ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΕΠΙΚΟΙΝΩΝΙΩΝ, ΗΛΕΚΤΡΟΝΙΚΗΣ & ΣΥΣΤΗΜΑΤΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μέθοδοι εξουσιοδότησης για δέσμευση πόρων σε Ευφυή-Προγραμματιζόμενα-Δίκτυα (Software-Defined-Networks)

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Σπυρίδων Γ. Μαστοράκης

Επιβλέπων : Βασίλειος Μάγκλαρης
Καθηγητής Ε.Μ.Π

Αθήνα, Μάιος 2014

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΕΠΙΚΟΙΝΩΝΙΩΝ, ΗΛΕΚΤΡΟΝΙΚΗΣ & ΣΥΣΤΗΜΑΤΩΝ
ΠΛΗΡΟΦΟΡΙΚΗΣ

Μέθοδοι εξουσιοδότησης για δέσμευση πόρων σε Ευφυή-Προγραμματιζόμενα-Δίκτυα (Software-Defined-Networks)

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Σπυρίδων Γ. Μαστοράκης

Επιβλέπων : Βασίλειος Μάγκλαρης
Καθηγητής Ε.Μ.Π

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 27η Μαΐου 2014.

Αθήνα, Μάιος 2014

....
Σπυρίδων Γ. Μαστοράκης

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Το Διαδίκτυο με τη σημερινή του δομή έχει συμβάλει τα μέγιστα στην ανάπτυξη εικονικών περιβάλλοντων. Ωστόσο, η δομή του αυτή εισάγει περιορισμούς ευρείας κλίμακας στην ανάπτυξη καινοτόμων εφαρμογών. Για το λόγο αυτό, εισήχθησαν τα «Ευφυή-Προγραμματιζόμενα-Δίκτυα» (Software-Defined-Networks), τα οποία αναμένεται να αποτελέσουν τη βάση του Διαδικτύου του μέλλοντος, συμβάλλοντας στην ακόμα μεγαλύτερη ανάπτυξη των Νέφων Υπολογιστών (Cloud Computing) και των εικονικών δικτύων (virtualized networks).

Σημαντικό ρόλο προς αυτή την κατεύθυνση αναμένεται να διαδραματίσει το πρωτόκολλο OpenFlow, το οποίο σε συνδυασμό με την αρχιτεκτονική των Ευφυών-Προγραμματιζόμενων-Δίκτυων επιτρέπει το διαχωρισμό του επιπέδου ελέγχου από το επίπεδο προώθησης πακέτων σε ένα δίκτυο. Επιπροσθέτως, επιτρέπεται η ύπαρξη πολλαπλών «ενοικιαστών» (tenants) κατά μήκος ενός κοινού μοιραζόμενου δικτυακού υποστρώματος. Ένας από τους βασικούς στόχους της εικονοποίησης δικτύων (network virtualization) είναι η παροχή σε κάθε «ενοικιαστή» της ψευαίσθησης ότι καταναλώνει μόνος του όλους τους διαθέσιμους δικτυακούς πόρους. Για το λόγο αυτό, κάθε «ενοικιαστής» μπορεί να ζητήσει το δικό του κομμάτι δικτυακών πόρων (network slice). Επίσης, οι δικτυακοί πόροι και οι ενέργειες κάθε «ενοικιαστή» δε θα πρέπει να συγκρούονται (conflict) με τους πόρους των υπολοίπων ενοικιαστών.

Για όλους τους παραπάνω λόγους, γίνεται επιτακτική η ανάγκη μελέτης τρόπων συνύπαρξης και απομόνωσης μεταξύ των «ενοικιαστών» κατά μήκος του φυσικού δικτυακού υποστρώματος.

Λέξεις κλειδιά: «Ευφυή-Προγραμματιζόμενα-Δίκτυα», Νέφη Υπολογιστών, εικονικά δίκτυα, κομμάτι δικτυακών πόρων, συνύπαρξη πολλαλών «ενοικιαστών» δικτύου, απομόνωση «ενοικιαστών».

