POLITECNICO DI TORINO

POLITECNICO DI TORINO

Master Degree course in Communications and Computer Networks Engineering

Master Degree Thesis

Data-plane assisted state replication with Network Function Virtualization

Supervisors
Prof. Paolo Giaccone
Prof. Andrea Bianco

Candidate
Iman Lotfimahyari

Academic Year 2019-2020

Acknowledgements

I would first like to thank my thesis advisors professor Paolo Giaccone and professor Andrea Bianco for their patient, guidance, and useful evaluations of this master thesis. The doors to their offices were always open whenever I ran into a trouble spot or had a question about my research or writing. They consistently allowed this paper to be my own work, but steered me in the right direction whenever they thought I needed it. I would like to show my very great appreciation to German Sviridov for his constructive professional hints at the beginning of this thesis. Many thanks to Politecnico di Torino for giving me the opportunity and also providing me the proper situation to do the master degree. Finally, I must express my very profound gratitude to my parents and to my beloved wife Mahboobeh for supporting me in all the steps and for their continuous encouragement throughout my whole years of the study and also through the process of researching and writing this thesis. These accomplishments could not be possible without them. Thank you.

Abstract

Nowadays, the continuous progression of NFV transformation is due to considering it as the platform for 5G. NFV based on SDN can extend the programmability and flexibility features of advanced network functions. Many of these network functions, however, need concrete state information to effectively process network flows. In this work we introduced a protocol to replicate the states between the VNFs in an SDN network, then we implemented two sample solutions with a slight difference in their schemes as the POC. Furthermore, we evaluated the amount of memory resources needed inside the SDN switches for successfully replicating the states between the VNFs through our protocol, and at last, we demonstrated by using the means of P4 programming language, we can efficiently reduce this value.

Contents

Chapter 1

Introduction

Nowadays, the continuous progression of the Network Function Virtualization(NFV) transformation is due to considering it as the platform for the 5G. The NFV based on SDN can extend the programmability and flexibility features of advanced network functions. Many of these network functions, however, need concrete state information to effectively process network flows. In this work, we have proposed a solution for sharing the states between different Virtual Network Functions(VNF) with the help of the data-plane programmable switches, while the generality of the proposed solution makes it applicable on the vanilla SDN switches as well. In the second chapter of this work, we briefly recalled the Software-Defined Networking and its most successful protocol. Then we took a look at a new approach for programming the data-planes of the new SDN switches. Furthermore, we reviewed the Network Function Virtualization, the state-full VNFs, and their important role in the 5G networks. In the third chapter of this work, we described our solution for the state sharing between the VNFs. Then we explained the whole protocol and its parts in detail and two different methods of implementing the solution. Finally, we evaluated theoretically the solutions and checked their scalability for the number of the NFs or the servers and the number of variables that can be participated in our solutions. In the fourth chapter, after introducing the needed tools for implementation and measurement, we implemented both solutions. Then for comparison, we implemented a vanilla SDN based structure with our proposed protocol and compared the three of them in terms of the switch resource consumption.

Chapter 2

Background

2.1 Overview

The traditional IP networks will not separate the control-plane from the data-plane. These planes usually are inside the same network device, making network structure highly decentralized to guarantee network resilience. They usually have a static architecture with a high degree of complexity, which makes their control even more complex and sometimes very hard. Because not efficient configuration will cause the network to behave not optimized and under the expectations, which can lead to having unwanted behaviors like loops, losing the packets and even more erroneous behaviors. Therefore, the vendors have to propose their proprietary management solutions for supporting network management. This includes the specific operating systems, and the network applications as well. Because of the proprietorship of these solutions, and considering that most of the time the network operators have no choice but to use different devices in different parts of the network, they need to maintain separate solution for different equipment of different vendors, they even have to separate supporting teams, which increases maintenance complexity and costs of the networks. For solving the lack of functionality problem within the network, the operators of the network have to use some middle-boxes, such as load-balancers, firewalls and other equipment, which the number of them, even if not larger, is comparable to the number of basic network devices. By using these middle-boxes, one will increase the complexity of the design and decreases the flexibility of network operations.

2.2 SDN

Software-Defined Networking (SDN) is an approach to network management that enables dynamic, programmatically efficient network configuration to improve network performance and monitoring. It is somehow a new paradigm to solve the limitations of traditional IP networks. Because the static architecture of traditional networks is decentralized and complex, current networks require more flexibility and easy troubleshooting. SDN attempts to centralize the intelligence of the network in one network device with the help of separating the packet forwarding process of the network or the data-plane from

the control-plane or the routing part. Figure 2.1 gives a general view over the difference between SDN and traditional IP networks. Separating the control-plane from a network

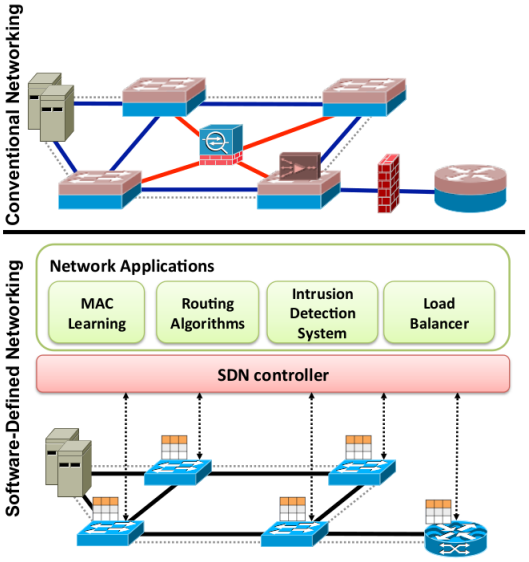


Figure 2.1. Traditional networking paradigm versus Software-Defined Networking (SDN) paradigm (reproduced from [1]).

device has several advantages:

- It gives us an affordable and non-complex test-bed to be used for developing SDN applications. The possibility to share or move the abstractions of the network programming languages and the platform of the control-plane, it will be easier to program the related applications.
- Different applications can integrate into each other in more straightforward ways. As an example, routing applications and load balancing ones can be used at the same time, where the load balancing decisions more priority over routing policies.
- The global network view can be shared between all applications, which by using this information by the control-plane software modules, policy decisions can be made more effectively.

The control-plane usually is made of one or sometimes more controllers which act as the brain of the SDN network. The intelligence centralization has its drawbacks when it comes to security [1], scalability [1], and dependability [1], which are some of the main concerns about the SDN. In Figure 2.2 we can see a tri-fold perspective presentation of the SDN.

Compared to a traditional network, an SDN infrastructure is made of several networking devices like switches, firewalls, and middle-box appliances. Their main difference is that the physical traditional devices are replaced with some simple forwarding devices

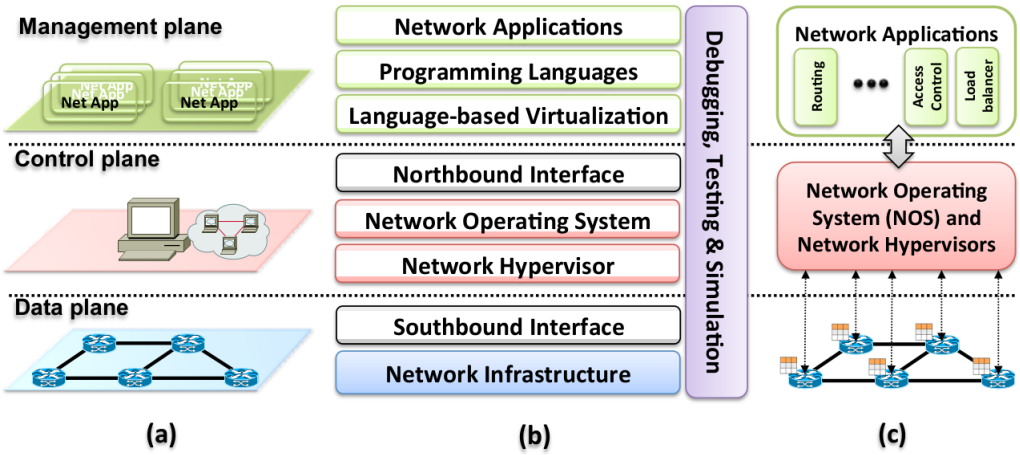


Figure 2.2. Software-Defined Networks simplified in (a) planes, (b) layers, and (c) system design architecture (reproduced from [1]).

without having an embedded control-plane or software to make autonomous decisions. As shown in Figure 2.2(a), the SDN helps by solving the problem related to the vertical integration, with separating control logic form the controlled hardware, which is responsible for forwarding the data. The control logic has been moved to a network operating system or a controller which is logically centralized, to simplify the network configuration and policy enforcement. An SDN architecture can be represented as a mixture of different layers, as illustrated in Figure 2.2(b). As illustrated, each of the layers has its special functions, while some of them are always used in an SDN deployment, like the southbound API(Application Programming Interface), the network operating systems, and the northbound API and even the network applications, others may participate only in special deployments, such as a hypervisor or maybe a language-based virtualization. Figure 2.2(c) depicts the integration of the layers in Figure 2.2(b) into a system design perspective, while the intelligence of the network is moved from the data-plane devices to a logically centralized control system, i.e., the applications and the network operating system. It is important to mention, that a logically centralized model does not signify a physically centralized system, Instead, control-planes of the industry SDN controllers are physically distributed, while they have a centralized logical structure. Separation of the control-plane from the data-plane requires a well-defined API between the SDN controller and the switches, as shown on the Figure 2.2(c). It is notable that, we can benefit more when the control logic becomes logically centralized. First of all, the modification of the network policies becomes less error-prone and much simpler, by using software components and high-level languages, when compared to the vendor-specific low-level device configurations. Second, a control program can automatically react to changes in the state of the network and maintain the high-level policies in any circumstances, including device/link outages and spikes of the data traffic. Finally, the control logic centralization in a controller gives global knowledge of the state of the network and therefore simplifies

the development of sophisticated networking functions, services, and applications. One can generally define the SDN by these abstractions:

- **Forwarding:** Any forwarding behavior which is desired by the control program should be allowed by the forwarding abstraction, while at the same time, it should hide the details for the hardware underly. E.g., OpenFlow has been one of the most successful realizations for such abstraction, which is considered equivalent to a device driver in an operating system.

- **Distribution:** The distribution abstraction should protect SDN applications from the problems of the distributed states. It will turn the distributed control problem into a logically centralized problem. For realizing it, you will require a common distribution layer, that in SDN it is placed in the Network Operating System (NOS). The two essential functions of this layer are:

  - `1.` to be responsible for installing any needed control commands on the forwarding devices, as requested.
  - `2.` should gather the information about the forwarding layer, to be able to provide for the network applications, a global network view.

- **Specification:** The specification abstraction allows a control program to indicate the desired behavior of the network without about implementation of that behavior. This goal can be reached with virtualization solutions or network programming languages.

Recalling in Figure 2.2(b) layers, two of the always present layers were the southbound API and the northbound API, which are mentioned again in Figure 2.3. In Software-Defined Networking (SDN), Northbound and Southbound APIs are used to describe how interfaces operate between the different planes (data-plane, control-plane and application plane).

Northbound interfaces define the way the SDN controller should interact with the application plane. Applications and services are things like load-balancers, firewalls, security services, and cloud resources. The idea is to abstract the inner-workings of the network so that application developers can hook into the network and make changes to accommodate the application needs without having to understand exactly what that means for the network. Common examples include Representational State Transfer (REST) [2], Common Object Request Broker Architecture (CORBA) [3] and Simple Network Management Protocol (SNMP) [4]. Southbound interfaces define the way the SDN controller should interact with the data-plane (aka forwarding plane) to make adjustments to the network, so it can adapt better to changing the requirements. Common examples include OpenFlow [5], Network Configuration Protocol (NETCONF) [6] and Extensible Messaging and Presence Protocol (XMPP) [7] which is a communication protocol for message-oriented middleware based on XML (Extensible Markup Language), which the most recognizable of these interfaces at the moment is OpenFlow.

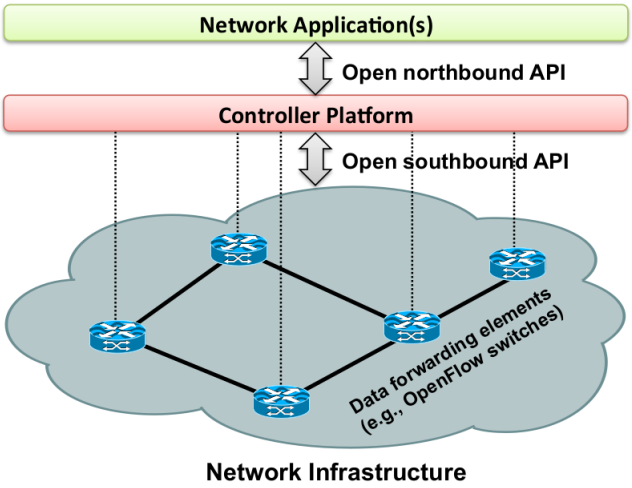


Figure 2.3. Simplified view of the SDN architecture (reproduced from [1]).

2.2.1 Open Flow protocol

The OpenFlow was the enabler of software-defined networking (SDN). It enabled the network controllers to determine the path of network packets across a network of switches. The separation of control from the forwarding allows for more sophisticated traffic management than is feasible using access control lists (ACLs) and routing protocols. Also, OpenFlow allows switches from different vendors, often each with their proprietary interfaces and scripting languages, to be managed remotely using a single, open protocol. The idea of the OpenFlow was initially proposed in 2006 at Stanford University. Open Networking Foundation(ONF) which promotes the adoption of SDN through open standards, released the OpenFlow v1.0 proceeding December 2009, as the first version of a communications protocol that gives access to the forwarding plane of a network switch or router over the network. It was started in a simple format, by having a single rules table which can match packets on a small number of header fields (e.g., MAC addresses, IP addresses, protocol, TCP/UDP port numbers, etc.). With different sets of rules, installed by the controller, an OpenFlow switch can behave like a router, switch, firewall, load-balancer, traffic-shaper, and any other middle-box. An OpenFlow-enabled switch is called an OpenFlow Switch [5]; Figure 2.4 shows a simple architecture for an OpenFlow switch.

OpenFlow switch is characterized by three components [5]:

- Flow table.
- Secured channel.
- OpenFlow protocol.

An OpenFlow switch may contain one or more flow tables, a secured channel that connects it to the controller, and OpenFlow protocol as the way for it to communicate

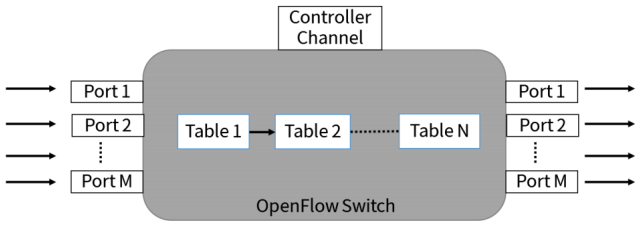


Figure 2.4. OpenFlow switch architecture (reproduced from [5]).

with the controller. A flow table is filled with the flow entries. The structure of the flow entry in OpenFlow 1.5 is shown in Figure 2.5.

Match Fields	Priority	Counters	Instructions	Timeouts	Cookie	Flags

Figure 2.5. Usual components of a flow entry in the OpenFlow 1.5 (reproduced from [5])

Each flow entry contains:

- **Match fields:** They are used to match against packet headers data.
- **Priority:** It clarifies the matching superiority of any of the flow entries.
- **Instructions:** A set of instructions that will be executed when a packet header data matches an entry.
- **Timeouts:** It is the maximum amount of idle time.
- **Cookie:** A non-transparent data value that is been chosen by the controller.
- **Flags:** They alter how the flow entries are managed.

When a packet arrives from a switch port, it is compared with the match fields in the flow entries. If the packet is matched, it will be processed as indicated in the instructions. The OpenFlow messages usually are transported on Transport Layer Security(TLS) or Transmission Control Protocol(TCP) connections. The simplest and most common messages in the OpenFlow protocol are Packet-in, Packet-out, and Flow-mod, which do the most of the job in this protocol. The Packet-in messages are originated from the switch and are destined to the controller, containing a copy of the packet (or possibly just the header), encapsulated as a packet-in message, to transfer the control of that packet to the controller. They are generated in the case of forwarding table misses. The Packet-out messages are made in the controller and will be sent to the switches, carrying the full packet or the buffer-ID of the switch that the original packet is held in it, to determine the action(s) that should be applied to the packet. The Flow-mod messages are again originated from the controller and are destined to the switch to modify the Flow-tables

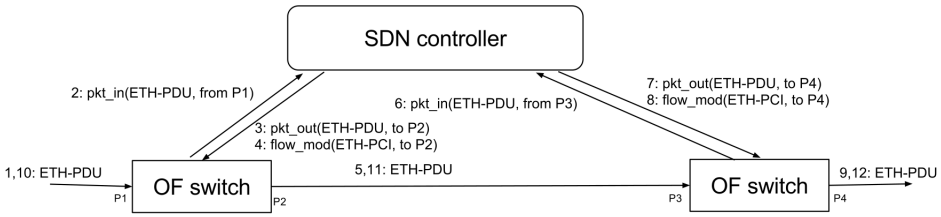


Figure 2.6. OpenFlow Packet-in, Packet-out and Flow-mode example diagram (reproduced from [8]).

while they carry the match-action rule(s) to be installed in the switch. An example of the above-mentioned messages is shown in the Figure 2.6.

The numbers before the packets show the order of the procedure.

- **(1)** By entering a packet that does not match any previously defined matches inside the switch tables.
- **(2)** The packet (or just the headers) is encapsulated inside a Packet-in message containing the information on the switch port the was entered from .e.g port P1, and will be sent to the controller.
- **(3)** The controller will examine the packet and will send a Packet-out message containing the packet or the Buffer-ID of the switch where the original packet is stored in (in the case of just headers) and the determined action e.g., (send to port P2).
- **(4)** A Flow-mod message then will be sent from the controller to the switch, to install a match-action rule for the similar packets .e.g (send to P2).
- **(5)** The switch then will send the packet out of port P2.
- **(6), (7), (8)** and **(9)** Now the packet enters to the port P3 of the second switch, which does not match to any previously defined matches inside the switch tables again, so the steps **(2), (3), (4)** and **(5)** Will be repeated.
- **(10), (11)** and **(12)** Now if a packet with those matches which previously was installed in steps 4 and 8 enters the switch, without any further communications with the SDN controller, through the related actions will be forwarded through ports P2 and P4

Considering the fact that there are many ways to program the forwarding elements, but having a common, open and vendor-independent interface (like OpenFlow) which allows a control-plane to control forwarding devices from different hardware and software vendors, will be more convenient. Nonetheless, the need for supporting the new header fields is increasing(see Figure 2.7). As an example, if the operators of a Data-Center

network, need to add to OpenFlow protocol new forms of packet encapsulation (e.g., STT [9], VXLAN [10], and NVGRE [11]), for which they resort to deploying the more flexible case, which is the software switches that are easier to extend with new functionality.

Version	Fields	Year
OpenFlow 1.0	12	2009
OpenFlow 1.1	15	2011
OpenFlow 1.2	36	2011
OpenFlow 1.3	40	2012
OpenFlow 1.4	41	2013
OpenFlow 1.5	44	2015

Figure 2.7. Header fields grow in the OpenFlow (reproduced from [8]).

Over almost a decade, the specifications have become more complicated and the match fields have been grown,(see Figure 2.8) so that with much more header fields and multiple stages of rule tables, allow switches to expose more advanced capabilities to the controller plane.

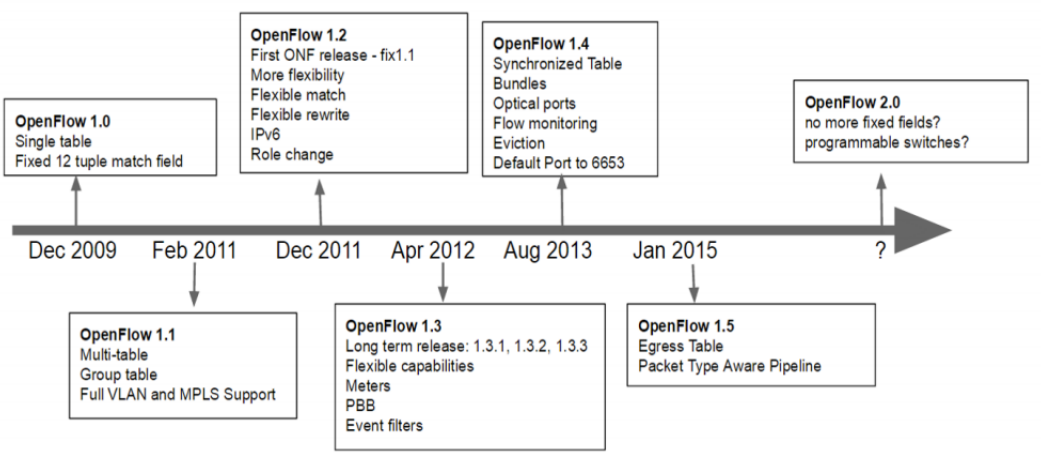


Figure 2.8. OpenFlow growing road-map (reproduced from [12]).

2.2.2 Stateful SDN

The very first big step of the SDN was to separate the control-plane from the data-plane. Introducing the Open-Flow, the separation results in having the forwarding plane represented as Dumb switches, which are processing packets by the match/action rules (flow entries) installed by the controllers as the Brain. The controller is responsible to change these rules due to needs and rewriting them to make the switches for reacting properly to changes of the states of the network. Anytime a switch is unable to match a flow or

packet to its table rules, it will send it to the controller and waits for the orders. This approach allows simple network programming, and a simplified view of the network as a one distributed big switch, but it has some drawbacks. especially, there can be a relatively long processing delay, caused by information exchange between switch and controller, which can be critical in cases of, for example, network outages or traffic management applications. In stateful Software Defined Network (SDN) data-planes, network switches hold some local flow-related states thanks to which they can perform decisions by locally executing simple algorithms. Stateful data-planes provide better reactiveness comparing to vanilla SDN. In stateful data-planes, switches are provided with the ability to take some local decisions based on the internal states, without any need for interaction with the SDN controller. This ability enables an enhanced level of programmability of the network comparing to standard SDN paradigms such as the ones based on OpenFlow. One of the possible ways of the advancement of the OpenFlow, according to [13], is to introduce a switch-driven adaptation of forwarding rules, based on the switch-local events. In OpenState [14] proposed a minimal architectural extension to the OpenFlow data-plane and control-plane to identify the flow to which a packet belongs to and to retrieve/update the associated state while it keeps the central SDN controller completely in control of all assigned tasks. The goal of the Open-State is to move some simple controlling tasks, which only needs local knowledge from the switch, out of the controller, while the controller remains responsible only for decisions that need network-wide knowledge. In OpenState, some custom states can be configured inside the switch, which is triggered by packet arrivals, measurements, and timers. SNAP [15] is a novel network programming abstraction, which allows defining quite complex network applications for stateful SDN and solves the problem of how to optimally place the states across the network switches, taking into account the dependency between states and the traffic flows. By design, SNAP is limited to just one replica of each state within the network. Another relevant work to the state replication in the stateful data-planes is described in Swing State [16], which introduces a mechanism providing state migrations entirely in the data-plane but, similarly to SNAP, assumes only one state that is on-demand migrated across the network. Regarding the implementation of stateful SDN, Open Packet Processor (OPP) [17] extended OpenState by adding additional features that allow the executions of Extended Finite State Machines (EFSM) directly in the data-plane. Forwarding metamorphosis [18] proposes a switch chip implementation based on the Reconfigurable Match Tables (RMT) model that permits, even if with some limitations indicated by the same authors, to manipulate some state within its pipeline. Finally, LODGE [19], which is a model according to which distributed network applications can make local decisions at each switch, based on some global variables shared across other switches, enables multiple replicas of the state, extending the single replica approach in SNAP.

2.3 Data-plane programmability

2.3.1 Overview

It is better to make future switches to support flexible mechanisms for parsing packets and matching header fields, allowing controller applications to use these capabilities through a

common, open interface, rather than extending the OpenFlow specification. This general, extensible interface would be much simpler and more future-proof than the OpenFlow standards. Modern chip designs demonstrate that such flexibility can be reached in custom ASICs at terabit speeds.

