system port is a port shared between multiple VNs, and packets leaving this port should be encapsulated using an MPLS label. On a customer port no encapsulation is applied nor expected, all traffic on these ports with the correct EtherTypes is assumed to belong to this VN. This is a rather simplified view of the customer ports, and the configuration could be extended to carry a more complex identifier of what traffic on each customer port should be assigned to this VN. For example, a per-customer port `ofp_match` structure could be included to provide a more granular filter. Finally, each VN is identified by an integer, useful for simplifying other configuration such as mapping controllers to VNs, removal of VNs etc. In the current implementation the VN identifier is directly mapped to the pre-shared key used to identify the role of a controller, directly assigning a virtual network to a connected controller.

```
struct ofl_exp_sparc_add_vn {
       struct ofl_exp_sparc_msg_header header;
       uint32_t vn_id;         // Virtual Network identifier
       uint32_t label_offset;  // Beginning of MPLS encapsulation label range
       // The length of the range is equal to the number
       // of EtherTypes to which the labels are mapped
       uint32_t table_start;   // Start of table-range
       uint32_t table_end;     // End of table-range
       uint32_t queue_id;      // Queue identifier
       uint32_t num_sysports;  // Number of system ports (shared ports)
       uint32_t num_custports; // Number of customer ports
       uint32_t num_etypes;    // Number of EtherTypes to support
       uint32_t sysports[];    // List of the system ports
       uint32_t custports[];   // List of the customer ports
       uint32_t etypes[];      // List of EtherTypes
};
```

Figure 13: SPARC vendor extensions message for creating new virtual network

```
struct ofl_exp_sparc_del_vn {
       struct ofl_exp_sparc_msg_header header;
       uint32_t vn_id;         // Virtual Network identifier
};
```

Figure 14: SPARC vendor extensions message for deleting virtual network.

3.5 BFD Based Continuity Check & Protection

In order to provide end-to-end monitoring of links, MPLS LSPs, and pseudo-wires, we have implemented a BFD-based solution inspired by the MPLS-TP standard[1]. Its functionality is discussed in detail in section 3.3.3 of SPARC deliverable D3.2. Here we provide a brief overview of the components and the protocol extensions needed to configure and use them.

[1] During the implementation these standards were still in the development phase and have since undergone several changes.

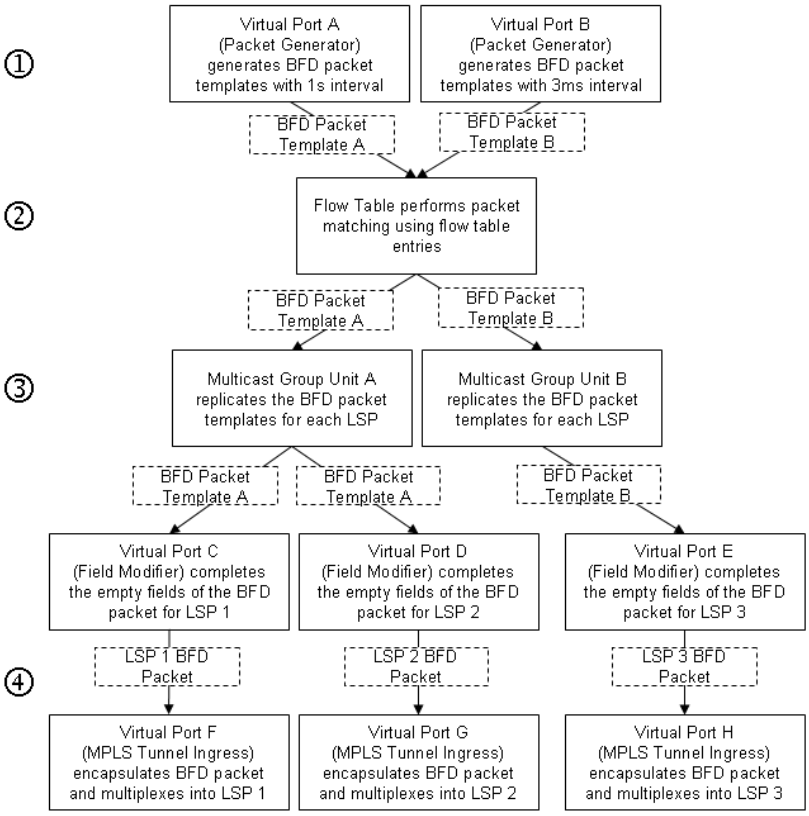

Figure 15: Functionality required for transmitting a BFD packet

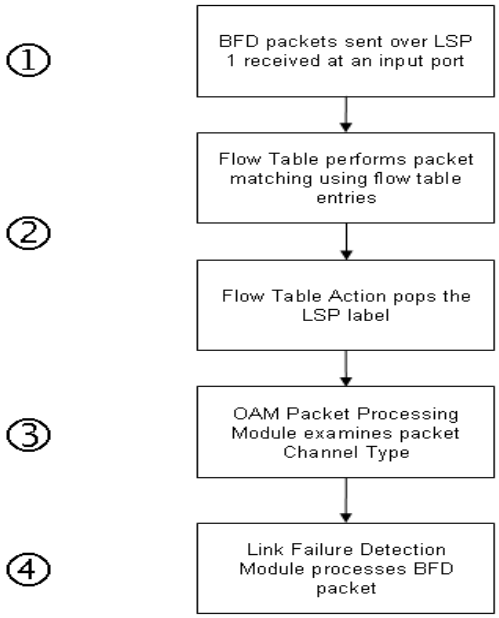

Figure 16: Functionality required for receiving BFD packets

In Figure 15 and Figure 16 the procedures for transmitting and receiving BFD packets can be seen respectively. The BFD packet insertion is split into several configuration phases, namely: packet generation, replication, content fill-in and packet injection. Although the first three steps could be done in a single step, the proposed multi-stage solution allows combining the common steps of several monitoring streams. This reduces the load on the OpenFlow forwarding pipeline and thus saves capacity.

To implement the above packet generation steps a new type of virtual ports has been applied. Such a typed virtual port extends the original concept of Section 3.2: it represents an instance of a generic processing entity and this entity can be configured with the help of OpenFlow together with the creation of the virtual port. It also provides means to define a subsequent physical, virtual or protocol reserved port to which a packet is passed executing the procedures at the current virtual port. These procedures can be quite complex and stateful. The basic or passive typed virtual ports are used to implement predefined packet manipulation steps, such as filling-in the BFD packet through rewriting various header fields, and adding additional headers and rewriting the existing ones to inject the BFD packets into the monitored flow. Active typed virtual ports are able to generate e.g., BFD template packets and to create internal hash-table holding BFD session information (step 1 in the left figure). In OpenFlow 1.1, the Multicast Group Entry is to be used for BFD packet replication. However, OpenFlow 1.0, as used in the SPARC prototype descried in D4.1, lacks such explicit multicast feature, though it supports implicit multicast through associating multiplicity of output port actions to a flow entry. However, the management of such extensions is much more complex compared to the explicit one, we excluded it from further consideration. Since BFD monitoring requires multicast feature; a further passive virtual port was defined for the OpenFlow 1.0 based case with functionality similar to Multicast Group Entries of OpenFlow 1.1.