ABSTRACT

Internet, with its current structure, has greatly contributed to the introduction and the development of virtual environments. However, this Internet structure introduces limitations on the development of innovative applications. In this context, Software-Defined-Networks (SDNs) were introduced and are expected to constitute the core of Future Internet contributing to the even greater development of Cloud Computing and network virtualization.

The introduction and standardization of the OpenFlow (OF) protocol plays an important role in this effort. SDN, based on the OpenFlow protocol, enables the decoupling of control and data plane. Furthermore, multi-tenancy is enabled across a shared physical network substrate. One of the major goals of network virtualization is to provide to each tenant the perception that it uses the available network resources exclusively on its own. In this context, each tenant can request its own network slice. Moreover, the requested network slices should not conflict with each other.

For all the reasons mentioned above, it is imperative to study various possible methods of coexistence and isolation among multiple tenants over a shared physical network substrate.

Key words: Software-Defined-Networking, Cloud Computing, network virtualization, network slice, multi-tenancy, tenant isolation.

Ευχαριστίες

Η διπλωματική αυτή εργασία αποτελεί το τελευταίο στάδιο των προπτυχιάκων μου σπουδών στο Εθνικό Μετσόβιο Πολυτεχνείο. Αρχικά, θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου, κ. Βασίλειο Μάγκλαρη, για τη συνολική και πολύπλευρη βοήθεια και καθοδήγηση του. Ακόμη, θα ήθελα να ευχαριστήσω όλα τα μέλη του εργαστηρίου NETMODE για τη συνεργασία και την άκρως φιλική τους διάθεση που με έκανε πραγματικά να μη θέλω να φεύγω τα βράδια από το εργαστήριο. Ειδικότερα, θα ήθελα να ευχαριστήσω τον υποψήφιο Διδάκτορα Χρήστο Αργυρόπουλο για την άψογη συνεργασία και την προθυμία του να με καθοδηγήσει όποτε αυτό χρειάστηκε.

Τέλος, θα ήθελα να ευχαριστήσω την οικογένειά μου για την αμέριστη υποστήριξη που μου έχουν προσφέρει όλα αυτά τα χρόνια σε όλα τα στάδια της ζωής μου, αλλά και τους φίλους για την κατανόηση που έχουν δείξει καθ’ όλη τη διάρκεια των σπουδών μου.

1. Table of Contents

Figures

Tables

Code snippets

2. Introduction

2.1. Research problem and Approach

Modern computer design is greatly based on the concept of virtualization in order to decouple service provisioning from physical resources. More specifically, the emerging cloud computing ecosystem and its major trends (e.g. Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Service as a Service (SaaS)) are mainly based on the concept of network virtualization.

The concept of network virtualization has become of even greater importance since the emergence of Software-Defined-Networking (SDN) [1]. SDN, based on the OpenFlow (OF) protocol [2],[3], introduced the decoupling of control and data plane and the concept of multi-tenancy over a shared physical network substrate. Multi-tenancy, as a feature of SDN, refers to the existence of multiple tenants across a common physical network topology.

In this context, each tenant can request its own network slice; a basket of allocated logical and physical network resources across one or more parts of physical network topology. In this way, tenants can run their own forwarding logic and develop advanced service functionalities within their virtual network (network slice), without being aware either of the physical network substrate or the existence of other tenants.

A typical method to achieve multi-tenancy is to introduce packet classification into flows via a logical separator, typically a field within the packet header. In this context, each OpenFlow controller considers a packet ID, preferably a Layer 2 header field, as the identifier of the network slices. Thus, in this thesis, three such methods, called “**network control plane slicing methods**”, are proposed, implemented in software and assessed. These methods are the following: **(i) domain-wide slicing**, **(ii) switch-wide slicing** and **(iii) port-wide slicing**.