2.3.2 PISA architecture

By taking a look in the OpenFlow protocol and the current switching chips, one can see that the current hardware switches are completely rigid, which allow the process of the Match-Action rules on only a fixed set of fields, while the OpenFlow specifications have only defined a limited collection of packet processing actions. To overcome these limitations one article [18] proposed the re-configurable match tables (RMT) model, as a new RISC-inspired pipelined architecture, used for switching chips that identified the indispensable minimum set of the action primitives for specifying that, how headers are going to be processed in the hardware. As an advantage in the RMT model, one can have a set of pipelined stages, while in each stage, it can have a match table of arbitrary depth and width. Regarding the reconfigurability, field definitions can be changed or new fields can be added, new actions can be defined(similar to creating new fields) and with some consideration in resource limits in the hardware; number, depths, widths, and topology of the match tables can be specified. All of this process of the configuration is managed by the SDN controller. The RMT model as a sequence of logical match-action stages is shown in Figure 2.9

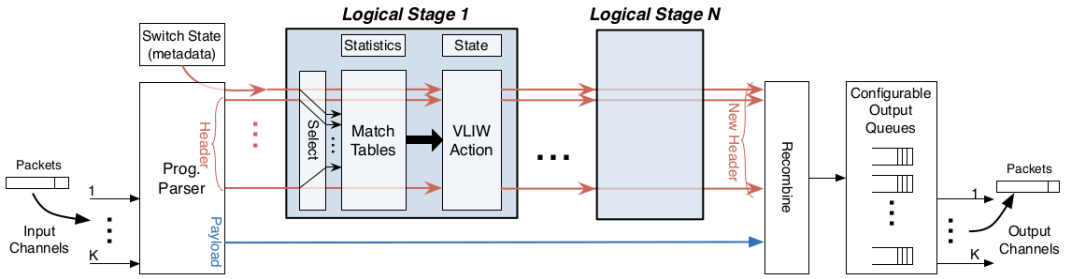
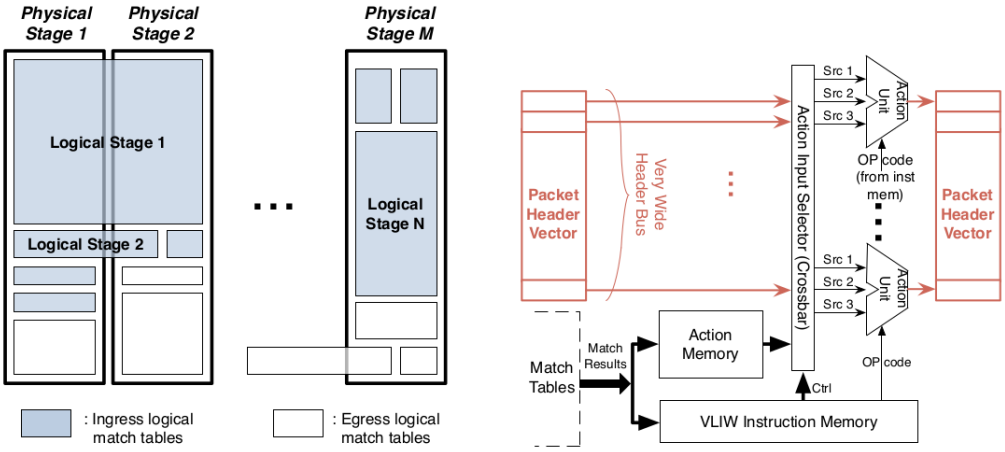


Figure 2.9. RMT model as a sequence of logical Match-Action stages (reproduced from [18]).

An entered packet to the switch will enter the programmable parser, where a vector of the packet header is separated from the payload. The word metadata is been used for all the information provided by the switch which that information is not available inside the packet header vector, such as the input port of the packet or the information on switch queues, etc. The packet header vector with the mentioned metadata will flow through a sequence of logical match stages, where each of them abstracts a logical unit of packet processing (e.g., IP processing), while the untouched payload with the packet header vector outputted from the last logical stage is merged to build the outgoing packet. The implementation of the RMT resulted in an architecture called Protocol-Independent Switch Architecture (PISA) [20], which allows modifying packet header vector through a wide instruction (VLIW - very long instruction word) that can operate on all fields in

the header vector concurrently, resulting in very high speed. Then by flexible resource allocation, they minimized the waste of resources knowing that a physical pipeline stage has some resources (e.g., CPU, memory) that it really needs, e.g., a firewall may require all ACLs, a core router may require only prefix matches, and an edge router may require some of each.(see Figure 2.10)



(b) Flexible match table configuration.

(c) VLIW action architecture.

Figure 2.10. Flexible match table configuration and VLIW action architecture (reproduced from [18]).

The Portable Switch Architecture (PSA) is a target architecture that describes common capabilities of network switch devices that process and forward packets across multiple interface ports. The Portable Switch Architecture (PSA) Model has six programmable blocks and two fixed-function blocks, as shown in Figure 2.11.

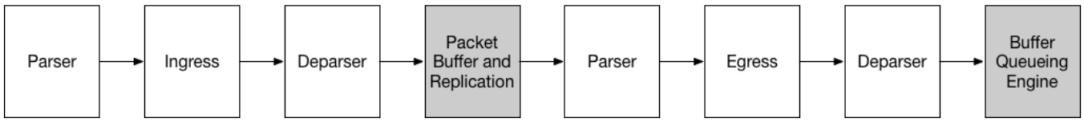


Figure 2.11. PSA: Portable Switch Pipeline (reproduced from [21]).

The behavior of the programmable blocks is specified by using the P4 language (will be described later in section 2.3.3). The Buffer Queuing Engine (BQE) and the Packet buffer and Replication Engine (PRE) functional blocks are target-dependent and they can be changed for some fixed set of operations. For each incoming packet, the headers are parsed and then validated, then they will be passed to an ingress match action pipeline, that can make some decisions on the path the packets will move on. An ingress deparser P4 code clarifies the contents from the packet that should be sent to the packet buffer and also determines which metadata of the packet should be carried with it. After the ingress pipeline, if needed the packet can be replicated for different reasons, and lastly, it will be

stored in the packet buffer. For each such egress port, the packet should pass through an egress parser and match action pipeline before it becomes deparsed and queued to leave the pipeline.

2.3.3 Programming Protocol-independent Packet Processors (P4)

Programming these kinds of switch chips, mentioned in section 2.3.2 is not very easy. The article [22] proposed a higher-level language for protocol-independent Switch Architecture (PISA) with the name **Programming Protocol-independent Packet Processors** or **P4**. P4 was used to configure the switch, by teaching it to know how packets should be processed, while the previously designed APIs (such as OpenFlow) is for populating the forwarding tables in fixed-function switches(see figure 2.12. According to the article [22], the authors' opinion is that the P4 can increase the level of abstraction for programming the network devices. Also, P4 language allows the SDN controller to be able to program the switch even on the switch itself, which makes the controller independent of the fixed function design of OpenFlow switches, and by this advances flexibility of operations of the network.

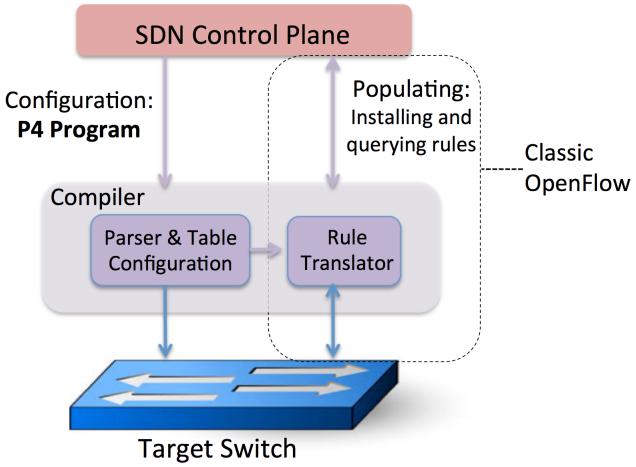


Figure 2.12. P4: a language to configure switches (reproduced from [22]).

The authors of the P4 language, tried to follow three goals in their design [23]:

- **Re-configurability:** The mentioned controller should be able to redefine the packet parsing and processing of already deployed switches, right in the field.
- **Protocol independence:** The target switch should not be designed just for specific packet formats. Indeed, the controller should be able to specify:
  - A kind of packet parser to extract the header fields with particular names and types.

  - A collection of the typed match plus action tables that are needed to process these headers.

- **Target independence:** As a C programmer does not need to know the specifics of the CPU, the controller programmer should not need to know the details of the switch. Instead, a vendor-supplied compiler should take care of the switch capabilities during the compilation of a target-independent description, written in P4, into a target-dependent program (used to configure the switch).

P4 is a language that describes how packets should be processed with the data-plane regarded to a programmable forwarding element free from being a hardware or a software switch, network interface card, router, or network appliance. P4 seems to be designed for only programming the switches, but its final goal has been grown to cover a diversity of network devices, such as ASIC, FPGA and in general any kind of network elements that can implement both a control-plane or a data-plane functionality. It was designed for clarification of the data-plane functionality for a target. As an example, Figure 2.13 shows the differences between a fixed-function regular switch with a P4-programmable switch. In a regular switch, the data-plane functions are constantly defined by the manufacturer, and the control-plane by managing entries in the tables will control those functions, might configure specialized objects (e.g., counters), and by means of some packets that are called the control-packets (e.g., routing protocol packets) or even other events, like changing the link states. In Figure 2.14 is illustrated a logical abstract for how the switches forward the

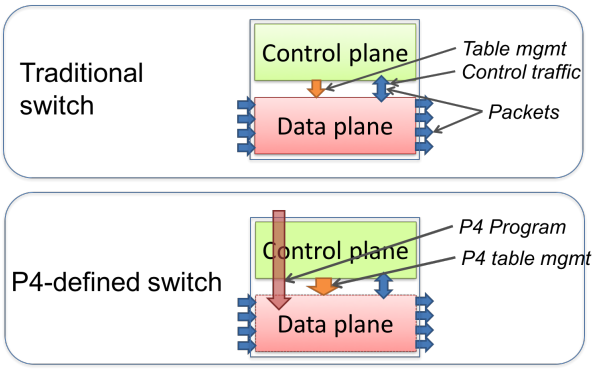


Figure 2.13. Difference between traditional and programmable switches (reproduced from [23]).

packets through a programmable parser which is followed by multiple stages of match-action, arranged in series, parallel, or a combination of both.

The difference between a traditional switch and a P4-programmable switch can be described in two things [23]:

- There are no predefined fixed functions in the data-plane, but they will be defined by the P4 program, at the time of the initializing the device. For building the functionality described by the P4 program, the data-plane that has no built-in

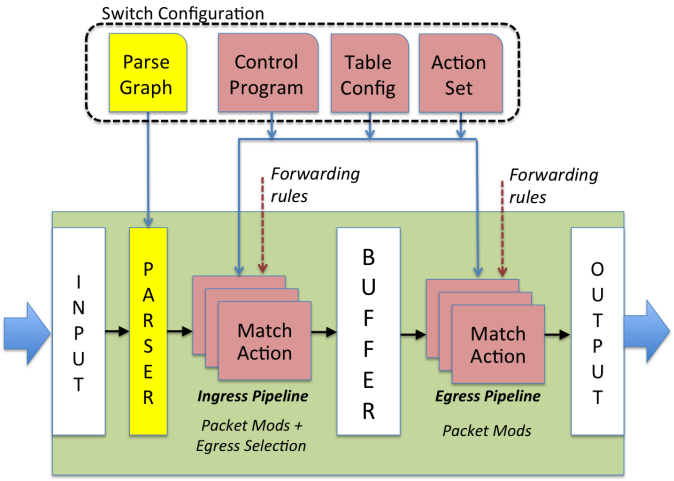


Figure 2.14. The abstract forwarding model (reproduced from [22]).

knowledge of existing network protocols will be configured, which is shown by the long red arrow.

- Then the control-plane can communicate with the data-plane with the same structure exists in a fixed-function device, while there is no longer any fixed tables and other objects in the data-plane, as they are defined by a P4 program. Then the control-plane will communicate with the data-plane through the API that the P4 compiler will generate.

There are some core abstractions that are provided by the P4 language [23]:

- **Header types:** They will describe the set of fields and their sizes of each header existing inside a packet.
- **Parsers:** THey will describe the allowed sequences of headers inside the received packets and describe how to identify their sequences and their fields to extract from the packets as well.
- **Actions:** They are some codes that describe how a packet header fields and their metadata should be manipulated. They can have some data, which will be supplied at the runtime by the control-plane.
- **Tables:** They will relate some user-defined keys to some actions. P4 tables are a general version of the traditional switch ones; they can be used to implement any user-defined table types, even including complex multi-variable decisions over the packets.

- **Match-action units:** They will perform these tasks:
  - They build lookup keys from computed metadata or the packet fields.
  - They do the lookups inside the tables with the help of the previously built keys, then they choose an action with the associated data (if any)to execute.
  - At last, they will execute the selected action.
- **Control flow:** THey will indicate an authoritative program that describes the way of the packet-processing for a network target, as well as the sequence of match-action unit invocations which are data-dependent. The deparsing can be done by using the control flow.
- **Extern objects:** They are the architecture-specific constructs that the P4 programs can change them through well-defined APIs, of course, their internal behavior is hard-wired and can not be programmed using P4.
- **User-defined metadata:** They are the user-defined data structures that are associated with each packet uniquely.
- **Intrinsic metadata:** THey are the metadata that has been provided by the architecture that is associated with each packet.

Any manufacturer that builds the targets will provide the software implementation framework or the hardware plus an architecture definition, and a specific P4 compiler for their target. Then the programmer will write its P4 program for one specific architecture, that explains a set of components that are P4-programmable for the target plus their external data-plane interfaces. Figure 2.15 illustrates a general workflow for programming a target using P4 language.

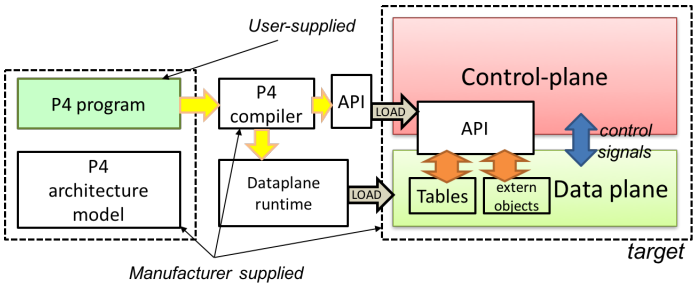


Figure 2.15. Programming a target with P4 (reproduced from [23]).

The P4(16) which is the second released version for the P4 language (v1.1), comparing to the previous version of the language (v1.0), so-called P4(14), made some big backward-incompatible changes to the syntax and semantics of the language. One can see the evolution from the previous version (P4 14) to the current version (P4 16) as is demonstrated in Figure 2.16.

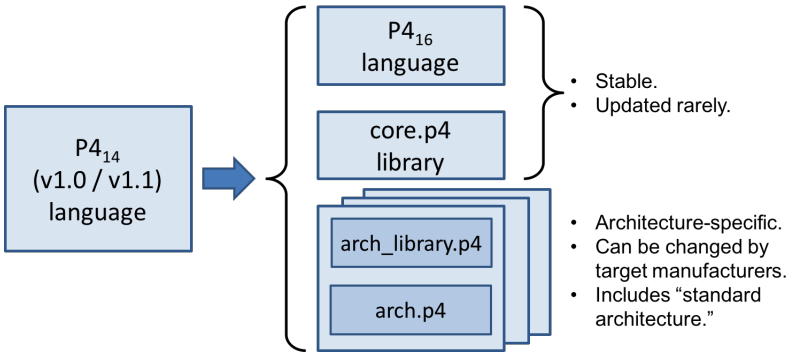


Figure 2.16. P4 Runtime: API Reference Architecture (reproduced from [23]).

Some of the language features have been moved into some libraries and the language itself, rather than a complex language has become a small core language. More than 70 keywords have been reduced to less than 40 keywords. The P4(16) language has a library of fundamental constructs that are needed for writing most P4 programs. The v1.1 version of P4 introduced a language construct called extern that can be used to describe library elements. Many constructs defined in the v1.1 language specification will thus be transformed into such library elements (including constructs that have been eliminated from the language, such as counters and meters). P4(16) also introduces and re-purposes some v1.1 language constructs for describing the programmable parts of an architecture. These language constructs are parser, state, control, and package. The revision of the P4 language was an effort to ensure that all programs written in P4(16) will remain syntactically correct and behave identically when treated as programs for future versions of the language. Compared to state-of-the-art packet-processing systems (e.g., based on writing microcode on top of custom hardware), P4 has some important advantages:

- **Flexibility:** The P4 language can express packet-forwarding rules as some programs, in comparison to the traditional switches, which show fixed-function forwarding rules to their users.

- **Resource mapping and management:** P4 programs describe in an abstract way the storage resources, the compilers will map these user-defined fields to available hardware resources and will handle low-level details such as scheduling and allocation.

- **Expressiveness:** The P4 language can do complex packet processing algorithms by using only the general-purpose operations and look-up tables. These programs are portable across devices that have similar architectures.

- **Software engineering:** The P4 program has significant benefits like hiding the information, checking the type, and reusing the software.

- **Component libraries:** The component libraries which are supplied by some manufacturers can be used to convert some hardware-specific functions into the portable high-level P4 constructs.
- **Decoupling hardware and software evolution:** Using the abstract architecture by the target manufacturers can help to more decoupling the evolution of high-level processing from low-level architectural details.
- **Debugging:** Providing software models of architecture by the manufacturers can help the debugging and development of P4 programs.

2.3.4 P4 Runtime

P4 Runtime is an API, which provides a new way by which, the control-plane software is capable to control the forwarding plane of a network device like switches, load-balancers, firewalls routers, etc. Maybe, the most fascinating aspect of the P4 Runtime is its ability, to allow us to control any forwarding plane, independently of whether it is built from a fixed-function or programmable switch ASIC, an FPGA, NPU or a software switch running on an ordinary x86 server. The framework of P4 Runtime stays unchanged and remains independent of forwarding plane capabilities, which of protocols and features the forwarding plane supports. The same API can be used to control a huge variety of different switches, whilst, as new protocols and features may be added to the forwarding plane, the P4 Runtime API will be automatically updated by changing the control scheme to describe how the new feature is to be managed, without restarting or rebooting the control-plane itself. P4 Runtime is completely independent of the placement of the control-plane. The control-plane could be a protocol stack running on a local switch operating system (switch OS), or a remote control-plane, running on x86 servers. The representation of the P4Runtime Reference Architecture is illustrated in Figure 2.17.

The target which is meant to be controlled is illustrated at the bottom of the picture, one or more controllers are shown at the top. A multi-master protocol allows more than one controller to participate, and a role-based arbitration scheme ensures only one controller has the write access to each read/write entity, or the pipeline configuration itself. Any controller may perform read access to any entity or the pipeline configuration. The P4Runtime API defines the semantics and the messages of the interface which is in the way between the server and the client. The API will be defined by the (p4runtime.proto) Protobuf [25] file, which is available on GitHub as a part of the standard [26]. It can be compiled by the Protobuf compiler, to generate implementation stubs for both the client and the server in a variety of languages. It will be the target implementer's responsibility for instrumenting the server. A future goal of the p4 consortium is to produce a reference gRPC [27] server which can be instrumented in a generic way, to reduce the burden of implementing P4Runtime. The controller can have access to the P4 entities which are defined in the P4Info metadata. The P4Info format is defined by the p4info.proto file, which is another available Protobuf file. The controller has also the ability to set the `ForwardingPipelineConfig`, which is used to install the compiled P4 program output and is included in the p4-device-config Protobuf message field, with installing the related P4Info metadata. At last, the controller can

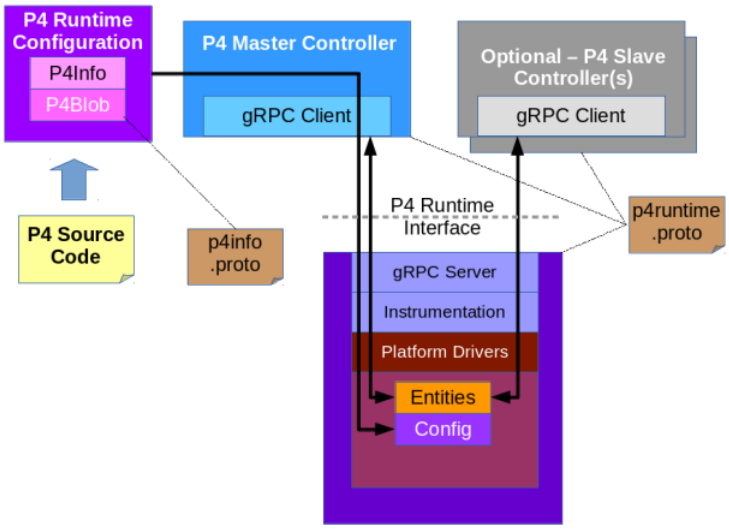


Figure 2.17. P4 Runtime: API Reference Architecture (reproduced from [24]).

check the target device for the `ForwardingPipelineConfig` to have the device config and the P4Info. The P4Runtime is able to handle more than one controller at the same time. Its mechanisms are explained well in its official document,(P4 Runtime: API Reference Architecture) [24]. Each of those use-cases shows a different aspect of the flexibility regarded to the P4Runtime. The different cases might mix different techniques and can be more complex. Here we show some use-cases. The Single Embedded Controller is shown in Figure 2.18. The Embedded Controller case, reminds us of the device or target, which has an embedded controller, that can communicate via P4Runtime with the on-board switch chipset. This might look appropriate for an embedded appliance that is not intended for SDN. P4Runtime is claimed to be a feasible embedded API that can operate as an ideal RPC and if needed as a viable IPC.

Now, the switches are generally being controlled by their proprietary APIs, that their owners prefer them to be closed and most of them have a set of predefined functions. These APIs are baked to the target chip of the device and to cover the needs, while there is almost no need for extending the API by passing the time. The software distribution ways, especially, license agreements and NDAs mostly forbid sharing their API with the others, which leads to being impossible to control switch ASICs from another chip vendor by using different APIs. So, adding new features and protocols becomes difficult, as one network owner cannot benefit from different features of another vendor, resulting in feeling constrained for innovation. By specifying the behavior of a switch in the P4 language, the P4 Runtime API is capable to be used for controlling any switch from any vendor. P4 Runtime can even be used for controlling the existing switches even with the fixed-functions. The developer needs to write the P4 program so that it documents the behavior of the switch, by using the P4 language. Then, the P4 compiler will automatically identify

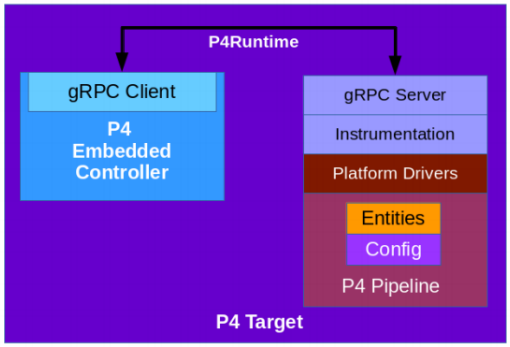


Figure 2.18. Single Embedded Controller (reproduced from [24]).

elements which are needed to be controlled, like the lookup tables defined in the P4 program and to which we need to delete and add the needed entries. The programming process of the network devices is possible with P4 language and the P4 Runtime API, they even increase the flexibility of this process to a new higher level.

2.4 NFV and 5G network

The Network Function Virtualization (NFV) and Virtualized Network Function (VNF) are very close terms in the modern network paradigms. As said above, the VNF is used for the virtualized network function, and generally is the software form of the network tools. it can act as a router, firewall or even a load-balancer. They are most of the time as some virtual machines on Linux KVM or hypervisors, on commercial off-the-shelf hardware. A physical network function refers to the traditional network tools on some proprietary hardware. The NFV means network function virtualization. It is about the operational framework which is automated and orchestrate the VNFs on some virtualized infrastructure on the commercial of the shelf hardware while managing VNF tools through their whole lifecycle. The NFV strongly depends on software-defined networking concepts. Those concepts are the separation of the control-plane from the data-plane plus the Management and Orchestration plane or the MANO. The Linux Foundation and the European Telecommunications Standards Institute actively trying to develop the standards and the reference architecture for the NFV framework. The most important open-source NFV projects are the Linux Foundation Open Network Automation Platform (ONAP) and the ETSI Open Source MANO (OSM). Following ETSI NFV Framework provides a great illustration of the relationship between VNF and NFV(see Figure 2.19).