At the egress side a BFD packet exception mechanism is required, which allows a reliable detection of the BFD frames with keeping information on the monitored entity (MPLS LSP or Link). However, at current OpenFlow actions on MPLS header manipulation do not have such capabilities. Therefore the MPLS label processing messages have been updated to detect and redirect the MPLS OAM packets for processing (step 3 and 4 in the right figure). In our MPLS-TP based example, the MPLS label pop action is extended to check if there is a second special GAL label after removing the first label. If there is a GAL label, the processed packet is removed from the processing pipeline and the G-ACh header expected after the GAL is processed. If the G-ACh carries an OAM packet (as indicated by payload field), the packet is passed to the BFD state handling module.

3.5.1 Configuration procedures

When configuring a BFD session, there are dependencies between the different stages, shown in Figure 17, together with the order of configuration. The dependencies are, for example, the packet fill-in virtual port (the "`FieldModifier`") needs to know where to send the modified packets; therefore an LSP ingress virtual port (or a physical port in case of link-only monitoring) has to be configured before creating the field modifier virtual port.

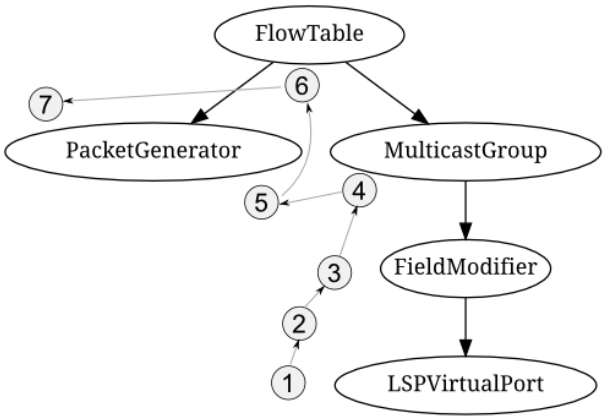

Figure 17: BFD session module dependencies

An appropriate order (seen in the figure above) for configuring a session would be:

1. Configuration of the outgoing port, be it a virtual or physical port. The assigned port number is necessary for step 2 & 3.

2. Creation of BFD State: The BFD state contains timing values, a MEP-ID, protection state, a paired MEP-ID (if independent failover is wanted), and the associated port that is monitored.
3. Creation of a packet filling-in port: the field modifier virtual port contains a MEP-ID as well as the associated monitored port.
4. Creation of a Multicast Group: The Multicast group contains multicast buckets for all BFD sessions running with the same timer value. Since this initially will be 1 second, multiple BFD sessions may be configured at once and put in a shared group at configuration time.
5. Allocation of a Virtual Port identifier for the Packet Generator: The Virtual Port identifier is needed in the next step, however, the packet generator should not be created before step 6 in order to avoid unnecessary traffic to the controller (generated packets will be unknown since no FlowTable entries have been installed).
6. Creation of a FlowTable entry to forward packets from the Packet Generator to the Multicast Group
7. Creation of a Packet Generator Active Typed Virtual Port: Once the whole chain has been established the packet generator can be installed

All these steps concern packet transmission. Receiving BFD packets are much simpler since they are handled automatically by the "exception mechanism" of the MPLS label processing actions. This will automatically detect OAM packets intended for the node and send them to the internal processing module.

3.5.2 Virtual port configuration options

The BFD packet generation and exception is represented by typed virtual ports in the OpenFlow pipeline, these typed virtual ports must be provisioned somehow. On one hand we have to manage them: create a new port instances, remove an existing one, etc. But on the other hand we also have to pass detailed configuration to those virtual port. In the considered protocol extensions, we assume that the port behavior and configuration is set through a vector of actions.

Section 3.2 describes a generic way of managing virtual ports in OpenFlow; however, it lacks any possibilities to pass detailed configuration of those ports. One possibility is to extend that message with an action vector that comprises of OpenFlow actions. The resulted structure is as follows:

```
struct ofp_vendor_ext_sparc_typed_vport {
        struct ofp_vendor_ext_sparc header;
        uint64_t controller_id;
        uint32_t result;
        uint32_t portno;
        uint32_t config;
        struct ofp_action_header actions[0];
};
```

Figure 18: Extended SPARC virtual port management message with action vector for virtual port configuration.

The only difference compared to the action presented in Section 3.2 is the substitution of the process id with an action vector. All other protocol operation and behavior defined in Section 3.2 also apply here.

The second option is to make use of virtual port management commands designed in the OpenFlow 1.0 version with Ericsson MPLS extensions [12]. This extension added so called virtual ports to OFP1.0 packet processing pipeline with. These virtual ports are addressable through the OUTPUT_PORT action of the flow entries and execute a list of actions on any packets redirected to that virtual port instance. Beside each virtual port has a parent port, which can be physical, virtual or protocol reserved ones, the packets will be passed to this latter parent port after performing the action list. This way it is possible to construct a chain of virtual ports between the flow table and the outgoing physical ports and OpenFlow reserved virtual ports.

Additional changes to the OFP1.0 with MPLS extensions version had to be made, primarily the virtual port identification. The OFP1.0 version uses a 16-bit value to identify ports. In order to accommodate Virtual Ports the port identifier was extended in the MPLS extended version to 32-bits. However, this extension was only made in selected messages, which managed virtual ports, while 16 bits port identifiers were used in all

other messages. To harmonize the messages and to support the packet generating virtual ports we introduced 32-bit port identifiers in all messages. This is necessary since Active Virtual Ports not only acts as targets for "output" actions, they can for example be used as `in_port` in the flow tables, which means that the `ofp_flow_mod` message had to be extended and similarly for all message types that includes a port identifier. This also requires updates how packets are directed into the flow table for processing.

Other additions are not directly related to OpenFlow as such, but rather the software switch implementation. These additions include things like support for high-resolution timers, which are needed to support microsecond resolution for BFD transmission and reception timers. Packet processing had to be updated to support packets arriving from virtual ports rather than physical ports only. Other changes include support for the BFD sessions to independently updating the flow tables in order to perform the switching between active and broken LSPs.