In a multi-tenant SDN environment, each tenant should not be able to exploit network resources that are delegated to other tenants (network slices). Thus, isolation should be enforced among network slices, so that they do not conflict with each other. In other words, a network slice should not be allowed to exhaust the network resources of other network slices. The rules, which should be generated in order to enforce an isolation policy among network slices, constitute the required flowspace [4]. Given that prerequisite, the proposed implementation also generates the required flowspace rules, so that the isolation enforcement among tenants is achieved.

2.2. Thesis Contribution

This thesis mainly intends to propose and assess various slicing policies that enable the efficient flowspace segmentation among tenants. Specifically:

- Three control plane slicing methods are proposed. These methods are applicable to diverse SDN architectures.
- An isolation policy across SDN environments is discussed and a rule reduction approach is proposed.
- The efficiency of the proposed slicing methods over multiple real network topologies is assessed and the required flowspace is generated.
- An evaluation of the slicing method feasibility is performed. Thus, the slicing methods are associated with FlowVisor, a popular OpenFlow proxy controller, and the generated flowspace rules are injected into this proxy controller.

2.3. Thesis outline

The remainder of this thesis is organized as follows: in section 3, the background of this research issue and the related work are summarized. In section 4, the overall design of the proposed implementation is analyzed. In section 5, the analysis of the software implementation is performed. In section 6, the evaluation of this software implementation is performed. In section 7, the conclusions are summarized and the future work is described. Finally, in section

8, the used references are listed and, in section 9, an appendix is included.

3. Background

In legacy network architectures, each network device constitutes an autonomous forwarding entity. Within such devices, the required forwarding, control and management functionalities are designed and implemented in distinct groups. These major groups constitute forwarding/data plane, control plane and management plane respectively. However, each vendor permits to a different, but always limited, extent the programmability and control of routers and Ethernet switches by network administrators. Moreover, each vendor designs network devices for specific markets. As a result, the mechanisms associated with the above functionalities are implemented in a different way by each vendor. This policy often results in major traffic management incompatibilities among devices of different vendors.

In order to overcome these limitations, we can take advantage of Software-Defined Networking (SDN) [1] based on the OpenFlow (OF) protocol [2],[3]. Nowadays, the majority of Ethernet switches are OpenFlow-enabled and, as a result, they contain flow tables for the implementation of services such as Network Address Translation (NAT), Quality of Service (QOS) and Firewall [5]. OpenFlow provides a protocol for the programmability of these flow tables. Each OpenFlow-enabled switch is controlled by an OpenFlow controller that can insert or delete flows in/from the flow table of each switch.

Generally speaking, SDN, based on the OpenFlow protocol, transforms network devices to fully programmable forwarding elements. Modern system design often employs virtualization to decouple the system service model from its physical realization. Thus, the OpenFlow protocol constitutes a concrete substrate for the development of multi-tenant virtualized environments. By using a network hypervisor [6], one can fully virtualize a physical network substrate, by inserting distinct abstraction layers in order to achieve operational goals divorced from the underlying physical infrastructure. On the other hand, the deployment of an OpenFlow transparent proxy controller (e.g. FlowVisor [4]) can result in the delegation of various network resources, under the form of network slices, to multiple tenants.

In the sections below, all the mentioned concepts are described in detail.

3.1. Networking planes

A plane, in networking context, is one of the three integral components of a telecommunication architecture. As mentioned above, these three integrals are: **(i) forwarding/data plane**, **(ii) control plane** and **(iii) management plane**. In legacy networks, all the three planes are implemented in the firmware of routers and switches.

3.1.1. Forwarding/Data plane

Typically, the forwarding/data plane is locally implemented within each network device and operates based on the line-rate. The forwarding/data plane refers to the underlying systems, which forward a packet to a selected destination. Said another way, the data plane is mainly responsible for the process of packet forwarding based on forwarding rules (e.g. longest-prefix match) and for the simultaneous check of Access Control Lists (ACLs). Moreover, queue management and packet scheduling are implemented in the context of this plane. All the above operations are based on hardware components.

Despite the fact that the forwarding/data plane implementation varies among vendors, network devices communicate with each other via standardized data forwarding protocols (e.g. Ethernet, Internet Protocol).