NFV MANO is responsible for:

- interacting with operations and business support systems (OSS/BSS) to deliver business benefits to service providers, such as rapid service innovation, flexible network function deployment, improved resource usage, and reduced CapEx and OpEx

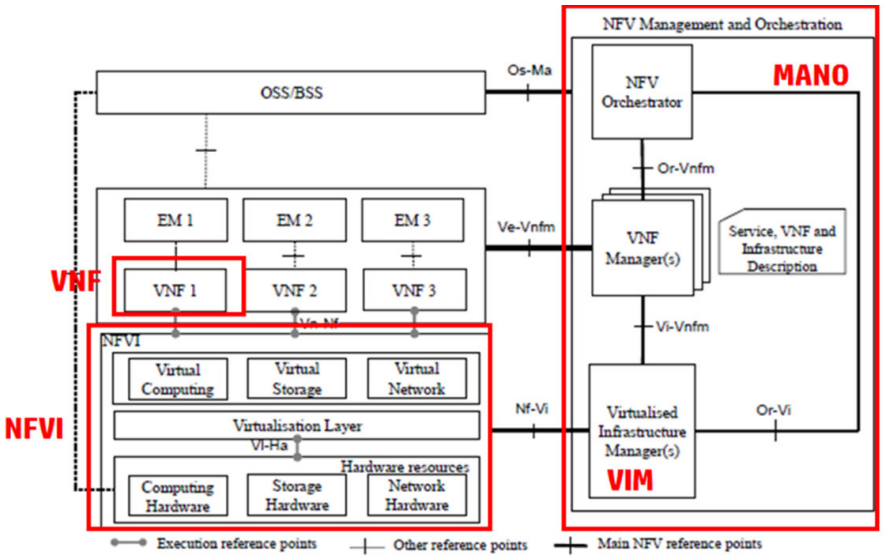


Figure 2.19. ETSI NFV Architectural Framework (reproduced from [28]).

costs;

- orchestrating VNFs into network services (NS), deploying and operating the VNF and NS instances on the virtualized resources, and managing the lifecycle of VNF and NS instances to fulfill the business benefits for service providers;
- interacting with element management (EM) to manage the logical function and assure service levels of the VNFs spanning across the management of VNF fault, configuration, accounting, performance and security (FCAPS);
- interacting with network function virtualization infrastructure (NFVI) to allocate, manage and orchestrate the virtualized resources including compute, storage and network, where VNFs are deployed.

One of the important needs for 5G networks is high flexibility. This means that it should be able to make new services and launch them without the need for changing the physical network, e.g., cables for connections, servers, and other network resources. The network should have the flexibility to adapt to the traffic patterns changes for day and night, weekends and weekdays, etc. By considering some IoT sensors like gas, electricity or water meters, that report their data at the end of the last day of each month, causing huge traffic over links, while they will generate very little traffic in all other times. So e.g., every month, at a specific hour, we need the 5G network to provide a huge capacity for these reports, while at the other times the common resources will be used for other needs. It is somehow attractive, but the need for managing a large variety of physical resources over different locations makes it again complex. By making a network virtualized and programmable, it will allow us to handle very well the complexity of the network orchestration and management. NFV and SDN are the best means to create different flexible

logical networks consisted of some virtualized network functions that are connected with some virtual links while installing on top of a programmable infrastructure. A fundamental and elemental change would be happened to the Telecom industry by this, from both the business and operational point-of-view. This kind of scalability and flexibility are the very important requirements in 5G networks, and along with ultra-low latency, the ultra-high capacity, and even the ability for supporting a huge number of concurrent sessions and users/things. To this end, Radio Access Network (RAN) Cloudification, Multi-access Edge Computing (MEC) and Network Slicing, are the most important enablers, which all of them are based on the capabilities of the SDN and the NFV. At present, the NFV framework is undergoing rapid development because of 5G business opportunities, and its ecosystem is growing with strong support from service operators and all varieties of solution providers.

Chapter 3

NFV state sharing

3.1 The proposed approach

There are some NFs, that are placed all over a network inside some servers, and we want to have an arbitrary composition of the state sharing among them. Each server may host more than one NF, while none of them are aware of the existence of the other NFs, and they have no specific information about the placement of the other NFs in the network. The variables, which their data is going to be shared, will be mapped to some unique variable-IDs. Each NF that wants to have access to any variable, will express its interest in that variables by using the corresponding variable-ID and some instructions embedded inside some special packets. In our proposed solutions, through a protocol, that will be explained in the next sections, the NFs will subscribe for data of the variables on the P4-programmable switches, publish data on variables and eventually share their local data on the variables. We decided to use the concept of the `publish/subscribe` as the key rule player for our proposal communication way between the NFs and the network. In our proposed schemes, the `Broker` task of the `publish/subscribe` paradigm, has been entrusted to the P4-programmable switches, where they record the requests by a procedure similar to what is done in a MAC learning algorithm. In the MAC learning paradigm when a packet enters a switch, the switch maps the source MAC Address of it to the input port of the packet in a key-value respectively. The switch can use that port number later for forwarding another packet which enters the switch with the destination MAC Address equal to this key. The differences in our case are:

- Our switch uses the `variable-ID`, which will be explained in the next sections, as a key for mapping information, instead of using MAC addresses.
- It uses a `MultiCast-Group-ID`, as will be explained in the next sections, as the value for the corresponding variable-ID key, instead of the port number.

3.2 Publish-Subscribe Model

Nowadays, Networking technologies enable a very high degree of connectivity among a large number of computers, applications, and users. In such environments, it is very

important to provide asynchronous communications for the group of distributed systems, that works in a loosely-coupled and autonomous way, where they need to have immunity from the network failures. One of the answers to this important need is the publish-subscribe model. The Publish/subscribe scheme is a pattern by which the exchange way of the messages between publisher and subscriber clients is done. Subscribers express interest in receiving messages and publishers simply publish messages without specifying the recipients for a message. In a same manner, the receiving applications or subscribers must receive only those messages that they have registered an interest in. The exchange of the publish/subscribe messages is anonymous and decoupled. This decoupling between senders and recipients is done by an entity that is placed in the route between the publisher and the subscriber, and the anonymity means, the publishers, should not know the subscribers identity or if any subscribers with matching interests does exist at all. This structure can support the many-to-many communication paradigm, where some sources are publishing the data and some receivers that needs those data will subscribe on them. Many types of the publish/subscribe schemes are explained. They mainly differ in the way that subscribers show their interest in some data, in the format and the shape of the messages which carry the data, in the architecture, and the degree of decoupling that they usually support. A simple example case for the publish/subscribe model is shown in Figure 3.1.

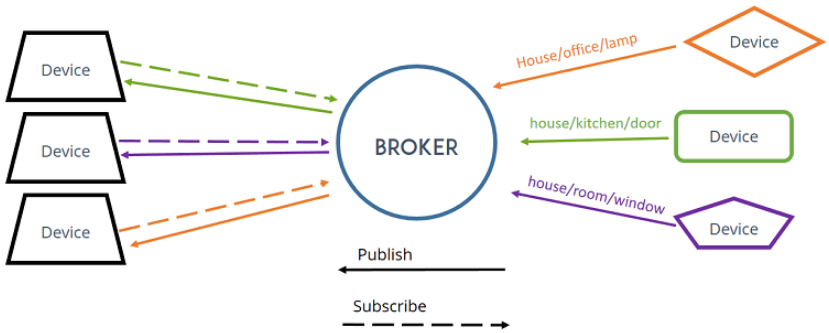


Figure 3.1. MQTT: Message Queueing Transport Telemetry scheme, a sample of publish/subscribe model (reproduced from [29]).

Here the broker or queue in some contexts is responsible to register the subscriptions requested by the subscribers, for requested topics. It also should deliver the published messages from the publishers to the subscribers over those published topics. The concept of the publisher and subscriber is logical. It means that a subscriber can be a publisher at the same time, on a different topic. The broker or queue capabilities depends on the design goals and can be very simple or very complex. Most of the simple design cases are used for constrained devices in the IoT frameworks, as they have limitations on power, and for the low-bandwidth or unreliable networks. In the simplest form, the publishers are publishing anytime they wish, by sending their messages to the broker. Any message from a publisher has an identification data, which we call it topic identification data, to be recognized and distinguished by the participants in the system. Anytime a subscriber wants to receive messages on one specific topic, it will send a registration request with

the topic identification data and some data about itself, that already been agreed on its structure previously in that system, to the broker. If already agreed in that system, those self introducing data can be sent by the publisher through the publish phase too. Anytime the broker receives a published message, it will check its registered subscriptions and will deliver the published message to those subscribed on that message through its topic identification data. The qualitative and quantitative characteristics of the scenario, including the publish, register, subscribe, deliver and any other customized steps, is completely design dependent. For example, the broker can store publishes for probably future subscription on them or simply discard them if no subscription is registered for them, It can make sure the subscriber is received the publishes by some algorithm, or just try one attempt without any further information about correctness of the delivery process.

3.3 State sharing protocol

In this section, we will describe our proposed protocol with the standard needed information for the steps and the algorithms of each step.

3.3.1 State sharing phases and general packet structure

Our state sharing protocol is recognized by two main phases:

1. Initializing phase
2. Data Replication phase

Every NF starts with the Initializing phase including an initial information request, which if succeeded, then it will move to the Data Replication phase, which due to the conditions might need to do the information request again. The information request can be considered as a sub-phase merged into both the Initializing phase and the Data Replication phase. The overall life-cycle diagram for an NF in our scheme is shown in Figure 3.2 and Figure 3.3.

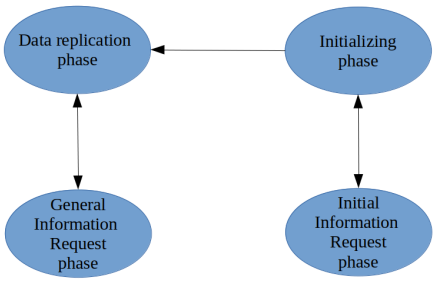


Figure 3.2. NF Life-cycle Diagram1.

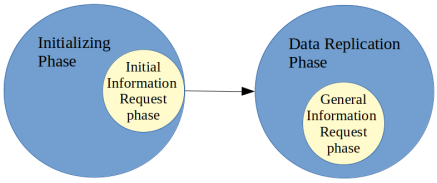


Figure 3.3. NF Life-cycle Diagram.2.

For the packet structure, we selected the standard IPv4 with UDP as the Transport layer. In the destination field of IPv4 Address, we used IP class D, dedicated for Multi-Cast purposes, and especially it's third group, Locally-Scoped MultiCast Address range, starting from `239.0.0.0` to `239.255.255.255`. This is due to carry some special information about our data, to later be used by some special parts of the network, including the REPLICA controller, the NFs and those REPLICA switches which are part of our proposed solution. In this Sub-Class the first 8-bits are constant, so we used the next 2-bits as the `flags` part for our design and the remaining 22-bits will be used as the variable-ID. We chose UDP for the Transport layer to support MultiCast for our `publish packets`. In the next sections, the reason for our choices will become more clear. The port number 65432 (0xff98) is chosen as the destination port in the UDP header for recognition of our special packets in the NFs and in the network. We defined seven(7) message kinds for this design due to different tasks:

1. `INIT-NF-ID request`
2. `INIT-PUB-VAR-ID request`
3. `PUBLISH`
4. `SUB-VAR-ID request`
5. `SUB-Register`
6. `SUB-Remove`
7. `RECOVER`

The `INIT-NF-ID request`, the `INIT-PUB-VAR-ID request` and the `SUB-VAR-ID request` are used in the Initializing phase and it is a sort of Information request from NF to the REPLICA controller. The `SUB-register`, `SUB-remove`, `PUBLISH`, and `RECOVER` packets are used in the Data Replication phase. They will be described in the section 3.3.2. The partitioning shape of the IPv4 IP-destination format for the protocol is described in Figure 3.4.

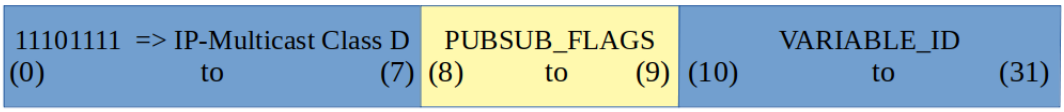


Figure 3.4. IPv4 IP-destination format for the protocol .

The bit usage of the IPv4 IP-destination format for the protocol is described in Table 3.1.

The destination for the IP layer will be generated from the variable-ID and FLAGS, based on the structure described above. The mac destination of the Ethernet layer will be set to `ff:ff:ff:ff:ff:ff`, differentiating our special packets from normal IP-MultiCast packets.

Bit Number	Description
0 to 7	**Constant:** (1110,1111)
8, 9	**PubSub-Flags**: 11 = `SUB-register` 10 = `SUB-remove` 01 = (`INIT-NF-ID-request`, `INIT-PUB-VAR-ID request`, `SUB-VAR-ID Request`, or `RECOVER`) 00 = `PUBLISH`
10 to 31	`variable-ID`

Table 3.1. IP Destination fields in details.

3.3.2 Initializing phase

In the Initializing phase, anytime an NF starts, it needs to be identified in the network by the REPLICA controller, and itself with a `Global-ID`, while the `Global-ID` will be assigned by the REPLICA controller to be unique for each NF in that network. This `Global-ID` will be added to the internal data structure of the NF as well as the REPLICA controller. The Initializing phase can be considered as the main information request in our system, and the reason of why we separated it from the Information Request phase, is its importance and that it happens once in the life-cycle of the NF.

Information Request

As mentioned before, it can be considered as a sub-phase merged in the other two phases, or as a different separate phase in relation with them. In the Information Request phase, the NF sends a request to the REPLICA Controller and will receive a response containing the information on its request. The NF will update its internal data structure based on the response from the REPLICA controller. The NF then, will be able to use these data in it's tasks e.g., `Subscribe` or `PUBLISH` in the `Data Replication phase`. The `INIT-NF-ID request`, `INIT-PUB-VAR-ID request` and `SUB-VAR-ID Request` are different kinds of the information requests.

Packet format

The NF uses the `INIT-NF-ID request` to ask for it's`Global-ID` from the REPLICA controller, or ask for a variable-ID through a `INIT-PUB-VAR-ID request` or a `SUB-VAR-ID Request` as explained in the Table 3.1. In both cases they are distinguished in the REPLICA switches through the `PubSub-Flags`, and will be forwarded to the REPLICA controller. Their data should carry some information about the variable names or other information that may be needed in the REPLICA controller and can be in any format, e.g., JSON. On the other hand, the replies from the REPLICA controller should contain the same information and the assigned values to the requested information.

Information request algorithm for Initializing phase

Depending on the `INIT-NF-ID-request` or the `VAR-ID requests`, the Time-Space Diagram for Information Request phase is shown in Figure 3.5 and Figure 3.6 respectively.

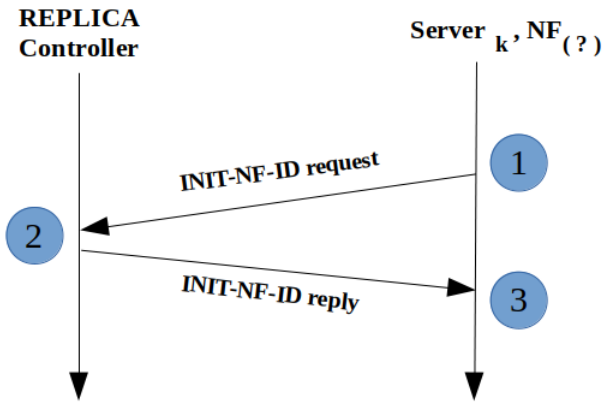


Figure 3.5. Time-Space Diagram for Information Request phase for `Initializing phase`.

- **(1)** Newly started NF wants to initialize, so it will send an `INIT-NF-ID request` to the REPLICA controller.
- **(2)** The REPLICA controller will assign a `Global-ID` in its data structure to the NF while updating its internal data structure, and replies to the request with the `Global-ID` added to the information were supplied on that request, to the NF.
- **(3)** The NF will store the received information inside its data structure.

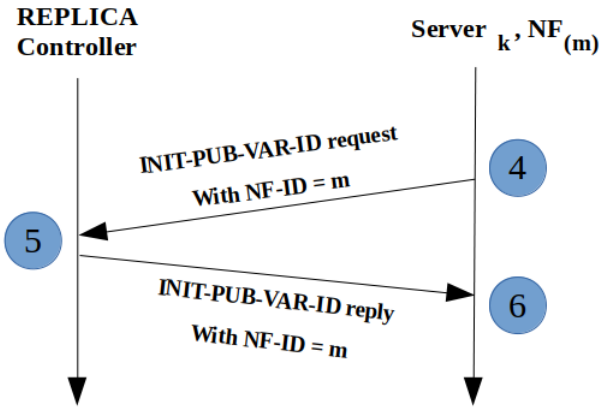


Figure 3.6. Time-Space Diagram for Information Request phase `Initializing phase`.

- **(4)** The initialized NF wants to `PUBLISH` the data of a variable or wants to `SUBSCRIBE` on data of a variable, so it needs to know the corresponding assigned variable-id in the network for the variable (if any). It will send a `VAR-ID request` to the REPLICA controller, containing its `Global-ID`.
- **(5)** The REPLICA controller receives the request, if it is asking for a variable-ID to publish on it, the REPLICA controller will assign a variable-ID to that variable and after updating its data structure, it will send the reply to the NF. If it is asking for a variable-ID to subscribe on it, the REPLICA controller will search its data structure for any matches and will respond with the answer to the NF.
- **(6)** The NF will update its data structure with the received information.

3.3.3 Data Replication phase

In Data Replication phase, if the NF wants to publish a variable or subscribe for a variable, it will use the variable-id instead of the variable name, and will publish the data or will subscribe on it, through the procedure which will be explained in the next sections. Through the rest of this thesis, we will presume that the variable-IDs are predefined by the REPLICA Controller.

Packet format

- **SUB-register and SUB-remove packets:** As is clear from the Table 3.1, the `Subscribe` packet has two kinds, `SUB-register` and `SUB-remove`. After UDP header, they carry no data and are just pure header packets. The variable-ID is used to fill the bits number 10 to 31 of the IP address destination of these packets(see Figure 3.4), while the `PubSub-Flags` are set through the Table 3.1.
- **Publish packets:** We defined an Application layer header after the UDP layer to provide some extra information on the `PUBLISH` packets destined to the NFs. This extra header which has implementation-dependent A-bits length, is been called from now the `Publish-Header`. The position of the `Publish-Header` header is shown in Figure 3.7.

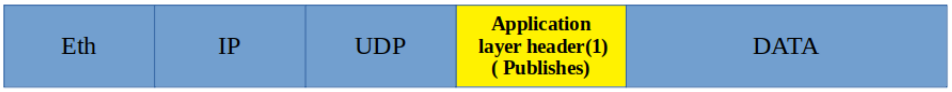


Figure 3.7. *Publish-Header* position.

`Publish-Header` is divided into four parts. The first a-bits length part after the UDP header is used to carry again the variable-ID for the NFs, while the length is implementation-dependent and should be enough to carry the `variable-ID` information, the next b-bits length part is for the update-Number, which been used to identify the set of packets that the publisher is sending in one `PUBLISH` attempt. The

next two c-bits length parts are used to show the Total number of the Fragments for this package and the `fragment-ID` of this `PUBLISH` packet. The description of the `Application layer header` for `PUBLISH` packets, is shown in Table 3.2.

Bit Number	Description
0 to a-1	The `variable-ID`
a to $a+b$-1	The `update-number`
$a+b$ to $a+b+c$-1	The `tot-fragments`
$a+b+c$ to $a+b+(2*c)$-1	The `fragment-ID`

Table 3.2. *Publish-header* in details.

If the data due to some limitations or consideration has to be fragmented, then the `update-number` that has been decided by the NF, is constant for all of those data fragments, while the `fragment-ID` is started from `0` to the last fragment number, due to the order of the fragments. It is notable that the length of the `update-number` and the `fragment-ID` are design-specific and is not constant in other cases.

- **RECOVER Packets:** For the `RECOVER` packet, the `Application Layer Header` is implementation-dependent. The purpose of this packet kind is to aware the system (mainly the REPLICA controller) from the incomplete delivery of the requested publish packets as we are using UDP and the multicast in our protocol. The mechanism of the response or how to RECOVER the data is implementation-dependent.

  The position of the `RECOVER-header` is shown in Figure 3.8 .

Figure 3.8. *RECOVER-Header* position.

3.4 Protocol algorithms and scenarios

3.4.1 The main algorithm

We presume that the NF has been done the `Initializing phase` , so the NF is aware of it's `NF-Global-ID` and we presume that the NF is already done the `Information request` sub phase for being aware of it's `PUB-variable-IDs` corresponding to the variables it wants to publish their values. At this point, the NF will start to publish on it's variables. Any time this NF wants to `subscribe` for a variable data, it will prepare the `SUB-register` packet through the previously explained packet formats and will send the request to the network. The same procedure is repeated for an the `SUB-remove` request for a variable from an NF, the difference with the previous case is in the `PubSub-Flags`. When the switch receives these packets, it will register or remove the subscription inside

its data structure. As we are using UDP, one can define a reply mechanism in the network for the `SUB-register` and `SUB-remove` packets, such that after sending the `SUB-register` and the `SUB-remove`, the NF may wait to receive back a subscription-reply, to make sure that the switch is alive and received the request, and after a time out if no reply is received from the switch, it can resend the request to the switch. Anytime an NF decides to publish on a variable, it will prepare the data to send through the previously explained publish packet format, and it will start to send the packets into the network. In the case of receiving a `PUBLISH` packet, the switch will read its data structure to check for the existence of subscriptions on the corespondent `variable-ID`. If subscription exists, the switch will send publishes to subscribers, otherwise, this packet will be dropped. Each NF that is receiving the published data, will check if there are any lost packets and if it finds lost fragments, it will prepare a `RECOVER` message and will send it to the REPLICA controller. If by any reason, it did not receive all the fragments, it has two choices:

- Keeping the received fragments related to that package.
- Discarding all received fragments related to that package.

In both cases, the `PubSub-Flags` bits will be set according to the table 3.1, to be recognized by the network. The switch will forward this packet to the REPLICA controller for further decisions. In the cases in which the NF needs to send the `variable-ID request`, the NF will set the `PubSub-Flags` according to the table 3.1, and the switch will forward this packet to the REPLICA controller too. Considering that for each step, the packets are already prepared, and the data transfer between NFs and the network is lossless, the time-space diagram for the general algorithm is illustrated in Figure 3.9.

the Figure 3.9 is described as:

- **(1)** and **(5)** The NF(*m*) with the `Global-ID` equal to (*m*) from the `server(`*k*`)`, publishes data on the `variable(`*X*`)`.
- **(2)** As no subscriptions on `variable(`*X*`)` is registered inside the P4-switch, so it will discard those packets.
- **(3)** The NF(*n*) with the `Global-ID` equal to (*n*) from `server(`*l*`)`, sends a `SUB-register request` or `SUB-remove request` for the `variable-ID` of the `variable(`*X*`)` to the P4-switch.
- **(4)** The P4-switch will store or remove the registration of the incoming port of this request packet, over the `variable-ID` mentioned in the request packet.
- **(6)** As there is a subscription about that `variable-ID` from NF(*n*) of the `server(`*l*`)`, the switch will forward a copy of the packet to the port that eventually is destined to the NF(*n*) of the `server(`*l*`)`.
- **(7)** The NF receives the publish packets of an update.
- **(8)** If all fragments of the `Update` is not delivered or some data is corrupted, the NF(*n*) will send a `RECOVER` message to the P4-switch.