3.5.3 New actions

BFD session state

BFD session state is created by sending a `vport_mod` command with the type set to `OFPVP_ADD_ACTIVE` and containing an `ofp_avport_bfd_create_state` command. Internally the information carried in this message will be inserted in a hash-table keyed by a hash of the MEP-ID. The message contains BFD specific timer values, the sessions MEP-ID, the associated virtual port used by the LSP being protected. Optionally an additional MEP-ID can be provided in order to pair two BFD sessions. This is used for implementing protection switching in which case the two BFD monitoring endpoints are attached to the protected and the protecting entities. These two BFD entities are associated with each other and this association binds the two protection involving entities. It is used for instance to see if the protecting path is active in case of the failure of the protected one and to what (virtual) port identifier traffic should be redirected. The message structure is defined as follows:

```
struct ofp_avport_bfd_create_state{
      uint16_t type;        // OFPAVP_BFD_CREATE_STATE
      uint16_t len;         // Length is 48
      // timer variables
      uint32_t desmintx;    // Desired minimum transmission interval
      uint32_t reqminrx;    // Required minimum receive interval
      uint32_t dtctmult;    // Detection multiplier
      // MEP-ID
      uint32_t tunnelid;    // Tunnel-ID
      uint32_t lspid;       // LSP-ID
      uint32_t nodeid;      // Node-ID
      // failover MEP-ID
      uint32_t pairtunnelid;    // Paired session Tunnel-ID
      uint32_t pairlspid;   // Paired session LSP-ID
      uint32_t pairnodeid;  // Paired session Tunnel-ID
      // working/backup
      uint32_t prot_state;  // (enum ofp_bfd_prot_state)
      uint32_t associated_port;    // associated vport
};
```

Figure 19: Action initiating the creation of a BFD session state.

The following enumeration (Figure 20) specifies code points for protection state description field (`prot_field`). The allowed values are as follows: (0) no failover actions are associated to this BFD session and there is no pair BFD session (FAILOVER_NONE), (1) the BFD session monitored entity is a working (protected) element; (2) the BFD session monitored entity is a backup or protecting element; (3) and (4) the BFD session monitored elements are the working/backup tunnel under reversion (i.e. if the *protected*

entity becomes operational again, it should automatically redirect the traffic back onto it from the protecting entity):

```
enum ofp_bfd_prot_state {     // Role of this BFD session in case of pairing
       FAILOVER_NONE = 0,        // Un-paired
       FAILOVER_WORKING = 1,     // Current working LSP
       FAILOVER_BACKUP = 2,      // Current backup LSP
       FAILOVER_WORKING_REV = 3, // Working with tunnel-reversion
       FAILOVER_BACKUP_REV = 4   // Backup, tunnel-reversion
};
```

Figure 20: Enumeration of protection states.

The virtual port creation command has been extended with two new code points (Figure 21). These indicate that the virtual port is a so called active virtual port, i.e., it implements state machines and other active procedures.

```
enum ofp_vport_mod_command {    // Additions to the vport mod command
       OFPVP_ADD,                  /* New virtual port. */
       OFPVP_DELETE,               /* Delete virtual port. */
       OFPVP_ADD_ACTIVE,           /* New active virtual port. */
       OFPVP_DEL_ACTIVE            /* Delete active virtual port. */
};
```

Figure 21: New code points for creating and deleting active virtual ports (bold).

Field modifier virtual port

A field modifier virtual port can be created by sending a `vport_mod` with the type set to `OFPVP_ADD`, containing an `ofp_vport_action_bfd_modifier` action. This virtual port should have its output port set to the LSP, pseudo-wire, or physical port that should be monitored by the BFD session. It only carries a MEP-ID which is used by the virtual port logic to find the BFD session state data used to modify the template packets.

```
struct ofp_vport_action_bfd_modifier
{
       uint16_t type;     /* OFPPAT_BFD_MODIFIER */
       uint16_t len;      /* Length is 16 */
       uint32_t lspid;    /* MEP-ID.LSPID */
       uint32_t tunnelid; /* MEP-ID.TunnelID */
       uint32_t nodeid;   /* MEP-ID.NodeID */
};
```

Figure 22: Action to configure the BFD fill-in function of a virtual port.

BFD template packet generator

A BFD template packet generator is created by sending a `vport_mod` command with the type set to `OFPVP_ADD_ACTIVE` and containing an `ofp_avport_bfd_create_pktgen` structure. This command contains two values: the transmission interval in milliseconds, and the originating virtual port number. When the datapath receives the message it will immediately create a virtual port and start transmitting packet templates that are sent to the flow table.

```
struct ofp_avport_action_create_pktgen
{
       uint16_t type;     /* OFPAVP_BFD_CREATE_PKTGEN */
       uint16_t len;      /* Length is  */
       uint32_t vport;    /* Transmit from this vport number */
       uint32_t timerval; /* Transmission interval (microsecs) */
       uint8_t pad[4];
};
```

Figure 23: Configuring a typed virtual port to be a BFD packet template generation with defined interval.

Multicast group setup

Multicast group configuration is an integral part of OpenFlow 1.1. However, the OpenFlow 1.0 lacks such construct; therefore, a protocol extension to that version has been defined (based on the discussions in the OpenFlow community before the release of the OpenFlow 1.1 specification). This group implementation basically consists of a virtual port which contains multiple *action buckets*. Each action bucket contains a list of actions which are executed on incoming packets based on the type of the group. We have only implemented multicast groups, in which the original unmodified incoming packet is sent to each of the buckets sequentially. Groups are created with the `ofp_group_mod` command which contains a number of buckets that in turn contain the same actions as is available in the flow table. In the BFD configuration case these buckets will contain a single action, *output*, which is used to distribute the incoming packet template to all the field modifier virtual ports that use the same timer value (i.e. there will be one Multicast group per transmission time value). The protocol structures for creating multicast groups are shown below.

```
struct ofp_group_mod {
    struct ofp_header header;
    uint32_t vport;       /* virtual port number. */
    uint16_t command;     /* One of OFPG_*. */
    uint8_t pad[6];       /* Align to 32-bits. */
    struct ofp_bucket_header buckets[0];
};
```

Figure 24: Group configuration command added to OpenFlow 1.0

```
struct ofp_bucket_header {
    uint8_t type;            // Bucket type, one of OFPG_BUCKET_* */
    uint8_t weight;          // Weight (not supported atm)
    uint16_t length;         // length of action list
    struct ofp_action_header actions[0];  // OFPPAT_*
};
```

Figure 25: Bucket structure (added to OpenFlow 1.0)

```
enum ofp_group_mod_command {
    OFPG_ADD,        /* Add bucket to existing group mod  */
    OFPG_DELETE,     /* Delete bucket from existing group mod  */
    OFPG_SET         /* Initialize and add buckets to a group mod  */
};
```

Figure 26: Enumerator for the group command actions

3.5.4 BFD & Protection Notification

The above specified configuration instructions allow the controller to deploy monitoring entities for links and MPLS LSPs and to configure these monitoring entities. However, the controller cannot detect the occurred changes either of the state of the monitoring toolset or of configured protection. Such events, like

BFD state changes or protection path activation, carry important information on the controller as it must maintain as accurate view of the network as possible to avoid misconfiguration.

During designing the extensions, two mechanisms have been considered: controller initiated polling and switch generated asymmetric notification. The OpenFlow protocol already implements statistic collection, which is a good example of polling various switch attributes. Like any other polling mechanisms, the controller must actively poll all configured monitoring & protection entities with proper frequency. However, such polling based state obtaining method raises serious scalability concerns: the proper polling frequency can be quite large and the number of the monitored and/or protected entities can be in orders of magnitude larger than the number of nodes that are polled during collecting statistics. Therefore, we excluded this option and we defined protocol extension only for the switch generated notification option.