3.1.2. Control plane

The control plane is the part of the network that carries signaling traffic and is responsible for system configuration, exchange and management of routing table information. The control plane feeds the data plane and, in this way, the data plane functionality is determined by the control plane rules. These rules are generated by specific algorithms. In legacy networks, the signaling traffic is in-band and the control plane refers to the component of a router that focuses on the way that this device interacts with its neighbors via state exchange.

One of the main control plane operations is to combine routing information (generated by a routing protocol) in order to populate FIB (Forwarding Information Base), which is used by the data plane.

Moreover, the control plane functionality can either be centralized or distributed. In case of a centralized control plane, decision-making regarding the entire infrastructure is a centralized process, whereas in case of a distributed control plane, the selected algorithms are distributed to each network device that is responsible for the control plane.

3.1.3. Management plane

The most widely used network management framework is FCAPS [7]. The five areas of function of this framework are described below:

- **Fault management:** its goal is to recognize, isolate, correct and log faults that occur in the network.
- **Configuration management:** its goals are to gather and store configurations from network devices, to simplify the configuration of the device, to track changes that are made to the configuration, to configure or “provision” circuits or paths through non-switched networks and to plan for future expansion and scaling.
- **Accounting management:** its goal is to gather usage statistics for users.
- **Performance management:** it focuses on ensuring that network performance remains at an acceptable level
- **Security management:** it refers to the process of controlling access to assets in the network.

3.2. Software-Defined Networking (SDN)

Software-Defined Networking (SDN) is an emerging architecture that is dynamic, manageable, cost-effective, and adaptable, making it ideal for the high-bandwidth, dynamic nature of today's applications. This architecture enables the control and the data plane decoupling. In this way, the network control plane becomes directly programmable and the underlying infrastructure can be abstracted for various applications and network services. The OpenFlow protocol is the cornerstone of building SDN solutions. SDN also constitutes the enabling technology for network virtualization. The most important features of a SDN architecture are:

- **Direct programmability**: The control plane is directly programmable because it is decoupled from the data plane.
- **Agility**: Abstracting the control plane from the data plane lets administrators dynamically adjust network-wide traffic flow to meet changing needs.
- **Central management**: Network intelligence is (logically) centralized in software-based SDN controllers. Such controllers maintain a global view of the infrastructure network, which appears to applications and policy engines as a single, logical switch.
- **Programmable configuration**: SDN lets network managers configure, manage, secure, and optimize

network resources very quickly via dynamic, automated SDN programs, which they can write themselves, because the programs do not depend on proprietary software.

- **Open standards implementation and vendor neutrality**: When implemented through open standards, SDN simplifies network design and operation because instructions are provided by SDN controllers instead of multiple, vendor-specific devices and protocols.