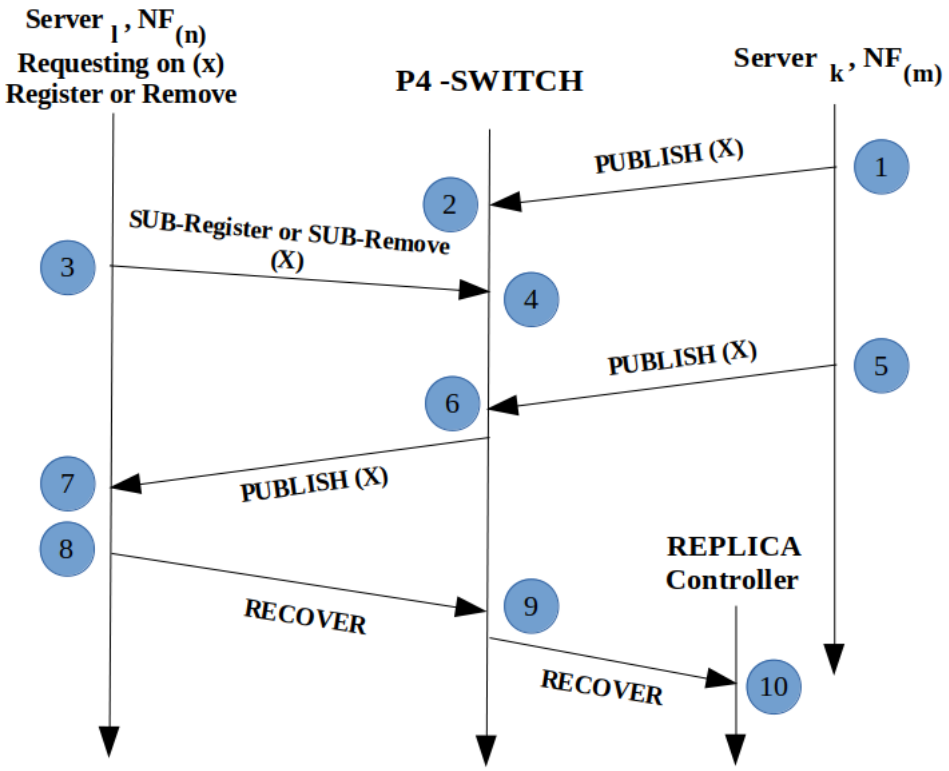


Figure 3.9. Time-Space Diagram for the main algorithm.

- **(9)** The P4-switch will forward the packet to the REPLICA controller.
- **(10)** The REPLICA controller receives the request and will decide about it.

It is notable to remind that, as described in section 2.3.3, the P4-switch header-parser acts like a state machine. So it can easily be programmed to do the normal tasks of an L2-switch, router or any other tasks done by the network devices, through separately defined pipelines, which will be selected by the header parser state machine, while it is serving for our special proposed task too.

3.4.2 The protocol scenarios

In this section, by considering our defined Packet formats and the P4-switch abilities, we proposed two different scenarios:

- Register-based scenario with P4 switch
- Embedded Controller-based scenario with P4 switch

Register-based scenario with P4-Switch

A simple topology for this scenario is shown in Figure 3.10.

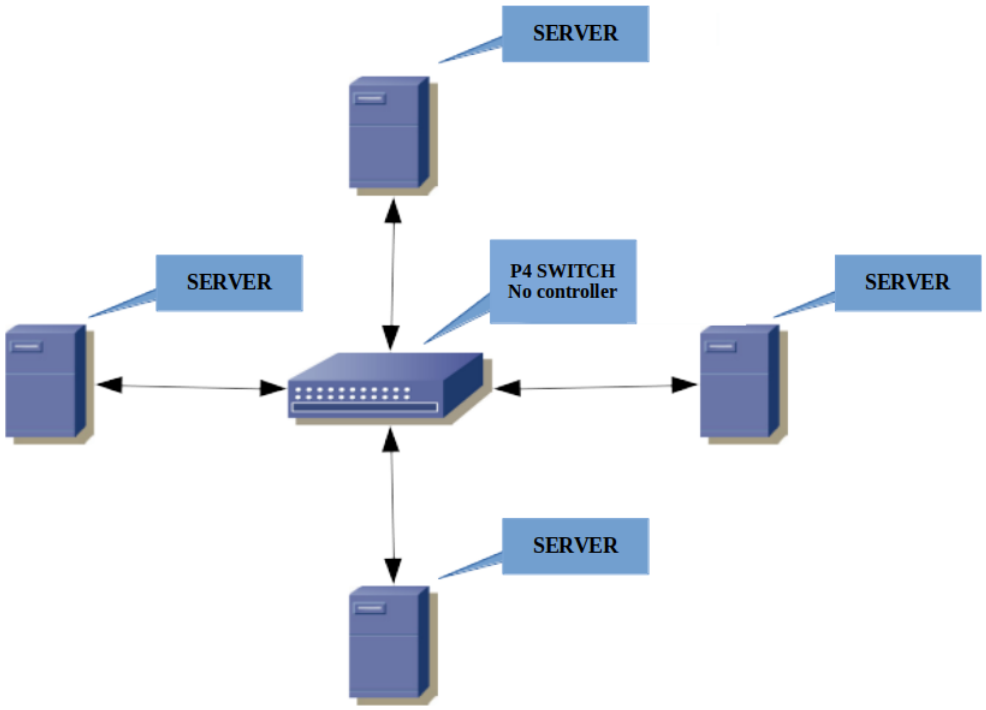


Figure 3.10. Register-based scenario with P4 switch.

As described in the P4 section of chapter Background, the switch sequentially parses the headers. If it finds an IP header in which the first 8-bits of its IP-destination address is not equal to (1110,1111), then it will check the packet for other instructions defined in our P4-switch, e.g., as a normal IPv4 packet. Otherwise, if the first 8-bits of its IP-destination address is equal to (1110,1111) and the switch parser reaches the UDP header and the destination port for the UDP was equal to a special port predefined in our protocol, e.g., 65432, then it will recognize it as a packet of our design kind.(see Figure 3.11 The P4-switch, then from the next 2-bits of the IP-destination address, will decide if this packet is meant to learn from it, e.g., the SUB-register and the SUB-remove packets in a way similar to the MAC learning key-value paradigm, where it uses those key-values for forwarding the PUBLISH Packets, or to be sent to the REPLICA controller, or should be done a multicast on it, due to existing subscription. In this scenario, for holding the key-values, the switch will use its internal registers. These registers are defined through the first programming step of the switch. The index of each register is used as the key, and the number saved inside the register will be the value. The switch may need to do a multicast on a received publish packet, as it may have more than one registration for that variable.

We defined the MultiCast-Group-IDs as an N-bit number, where N is the number of

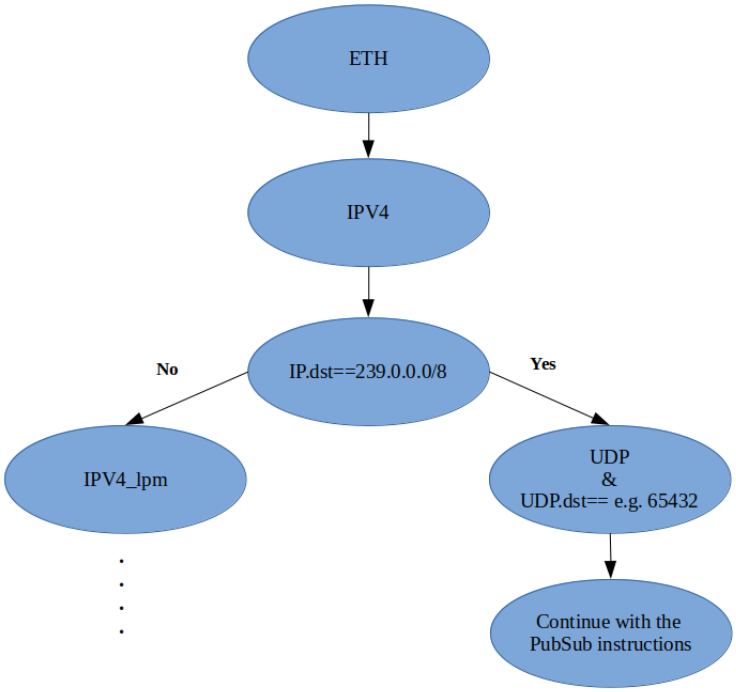


Figure 3.11. The parser logic states of P4 switch for the PubSub protocol.

the switch ports, such that in the binary shape they can show exactly which ports are involved. We defined a port-mask for each port, which is an `N`-bit all zero number, except only k-th bit equal to one, where N is the number of the switch ports, and k is the port number plus `1`.

Let's clear this part with an example. Assume that we want to define a MultiCast group that is related to ports `2`, `5` and `7` of a switch with `8` ports. The corresponding `MultiCast-Group-ID` will be `01010010`, while in this `8`-bit length id, equal to the number of the switch ports, the second, fifth and seventh bits are one, showing the port numbers involved, so the `MultiCast-Group-ID` value becomes `162` and vise Versa. The P4 language has its limitations, for example, it does not support loops or some arithmetic operation like division and power calculation. The simplest way to map the input port number to the binary position of the bits is to shift a `1` by a value equal to that number minus 1, e.g., We know `(3 = 0011)` in a `4`-bit system and we need to have `0100` as `3`, so:

$$0001 \ll (3 - 1) \Rightarrow 0100$$

In Figure 3.12 a sample structure of our registers in the P4-switch is demonstrated. Each time a `SUB-register` packet enters the P4-switch, it generates the input-port bit-mask, then it reads the value of the register that its index is 1 times lower than the `variable-ID` in the packet header, after that it makes a bit-wise `OR` between the value and port-mask for adding the port-mask, and writes back the result into the register again. It will swap the `IP-source` and `IP-destination` fields of the packet, and will mirror it back to the port

(INDEX) = Variable_ID - 1	(VALUE) = The bit-mask for the switch port connected to the Subscriber				
	Port *k*	**. . .**	**Port 3**	**Port 2**	**Port 1**
0	1	. . .	1	0	1
1	0	. . .	0	1	0
. . .	. . .	. . .	. . .	. . .	. . .
n	0	. . .	0	0	0

Figure 3.12. The sample structure of our registers in the P4-switch for the PubSub protocol.

it was entered. This packet will be act as a `subscribe-reply` message for the subscriber NF in the corresponding middle-ware related to that NF. The same process is repeated when a `SUB-remove` packet arrives, with one difference, by replacing the bit-wise `OR` with a bit-wise `XOR` to remove the port-mask. These two packet kinds are the only ones in our protocol, which make a change inside the P4-switch. For the case of a `PUBLISH` packet, if there is a registration for that `variable-ID` inside the corresponding register, the switch will consider that value as the destination `multicast-group-ID` and will send a copy of the packet to each port which was participating in that `multicast-group`. For the case of `RECOVER` packet, `variable-ID request` or `INIT-NF-ID request` packets, the switch will forward them through the port which is eventually destined to the REPLICA controller. The last three packet kinds mentioned here, make no changes inside the switch logic.

Local Controller-based scenario with P4-Switch

A simple topology for this scenario is shown in Figure 3.13. In this scenario, the switch will use an empty Match-Action table predefined by the P4 program of the P4-switch, instead of using registers for holding the subscriptions as key-values. The empty structure of this table is made once through the first programming step of the switch, then entries for this table will be added, modified or removed, dynamically through a local embedded controller inside the switch. The P4Runtime API is used for controlling the table which was defined by our P4 program. It uses the gRPC protocol for communication between the P4 target and the P4 controller as discussed in section 2.3.4(see Figure 3.14).

Anytime a `SUB-register` or a `SUB-remove` packets arrive at the switch ports, the switch will add a controller header to the packet, containing the input port of the packet,

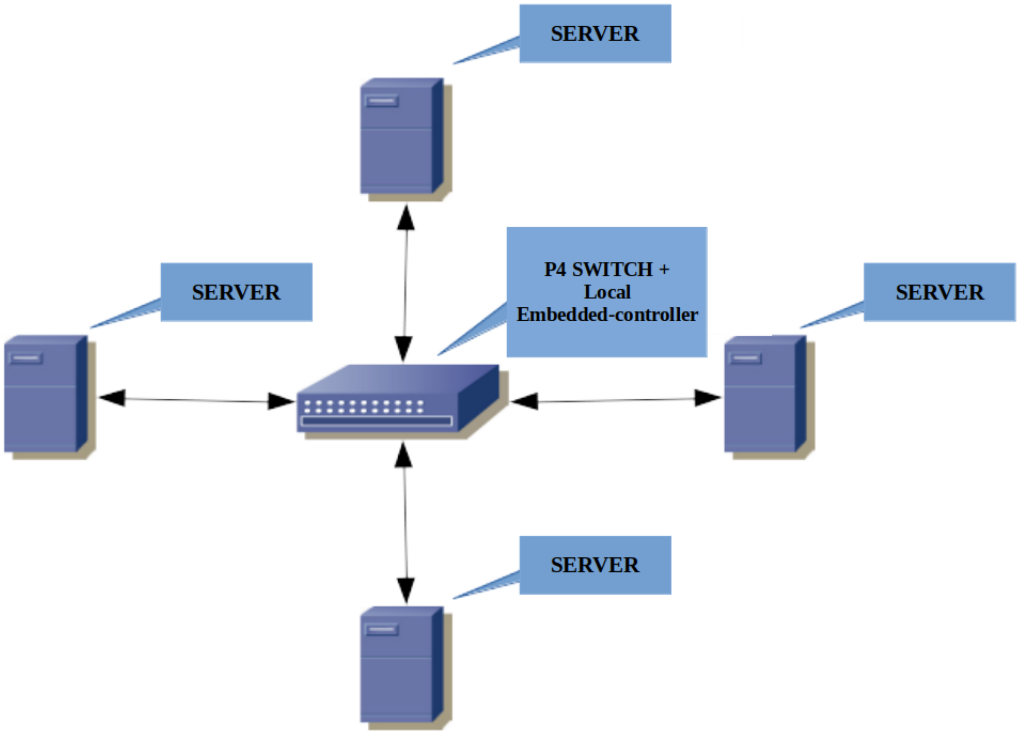


Figure 3.13. Register-based scenario with P4 switch and no Controller topology.

Match	Action
Variable_ID == 1	Send to ports : [1, 3, …, *k*]
Variable_ID == 2	Send to ports : [2]
. . .	. . .

Figure 3.14. The sample structure of our table in the P4-switch for the PubSub protocol.

and sends it to the local controller. The controller will extract and saves the needed information from the packet headers, then it will modify the entries in the switch table through P4Runtime API (e.g., adding or removing a port number from the action list of the table entry which has a match field equal to this `Variable-ID`), and will save the results of such operations for later needs. All the other definitions and procedures, e.g., the shape of the key-value or the format for the `MultiCast-Group-ID`, if not described here, are similar to the previously proposed scenario. There is an extended part between

the P4-switch and the local P4-controller(see Figure 3.15), which shows :

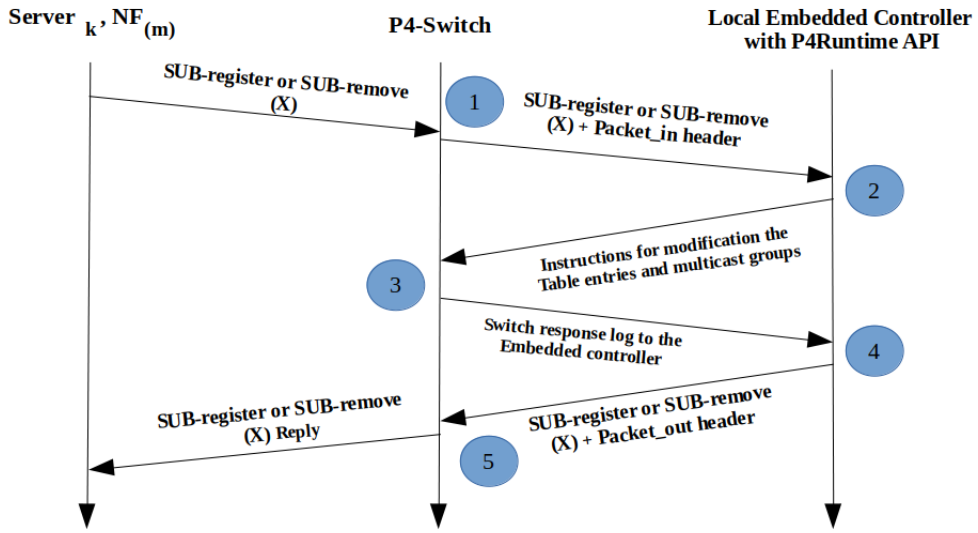


Figure 3.15. Time-Space Diagram for the Local-Controller part.

- **(1)** When a `SUB-register` or a `SUB-remove` packet arrives at the switch ports, it will be sent through P4Runtime API to the local Embedded-controller with an extra `Packet-In` header containing the input port of the packet.
- **(2)** The controller will use the information of the packet header fields, the input port and its previous data structure on table entries and defined multicast groups of the switch, and will send proper instructions to the switch. The instructions also include the request for the report after finishing the switch configuration.
- **(3)** Switch will report the configuration procedure and results to the local Embedded-controller.
- **(4)** The local Embedded-controller will update its data structure with the switch report and will send the `Subscribe` packet it was received at **(1)** after swapping the `IP-destination` and `IP-source` fields, removing the `Packet-In` header and adding a `Packet-Out` header containing the same input port existed in `Packet-In` header, as the new output port for the switch.
- **(5)** The switch will remove the `Packet-out` header after extracting its information, and will send the packet to the NF, through the port mentioned in that information.

3.5 Scalability analysis

In this section, we try to theoretically evaluate the scalability of our proposed schemes in terms of the number of servers and the number of NFs living inside each server. Then

we evaluate the effect of changing the number of variables that each NF may publish on it or subscribe for it and at last, we will consider the effect of these factors on the switch internal resource consumption.

Scalability analysis as a function of the number of servers and NFs

As a matter of fact, in our evaluation, we will take into account only the servers, which has at least one NF in it, and in our scenario, the NF will be the one which is publishing or subscribing on at least one variable. In general, the number of physical servers indirectly connected to a switch is not bounded to the number of switch physical ports. By considering a group of physical servers that are eventually connected to one port of the P4-switch, the traffic of the multiple physical servers can be aggregated to one link that is destined to one port of our P4-switch.(see Figure 3.16). In this case, from the

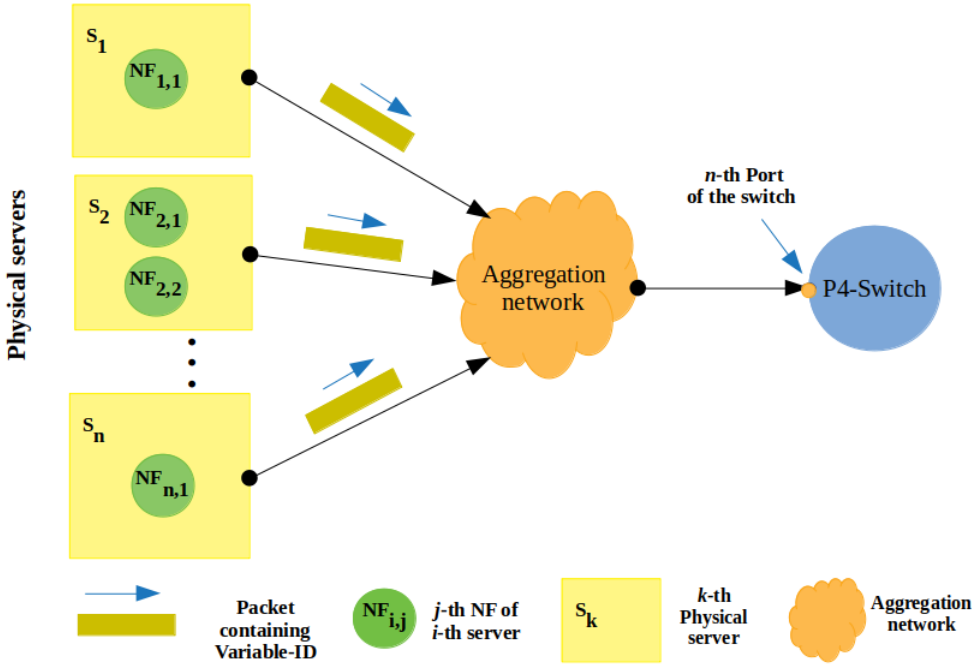


Figure 3.16. Physical servers and their NFs with physical connection to the P4-Switch.

P4-switch point of view, for each of its physical ports, if connected, the switch only sees its next hope, while the packets coming from the next hope are carrying the variable-ID (see Figure 3.17). From now for simplicity, we presume that only one physical server is directly connected to each port of the switch. Figure 3.16 demonstrates two problems. Assume that in this figure, more than one NF has subscribed to one specific variable in the P4-switch, no matter from the same or different physical servers. If only one of them tries to unsubscribe, due to have only one subscription per each variable, the switch will remove the subscription on that variable destined to that port. It will cause to starve the remaining NFs that had a subscription on that variable and through that port but are

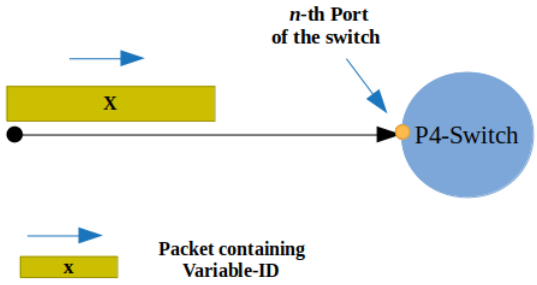


Figure 3.17. P4-Switch point of view from physical connection to it.

on different physical servers connected to the same port. The solution has two steps:

- **(1)** For NFs inside one server, we need a mechanism to have a trace of the local registrations of its NFs inside the server. The `SUB-remove request` will leave the server, if and only if, there is no other existing registration for that variable inside the server. The `SUB-register request` will leave the server, if and only if, there is no existing registration for that variable inside the server. This will eliminate sending non-necessary `SUB-register request` to the switch. Other than that, we did not cover all the combinations, like when there is a need to share data locally and inside one server. The developer of this protocol should consider these steps into account.

- **(2)** A similar mechanism should be used in the aggregation points of the physical server through the path to the switch port. This will solve the problem for NFs inside separate physical servers.

While providing a solution for the first case is completely implementation-dependent, providing a solution for the second case would be out of the scope of this thesis and can be considered as future work. In the next chapter as a Proof Of Concept (POC), we implemented our solution for the first case.

Scalability analysis as a function of the number of variables

For the case of the different number of variables inside each NF, in theory, due to having 22-bit for defining variable-IDs, we are limited to a maximum number of $(2^{22} \approx 4 \times 10^6)$ for maximum different variables. For the `Register-based solution`, as the number of needed registers inside the switch is exactly equal to the number of the variables, by increasing the number of variables, the number of needed registers inside the switch grows linearly, and this number is limited by the available memory inside the switch for defining registers. It is notable that, if changing the number of variables leads to having more variables than already defined registers, the switch should be completely re-programmed. Nonetheless, this procedure, if needed, can be done very fast. In the case of `Embedded controller-based` instead of the registers, we use a match-action table. Except for the

available memory inside the switch, there is no other limitation on real-time changing of the number of variables, as the definition of variables inside the switch is done dynamically and in real-time upon request and by the local Embedded-controller and there is no need to reprogram the switch. The scenario with the registers is fast and simple but scalable for small cases with limited variables and barely changing structures or the cases that the maximum number of the variables barely crosses the number of defined registers. The scenario with Local P4Runtime is fast, dynamic and scalable for any structure. Both schemes make the scenario needless for the standard REPLICA remote Controller to be involved in the major parts of the registration of or distribution of the publishing variables, by offloading some duties of the REPLICA controller to the P4-switches. The two scenarios with P4 switches are briefly evaluated in Table 3.3, Table 3.4.

SCALABILITY IN TERMS OF THE NUMBER OF:	THEORETICAL UPPER BOUND	LIMITATIONS OR CONSIDERATIONS
SERVERS	Unlimited	—
NETWORK FUNCTIONS	Unlimited	—
VARIABLES	$2^{22} \approx 4 \times 10^6$	Definable variables are Upper Bounded by the number of the registers which are defined at the compile time.

Table 3.3. Register-based with P4-switch

- **Pros:** Fast, Simple, Stable.
- **Cons:** Static number of Registers, change in number of registers, Needs recompiling and installing new P4 program inside the P4-switch .