According to OpenFlow specification, the datapath elements are able to generate asynchronous messages to the controller, i.e. the "Packet-in", the "Flow Removed", the "Port Status" and the "Error" messages. Among these messages, the "Flow Removed" and the "Port Status" are for reporting changes in the datapath element configuration: the deletion of flow entry or the change of the state of a port, respectively. While the first three examples are dedicated for a single purpose, the "Error" message is a generic placeholder for reporting any problem to the controller.

In our currently proposed extension we propose an extension to the `ofp_error_msg` message, which acts as a generic placeholder to carry any notifications to the controller. The notification is encoded through the definition of a new code point in the error type (OFPET_NOTIFY, 8). The type of the notification, indicating an OAM related event or a switchover, is expressed in the error code field. Note that the error type defines a namespace for the error codes; therefore it is possible to differentiate 65536 notification types. In our case two notification types are enumerated as shown below:

```
enum ofp_notification_code {
  OFPN_PROTECTION, /* BFD Failover triggered */
  OFPN_OAM         /* BFD Session status changed */
};
```

Figure 27: Notification code points.

A possible alternative of defining a new error code point would be to extend the SPARC vendor extension type set (see section 3.1) and a new structure for encoding the details of the notification can be used.

This extension proposes using either of the above placeholders to notify the controller about changes in the monitoring and protection state. Whenever a BFD session enters the UP or DOWN state, an OAM notification is sent to the controller. This can be used by the controller to use the BFD module for switch based monitoring, without failover. The same message will also be sent whenever the BFD module receives an unexpected BFD message in order to detect misconfigurations and discover where the misconfiguration has occurred (for this reason the received unexpected MEP-ID is included).

If the BFD module has been configured to independently perform failover, it will use the same mechanism (and the same message payload) to notify the controller of the outcome of the operation. This includes both either success or failure of primary to backup path failover (which may fail if the backup path is currently down). If the BFD sessions have been configured for reversion (i.e. they automatically trigger falling back to the primary path when it becomes available again), this will be notified as well.

Regardless of the notification type (BFD state change or failover success/failure) the same structure is used, carried in an OpenFlow error message as is shown below. It contains a MPLS MEP-ID[2], which comprises of `tunnelid`, `lspid` and `nodeid` fields, used for identifying which BFD session the notification refers to, a Virtual Port identifier (`vport`) used to identify the affected MPLS LSP virtual or physical port (e.g. `LSPVirtualPort`, see Figure 17) in and two code fields (`oam_type` and `code`) used for identifying the notification sub-type and error code value.

[2] The MEP-ID is inherited from the MPLS-TP standardization (RFC6370) and is used to uniquely identify an LSP within an operator network.

```
struct ofp_bfd_oam_data {
        uint32_t oam_type;
        uint32_t code;
        uint32_t tunnelid;
        uint32_t lspid;
        uint32_t nodeid;
        uint32_t vport;
};
```

Figure 28: Common data structure carrying details of OAM and protection notifications.

The notification code (carried in the `code` field of `ofp_error_msg`) defines namespaces for the `oam_type` field. If it is an OAM status message, i.e. the `code` field is set to `OFPN_OAM`; the `oam_type` value will be one of the `ofp_oam_types` code points. But when a protection notification is sent, i.e. the `code` field is set to `OFPN_OAM`; the `oam_type` value will be from the `ofp_protection_type` code points. Both enumerators can be extended to support other types of protection or OAM mechanisms, in order to, for example, support an Ethernet-based OAM tool. Defining separate name spaces for protection and OAM types adds flexibility through supporting such toolsets that are able to monitor but not able to initiate restoration.

```
enum ofp_protection_types {
        OFPN_PROT_BFD
};
enum ofp_oam_types {
        OFPN_OAM_BFD
};
```

Figure 29: OAM type code points for protection and OAM state notification messages.

In the case of BFD protection notification, the `code` field of `ofp_bfd_oam_data` structure (Figure 28) can be set to four different values in order to notify about successful or failed execution of switchover and switchover reversion:

```
enum ofp_protection_notification{
  SWITCHOVER_SUCCESSFUL,
  SWITCHOVER_FAILED,
  REVERSION_SUCCESSFUL,
  REVERSION_FAILED
};
```

Figure 30: Enumerator for switchover / reversion status

In the case of BFD state change notification, the `code` field of `ofp_bfd_oam_data` (Figure 28) can inform if the session has gone up or down, or if some kind of misconfiguration has occurred, e.g. the session is receiving BFD data packets belonging to another session.

```
enum ofp_oam_notification {
  CC_DOWN,
  CC_UP,
  CC_MISCONF
};
```

Figure 31: Enumerator for BFD status

An example of the structure of a complete notification message for a detected failure can be seen below:

```
struct ofp_error_msg{
	struct ofp_header_msg header {
		uint8_t version = 0x01;		// OpenFlow protocol version
		uint8_t type = OFPT_ERROR;	// Error message
		uint16_t length = 36;		// Total length of the packet
		uint32_t xid ;			// Packet transaction ID
	}
	uint16_t type = OFPET_NOTIFY;		// Notification error type
	uint16_t code = OFPN_OAM;		// OAM notification subtype
	struct ofp_bfd_oam_data {
		uint32_t oam_type = OFPN_OAM_BFD;// BFD OAM subtype
		uint32_t code = CC_DOWN;	// BFD OAM session status
		// MPLS-TP MEP-ID
		uint32_t tunnelid = 1234;	// Tunnel-ID of the particular LSP
		uint32_t lspid = 1234;		// LSP-ID of the particular LSP
		uint32_t nodeid = 10.0.0.1;	// IPv4 address of the node
		uint32_t vport = 0x10004;	// MPLS LSP virtual port identifier
	}
};
```

Figure 32: Example BFD state change notification message

An example of the structure of a complete notification message for a successful protection action can be seen below:

```
struct ofp_error_msg{
	struct ofp_header_msg header {
		uint8_t version = 0x01;		// OpenFlow protocol version
		uint8_t type = OFPT_ERROR;	// Error message
		uint16_t length = 36;		// Total length of the packet
		uint32_t xid;			// Packet transaction ID
	}
	uint16_t type = OFPET_NOTIFY;		// Notification error type
	uint16_t code = OFPN_PROTECTION;	// Protection notification subtype
	struct ofp_bfd_oam_data {
		uint32_t oam_type = OFPN_PROT_BFD;	// BFD protection subtype
		uint32_t code = SWITCHOVER_SUCCESSFUL;	// BFD protection result
		uint32_t tunnelid = 1234;	// Tunnel-ID of the particular LSP
		uint32_t lspid = 1234;			// LSP-ID of the LSP
		uint32_t nodeid = 10.0.0.1;		// IPv4 address of the node
		uint32_t vport = 0x10004;		// MPLS LSP virtual port ID
	}
};
```

Figure 33: Example BFD protection action result notification message

3.6 Pseudo wire

Pseudo wire emulation (PWE) is a mechanism for emulating point-to-point connections over packet switched networks (PSN). In order to transport L1, L2 or L3 data signals over a PSN, PWE defines several layers. The payload encapsulation maps the client data signal to a sequence of packets and is responsible for carrying any information required to properly regenerate the carried signal at the remote endpoint. The PW

demultiplexing layer attaches an identifier (or label) to each packet. Then all packets extended with PW labels are encapsulated into the PSN defined transport connection. Although PWE allows carrying different client signals over a PSN, e.g., ATM, TDM, and Ethernet, we focused on only supporting Ethernet packets. We excluded L1 client signals as the current OpenFlow forwarding model does not consider them at all.