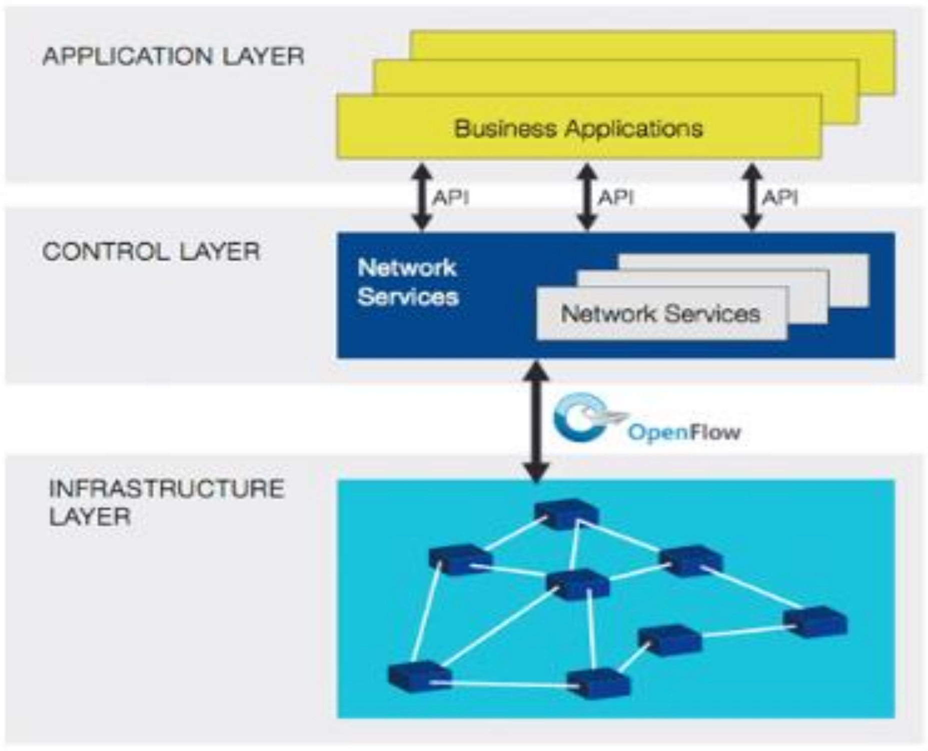


Figure 3.2-1: Decoupling of control and data plane introduced by SDN [Source: Open Networking Foundation]

3.3. Network virtualization

The primitive principles of virtualization have been implemented in many well-known and widely used network protocols. In the past, network virtualization was used for the increase of utilization, establishment of logical separation among different network instances, simplification of network management (e.g. Virtual Private Networks – VLANs) and security over untrusted networks (e.g. Virtual Private Networks – VPNs).

Cloud computing brought network virtualization to prominence because cloud providers needed a way to allow multiple customers (or "tenants") to share a common infrastructure. SDN architecture, based on the OpenFlow protocol, constitutes a solid background for the development of multi-tenant virtualized environments.

There are two major approaches of network virtualization: **(i) full network virtualization** and **(ii) control plane "slicing"**.

3.3.1. Full network virtualization

As it is mentioned in [8], network virtualization presents the abstraction of a network that is decoupled from the underlying physical equipment. Network virtualization allows multiple virtual networks to run over a shared infrastructure and each virtual network can have a much more abstract topology than the underlying physical substrate. Important semantics of the full network virtualization concept are link/node abstraction [6] and path splitting and migration [9].

3.3.2. Control plane slicing

The main idea is to divide traffic flowspace (physical resources) into “slices” (a concept initially introduced in PlanetLab [10]), where each slice has a part of network resources and is managed by a different SDN controller. An intermediate controller can act as a transparent proxy controller, speaking OpenFlow to each SDN controller and OpenFlow switch.

The behavior of such a proxy controller is specified by establishing flowspace rules. In this context, each network slice is associated with a certain number of flowspace rules, which specify the way that the physical resources of a particular slice are utilized.

The (up to now) de-facto software-based OpenFlow proxy controller is FlowVisor. Other undergoing promising efforts are OVX (OpenVirtex) [11] and Flowspace Firewall [12].

3.4. Algorithmic basis

Algorithms for the creation of various data structures (e.g. hash tables and multi-dimensional binary trees) and the lookup process in them were studied and implemented. Moreover, disjoint path finding was studied and an algorithm for routing between a given pair of nodes over two physically disjoint paths was implemented.

3.4.1. Search algorithms

In computer science, a search algorithm is an algorithm for finding an item with specified properties among a collection of items.

The items may be stored individually in a database or a hash table or may be elements of a search space defined by a mathematical formula or procedure.

An important step in evaluating the efficiency of an algorithm is algorithmic asymptotic analysis. This gives us a solid view of the algorithmic behavior at large inputs and forms a good basis for the comparison of various algorithms. The goal of asymptotic analysis is to categorize algorithms in large complexity classes (using the "Big O" notation) without focusing on "constants" that differentiate execution behavior to a quite smaller extent.

In the context of this thesis, the following search algorithms were implemented:

- Linear Search
- Single hashing
- Open addressing with double hashing
- Multi-dimensional binary search tree

The **linear search** algorithm [13] is a method for finding a particular element in a data structure and consists of serially checking every one of its elements. It has a worst-case time complexity of O(N), where N is the amount of elements that have to be accessed. Despite its simplicity and its good storage requirements, this algorithm results in slow lookup rates, especially in cases that the accessed structure has many elements and the requested element is at its end.