The P4-switch only uses the `variable-ID` and the port numbers for the subscriptions, then our scenarios are only dependent on the number of switch ports and the number of variables. For each rule, the `variable-ID` is a unique key, while the combinations of the participating ports, forms the corresponding value. So, the number of rules needed to be installed inside the P4-switch in both scenarios is only proportional to the number of variables.

$$\#\ \text{Rules to be installed} \propto \#\ \text{Variables}$$

Furthermore, recalling on section 2.3.2, as the P4-switches use Reconfigurable Match Tables (RMT), we expect to have less internal memory consumption for installing the needed rules for the subscriptions, with respect to OpenFlow switches.

SCALABILITY IN TERMS OF THE NUMBER OF:	THEORETICAL UPPER BOUND	LIMITATIONS OR CONSIDERATIONS
SERVERS	Unlimited	—
NETWORK FUNCTIONS	Unlimited	—
VARIABLES	$2^{22} \approx 4 \times 10^6$	—

Table 3.4. Embedded Controller-based with P4-switch.

- **Pros:** Dynamic, No Register-Limited.
- **Cons:** More delay, More complexity and More activity by the switch than Register-based scenario.

Chapter 4

Implementation and experimental validation

In this chapter, we will describe our simulation methodology, the tools we used to emulate our network environment, network components and our evaluation method for comparing our schemes described in chapter 3. Then as a Proof Of Concept(POC), we build a simple test configuration, and we run the solution to prove the correctness of it. After that we implement our proposed protocol with OpenFlow switch, to show its applicability in the vanilla SDN. At last, we tried to simulate the behavior of the internal resource consumption of the three implementations to show the excellence of our solutions.

4.1 Tools and Components

In this section we describe in brief , the tools we used for emulating our schemes.

4.1.1 Mininet

Mininet is a powerful network emulator, which is very convenient to be used for prototype and test almost any SDN solution, by having a virtual experimental Network inside our PC. Mininet creates a realistic virtual network, running real kernel, switch and application code, on a single machine (VM, cloud or native), in seconds, with a single command.(See Figure 4.1).

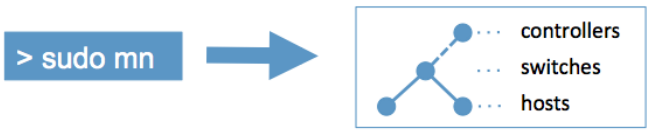


Figure 4.1. Mininet Diagram (reproduced from [30]).

4.1.2 BMV2

BMV2 is just a prototype virtual switch, recently made and introduced to show the main capabilities of the P4 language. It is the abbreviated form for (Behavioral Model Version 2). It is a software switch that is emulating a P4 data-path, written in C++(11) language. It takes as input a JSON file generated from your P4 program by a P4 compiler and interprets it to implement the packet-processing behavior specified by that P4 program [31]. Figure 4.2.

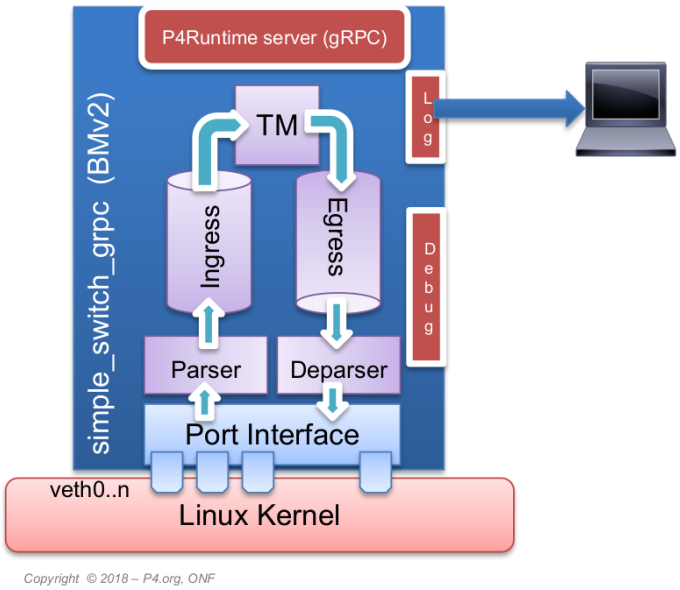


Figure 4.2. BMV2(Behavioral Model Version 2) (reproduced from [32]).

4.1.3 bm_CLI

It is a very handy process, written by the developers of the BMV2. By connecting to a thrift port that is defined in the BMV2 through the compile phase, it gives a very powerful and easy command-line interface to the P4 switch for reading the internal externs of the switch and manipulating its tables.

4.1.4 Open vSwitch

Open vSwitch is a production quality, multilayered virtual switch licensed under the open-source Apache 2.0 license. It is designed to enable massive network automation through programmatic extension, while still supporting standard management interfaces and protocols (e.g., NetFlow, sFlow, IPFIX, RSPAN, CLI, LACP, 802.1ag) [33]. The Open vSwitch has been developed for some years and has been supported and debugged by a vast community. In many ways, Open vSwitch targets a different point in the design space than the hypervisor networking stacks which use the built-in L2 switch (the Linux

bridge), by focusing on the need for automated and dynamic network control in large-scale Linux-based virtualization environments. The goal with Open vSwitch is to keep the in-kernel code as small as possible (as is necessary for performance) and to re-use existing subsystems when applicable (for example Open vSwitch uses the existing QoS stack). As of Linux 3.3, Open vSwitch is included as a part of the kernel and packaging for the userspace utilities, which are available on most popular distributions [34].

4.1.5 RYU Controller

Ryu which is pronounced as (ree-yooh), means flow in Japanese language. It is a component-based software defined networking framework written in python language, that provides some software components with a well defined API, which makes it very easy for developers to control applications and to create new network management. It supports many protocols such as OpenFlow, Netconf and OF-config to manage network devices. Ryu supports fully the OpenFlow versions 1.0, 1.2, 1.3, 1.4 and 1.5. All of its code is freely available under the Apache 2.0 license.

4.1.6 Traffic generators

In order to build our specific packet shapes for our tests we used some tools:

- **SCAPY [35]:** It is a very flexible and powerful Python library, which is capable of packet, creation, manipulation, forging and even doing many network tasks including network discovery, testing, trace-routing, and many others. In our experiments, we use widely some power of this library to debug our codes.
- **Socket Generator and Listener [36]:** After testing many packet shapes we decided to use standard Python socket library, as our needs were satisfied by using the standard network protocol headers and there was no need to use Scapy library.

4.1.7 Measurement tools

For measuring the memory resources used by the proposed schemes, we used the Resident Set Size(RSS) measurement. RSS is used to show approximately, how much of the memory is assigned to a process and is existing in the RAM. It does not include the memory part that is swapped out, but it does include all the stack and heap memory, and at last, it includes the memory used by the shared libraries as long as those libraries pages are actually in the memory. There is another measure factor which is called Virtual Memory Size(VSZ). It will include all the memory parts that a process can have, including the allocated memory which is not used, the swapped out memory, and the shared libraries memor. As an example, if a process has 1000K binary and is linked to some shared libraries of size 2000K, while it has allocated 500K of stack/heap, which 200K from it is actually in memory, and it has only actually loaded 800K of the shared libraries and 300K of its binary then we have:

$$TheVSZ : 1000K + 2000K + 500K = 3500K$$

$$TheRSS : 300K + 800K + 200K = 1500K$$

Since part of the memory is shared, many processes may use it, so if you add up all of the RSS values you can easily end up with more space than your system has. The memory that is allocated also may not be in RSS until it is used by the program. So if your program allocated a bunch of memory upfront, then uses it over time, you could see RSS going up and VSZ staying the same. The RSS measurement can give us a much realistic measurement over the actual memory resources consumption of a process than the VSZ. To measure the RSS we used:

- **`ps(Process Status):`** The `ps` program is a process, which in most Unix and Unix-like operating systems, displays the information about a selection of the active (currently-running) processes. It gives us information on the Resident Set Size (RSS). This process does not indicate precisely how much of the shared libraries are used exactly by our process, because it divides the size of the shared libraries over the number of running processes that are using those shared libraries. IN this way it gives us an approximation over the RSS for that process.

- **time:** `time` is a process which runs the program command with any given arguments, when the command finishes, time displays information about resources used by the command (on the standard error output, by default), and it gives us some information on the command including the maximum RSS. So it gives us an upper bound for the RSS measurement.

both of them give us an upper-bound approximated measure on the memory usage for installing the rules inside the switches. It means the values are like:

$$Measured(RSS) = Actual(RSS) + error$$

We used `ps` for the Local Embedded-controller scenario and the `Time` for the Register-based scenario.

4.2 Architecture

As we are using a virtual environment and virtual components, there are some limitations to check the efficiency of the two mentioned methods. While it is not possible having the correct results for cases such as end-to-end delays that need to be tested in a real physical environment, it is possible to check the correctness of the proposed algorithms in such environments. As a Proof Of Concept(POC), we emulated both proposed schemes with a simple topology. For our experiments we used components placement for the `Register-based` and the `Embedded controller-based` solutions according to the Figure 4.3.

The only difference between the two solutions is in the way they keep the records of the subscriptions inside the P4-switch, which results in a small structural difference in the P4 programs and having an extra controller process for the Embedded-controller scenario. The hosts shown in the topology, (H1, H2, H3, and H4), are virtual samples of the physical servers in the Mininet, where H1, H2, and H3 are hosting our sample NFs

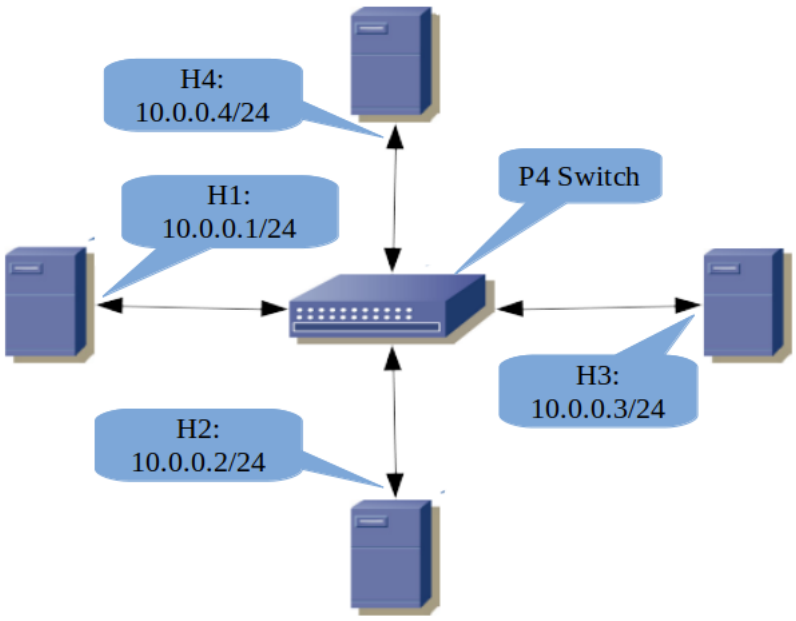


Figure 4.3. Experimental topology with P4 switch.

and a simple controller has been placed inside the H4 to play the role of the REPLICA controller in our solutions.

4.2.1 The Middle-ware

The fact that there could exist more than one NF inside one server, crystallizes the need for an internal mechanism inside the server to handle efficiently and correctly the communication between the internal NFs and between NFs and the network. As an example, in the case of two NFs inside one host, that both are subscribed on the `variable(X)`, if one of them decides not to receive any further updates for the `variable(X)`, as it is unaware of the possible existence of other NFs and their subscriptions, so it will send a `SUB-remove request` to unsubscribe from the variable(X), then the switch will not forward any `PUBLISH` data of the `variable(X)` to that host and the second NF will no longer receive the updates from the `variable(X)`. Another example is the case of two NFs inside the same host, which one of them is publishing on the `variable(X)` and the second one wants to subscribe to `variable(X)`, which is more reasonable to have state-sharing being handled locally. To cover the above-mentioned situations, either we need to have direct communications between the NFs, or using an outside mechanism that is shared between all of them, which can be another process that is discover-able for all of them and is able to communicate locally with all the NFs inside the same host. The NFs are distinct processes, which are not aware of each other existence, which means the first approach is not applicable to our case as there is not enough primitive information to

start the communication between them. So to implement the second approach, we introduced a process as a `Middle-Ware` inside each host which is placed in the way between the NFs and the host. By using Middle-ware, they do not need anymore to take care of consensus between them and they will be separated from the outside network, which results in simplifying the NFs internal structure, and lighter in terms of resource usage and programming complexity. For the communication between this new process and the NFs, we need `Inter Process Communication`(IPC). The well known approaches are using the `Un-named Pipes`, the `Named Pipes`, the `Shared memory` and the `Sockets`. The `Un-named Pipes` needs a hierarchical `parent` and `child` paradigm, which by considering the fact that our NFs should be completely unrelated and distinct processes, even from our `Middle-Ware`, will be eliminated from the choices. The choice between the `Un-named Pipes`(also known as `FIFOs`), the `Shared memory` and the `Sockets` is dependent on the application of the protocol, which here we decided to use the `Sockets`, as they are the clearest concept of the communication for the network. We decided to use TCP sockets, as they are reliable for ensuring the data transfer between the endpoints. We will discuss in detail all the parts and all the steps for the proposed scheme in the next sections. For running our implementations, we needed to build a special virtual machine or installing several dependencies in our Linux Operating System to use the BMV2 virtual switch, so for the sake of simplicity we used the special virtual machine which is prepared by the **P4 Language Consortium** [37]. In our implementation, first of all, the `REPLICA controller` will be started in the H4, then in each of the H1, H2 and H3 hosts, one `Middle-Ware` will be started to listen for the NFs messages, then the NFs will be started in all hosts except the H4. By adding the `Middle-Ware` to our structure, an example view of our architecture inside the hosts can be shown as Figure 4.4.

In the next section, we will describe in detail our designs for the `Middle-Ware` and the NFs.

4.3 Software implementation

The NFs and the Middle-ware are written with the python language.

4.3.1 NF design structure

By using the Middle-ware, we are separating the NF from the real network, so for communication with the Middle-ware, we just need to define the data part of the packet formats described in the section 3.3.

Message format

All messages have two parts, a common part that has the same structure in all the messages, and a specific part which is message dependent:

- **Common part:** It is the first 6-bytes of the message, and is divided into three parts:

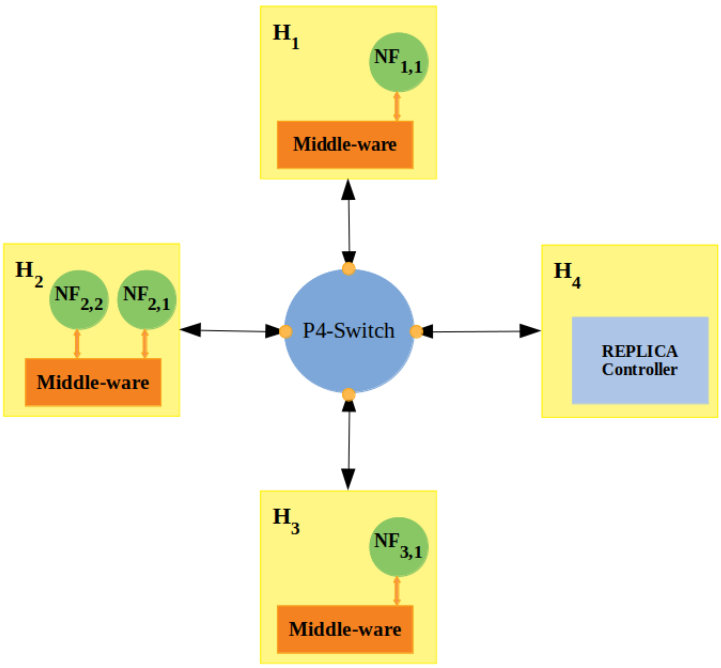


Figure 4.4. An internal view of our hosts architecture.

- **Message length:** It is the first 2-Bytes of each message and contains an unsigned integer number which is the length of the message including itself in bytes. It is used in the TCP sockets for recovering the messages.
- **Message kind:** It is the second 2-Bytes of each message and contains an unsigned integer number between zero and six, which is showing the message type:
  - **Type (0):** It is an `INIT-NF-ID request` message.
  - **Type (1):** It is an `INIT-PUB-VAR-ID request` message.
  - **Type (2):** It is a `PUBLISH` message.
  - **Type (3):** It is a `SUB-register` message.
  - **Type (4):** It is a `SUB-remove` message.
  - **Type (5):** It is a `SUB-VAR-ID Request` message.
  - **Type (6):** It is a `RECOVER` message.
  - *
- **NF-Global-ID:** It is the third 2-Bytes of each message and contains an unsigned integer number between 0 and 65535. The 0 is only being used for asking the `INIT-NF-ID` from the REPLICA controller and in the rest of the cases it is a constant non-zero number distinguishing the NF from the others.

- **Specific part:** It contains the rest of the message, the details and the positions are described for each kind as bellow:

- **Kind (0):** It carries the `NF-name`(m-Bytes). The m is the length of the `NF-name`. This message will have a reply similar to the request, while the number 0 in the `Global-NF-ID` field has been replaced with the assigned `Global-NF-ID` by the REPLICA controller.
- **Kind (1):** It carries the `Variable-name`(n-Bytes). The n is the length of the `Variable-name`. This message will have a reply similar to the request, while a part with 2-Bytes length is inserted between the `Global-NF-ID` field and the `Variable-name`, containing the `INIT-PUB-VAR-ID`.
- **Kind (2):** It carries the `Variable-ID`(2-Bytes) + the `Update-number`(2-Bytes) + the `Total-fragments`(2-Bytes) + the `Fragment-ID` + the `Published data`(k-Bytes). They are based on definitions on the Table 3.2.
- **Kind (3):** It only carries the `Variable-ID`(2-Bytes).
- **Kind (4):** It only carries the `Variable-ID`(2-Bytes).
- **Kind (5):** It carries the `Variable-name`(n-Bytes). The n is the length of the `Variable-name`. This message will have a reply similar to the request, while a part with 2-Bytes length is inserted between the `Global-NF-ID` field and the `Variable-name`, containing the `SUB-VAR-ID`. There are two kinds of replies to this request, if the `SUB-VAR-ID` in the reply is a non-zero number, the request was successfully answered, otherwise, zero means the `Variable-name` was not assigned before, so the NF will retry its request after a back-off time. This procedure will be repeated until receiving a non-zero `SUB-VAR-ID`.
- **Kind (6):** It carries the `Variable-ID`(2-Bytes) + the `Update-number`(2-Bytes) of the lost message parts.

Communication steps

Figure 4.5 demonstrates how our implementation is using the steps described above .

- **(1)** Newly started NF wants to initialize, so it will send an `INIT-NF-ID request` containing `Length = 15`, `Kind=0`, `Global-NF-ID: 0` and `NF-name='scanner-1'`, to the middle-ware.
- **(2)** The `Middle-ware` replies to the NF with the same message format while rewriting the `Global-NF-ID: 5001`.
- **(3)** The NF needs to publish on its variable with the name e.g., 'a'. So it will send an `INIT-PUB-VAR-ID request` containing `Length=9`, `Kind=1`, `Global-NF-ID: 5001` and `PUB-VAR-name='a'`, to the middle-ware.
- **(4)** The `Middle-ware` replies to the NF with the same message format while rewriting the `Variable-ID: 35`.
- **(5-a)** till **(5-b)** The NF wants to publish on variable with the `PUB-VAR-name='a'` and the `Variable-ID: 35`, so after generating the update, it breaks the data into the chunks with the maximum size equal to e.g., 1400 Bytes, then it calculates the

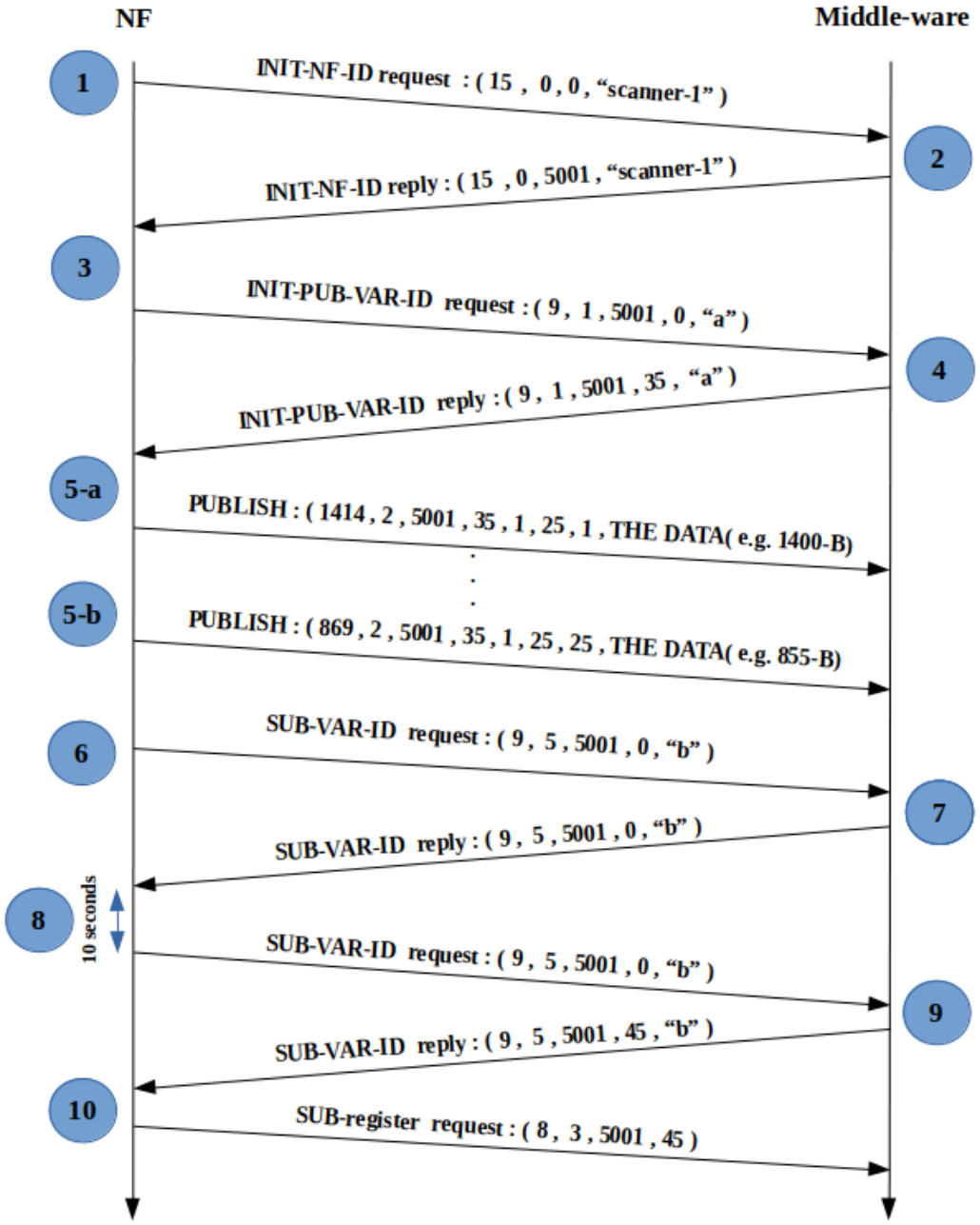


Figure 4.5. The time-space diagram for the NFs.

number of the fragments and will send sequentially the fragments one after each other with the `Kind=2`, the `Update-number=1`, the `Total-fragments=(e.g., 25)` and the`Fragment-ID` following by the `DATA`.

- **(6)** The NF decides to subscribe on a variable that it is already aware of its

name. So it will send a `SUB-VAR-ID request` containing `Length=9`, `Kind = 5`, `Global-NF-ID: 5001` and `SUB-VAR-name='b'`, to the middle-ware.