In our prototype, we have designed and implemented basic support for MPLS Pseudo Wire Emulation (PWE) according to RFC3985 and RFC4385, for OpenFlow 1.0 with the Ericsson MPLS extensions. Our implementation gives the possibility to tunnel Ethernet frames transparently through a MPLS transport network, and therefore enables implementing Ethernet services. Note that these extensions focus on PWE encapsulation/decapsulation, and the so called native service processing related functionalities (RFC 3985) are out of scope. For OpenFlow 1.1, the entire pseudo-wire implementation should be adapted to the updated switch processing pipeline model that comprises of the concatenation of multiple flow tables as well as the introduction of the group table.

3.6.1 General pseudo-wire processing

The first step when tunneling an Ethernet frame in MPLS PWE is to strip the preamble and the FCS of the Ethernet header (if they are still there), while the other fields like destination and source addresses, the EtherType and the payload will form the remaining packet. Then, if necessary, a PWE control word is prepended to the original packet. This control word is used in various optional features, for example to support ordered delivery of tunneled frames. This is not necessary for Ethernet encapsulation as Ethernet does not require ordered delivery. Therefore the control word can be filled with zeros and still fulfill the standards. For legacy reasons[3] the control word may also be omitted in case of Ethernet payload, when either no ECMP is implemented in the PSN or the ordered delivery of Ethernet payload is not required. The PWE label, which has the same format as a regular MPLS label, is then added to carry the PWE demultiplexer value. Then the packet is encapsulated into the MPLS transport tunnel by adding an additional MPLS label. The frame now contains all required parts of a MPLS PWE frame, but it is common to tag the frame with additional MPLS tags.

At egress side the following steps will happen: first the MPLS transport label is removed, and the PWE label identifies the PW entity that will further process the packet. After receiving the packet the PWE entity strips the PWE label and it will process the control word as well. If everything is fine, the payload is passed to appropriate networks service processing unit or to a client facing port.

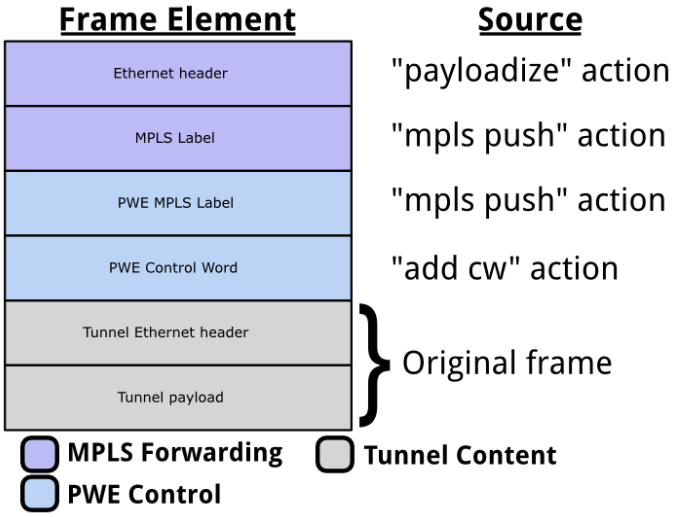

Figure 34: Outline of a PWE frame with sources for the different frame elements

[3] How a pseudowire without a control word should be handled is not entirely clear in the standards, RFC 4448 says that a lack of control word MUST be supported in order to support legacy hardware. However, RFC 4385 says that a control word must be present.

3.6.2 OpenFlow 1.0 pseudo wire processing

In an OpenFlow context, the above described procedures are modeled with the help of actions. Pairs of actions are assigned to each step and the first member of each pair is used at the ingress side (for encapsulation) while the second one is applied at the egress (for decapsulation).

At the ingress side, the first step is to reinterpret the just processed Ethernet packet as payload. The action, referred to as *payloadize (*`OFPPAT_PAYLOADIZE`*)*, treats the packet as payload and builds an encapsulating Ethernet header by inserting a new Ethernet header around the client frame. The parameters of the action encode the source/destination MAC and EtherType used when creating this new header. Internally in the datapath element, this action will push a new Ethernet header on to the packet buffer, in front of the client packet. It will then proceed by resetting the pointer in the buffer so the buffer now only contains a pointer to our newly created Ethernet header

```
struct ofp_vport_action_payloadize
{
    uint16_t type;                  /* OFPPAT_PAYLOADIZE. */
    uint16_t len;                   /* Length is 20. */
    uint8_t dl_src[OFP_ETH_ALEN];   /* Ethernet source address. */
    uint8_t dl_dst[OFP_ETH_ALEN];   /* Ethernet destination address. */
    uint16_t dl_type;               /* Ethernet frame type. */
    uint8_t pad[6];
};
```

Figure 35: SPARC payloadize PWE virtual port action.

The second required action is the “push control word” action (`OFPPAT_PUSH_CW`), which takes no parameters, and adds the control word after the Ethernet header. This action could be extended with optional argument when support for more PWE features is implemented. After the control word is added the PWE MPLS label needs to be pushed to the buffer, which can be done using the existing MPLS push action. The frame is now a standard compliant MPLS PWE frame and may be handled as any other MPLS frame. Both of the above mentioned actions are use in the ingress side of the PWE tunnel.

At the egress side, the counterparts of the “payloadize” and the “push control word” actions must be implemented. However, the processing methods associated to the control word depend on the pseudo wire type; therefore, the two functionalities are implemented in a single “packetize” action (`OFPPAT_PACKETIZE`). The attributes of this “packetize” action depends on the functionality it implements. For instance, in case of Ethernet PWE it may carry an attribute on indicating whether a control word is expected on the stack or not, and a further flag may enable/disable to control word processing.

The implemented “payloadize” action implicitly assumes that the control word is in the packet but it does not processes it. After removing the control word it removes the encapsulating Ethernet header. The “packetize” action is also responsible for reconstructing the correct pointer in the packet buffer, for example the pointer to an IPv4 headers, so other actions applied after this action can find the correct pointer to work with.

In the current implementation neither “push control word” nor ‘payloadize” actions carry attributes. Therefore they are actually empty actions, i.e., they comprises only of the common action header that contains the action type. Therefore, the format of these two actions does not need to be detailed in this deliverable.