The **single hashing** algorithm, described in [13], searches for a given key K in a table of existing keys (hash tables). This algorithm make use of a hash function h(K) (e.g. MD5, SHA-1 and CRC32) in order to map the requested data of arbitrary length into data of a fixed length. Its average time complexity is O(1) and its worst-case time complexity is O(N). Generally speaking, the lookup process is quite fast (more apparent when the number of entries is thousands or even more). Moreover, in a well-dimensioned hash table, the average cost (number of instructions) for each lookup is independent of the number of elements stored in the table. On the other hand, if the hash table uses dynamic resizing, an insertion or a deletion operation may occasionally take time proportional to the number of entries and this may be a serious drawback in real-time or interactive applications. Hash tables also require the design of an effective hash function for each key type, which in many cases is quite difficult and time-consuming to design and debug.

The **open addressing with double hashing** algorithm, described in [13], probes the table in a slightly different fashion by making use of two hash functions $h_1(K)$ and $h_2(K)$. $h_1(K)$ produces a value between 0 and M-1 , inclusive (M is the table size) . However, $h_2(K)$ must produce a value between 1 and M-1 that is relatively prime to M. The steps of this algorithm are described below:

Step 1 [First hash] Set $i \leftarrow h_1(K)$.

Step 2 [First probe] If TABLE[i] is empty, go to **Step 6**. Otherwise if KEY[i] = K, the algorithm terminates successfully.

Step 3 [Second hash] Set $c \leftarrow h_2(K)$.

Step 4 [Advance to next] Set $i \leftarrow i - c$; if now $i<0$, set $i \leftarrow i + M$

Step 5 [Compare] If TABLE[i] is empty, go to **Step 6.** Otherwise if KEY[i] = K, the algorithm terminates successfully. Otherwise go back to **Step 4.**

Step 6 [Insert] If N = M – 1, the algorithm terminates with overflow. Otherwise set N ← N + 1, mark TABLE[i] occupied and set KEY[i] ← K.

The average time complexity of this algorithm is O(1), while the worst-case time complexity is O(N). Open addressing resolves the problem of hash collisions (that is to say the problem of different key values that are assigned by the hash function to the same bucket). Moreover, by applying the second hash function to produce values relatively prime to the maximum value produced by both the hash functions, the appearance of consecutive key values is now actually a help instead of a hindrance. Furthermore, the two hash functions are independent, in the sense that different keys would have the same value for both the hash functions with probability O(1/M^2) instead of O(1/M), where M-1 is the maximum value produced by the hash functions. On the other hand, the lookup becomes somewhat slower and the memory needed is increased compared to the case of single hashing.

The **k-dimensional binary search tree** algorithm (or **k-d tree** algorithm, where k is the dimensionality of the search space) is described in [14]. In general terms, if a file is represented as a k-d tree, then each record in the file is stored as a node in the tree. In addition to the k keys, which comprise the record, each node contains two pointers, which are either null or point to another node in the tree. Each pointer can be considered as specifying a subtree. Based on this data structure, various utility algorithms are developed, such as insertion, deletion of the root, deletion of a random node and optimization (guarantees logarithmic performance of searches).