- **(7)** If there is no previous id assignment for the variable 'b', then the `Middle-ware` replies to the NF with the same message format while rewriting the `Variable-ID: 0` as an 'ERROR' message.
- **(8)** By receiving this 'ERROR' message, the NF will wait for 10 seconds and will repeat the step **(6)**.
- **(9)** The `Middle-ware` replies to the NF with the same message format while rewriting the `Variable-ID: 45`.
- **(10)** The NF sends a `SUB-register request` with the `Kind=3` and the `Variable-ID: 45` to the `Middle-ware`

Internal structure

Inside each NF, five threads will be made and started sequentially:

- **1.Receive thread:** When the `Receive` thread starts, it tries to connect to the address tuple `('localhost', special-port-1)`. This thread will act as a blind receiver and tries to receive in an infinite loop and write the received data in a queue made for it.
- **2.Send thread:** When the `Send` thread starts, if any, it tries to pop messages from a queue made for it, and send them to the socket connection made previously by the `Receive` thread.
- **3.Msg-handlr thread:** When the `Msg-handlr` thread starts, if any, it continuously pops items from the output queue of the `Receive` thread, and rebuild the received packets by the NF through some information that we already put in the original messages. After that it decides what to do with the messages, based on the message kinds explained in the previous chapters as below:
  - **`INIT-NF-ID reply`:** It will store the `INIT-NF-ID` and will change the flag related to the `Init-publish` thread for moving to the `INIT-PUB-VAR-ID request` step.
  - **`INIT-PUB-VAR-ID reply`:** It will store the `INIT-PUB-VAR-ID` and will change the flag related to the `Init-publish` thread for moving to the `PUBLISH` step.
  - **`SUB-VAR-ID reply`:** It will store the `SUB-VAR-ID reply` for the `Subscribe` thread to decide whether resend the request after some waiting, or making the `SUB-register` message.
  - **`PUBLISH` messages:** it will check for the possible packet losses, if there is no packet loss, it will store the `PUBLISH` messages in a file, otherwise it will make a `RECOVER` message and will add it to the input queue of the `Send` thread.

- **4.Init-publish thread:** When the `Init-publish` thread starts, it tries to do the initializing phase by making the `INIT-NF-ID request` message and adding this message to the input queue of the `Send` thread. It waits until receiving the proper answer, then it makes the `INIT-PUB-VAR-ID request` message and adding this message to the input queue of the `Send` thread. It waits until receiving the proper answer, then it will start to make the `PUBLISH` packets and will add them to the input queue of the `Send` thread. It also stores a copy of the `PUBLISH` packets it is making in a file for further possible needs.
- **5.Subscribe thread:** When the `Subscribe` thread starts, it tries to do information request sub-phase, by making the `SUB-VAR-ID Request` message and adding this message to the input queue of the `Send` thread. If it receive an `ERROR` message, it waits for 10 seconds and will retry by making the `SUB-VAR-ID request` message and adding this message to the input queue of the `Send` thread, until receiving the proper answer, then it will make the `SUB-register` message, and will add this message to the input queue of the `Send` thread.

The internal diagram of our NF is demonstrated in the Figure 4.6.

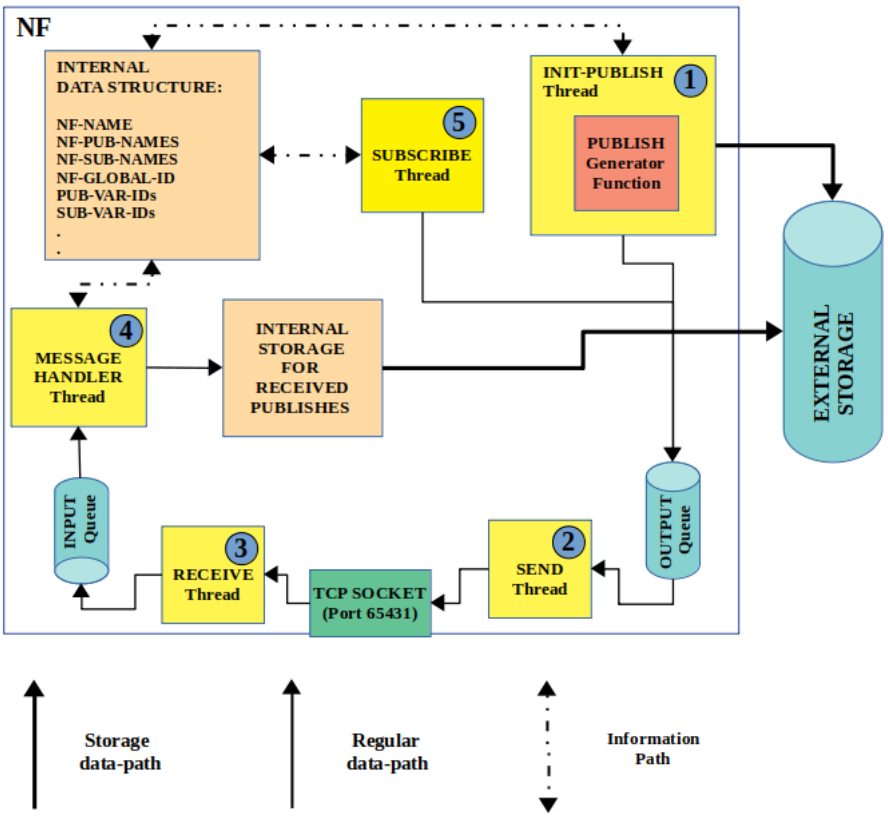


Figure 4.6. The internal diagram for the NFs.

4.3.2 Middle-ware design structure

Message format

The `Middle-ware` does not create any messages from zero by itself. If it receives the messages from the NFs, based on the `Middle-ware` internal data structure, if this message is needed to be sent to the network, then the `Middle-ware` will send the message to the proper address tuple (e.g., (('239.0.0.35, 65432))) on the network. On the other hand, if it receives a message from the network, then based on the message kind and the `Middle-ware` internal data structure, it will forward the message to the proper NF/s. The `Middle-ware`, except in the cases of the `INIT-NF-ID request` and the `INIT-NF-ID reply`, will not change any field of the received messages. In those mentioned cases, for mapping the received `INIT-NF-ID reply` to the proper NF, the `Middle-ware` will assign a `LOCAL-NF-ID` to each of the NFs once it receives a `INIT-NF-ID request` in its internal data structure. Then it will add this value as a 2-Bytes length field after the `GLOBAL-NF-ID` field in the message so that it can be recognized in the reply message. The `Middle-ware` will eliminate this field when it received the `INIT-NF-ID reply` message, the NF will be completely unaware of this process.

Communication steps

Figure 4.7 demonstrates how our implementation is using the steps described above .

- **(1)** The `Middle-ware` adds the already assigned `LOCAL-NF-ID` to the `INIT-NF-ID request` as mentioned previously, then it sends the message as a Unicast UDP packet to the UDP port number=(65432) and the `IP address` of the `REPLICA controller`.
- **(2)** The `P4-switch` forwards the received packet based on its `IPV4.lpm` routing table.
- **(3)** The `REPLICA controller` will replace the `Global-NF-ID` field of the message with the one it assigned to the NF, and sends back the `INIT-NF-ID reply` message as a Unicast UDP packet to the UDP port number=(65432) and the `IP address` of the server which is hosting the `Middle-ware`.
- **(4)** The `Middle-ware` removes the `LOCAL-NF-ID` field and sends the message to the related NF.
- **(5)** The `Middle-ware` sends the `INIT-PUB-VAR-ID request` message as a Unicast UDP packet to the UDP port number=(65432) and the `IP address` of the `REPLICA controller`.
- **(6)** The `REPLICA controller` will replace the `Global-PUB-VAR-ID` field of the message with the one it assigned to the `Variable`, and sends back the `INIT-PUB-VAR-ID reply` message as a Unicast UDP packet to the UDP port number=(65432) and the `IP address` of the server which is hosting the `Middle-ware`.

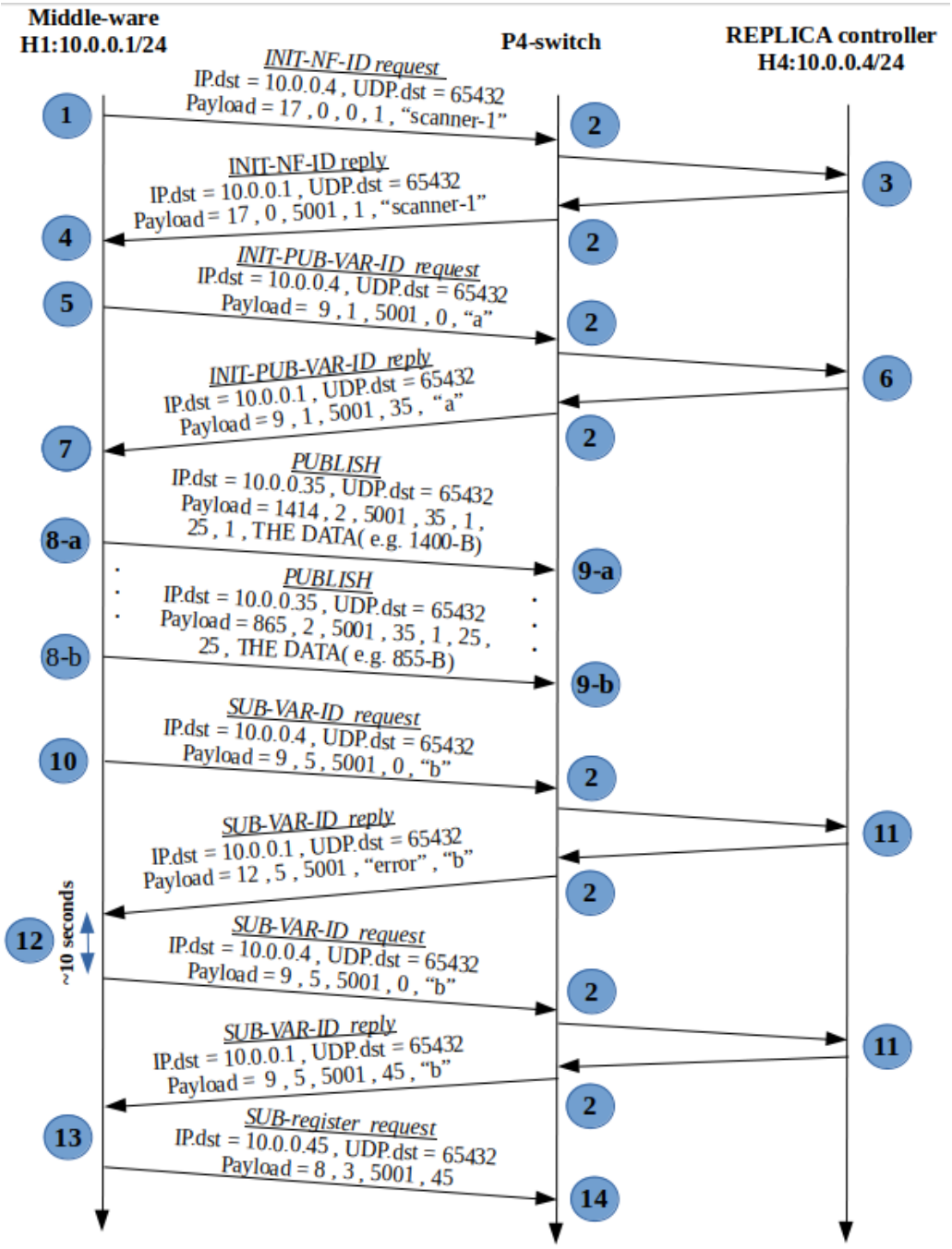


Figure 4.7. The time-space diagram for the `Middle-ware`.

- **(7)** The `Middle-ware` sends the `INIT-PUB-VAR-ID reply` message to the related NF.
- **(8-a)** to **(8-b)** The `Middle-ware` will send the `PUBLISH` messages as UDP packets

to the UDP port number=(65432) and the `IP address` of the corresponding IP-Multicast group.

- **(9-a)** to **(9-b)** The `P4-switch` through its internal data structure will forward them to the registered ports, otherwise it will drop the packets.
- **(10)** The `Middle-ware` sends the `SUB-VAR-ID request` message as a Unicast UDP packet to the UDP port number=(65432) and the `IP address` of the `REPLICA controller`.
- **(11)** The `REPLICA controller` will check its internal data structure to see if any `Variable-ID` has already been assigned to this variable. If the answer is positive, then the `REPLICA controller` will put this number in the `Variable-ID` field of the message, otherwise, it will put 'error' in the message. At last, it sends the `SUB-VAR-ID reply` message as a Unicast UDP packet to the UDP port number=(65432) and the `IP address` of the server which is hosting the `Middle-ware`.
- **(12)** The `Middle-ware` sends the `SUB-VAR-ID reply` message to the related NF. But as it was a reply containing 'error' message, after almost 10 seconds The `Middle-ware` will receive a repeated `SUB-VAR-ID request` message, and will send it to the `REPLICA controller` again. Steps **(10)** to **(12)** will be repeated until instead of the 'error' message, a `Variable-ID` is received by the NF.
- **(13)** The `Middle-ware` will send the `SUB-register` messages as UDP packets to the UDP port number=(65432) and the `IP address` of the corresponding IP-Multicast group.
- **(14)** The `P4-switch` will register the request inside its internal data structure.

Internal structure

Inside the `Middle-ware`, four main threads will be made and started sequentially, and for each NF connecting to the `Middle-ware`, it will make three threads specifically for handling the communication with that NF. The number of threads inside the `Middle-ware`, n, is calculated from:

$$n = 4 + m + (3 \times k)$$

where the k is the number of the connected NFs and m is a value equal to the number of subscribed variables divided by the `maximum IGMP membership` limitation of the OS. They can be described as bellow:

- **1.INITIAL thread:** The `Middle-ware` starts by making the first thread, which is responsible to make a server for a TCP socket, listening on a specific port (e.g., 65431 in our case) for incoming connections from the NFs. This port is predefined in all NFs, and is used to start the primary connection between each NF and the `Middle-ware`. After accepting the connection from the NF, three threads will be made, responsible for the communications related to this NF:

  - **SEND thread:** It is responsible to send messages to the connection of that NF. There is a queue assigned to this thread, which continuously is checked by this thread to see if any message is available for sending.
  - **RECEIVE thread:** It is responsible to receive messages from the connection of that NF. There is a queue assigned to this thread. It continuously adds the data it receives from the NF connection to this queue.
  - **MESSAGE-HANDLER thread:** For each NF connected to the `Middle-ware`, a third thread will be made and will be responsible to read from the output queue of the receiver thread and if needed, rebuild the original message sent by the NF by using the Length field of the messages. This thread based on the message kind and the available information inside the internal data structure of the `Middle-ware`, may update the internal data structure of the `Middle-ware` or add a copy of this message into the queue for the send thread of the other NFs, but for sure, it adds the original message to the main outgoing queue of the `Middle-ware` destined to the network.

- **2.MAIN SEND thread:** This thread is responsible to make a UDP socket and if there is a message in the main outgoing queue of the `Middle-ware`, the thread will make an IP packet, and based on the message kind it will make a proper destination for the IP layer, sets the UDP layer destination port to another special port (e.g., 65432 in our case), and will send the packet to the network. In the case of subscriptions, it will do the membership in the proper IP-multicast group in the OS and if reached the `maximum IGMP membership` limitation of the OS, will build a new thread for a new range of the IGMP membership.

- **2.MAIN RECEIVE thread:** This thread is making another UDP socket and will bind this socket to an address tuple made of the server IP address and the second-mentioned spacial port (e.g., ((' ', 65432)). It is responsible to receive from this socket and will add whatever it receives to the main incoming queue of the `Middle-ware`.

- **3.MAIN MESSAGE-HANDLER thread:** This thread is responsible to read from the main incoming queue of the `Middle-ware`, updating the internal data structure of the `Middle-ware`, and adding a copy of the message to the input queue of the proper NFs.

The internal diagram of our `Middle-ware` is demonstrated in the Figure 4.8.

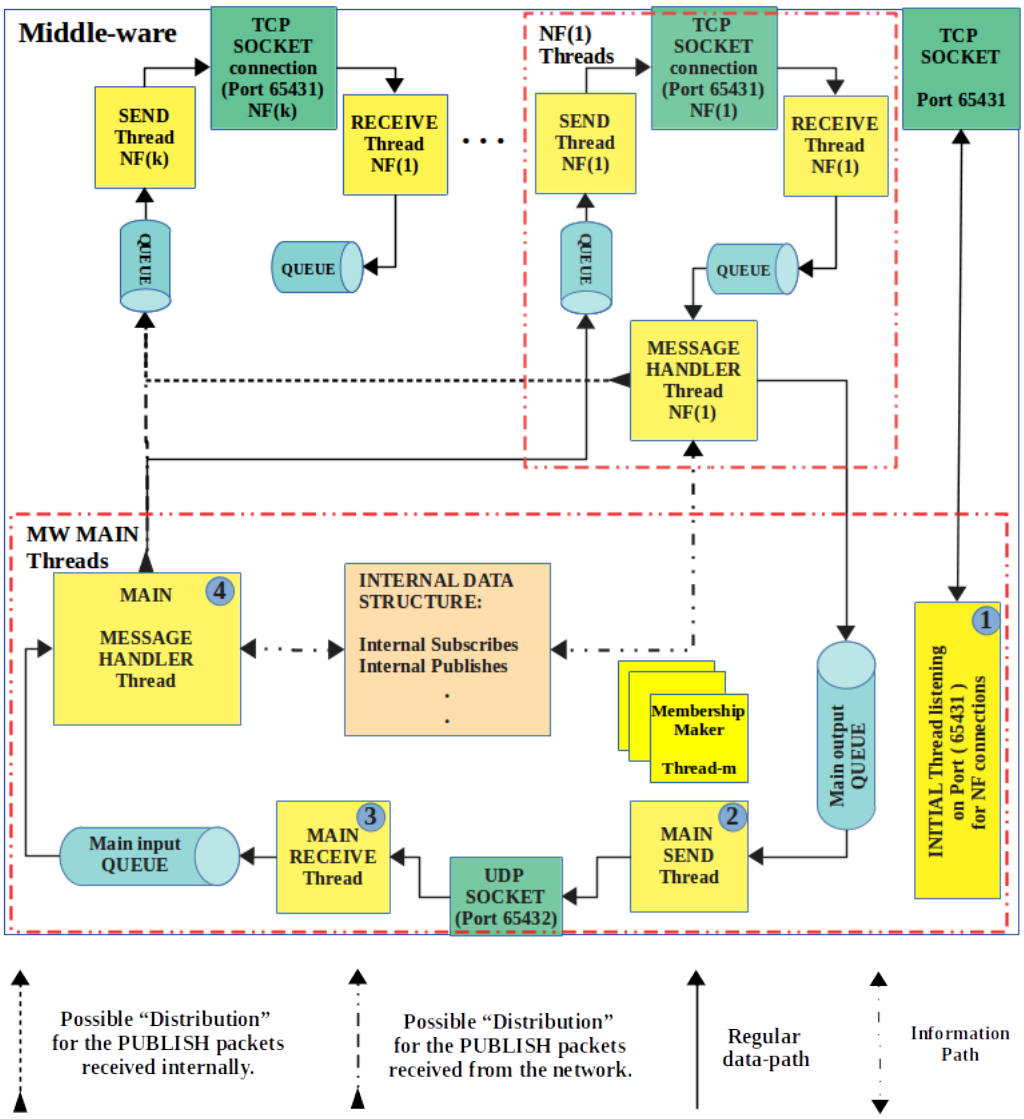


Figure 4.8. The internal diagram for the `Middle-ware`

4.4 Experimental results

For simplicity we presumed that the channels are lossless, the `PUBLISH` packets are sequential and there is a long enough free time between two consecutive updates for each variable-ID. For simplicity, we start only one NF in each of the three first hosts(H1, H2, and H3). Each of these three NFs supposed to publish only one variable and not subscribe to any variables.

- **(1).** The `REPLICA controller` is started in the H4.

- **(2).** One `Middle-ware` is started in each of the H1, H2 and H3.
- **(3).** One NF from the H1 starts and sends an `INIT-NF-ID request` for the NF-name: `'/onem2m/torino_5g/libeliumscanners/wifi/scanner1'` to the `REPLICA controller`.
- **(4).** The `REPLICA controller` will assign the `Global-NF-ID: 5001` for the NF in the H1.
- **(5).** One NF from the H2 starts and sends a `INIT-NF-ID request` for the NF-name: `'/onem2m/torino_5g/libeliumscanners/wifi/scanner2'` to the `REPLICA controller`.
- **(6).** The `REPLICA controller` will assign the `Global-NF-ID: 5002` for the NF in the H2.
- **(7).** One NF from the H3 starts and sends a `INIT-NF-ID request` for the NF-name: `'/onem2m/torino_5g/libeliumscanners/wifi/scanner3'` to the `REPLICA controller`.
- **(8).** The `REPLICA controller` will assign the `Global-NF-ID: 5003` for the NF in the H3.
- **(9).** The NF(5001) sends a `INIT-PUB-VAR-ID request` for the Variable-name: `'logs/wifi/scanner1'` to the `REPLICA controller`.
- **(10).** The `REPLICA controller` will assign the `Variable-ID: 1` for the NF(5001).
- **(11).** The NF(5001) starts to publish on the Variable-name: `'logs/wifi/scanner1'` with the `Variable-ID: 1`.
- **(12).** The NF(5002) sends a `INIT-PUB-VAR-ID request` for the Variable-name: `'logs/wifi/scanner2'` to the `REPLICA controller`.
- **(13).** The `REPLICA controller` will assign the `Variable-ID: 2` for the NF(5002).
- **(14).** The NF(5002) starts to publish on the Variable-name: `'logs/wifi/scanner2'` with the `Variable-ID: 2`.
- **(15).** The NF(5003) sends a `INIT-PUB-VAR-ID request` for the Variable-name: `'logs/wifi/scanner3'` to the `REPLICA controller`.
- **(16).** The `REPLICA controller` will assign the `Variable-ID: 3` for the NF(5003).
- **(17).** The NF(5003) starts to publish on the Variable-name: `'logs/wifi/scanner3'` with the `Variable-ID: 3`.
- **(18).** Another NF from the H2 starts and sends a `INIT-NF-ID request` for the NF-name: `'/onem2m/torino_5g/libeliumscanners/wifi/scanner4'` to the `REPLICA controller`.

- **(19).** The `REPLICA controller` will assign the `Global-NF-ID: 5004` for the second NF in the H2.
- **(20).** The NF(5004) sends a `INIT-PUB-VAR-ID request` for the Variable-name: `'logs/wifi/scanner4'` to the `REPLICA controller`.
- **(21).** The `REPLICA controller` will assign the `Variable-ID: 4` for the NF(5004).
- **(22).** The NF(5004) starts to publish on the Variable-name: `'logs/wifi/scanner4'` with the `Variable-ID: 4`.
- **(23).** The NF(5004) sends a `SUB-VAR-ID request` for the Variable-name: `'logs/wifi/scanner1'` to the `REPLICA controller`.
- **(24).** The `REPLICA controller` will reply with the `Variable-ID: 1` for the `'logs/wifi/scanner1'` to the NF(5004).
- (25). The NF(5004) sends a `SUB-register` request for the `Variable-ID: 1`.
- **(26).** The NF(5004) sends a `SUB-VAR-ID request` for the Variable-name: `'logs/wifi/scanner1'` to the `REPLICA controller`.
- **(27).** The `REPLICA controller` will reply with the `Variable-ID: 1` for the `'logs/wifi/scanner1'` to the NF(5004).
- **(28).** The NF(5004) sends a `SUB-register` request for the `Variable-ID: 2`.
- **(23).** The NF(5004) sends a `SUB-VAR-ID request` for the Variable-name: `'logs/wifi/scanner1'` to the `REPLICA controller`.
- **(29).** The `REPLICA controller` will reply with the `Variable-ID: 1` for the `'logs/wifi/scanner1'` to the NF(5004).
- **(30).** The NF(5004) sends a `SUB-register` request for the `Variable-ID: 3`.