3.6.3 Implementation example

This example illustrates the usage of the PWE extensions through the configuration of a point-to-point Ethernet service between two interfaces of two edge nodes. Since similar configuration steps are performed in both directions, here we only discuss configuration of one direction. At the ingress side a typed virtual port is created and the PWE encapsulation actions, namely the “payloadize” and the “push control word” actions are attached to it. Then this typed virtual port is configured to pass the processed packet to other virtual port representing the ingress endpoint of the MPLS tunnel, which will carry the frames to the remote edge node. A typical setup therefore could have the following structure:

1. The Ethernet frames belonging to the provided service are identified with a help of a flow entry, which points to the PWE encapsulation. In our example the identification is based only on the identifier of the incoming port of the Ethernet frames.
2. A PWE encapsulation virtual port consists of:
    a. A ”Payloadize” action, with Ethernet source/destination MAC and EtherType
    b. A “Push Control Word” action
    c. A ”Push MPLS label” action, for PWE tunnel identification
    d. And finally, an “outport” action to forward the packet to the MPLS ingress virtual port
3. The MPLS ingress virtual port will carry actions for
    a. Pushing the transport MPLS label
    b. Updating the source and destination MAC headers according to the MPLS operation (i.e., the source set to the current node, while the destination set to the next-hop one)
    c. Forwarding a packet to physical port

A good property of this solution is that it is flexible in its configuration but still gives the possibility to chain different virtual ports and thereby increase the scalability of the PWE handling. Also the solution is simple to design and gives some flexibility for future extensions.

3.7 OpenFlow extensions for energy-efficient networking

3.7.1 Introduction

An important goal for future networking is the reduction of its carbon footprint. The attention for climate change is influencing the ICT sector. ICT accounts for 2 to 4% of the worldwide carbon emissions. About 40 to 60% of these emissions can be attributed to energy consumption in the user phase, whereas the remainder originates in other life cycle phases (material extraction, production, transport and end-of-life) [13].

Decreasing the power consumption of network elements is an obvious solution reducing the carbon footprint. The Energy-Efficient Ethernet concept aims reducing the energy consumption of the network elements without impairing its performance. The IEEE802.3az standards defined the Low Power Idle mode of Ethernet interfaces, i.e., it detects when there is nothing to transmit and switches the interface to a low consumption idle mode [13]. Nevertheless, other concepts have been proposed, for instance burst mode (BM) and adaptive link rate (ALR) operation modes. In BM, the idea is to conserve energy by buffering packets and sending them in bursts when a certain buffer capacity threshold is reached. In ALR, the switch ports are optimized for different line rates and, according to the current traffic load, can switch to the desired rate. An example of ALR was also proposed for implementation in Energy-Efficient Ethernet [15].

The OpenFlow based split architecture allows us to implement energy consumption reducing strategies in the controller, taking benefit of turning off power-hungry components. As discussed in section 3.7 of D3.2, several possible energy saving approaches are for instance network topology optimization (e.g. multilayer traffic engineering, MLTE). MLTE adapts the network layout in times of low traffic loads in order to allow complete switching off of certain links, i.e. switch ports. OpenFlow currently has limited support for the control of power management in the switches. For enabling efficient power management, OpenFlow must support three specific functionalities: advertisement of specific switch capabilities, control of these capabilities, and monitoring of switch parameters. In the following sections, we will detail the proposed extensions for energy efficient networking.

3.7.2 Extensions for dissemination of capabilities

First of all, if the OpenFlow application is to make use of energy efficiency innovations in the switch hardware, the application must learn which capabilities are available, e.g. BM and ALR. While such port features can be configured locally without input from the OpenFlow controller, it does make sense to allow centralized configuration from the controller in order to make more efficient use of them. If the OpenFlow application knows characteristics for the flows in the network, it can fine tune the ports. If the OpenFlow application knows the average, peak and minimum bitrates for a certain flow in advance, it can prevent the

port from using its own statistics in setting the line rate, which may be inefficient. For instance, in movies where a low-motion scene is followed by an intense action sequence, automatic ALR based on average measurements may lead to choppy performance. Dedicated OpenFlow applications which have a-priori statistics on the streaming movie can raise the line rate just before the action sequence to avoid performance issues.

Port capabilities are advertised in the `ofp_port` structure. This structure contains the features which are currently enabled (`curr`), `advertised`, `supported`, and `peer-advertised`. These features are formatted in an `ofp_port_features` bitmap. It also contains a `curr_speed` and `max_speed` field, which are used if the features bitmap is not used.

```
/* Description of a port */
struct ofp_port {
    uint32_t port_no;
    uint8_t pad[4];
    uint8_t hw_addr[OFP_ETH_ALEN];
    uint8_t pad2[2];                    /* Align to 64 bits. */
    char name[OFP_MAX_PORT_NAME_LEN];   /* Null-terminated */
    uint32_t config;                    /* Bitmap of OFPPC_* flags. */
    uint32_t state;                     /* Bitmap of OFPPS_* flags. */
    /* Bitmaps of OFPPF_* that describe features. All bits zeroed if
     * unsupported or unavailable. */
    uint32_t curr;                      /* Current features. */
    uint32_t advertised;                /* Features being advertised by the port.
*/
    uint32_t supported;                 /* Features supported by the port. */
    uint32_t peer;                      /* Features advertised by peer. */
    uint32_t curr_speed;                /* Current port bitrate in kbps. */
    uint32_t max_speed;                 /* Max port bitrate in kbps */
};
OFP_ASSERT(sizeof(struct ofp_port) == 64);
```

Figure 36: OpenFlow port describing structure from OFP1.1.

To indicate the considered BM and ALR capabilities of a port we extended the `ofp_port_features` bitmap. It contains two extra bits (OFPPF_BURST_MODE and OFPPF_ALR_MODE), one for BM and one for ALR as shown below.

```
enum ofp_port_features {
OFPPF_10MB_HD       = 1 << 0,    /*10 Mb half-duplex rate support. */
OFPPF_10MB_FD       = 1 << 1,    /*10 Mb full-duplex rate support. */
OFPPF_100MB_HD      = 1 << 2,    /*100 Mb half-duplex rate support. */
OFPPF_100MB_FD      = 1 << 3,    /*100 Mb full-duplex rate support. */
OFPPF_1GB_HD        = 1 << 4,    /*1 Gb half-duplex rate support. */
OFPPF_1GB_FD        = 1 << 5,    /*1 Gb full-duplex rate support. */
OFPPF_10GB_FD       = 1 << 6,    /*10 Gb full-duplex rate support. */
OFPPF_40GB_FD       = 1 << 7,    /*40 Gb full-duplex rate support. */
OFPPF_100GB_FD      = 1 << 8,    /* 100 Gb full-duplex rate support. */
OFPPF_1TB_FD        = 1 << 9,    /* 1 Tb full-duplex rate support. */
OFPPF_OTHER         = 1 << 10,   /* Other rate, not in the list. */
OFPPF_COPPER        = 1 << 11,   /* Copper medium. */
OFPPF_FIBER         = 1 << 12,   /* Fiber medium. */
OFPPF_AUTONEG       = 1 << 13,   /* Auto-negotiation. */
OFPPF_PAUSE         = 1 << 14,   /* Pause. */
OFPPF_PAUSE_ASYM    = 1 << 15,   /* Asymmetric pause. */
/* SPARC ENERGY EFFICIENCY EXTENSIONS */
OFPPF_BURST_MODE    = 1 << 30,   /* Burst Mode capability)
OFPPF_ALR_MODE      = 1 << 31,   /* Adaptive Line Rate capability*/
};
```

Figure 37: OpenFlow port feature list extended with new bits for burst and ALR modes.