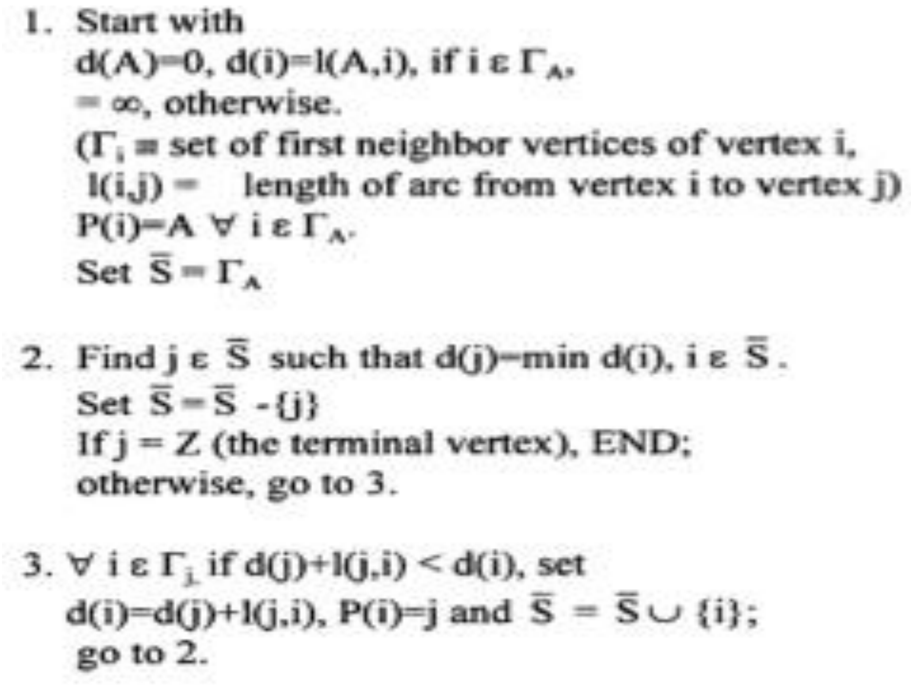
As a consequence of the aforementioned optimization, the average search time complexity is O(logN), while the worst-case search time complexity is O(N). A great advancement of this algorithm is that a single data structure facilitates many different and seemingly unrelated query types. Moreover, this algorithm is efficient for large trees (which consist of more than 8,000-9,000 nodes) and flexible enough to allow intersection queries. On the other hand, it is less efficient than Linear Search for small or medium sized trees (up to 6,000-7,000 nodes approximately). In cases that the requested element is not a part of the tree, the lookup task takes too much time to be terminated.

3.4.2. Disjoint paths algorithm

An optimal algorithm for k-disjoint path finding (k greater or equal to 2) in a graph of vertices (nodes) and edges (links) are presented in [15]. The used algorithm is a slight variant of the original Dijkstra algorithm [16]. It is different (in **step 3** below) in that it scans all the neighbors of the node selected in **step 2.** Let d(i) denote the distance of node i from starting node A. Let P(i) denote the predecessor of node i. The ending node is Z.

In each iteration, a node with the least path length is selected from the set $\bar{S}$. The search process includes one move at a time and terminates when the node selected from the set $\bar{S}$ is Z. In the original Dijsktra algorithm, when a node with the least path length is selected from the list of tentatively labeled nodes, the selected node is said to have been labeled “permanently” and no further scanning from any other node in the graph can update the label of this node. However, in the algorithm described in Figure 3.4.2-1, because of the presence of negative arcs in the modified graph, rescanning can update the label of the previously selected (or “permanently” labeled) node. That is why the algorithm given in Figure 3.4.2-1 permits rescanning.

However, achieving vertex-disjointness and edge-disjointness for the generated shortest pair of paths is not a trivial process. These algorithms are analyzed in sections 3.1 and 3.2 of [15] and require a number of runs of the shortest path algorithm described in Figure 3.4.2-1.



```
1. Start with
   $d(A)=0$, $d(i)=l(A,i)$, if $i \in \Gamma_A$,
   $= \infty$, otherwise.
   ($\Gamma_i$ = set of first neighbor vertices of vertex i,
    $l(i,j)$ =    length of arc from vertex i to vertex j)
   $P(i)=A \; \forall \; i \in \Gamma_A$.
   Set $\bar{S} = \Gamma_A$

2. Find $j \in \bar{S}$ such that $d(j)=\min d(i)$, $i \in \bar{S}$.
   Set $\bar{S} = \bar{S} - \{j\}$
   If $j = Z$ (the terminal vertex), END;
   otherwise, go to 3.

3. $\forall \; i \in \Gamma_j$ if $d(j)+l(j,i) < d(i)$, set
   $d(i)=d(j)+l(j,i)$, $P(i)=j$ and $\bar{S} = \bar{S} \cup \{i\}$;
   go to 2.
```