Each NF generates a random table to `PUBLISH`, whit the `Variable-ID:` X, similar to:

```
1 {RSSI:-75, Vendor:Intel Corporate, TimeStamp:2019-06-05 19:20:43, MAC:
      E305A17508E4C60070513C08A3A7134E715FED1C49AAF4C679B37744},
2 {RSSI:-75, Vendor:Unknown, TimeStamp:2019-06-05 19:20:43, MAC:42914
      BBC0E3BEC2D0F5107A6F2D68964C5F235CA2B445E2824CBEA58},
3 {RSSI:-69, Vendor:Unknown, TimeStamp:2019-06-05 19:20:43, MAC:1532
      B055C9FFD6D3A45B4AE9A71091F70DFD49203AB95C20D9A6D768},
4 {RSSI:-69, Vendor:Google, TimeStamp:2019-06-05 19:20:43, MAC:0
      E04268AAFCC8D52DD640ADFECF0B2C43D400CD04DDCA2AABC4B222E},
5 ........
```

Where for each NF the X is determined by the `REPLICA controller` as explained in the previous steps. The tables are filled with randomly generated values, sorted by the time and in a Comma Separated Values (CSV) format. Generating the tables is started after receiving the reply to the `INIT-PUB-ID request` from the `REPLICA controller`, and every 20 seconds, a random number of lines (between 10 and 50) will be added to

the end of each table. By using the `bm_CLI` process, we can see the current values of the registers and entries of the tables for the BMV2.

- The results for the `Register-based:`

```
$ bm_CLI --thrift-port 9090 --json build/pub_sub.json --pre SimplePreLAG
↵
Control utility for runtime P4 table manipulation
RuntimeCmd: register_read subIndxPort ↵
register index omitted, reading entire array
subIndxPort= 0, 0, 0, 0
RuntimeCmd: register_read subIndxPort ↵
register index omitted, reading entire array
subIndxPort= 2, 0, 2, 0

```

The `subIndxPort` is the name of the registers array defined in our P4 program with indexes from 0 to 4, for four variable-IDs, to hold the corresponding `bitmask` of the input port of the requests. Their first values are all 0, while after the subscription phase, they are 2, 0, 2 and 0 respectively, while the lower indexed register is shown on left and the higher indexed one is on the right. It means through our definitions in chapter 3.4.2, subscriptions are:

  - **First** register value is 2, or $'b0010'$, which means on `variable_ID=1`, a subscription is made from port 2.
  - **Second** register value is 1, or $'b0001'$, which means on `variable_ID=2`, no subscription is made from any port.
  - **Third** register value is 2, or $'b0010'$, which means on `variable_ID=3`, a subscription is made from port 2.
  - **Forth** register value is 3, or $'b0000'$, which means on `variable_ID=4`, no subscription is made from any port.

- The results for the `Embedded Controller-based:`

```
$ bm_CLI --thrift-port 9090 --json build/pub_sub.json --pre SimplePreLAG
↵
Control utility for runtime P4 table manipulation
RuntimeCmd: table_dump MyIngress.pubsub_forward ↵
==========
TABLE ENTRIES
==========
Dumping default entry
Action entry: MyIngress.drop -
==========
RuntimeCmd: table_dump MyIngress.pubsub_forward ↵
==========
TABLE ENTRIES
**********
Dumping entry 0x0
Match key:
* scalars.local_metadata_t.pubsub_indx: EXACT 000001
Action entry: MyIngress.set_mcast_grp - 02
```

```
18      **********
19      Dumping entry 0x1
20      Match key:
21      * scalars.local_metadata_t.pubsub_indx: EXACT 000003
22      Action entry: MyIngress.set_mcast_grp - 02
23      **********
24      ==========
25      Dumping default entry
26      Action entry: MyIngress.drop -
27      ==========
28
```

The `pubsub_forward` is the empty table defined to keep the corresponding bitmask of the input port of the requests. Before subscriptions it is empty and after the subscription phase, it has three match-action rule, which means there exists only subscription for 3 `variable_ID` out of 4. Each rule has an `EXACT` match check on the `variable_ID` and will return an integer as the `multicast_group_ID`. The subscriptions are:

- **First** rule is an `EXACT` match on value 1, which return the 02 or $'b0010'$ as the `multicast_group_ID`, which means on `variable_ID` 2, a subscription is made from port 2.
- **Second** rule is an `EXACT` match on value 3, which return the 02 or $'b0010'$ as the `multicast_group_ID`, which means on `variable_ID` 3, a subscription is made from port 2.

As mentioned in section 3.5, the `Embedded Controller-based` seems to be more efficient than the `Register-based` in resource usage. In both solutions, the P4 switch will drop all received `PUBLISH` packets of the `Variable-ID: 2`, and the `Variable-ID`=4, due to not received the`SUB-register` for any of them from the NFs. It is notable that, nevertheless the NF(5004) which is running in the `H2` has send a `SUB-register request` for the `Variable-ID: 2`, which is being published by the NF(5002) that is also running in the `H2`, there is no subscription for the `Variable-ID: 2` in the switch for both solutions. The file records for received publishes by the NF(5004) show that it is receiving the `PUBLISH` updates for the `Variable-ID: 2`. This is due to the `Middle-ware` running in the H2, which recognized that an internal `PUBLISH` for the requested `Variable-ID` exists, so it did not send the request to the network and is handling a local state sharing inside the H2.

4.4.1 Remote Controller-based scenario with OpenFlow Switch

In this part, we implemented our scenario by using the traditional SDN OpenFlow-enabled switches and OpenFlow protocol. Considering the descriptions mentioned in section 4.1.4 and section 4.1.2, it is clear that, comparing BMV2 and Open vSwitch for speed test or the throughput, will not give us any real picture of the physical OpenFlow switches versus P4-programmable switches. So, while showing the applicability of our proposed protocol in the vanilla SDN, we tried to make a point of view for comparing the internal memory consumption, between P4-switches and a traditional OpenFlow switch in our scenario.

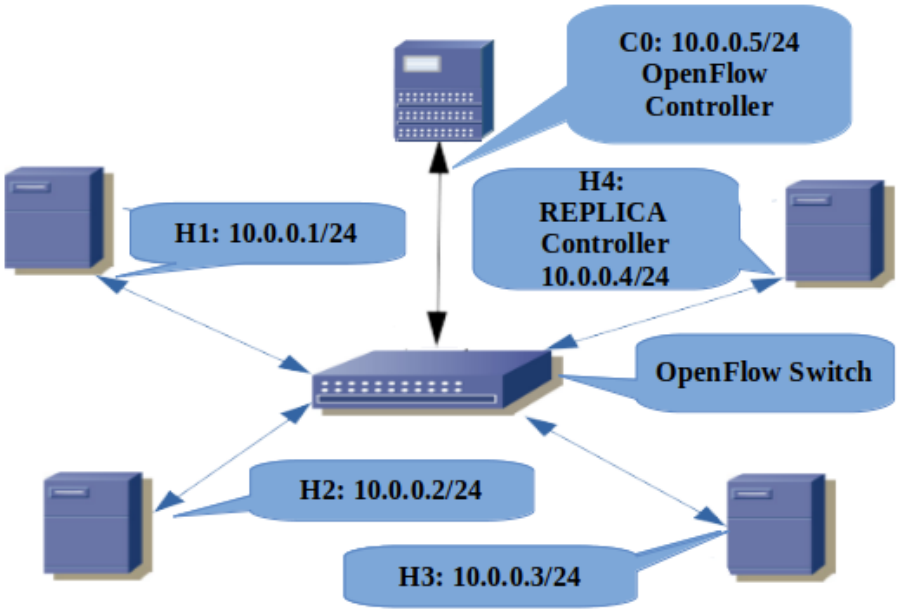


Figure 4.9. Remote Controller-based scenario with OpenFlow switch.

A simple topology for this scenario is shown in Figure 4.9.

This scenario needs to have two rules pre-installed inside the Open vSwitch, as the initialization phase, so at the start time, the controller will install those two initial rules inside the OpenFlow-enabled switch. After pre-install these two rules, the switch will send the all controller related packets to the controller and will drop Non-Registered Publish packets.

- The first rule is to recognize the `SUB-register request` and the `SUB-remove request` packets. By this rule, the switch will send them to the remote SDN controller, so that the controller can install proper rules inside the Switch for forwarding the requested publish packets. Its match rule in the simplest form should contain:

```
1      OXM_OF_ETH_DST = ff:ff:ff:ff:ff:ff // MAC destination,
2      OXM_OF_ETH_TYPE = 0x800            // Ethernet protocol=(IPv4),
3      OXM_OF_IPV4_DST = '239.128.0.0/9'  //IP-destination,
4      OXM_OF_IP_PROTO = 0x11             // IP-protocol=(UDP),
5      OXM_OF_UDP_DST = 65432             // UDP-destination-port
6
```

  while the other fields should be wild-carded. The Action will be (send to the `OFPP_CONTROLLER` port). Similar to the previous two scenarios, we presume that the route to the REPLICA controller for reply to the NF with the proper `NF-Global-ID` (for the `INIT-NF-ID request` and the `variable-ID` (for the `SUB-VAR-ID Request` and the `INIT-PUB-ID Request`) has already pre-installed by the remote SDN controller.

- The second rule is for dropping the Publishes that there were no registrations for them and so there are no Match-Action rules installed for them. This rule is OpenFlow version and switch configuration dependent and makes backward compatibility for OpenFlow before `v1.3`. In those versions default action for table-miss is, send to the controller, which floods the controller with unwanted publish packets. In higher versions, the default Action for this situation is `Drop` the packet. This matching rule in the simplest form should contain:

```
1       OXM_OF_ETH_DST = ff:ff:ff:ff:ff:ff // MAC destination,
2       OXM_OF_ETH_TYPE = 0x800            // Ethernet protocol=(IPv4),
3       OXM_OF_IPV4_DST = '239.0.0.0/10'   // IP-destination,
4       OXM_OF_IP_PROTO = 0x11             // IP-protocol=(UDP),
5       OXM_OF_UDP_DST = 65432             // UDP-destination-port,
6       priority = 0                       //(using a very low priority)
7
```

  and other fields wild-carded. The Action will be `Drop`.

The Algorithm and the controllers role

Anytime a `Pub-Sub` packet other than `PUBLISH` arrives at the switch ports, due to the pre-installed Match-Action rules, in the case of the `SUB-register request` or the `SUB-remove request`, the switch will encapsulate these messages and will send them to the remote SDN controller for inspecting and installing the necessary rules inside the Open vSwitch.

The general time-space diagram for this situation is demonstrated in Figure 4.10.

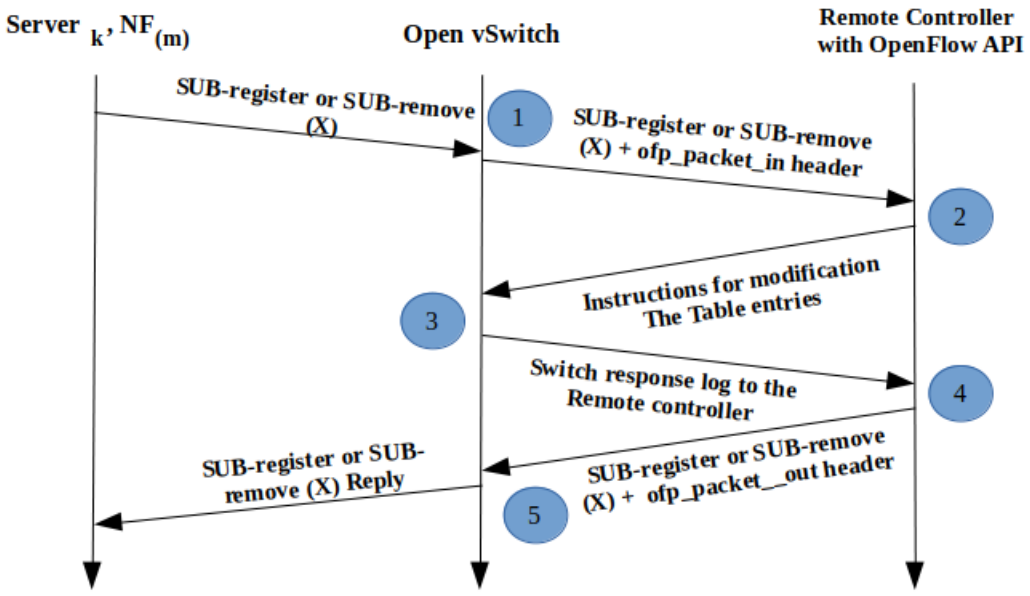


Figure 4.10. Time-Space diagram for Remote Controller-based scenario with OpenFlow switch.

The other steps including the communication between the NFs and the `Middle-ware` is topology independent and similar to the previously described one.

- **(1)** When a `SUB-register` or a `SUB-remove` packet arrives at the switch ports, it will be sent through OpenFlow API to the Remote SDN controller with an `OFP-Packet-In` header.

- **(2)** Due to the definition structure of the match fields in the OpenFlow SDN switches, if it is a `SUB-register`, the Controller will install a match rule inside the switch to forward any further publishes of this `Variable-ID`(e.g.15), to the port that this request already comes from it, with the priority higher than the priority of the default dropping rule, pre-installed for the `publishes` at the start of the controller (e.g., 2). This matching rule in the simplest form should contain:

```
1       OXM_OF_ETH_DST = ff:ff:ff:ff:ff:ff // MAC destination,
2       OXM_OF_ETH_TYPE = 0x800            // Ethernet protocol=(IPv4),
3       OXM_OF_IPV4_DST = '239.0.0.15/10'   // IP-destination(for var-id=15),
4       OXM_OF_IP_PROTO = 0x11             // IP-protocol=(UDP),
5       OXM_OF_UDP_DST = 65432             // UDP-destination-port,
6       priority = 2                       //(using a very low priority)
7
```

  and other fields wild-carded. The Action will be `send to port` k, where k is the port number that the `SUB-register` packet enters the switch. If the match already exists, the controller just updates the Action part by adding this port to the previous ones. In the case of Remove, the controller will update the Action part of the corresponding match rule by removing the mentioned port from the Action list if it exists or will remove the rule if no other actions remain for that rule. The controller will use the information of the packet header fields, the input port and its previous data structure on table entries, and will send proper instructions to the switch for installing new rules or updating previous ones.

- **(3)** Switch will report the configuration procedure and results to the SDN controller.

- **(4)** The Remote SDN controller will update its data structure with the switch report and will send the `Subscribe` packet it was received at **(1)** after swapping the `IP-destination` and `IP-source` fields, removing the `Packet-In` header and adding a `Packet-Out` header containing the same input port existed in `Packet-In` header, as the new output port for the switch.

- **(5)** The switch will remove the `Packet-out` header after extracting its information, and will send the packet to the NF, through the port mentioned in that information.

If the incoming packet to the switch is one of the `INIT-NF-ID request`, the `INIT-PUB-ID Request` or the `SUB-VAR-ID Request`, the switch will send the packet to the REPLICA Controller.

4.4.2 Switch memory consumption comparison

In this section, we compared the behavior of the BMV2 and Open vSwitch for the internal resource consumption as an estimated picture of their physical switches. Considering the descriptions in sections 4.1.2 and 4.1.4, one should expect to have a better approximation to the physical switch for the Open vSwitch. But due to the very efficient usage of memory that is clear from the structure of the RMT used in P4 switches in comparison to the structure of match tables in OpenFlow, the not efficiently developed BMV2 should use fewer resources than the well-optimized Open vSwitch. It means in a real physical comparison, the difference could be even more. As discussed in section 3.5, in our designs number of the rules is proportional to the number of the `variable_ID`s subscribed for. Knowing that each NF can subscribe on as many `variable_ID`s as we want, we can simply use only one NF for testing the switch memory consumption per each rule installation. For this part, the structure is simplified to the switch and one host containing only one NF. Following the description of the RSS in section 4.1.7, it sounds like a proper metric for us to measure the proportional memory consumption in the switches per each subscription, as an upper-bound estimation for the memory consumption of our proposed solutions, using the real switches. All the measurements start with one subscription and will end with 2049 subscriptions.

Registered-based with P4 switch

In this scenario, as we are using registers, so theoretically for each rule registration, we will occupy at list one register and some memory for the index of the register. Regarding the P4-16 language documents and our implementation, each register will be mapped to a 32-bit length memory area after compilation, regardless of the programmer register length definition in the main P4 program. Also, it will use a 32-bit number as indexing the mentioned register. So as a minimum lower band for the needed memory we expect that in the experiments we occupy at least 64-bits for each rule. We used one NF inside one host of the Mininet and one P4 switch. For this part of the experiment, we used the `ps` Linux command. After the initializing part, the subscription part of the NF program was changed to do the measurement. Regarding the architecture of the P4-program of the switch described in the section 3.4.2, we defined 2049 registers inside the P4 program at the compile time, to have enough pre-defined registers for the tests. The test algorithm is described in the below pseudo-code:

```
For i in range(2049):
    MEASURE the RSS of the BMV2 process
    SEND subscribe packet for (VARIABLE_ID = i)
    WAIT for (1) second # To receive the reply
    WHILE no subscription confirm received:
        SEND subscribe packet for (VARIABLE_ID = i)
        WAIT for (1) second
    i = i+1
```

The measurement phase is done inside the NF, through the steps are shown in the Figure 4.11:

- **(1)** The NF will measure the RSS of the BMV2.

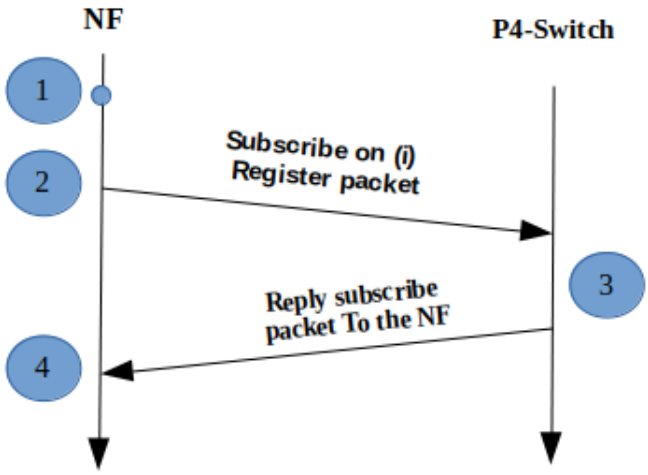


Figure 4.11. Time-Space diagram for the Register-based RSS measurement steps.

- **(2)** The NF will send the `subscription-register` packet for i-th `variable_ID` and waits 1 second for receiving the reply. If it does not receive the reply it will resend the same packet.
- **(3)** The P4-switch generates the bit-mask of the input port of the packet and will save the bit-mask inside its $i-1$-th internal register. Then it mirrors the received packet, back to the sender NF as a register-reply.
- **(4)** If the NF receives the reply, it increases the value of the i to the $i+1$ and goes to the step **(1)**.

The result of the measurement is shown in the Figure 4.12:

It is clear from the Figure 4.12 that the RSS size of the switch is not showing any changes. By checking the register's value of the switch with the help of the `bm_CLI` process, it becomes clear that the registration was done correctly and completely as all the registers had non-zero and equal values. By repeating several times the whole test, only from one total test to another total one, the total RSS size was changing. From the definition of the RSS, the variation of the measured RSS, in the case of recompiling and restarting the switch at the beginning of each total test was reasonable, as, in each start, the amount of the shared libraries inside the memory was changing. But the constant value for the RSS during our rule installations, which is a kind of writing in the memory, may have only one explanation: **That part of the memory assigned to the `Registers` is occupied somehow by the BMV2 at each start!**, this might be due to how the developers of the BMV2 defined the registers. If this assumption is right, then in this scenario, we expect that the effect of defining each of the registers inside the program, to be equal to what we expect from installing one rule inside the switch in terms of the memory occupancy increment. So for simulating the effect of the rule installation, instead of just sending a `SUB-register` request and wait for the reply from

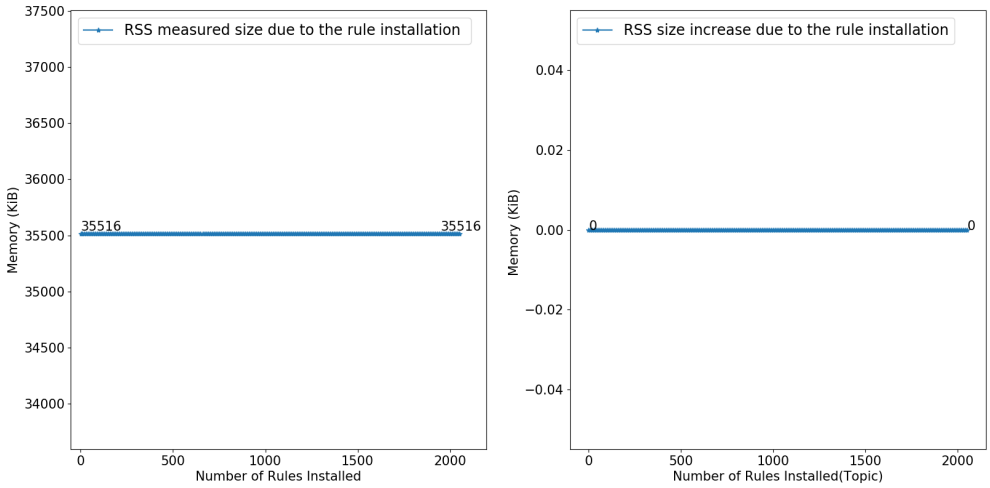


Figure 4.12. Register-based RSS measurement with the `ps` command.

the switch, we needed to repeat some more steps. We needed to change the amount of the registers inside the P4 program of the switch, compile the program, prepare and start the needed Mininet environment, start the switch, measure the RSS, shutdown properly the switch and the Mininet environment, for each test value. For automation the test steps we used the `time -v` command. Due to the RSS definition, and its behavior inside the `Linux`, for different runs with the same number of the registers, we will have different measured values, we decided to repeat the test several times for each value of the registers and calculate the 95% confidence interval for the measured values. The test algorithm is described in the below pseudo-code:

```
for i in range(1, 129): # Number of the test steps(128)
    SET the number of registers equal to i
    for j in range(10): # Number of the measurements per each test step(10)
        RUN the scripts of starting the switch as an argument of the time -v command
        STORE the RSS upper-bound measurement of the BMV2 process by the time
    -v command
        j = j+1
    i = i+1
```

To have a better view of the memory occupancy behavior, we added a linear regression over the mean values of the calculated 95% confidence intervals. For showing the stability of our results, we repeated the whole tests with three different numbers of the test steps and three different numbers of the measurements per each test step. The result of the measurements is shown in the Figures 4.13, 4.14 and 4.15:

The linear regressions of the results have the same slope, which shows the stability of our measures.

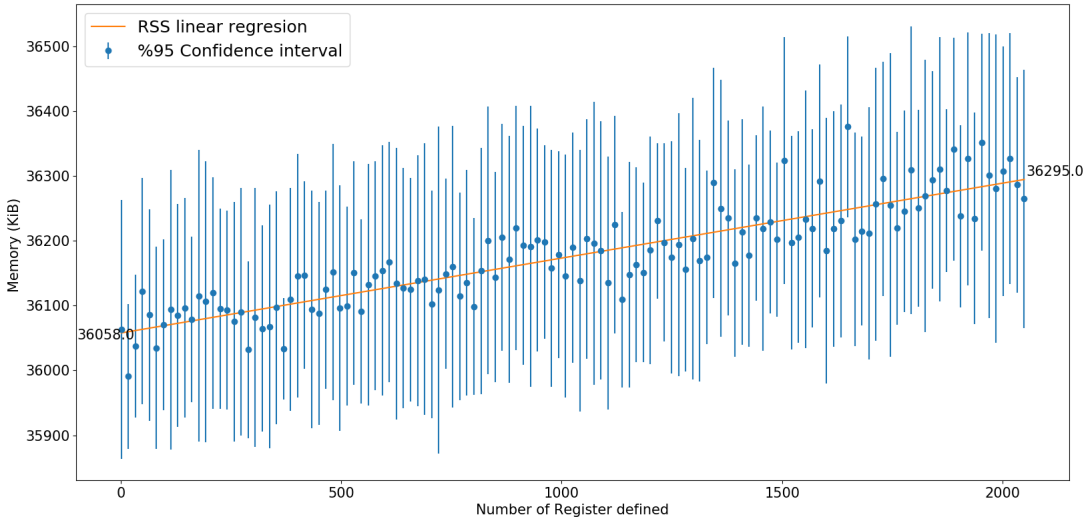


Figure 4.13. Register-based RSS measurement using `time` command, with 128 steps and 10 measures for each step.