3.7.3 Extensions for monitoring of switch parameters

Energy-efficient applications, globally optimizing the network by power consumption, need a way for monitoring key switch parameters. OpenFlow currently provides counters which gather statistics on flows, ports, flow tables, etc, but currently parameters governing the entire switch as a whole, such as its current power consumption, temperature, etc. are not considered. The protocol does contain the `OFPST_DESC` statistics request type, which contains descriptions of the switch such as manufacturer and serial number, and could be extended to contain such parameters; however we opt for a new structure `OFPST_SPAR` to contain live switch parameters. Furthermore, current proposals for energy efficient networking include solutions such as “follow the wind / follow the sun”, which envision redirecting traffic towards datacenters abundant in green energy sources [13]. To enable such innovative solutions we need to know the output of such green energy sources at the datacenter/switch. A pragmatic solution may be to have switches with two dedicated power units, one coming from local “clean” sources (such as solar panels) and one from the conventional grid.. Monitoring both sources will indicate where clean energy is most abundant.

```
struct ofp_spar_stats {
        uint32_t poll_intrvl;      /* interval for gathering stats. (sec) */
        uint32_t temp_inst;        /* instantaneous temp. reading (K) */
        uint32_t temp_min;         /* min switch temp. in interval (K)*/
        uint32_t temp_max;         /* max switch temp. in interval (K)*/
        uint32_t temp_avg;         /* average switch temp. in interval (K) */
        uint32_t pwr_cnsmptn_inst;/* instantaneous power consumption (W) */
        uint32_t pwr_cnsmptn_min; /* min power consumption in interval(W) */
        uint32_t pwr_cnsmptn_max; /* max power consumption in interval(W) */
        uint32_t pwr_cnsmptn_avg; /* average power consumption in interval(K) */
        uint32_t green_pwr_inst;  /* instantaneous available green power (W) */
        uint32_t green_pwr_min;   /* min available green power (W) */
        uint32_t green_pwr_max;   /* max available green power (W) */
        uint32_t green_pwr_avg;   /* avg available green power (W) */
};
```

Figure 38: Body of a new statistics reply message reporting energy consumption statistics

The polling can be done using the already existing statistic polling `OFPT_STATS_REQUEST/REPLY` mechanism of OpenFlow. First of all, the switch capability reporting bitmap (`ofp_capabilities`) must be extended with a new bit (`OFPC_GREEN_STAT`) that indicates if the switch supports reporting the above specified statistics. Then a new statistic type is also defined (`OFPST_GREEN_STAT, 0xFFFE`), which is used in the statistic requests and reply. In a request message it indicates that the controller polls the above statistic and in the reply message it indicates that the content of the reply conforms to the structure of Figure 38.

In order to provide live updates to the controller, the switch might notify the controller on above data without any specific controller requests. To implement such reports we can make use of the Notification mechanism described in Section 0. First a notification types is extended with a new code point, the `OFPN_GREEN_STAT`. The payload of a notification message with this type will be the statistics defined in Figure 38.

3.7.4 Extensions for control of capabilities

For enabling features on the port, we use the `ofp_port_config` structure. We can use the `OFPPC_PORT_DOWN` field for turning on/off the ports from the application. If Burst Mode and ALR are available we can also turn them on/off here.

```
enum ofp_port_config {
        OFPPC_PORT_DOWN     = 1 << 0,    /* Port is administratively down. */
        OFPPC_NO_RECV       = 1 << 2, /* Drop all packets received by port. */
        OFPPC_NO_FWD        = 1 << 5,    /* Drop packets forwarded to port. */
        OFPPC_NO_PACKET_IN  = 1 << 6,    /* Do not send packet-in msgs for port.*/
        /* SPARC ENERGY EFFICIENCY EXTENSIONS */
        OFPPC_BURST_MODE    = 1 << 14,   /* Enable Burst Mode */
        OFPPC_ALR           = 1 << 15,   /* Enabling Adaptive Line Rate */
};
```

Figure 39: Extended port configuration flags.

We also need to control some parameters for BM and ALR. Some features can be controlled by the standard OFP1.1 port modification message. Note that `ofp_port_mod` is not part of the OpenFlow 1.0 specification, so in order to support these extensions also in OFP1.0 we adopt `ofp_port_mod` from OFP1.1.

```
struct ofp_port_mod {
        struct ofp_header header;
        uint32_t port_no;
        uint8_t pad[4];
        uint8_t hw_addr[OFP_ETH_ALEN];
        uint8_t pad2[2];     /* Pad to 64 bits. */
        uint32_t config;    /* Bitmap of OFPPC_* flags. */
        uint32_t mask;      /* Bitmap of OFPPC_* flags to be changed. */
        uint32_t advertise; /* Bitmap of OFPPF_*. Zero all bits to prevent
                             any action taking place. */
        uint8_t pad3[4];    /* Pad to 64 bits. */
};
```

Figure 40: Configure the behavior of the virtual port (adapted from OFP1.1 to OFP1.0)

We use the `config` field which sets the `OFPPC_*` to enable/disable BM and ALR and we use the `advertise` field which sets the `OFPPF_*` structures to set the Line Rate. For Burst Mode, we need to set the burst length and maximum waiting time to send the buffer. These features are implemented using queues:

```
enum ofp_queue_properties {
        OFPQT_NONE = 0,           /* No property defined for queue (default). */
        OFPQT_MIN_RATE,           /* Minimum datarate guaranteed. */
        /* Other types should be added here (i.e. max rate, precedence, etc). */
        /* SPARC ENERGY EFFICIENCY EXTENSIONS */
        OFPQT_BM_BURST_LENGTH,    /* Length at which the burst is transmitted */
        OFPQT_BM_MAX_WAIT,        /* Max waiting time to transmit buffer */
};
```

Figure 41: OpenFlow queue configuration extensions

Switches that natively support Burst Mode must implement the `burst_mode_queue`:

```
struct ofp_queue_burst_mode {
        struct ofp_queue_prop_header prop_header;
        /* prop:OFPQT_BURST_MODE,len:16.*/
        uint32_t burst_length;    /* In bytes. */
        uint32_t max_wait;        /* In ns */
        uint8_t pad[6];           /* 64-bit alignment */
};
```

Figure 42: Burst-Mode queue property description.

The `max_wait` is specified in nanoseconds; allow a range from one nanosecond up to 4 seconds for the buffer send delay.

4 Summary and Conclusions

This deliverable reports OpenFlow protocol extensions for supporting the architectural and functional extensions required by large-scale wide area networks, such as carrier-grade operator networks. OpenFlow appears to be an interesting evolving technology, reasonably well-suited for realizing split-architecture operations in networks. Previous SPARC deliverables (D2.1 and D3.1) made it clear that current OpenFlow implementations do not fulfill all carrier requirements. According to the work done in WP3, D3.2 provided a set of functional extensions to the state-of-the-art OpenFlow protocol. These functional extensions, however, were at different level of discussion. Some aspects were thoughtfully discussed and mature proposals have been provided. But some other aspects were discussed only at conceptual level which resulted in only initial concepts and proposals covering only some parts.

The present deliverable reports OpenFlow protocol extensions, which have been specified during the prototyping activities and which are based on the functional extensions given in D3.2. Legacy control protocols, like GMPLS, have been not been discussed in this document as earlier use case and architecture studies did not indicate the need for extending this type of protocols. In this final section, we will summarize the proposed OpenFlow protocol extensions. We will also report the implementation status of these extensions, and how they are integrated into the different demonstrators.

4.1 Implementation of proposed OpenFlow protocol extensions

In this deliverable we provided OpenFlow protocol extensions of selected study topics of D3.2. The mapping of the individual protocol extensions to the study topics is summarized on Table 2.

Study topic (as named in D3.2)	**Proposed extension (as named in D4.2)**	**Defined in section**	**Prototyping status**
Openness and Extensibility	Dynamic Configuration of Virtual Ports	3.2	DONE, Split BRAS prototype
	Flowspace registration	3.3	DONE, Split BRAS prototype
Virtualization and Isolation	Configuration of network virtualization	3.4	NO
OAM: technology-specific MPLS OAM	BFD Based Continuity Check & Protection	3.5	DONE, SPARC prototype (D4.1)
Service Creation	Pseudo wire	3.6	DONE SPARC prototype (D4.1)
Energy-Efficient Networking	OpenFlow extensions for energy-efficient networking	3.7	NOT PLANNED

Table 2: Mapping proposed protocol extensions to D3.2 study items.

The above table also indicates the prototyping status of the different extensions.

Dynamic Configuration of Virtual Ports: As it was pointed out in D3.2, the main idea of the virtual port concept is to hide the complexities of processing units and not to require continuous extensions of the OpenFlow protocol when new extensions are added. According to the concept, the detailed configuration of the processing entities happens by other means than OpenFlow. However, due to lack of such auxiliary configuration protocol, in the implementation the specified messages are extended with the configuration instructions of considered processing unit, namely the PPP termination function.

Flowspace registration: The specified extensions have been fully prototyped in a targeted prototype demonstrating the distributed BRAS concept.

Configuration of network virtualization: The implementation of the specified extensions has not started, since the specification of the virtualization model and the extensions consumed most of the allocated resources.

BFD Based Continuity Check & Protection: The defined extensions have been fully prototyped in the software switch implementation and the support of these extensions is added to the LibOFP and to the NOX based controller. These extensions are integrated part of the SPARC MPLS OpenFlow demonstrator.

Pseudo wire: The defined extensions have been fully prototyped in the software switch implementation and the support of these extensions is added to the LibOFP and to the NOX based controller. This extension is integrated part of the SPARC MPLS OpenFlow demonstrator.

OpenFlow extensions for energy-efficient networking: No implementation work has been done, due to lack of appropriate hardware devices. In the current prototype we are using software switches running on Linux machines. This platform does not provide such capabilities that could have been used when the proposed extensions were prototyped. Besides, it is not planned to do any implementation on real hardware platform.

The implemented extensions have been integrated into prototypes. The virtual port management and the flow space registration extensions have been presented in the "Split BRAS" targeted prototype: this smaller prototype focused on validating the developed extension in the context of realizing a distributed Broadband Remote Access Server or BRAS. The other finalized extensions have been added to the main MPLS OpenFlow prototype presented in D4.1. Both prototypes, containing the above specified and implemented extensions have been presented at the 1st annual review meeting, in Poznan, October 2011. The possible integration of the different prototypes is under discussion at the time of writing of this deliverable.

References

[1] ITU-T G.8080

[2] "Initial Split Architecture and Open Flow Protocol study", SPARC Deliverable D3.1, October 2010.

[3] "Update on Split Architecture for Large Scale Wide-area Networks", SPARC Deliverable D3.2, December 2011.

[4] Nick McKeown et al., "OpenFlow: Enabling Innovation in Campus Networks", March, 2008. http://www.openflow.org/documents/openflow-wp-latest.pdf

[5] "Status Report on the Prototype of Integrated OpenFlow Reference Architecture", SPARC Deliverable D4.1, July 2011.

[6] Teemu Koponen et al, "Onix: A Distributed Control Platform for Large-scale Production Networks", *9th USENIX Symposium, Vancover, Canada Oct 4-6 (2010),* http://www.usenix.org/events/osdi10/tech/full_papers/Koponen.pdf

[7] Amin Tootoonchian, Yashar Ganjali, "HyperFlow: A Distributed Control Plane for OpenFlow Networks", *NSDI Internet Network Management Workshop/Workshop on Research on Enterprise Networking (INM/WREN),* San Jose, CA, April 2010.

[8] L. Martini, N. El-Anwar and G. Heron, "Encapsulation Methods for Transport of Ethernet over MPLS Networks", *IETF RFC-4448,* April, 2006.

[9] S. Bryant and P. Pate, "Encapsulation Methods for Transport of Ethernet over MPLS Networks", *IETF RFC-3985,* March 2005.

[10] S. Bryant et al., "Pseudowire Emulation Edge-to-Edge (PWE3) Control Word for Use over an MPLS PSN", *IETF RFC 4385,* February 2006.

[11] "Config protocol", collection of suggestions by the OpenFlow community, Online. Accessed January, 2012. http://www.openflow.org/wk/index.php/Config_Protocol

[12] Ericsson MPLS extensions to OpenFlow V1.0. online: http://www.openflow.org/wk/index.php/OpenFlowMPLS

[13] W. Vereecken et al., Power consumption in telecommunication networks: overview and reduction strategies, IEEE Communications Magazine, vol. 49, no 6, 62-69, 2011.

[14] IEEE P802.3az Energy Efficient Ethernet Task Force http://www.ieee802.org/3/az/index.html

[15] F. Blanquicet and K. Christensen, S. Suen, and B. Nordman, "Reducing the"PAUSE Power Cycle: A New Backwards Compatible Method to Reduce Energy Consumption Use of Ethernet with an Adaptive Link Rate (ALR)," IEEE Transactions on Computers, Vol. 57, No. 4, pp. 448-461, AprilSwitches", ethernet alliance, 2008.

[16] Ward Van Heddeghem et al., "Distributed computing for carbon footprint reduction by exploiting low-footprint energy availability", Future Generation Computer Systems, Volume 28, Issue 2, February 2012, Pages 405-414.