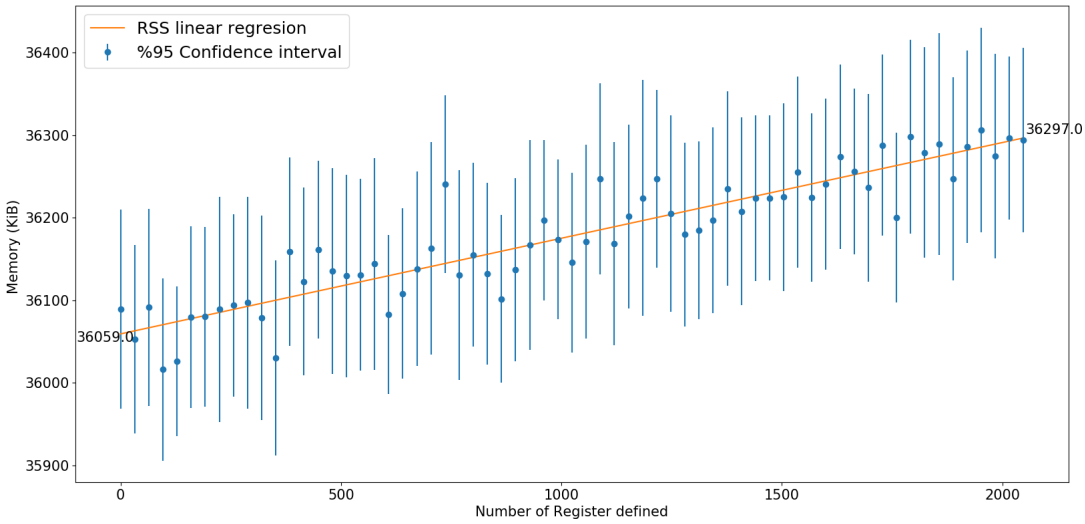


Figure 4.14. Register-based RSS measurement using `time` command, with 64 steps and 20 measures for each step.

Embedded controller-based with P4 switch

In this scenario, as we are using a `Match-Action` table instead of registers. By considering the fact that, we are using the RMT(Re-configurable Match Table), so theoretically for each rule registration, at least we need to store a `Key` for the `Match`, and a `value` for the `Action`. Because we are using the `Exact` matching on an IP address, so 32 bits are needed,

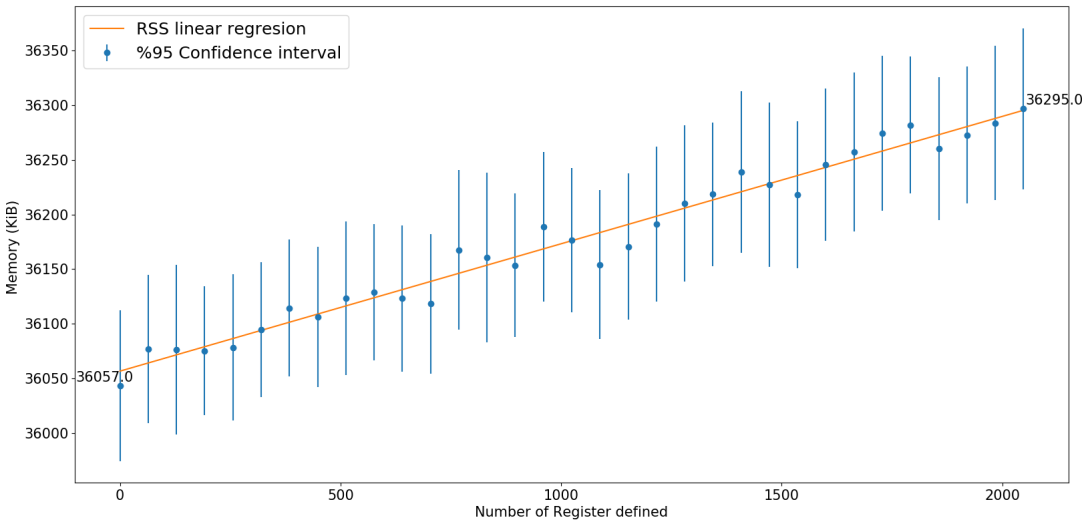


Figure 4.15. Register-based RSS measurement using `time` command, with 32 steps and 50 measures for each step.

and action is an integer number representing our `Multi-cast` group id, that adds another 32 bits. While as a minimum lower band for the needed memory we expect that, in the experiments, we occupy at least 64-bits for each rule, the more complex structure of the tables gives the sense that it would need more memory than the register-based solution as it needs at least more mapping steps and for defining the table structure. We used one NF inside one host of the Mininet and one P4 switch. For this part of the experiment, we used the `ps` Linux command. After the initializing part, the subscription part of the NF program was changed to do the measurement. The test algorithm is described in the below pseudo-code:

```
for i in range(2049):
    MEASURE the RSS of the BMV2 process
    SEND subscribe packet for (VARIABLE_ID = i)
    WAIT for (1) second # To receive the reply
    WHILE no subscription confirm received:
        SEND subscribe packet for (VARIABLE_ID = i)
        WAIT for (1) second
    i = i+1
```

The measurement phase is done inside the NF, through the steps are shown in the Figure 4.16:

The steps in the Figure 4.16 are described as bellow:

- **(1)** The NF will measure the RSS of the BMV2.
- **(2)** The NF will send the `subscription-register` packet for i-th `variable_ID` and waits 1 second for receiving the reply. If it does not receive the reply it will resend the same packet.
- **(3)** If the NF receives the reply, it increases the value of the i to the $i+1$ and goes

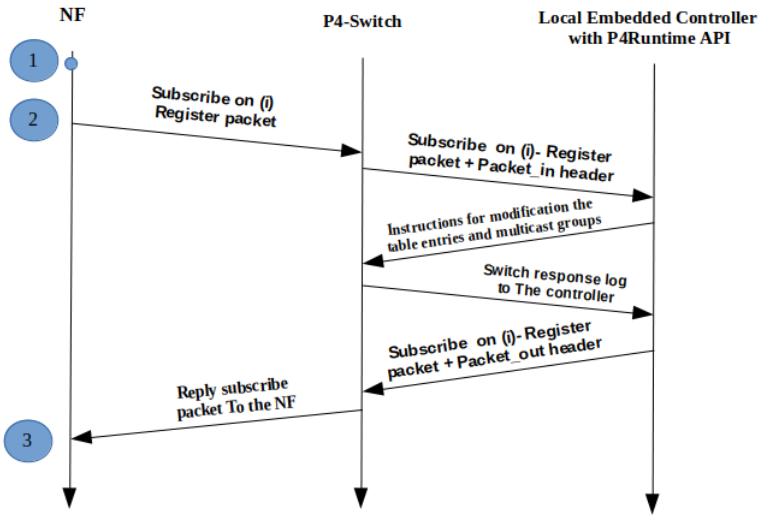


Figure 4.16. Time-Space diagram for Embedded Controller-based RSS measurement steps.

to the step **(1)**.

The result of the measurement is shown in Figure 4.17:

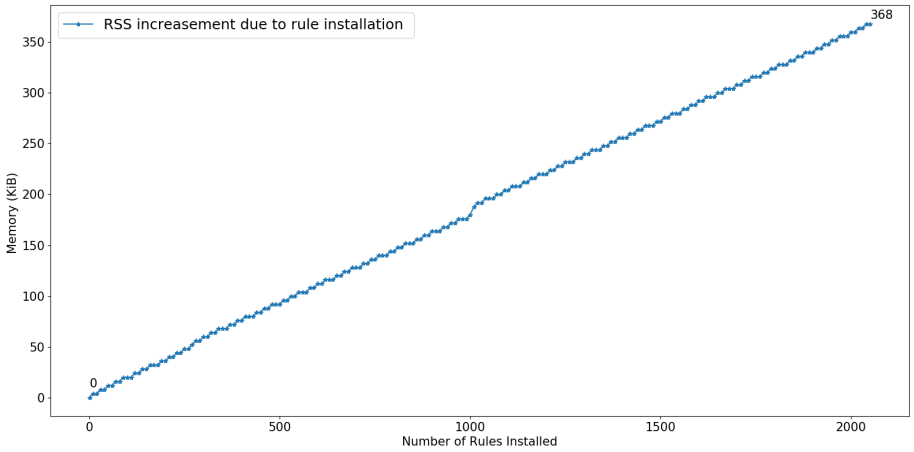


Figure 4.17. Embedded Controller-based RSS measurement.

Remote controller-based with OpenFlow and the Open vSwitch

In this scenario, due to the OpenFlow definitions, we are using an MMT(Multiple Match Table) `Match-Action` table instead of registers or RMT tables used with the P4. At least we need to store a `Key` for the `Match`, and a `value` for the `Action`. Because we are using the `Exact` matching on an IP address and we are using the MMT tables, so theoretically for each rule registration, we need to match the `MAC destination address`(48 bits), the `Ether-type`(16 bits) for IPv4, the*IPv4 destination address*(32 bits), the `IP protocol`(8 bits) for the UDP, and the `UDP destination port`(16 bits). It means at least 120 bits are needed for the matching part, and the action is an integer number representing our `Multi-cast` group id that adds another 32 bits, which ends in 152 bits. While as a minimum lower band for the needed memory we expect that, in the experiments, we occupy at least 152 bits for each rule, the more complex structure of the tables gives the sense that it would need even more memory. For this part of the experiment, we used Open vSwitch(version 2.5.5), which supports OpenFlow versions 1.0, 1.1, 1.2 and 1.3 completely, and the 1.4 and 1.5, which are supported, but missing some features until be enabled by the user. We used the `RYU` controller, which by default supports all the OpenFlow protocols including 1.0, 1.1, 1.2, 1.3, 1.4 and 1.5. Because we are using a rule which has only one changing part, (`IPv4 Destination Address`, so using different OpenFlow versions (e.g., 1.0 and 1.3) should not make much difference between using SMT(used in OpenFlow v1.0) and the MMT(used in other OpenFlow protocols)due to their structural definitions. To begin the test, we used one NF inside one host of the Mininet and one Open vSwitch. After the initializing part, the subscription part of the NF program was changed to do the measurement and we used the `ps` Linux command as the measurement tool for the RSS. The test algorithm is described in the below pseudo-code:

```
for i in range(2048):
    MEASURE the RSS of the OVSK process
    SEND subscribe packet for (VARIABLE_ID = i)
    WAIT for (1) second # To receive the reply
    WHILE no subscription confirm received:
        SEND subscribe packet for (VARIABLE_ID = i)
        WAIT for (1) second
    i = i+1
```

The measurement phase is done inside the Remote SDN controller, through the steps are shown in the Figure 4.18:

- **(1)** The NF will measure the RSS of the Open vSwitch.
- **(2)** The NF will send the `subscription-register` packet for i-th `variable_ID` and waits 1 second for receiving the reply. If it does not receive the reply it will resend the same packet.
- **(3)** If the NF receives the reply, it increases the value of the i to the $i + 1$ and repeats the scheduler from step **(1)**.

The measurements for both OpenFlow V1.0 and V1.3 showed the same behavior, while the measured values for the OpenFlow V1.3 were slightly bigger than the OpenFlow

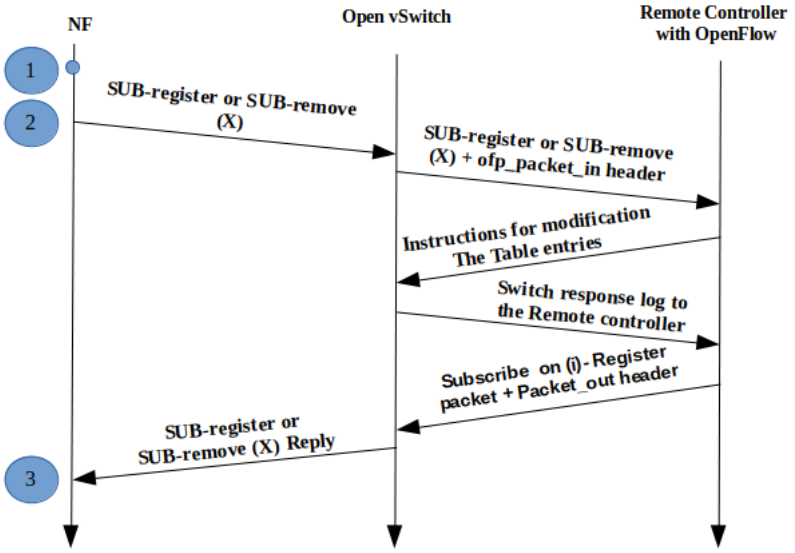


Figure 4.18. Time-Space diagram for Remote Controller-based RSS measurement steps.

v1.0. It can be due to the probable existence of extra wild-carded header fields before the `UDP destination port` in the OpenFlow V1.3 (16 bits). It is worthy to recall that OpenFlow V1.3 has 40 match fields, while OpenFlow V1.0 has only 12 match fields. Due to their slight difference in their results we just used one of the results. The result of the measurement is shown in Figure 4.19:

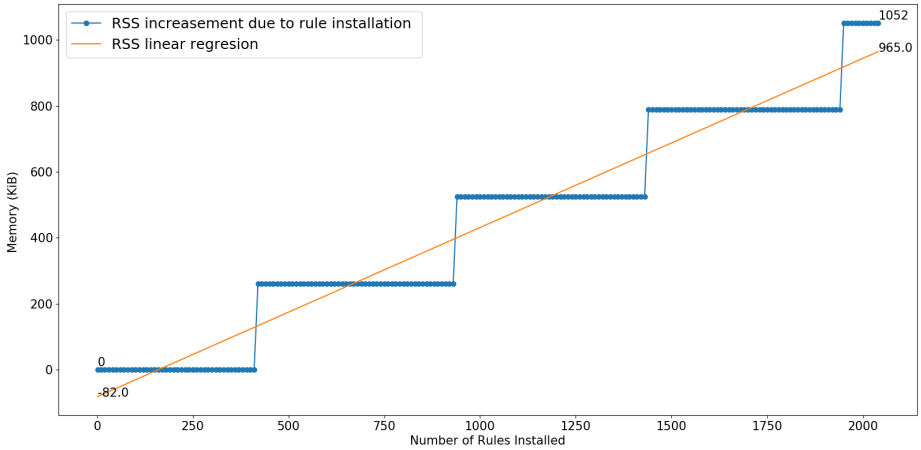


Figure 4.19. Remote Controller-based RSS measurement.

Overall comparison

By using the linear regression of the above results, it is possible to see the estimated behavior of the three scenarios in switch internal resource consumption for 2048 rule installation.(see Figure 4.20)

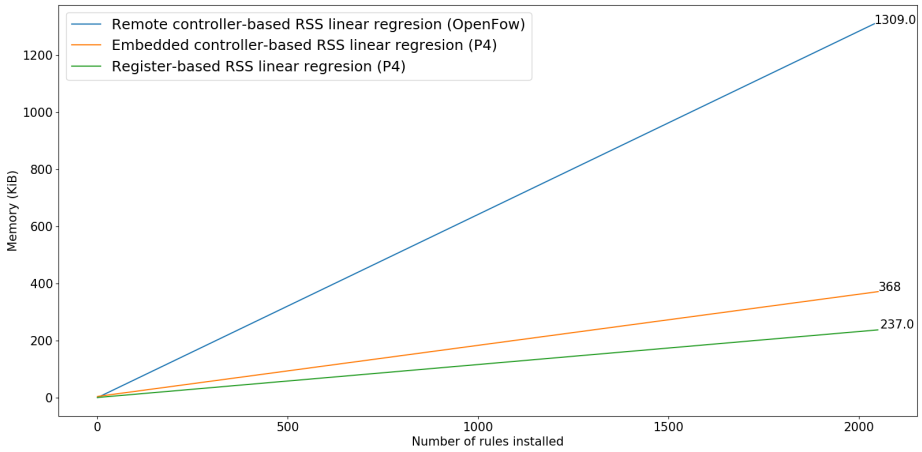


Figure 4.20. The upper-bound measured memory consumption comparison.

The upper-bound measured and the lower-band estimated memory occupancy per rule installation in the switch for the different scenarios are shown in the Table 4.1

SCENARIO	Upper-bound measured memory occupancy (Bytes)	Lower-bound estimated memory occupancy (Bytes)
Register-based	$\frac{237\times1024}{2048} \approx 118.5$	$\frac{64}{8} = 8$
Local Embedded Controller-based	$\frac{368\times1024}{2048} \approx 184$	$\frac{64}{8} = 8$
Remote Controller-based	$\frac{1309\times1024}{2048} \approx 654.5$	$\frac{152}{8} = 19$

Table 4.1. The upper-bound measured and the lower-band estimated memory occupancy per rule installation

Chapter 5

Conclusion

In this work, we have proposed a solution for sharing the states between different Virtual Network Functions(VNF) with the help of the P4 language programmable switches, while the generality of the proposed solution makes it applicable on the OpenFlow enabled switches as well. The flexibility of the P4 programmable switches and the P4 language made us capable to have two different implementations in terms of switch pipeline structure for the solution, and each of the implementations has some advantages for different applications. In the second chapter of this work, we briefly recalled the Software-Defined Networking and its most successful protocol, OpenFlow. Then we took a look at the new approach for refining the Multi-Match Table(MMT), by introducing the Re-configurable Match Table(RMT) as an efficient less resource-demanding approach. The Re-configurable Match Table(RMT) led to the birth of a new, novel and state of the art language for programming the data-planes of the new switches, the P4 language. Furthermore, we reviewed the P4Runtime, an API like OpenFlow, but made for providing the controllers with the ability to communicate with the P4-programmable switches for different purposes, while allows them to remotely reprogram partially or the whole P4 program of the P4-programmable switches by acting as a Remote Procedure Call(RPC) tool, or even locally by acting as an Inter-Process Communication(IPC) tool too. After that, we talked about Network Function Virtualization, the state-full VNFs, and their important role in the 5G networks. In the third chapter, we described our solution for state sharing between the VNFs, which is using the Publish/Subscribe scheme. We explained the whole protocol, the phases, the general message formats, and we derived two ways of implementing the solution, by two different methods of storing the subscriptions inside the P4 switches, thanks to the flexibility of the pipeline programming and state storing provided by the P4 programming language. Finally, we evaluated theoretically the solutions and showed that their scalability in terms of the number of the NFs or the servers is not bounded, while the P4 programmable switches internal resources is the only upper-bound limitation for the number of variables that can be participated in our solutions if the number of the variables do not cross the 2^{22} variables. In the fourth chapter, as a Proof Of Concept(POC), we implemented both solutions by using the Python programming language. Then we described the possible problems or inefficiencies that

might occur due to sharing a virtual machine or host by two or more VNFs, and as a solution, we introduced and implemented a middle-ware, placed as a node between the VNFs and their virtual machine. Finally, for comparison, we implemented a vanilla SDN based structure with our proposed protocol and compared our solutions with it in terms of the switch resource consumption. We showed that our solutions have more flexibility with the help of the P4 language compared to the OpenFlow. We also showed that with the help of the P4 language compared to the vanilla SDN using OpenFlow enabled switches, the amount of the internal switch resources needed to successfully store the instructions for state sharing, can be efficiently reduced to less than one third.

Bibliography

[1] Diego Kreutz, Fernando M. V. Ramos, Paulo Esteves Veríssimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. "Software-Defined Networking: A comprehensive survey". IEEE, 2014.

[2] REpresentational State Transfer (REST). https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

[3] Common Object Request Broker Architecture (CORBA). https://www.ibm.com/support/knowledgecenter/en/SSMKHH_9.0.0/com.ibm.etools.mft.doc/bc22400_.htm.

[4] Simple Network Management Protocol APIs (SNMP. https://www.ibm.com/support/knowledgecenter/en/ssw_ibm_i_74/apis/unix6.htm.

[5] OpenFlow switch specification: version 1.5.1 (Protocol version 0x06). https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf.

[6] Network Configuration Protocol (NETCONF). https://tools.ietf.org/html/rfc6241.

[7] XMPP : The open standard for messaging and presence. https://xmpp.org/.

[8] OpenFlow messages example. https://www.telematica.polito.it/app/uploads/2018/07/sdn-switches.pdf.

[9] Stateless transport tunneling protocol for network virtualization. https://tools.ietf.org/html/draft-davie-stt-08.

[10] Virtual eXtensible Local Area Network (VXLAN). https://datatracker.ietf.org/doc/rfc7348/.

[11] Network Virtualization Using Generic Routing Encapsulation (NVGRE). https://datatracker.ietf.org/doc/rfc7637/.

[12] Chang Ching-Hao and Dr. Ying-Dar Lin. "OpenFlow version roadmap". Semantic-Scholar, 2015.

[13] Masoud Moshref, Apoorv Bhargava, Adhip Gupta, Minlan Yu, and Ramesh Govindan. "Flow-level state transition as a new switch primitive for SDN". ACM SIGCOMM '14, 2016.

[14] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. "OpenState: programming platform-independent stateful OpenFlow applications inside the switch". ACM SIGCOMM, 2016.

[15] Giuseppe Bianchi, Marco Bonola, Antonio Capone, and Carmelo Cascone. "SNAP: Stateful Network-wide Abstractions for Packet Processing". ACM SIGCOMM, 2014.

[16] Marco Bonola, Roberto Bifulco, Luca Petrucci, Salvatore Pontarelli, Angelo Tulumello, and Giuseppe Bianchi. "Swing State: consistent updates for stateful and programmable data planes". ACM SOSR '17, 2017.

[17] Marco Bonola, Roberto Bifulco, Luca Petrucci, Salvatore Pontarelli, Angelo Tulumello, and Giuseppe Bianchi. "Implementing advanced network functions for datacenters with stateful programmable data planes". IEEE, 2017.

[18] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeowna nd Martin Izzard, Fernando Mujica, and Mark Horowitz. "Forwarding Metamorphosis: fast programmable match-action processing in hardware for SDN". ACM SIGCOMM '13, 2013.

[19] German Sviridov, Marco Bonola, Angelo Tulumello, Paolo Giaccone, Andrea Bianco, and Giuseppe Bianchi. "LODGE: LOcal Decisions on Global statEs in prograna naable data planes". IEEE, 2018.

[20] Pisa: Protocol-independent switch architecture. https://events19.linuxfoundation.org/wp-content/uploads/2018/08/P4-based-Programmable-Forwarding-Plane-1.pdf.

[21] P4(16) Portable Switch Architecture (PSA). https://p4.org/p4-spec/docs/PSA.pdf.

[22] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talaycoa nd Amin Vahdat, George Varghese, and David Walker. "P4: Programming Protocol-independent Packet Processors". ACM SIGCOMM '14, 2014.

[23] P4-16 Language Specification version 1.2.0-rc. https://p4.org/p4-spec/docs/P4-16-v1.2.0.pdf.

[24] P4Runtime Specification version 1.0.0. https://p4.org/p4runtime/spec/master/P4Runtime-Spec.pdf.

[25] What are protocol buffers? https://developers.google.com/protocol-buffers/docs/overview.

[26] p4lang/p4Runtime repository:P4Runtime protobuf definition files and specification. https://github.com/p4lang/p4runtime..

[27] gRPC ducumentation. https://grpc.io/docs/guides/.

[28] ETSI NFV MANO. https://sdn.ieee.org/newsletter/july-2016/opensource-mano.

[29] MQTT: message queueing transport telemetry scheme. https://randomnerdtutorials.com/what-is-mqtt-and-how-it-works/.

[30] MININET An instant virtual network on your laptop (or other PC). http://mininet.org/.

[31] BMV2 Behavioral Model Version 2. https://github.com/p4lang/behavioral-model.

[32] BMV2 Behavioral Model Version 2. https://github.com/p4lang/tutorials/blob/master/P4_tutorial.pdf.

[33] OVSK Open Virtual Switch. https://www.openvswitch.org/.

[34] Open-vSwitch: Why OVS? https://github.com/openvswitch/ovs/blob/master/Documentation/intro/why-ovs.rst.

[35] SCAPY Packet crafting library for Python2 and Python3 . https://scapy.net/.
[36] SOCKET Low-level networking interface for Python2 and Python3 . https://docs.python.org/3/library/socket.html.
[37] P4 Tutorial: acm sigcomm august 2019 tutorial on programming the network data plane. https://github.com/p4lang/tutorials.