Figure 3.4.2-1: Modified Dijkstra Algorithm for Shortest Path from node A to Z [Source: Optimal physical diversity algorithms and survivable networks**]**

3.5. OpenFlow protocol

OpenFlow (OF) is a communication protocol that enables the network control plane to define cross-layer forwarding rules, which can be established and handled by OpenFlow-enabled devices. Based on the SDN architecture, together with the OF protocol, network devices are transformed to fully programmable forwarding elements. OpenFlow Switch Specification (its latest version is described in [17]) provides a standardized and secure interface (secure channel) between a centralized control plane entity (OpenFlow controller) and distributed data plane entities (OpenFlow-enabled switches).

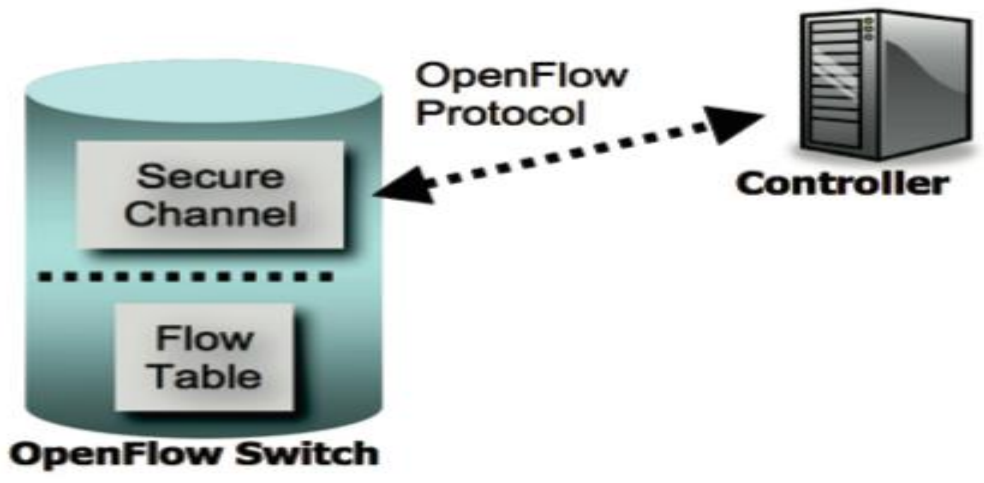


Figure 3.5-1: An OpenFlow-enabled switch communicates with an OpenFlow controller over a secure channel using the OpenFlow protocol [Source: OpenFlow Switch Specification Version 1.4]

3.5.1. Flow Table

A flow table consists of flow entries. Each flow entry (Table 3.5.1-1) contains the following fields:

- **Match fields**: to match against OpenFlow packets. These fields consist of the ingress port and the packet headers and, optionally, some metadata specified by a previous flow table.

- **Priority**: matching precedence of the flow entry. Higher values are higher priorities.
- **Counters**: increased by one when a packet is matched.
- **Instructions**: modification of the action set or pipeline processing.
- **Timeouts**: maximum timespan or idle time before a flow is expired by the switch.
- **Cookie**: opaque data value handled and selected by the controller. May be used by the controller to filter flow statistics, flow modification and deletion.

The match fields and priority taken together, uniquely identify each flow table entry in a flow table.

In Table 3.5.1-2, the required match fields are presented. These

Match	Priority	Counters	Instructions	Timeouts	Cookie

Table 3.5.1-1: Major components of a flow entry in a flow table

fields are matched against the corresponding fields of each OpenFlow packet that arrives at an OpenFlow-enabled switch. Each flow entry

Ingress Port	Ether src	Ether dst	Ether type	VLAN id	VLAN Priority	IP src	IP dst	IP proto	IP ToS Bits	TCP/UDP Src Port	TCP/UDP Dst Port

Table 3.5.1-2: Required match fields of a flow table entry

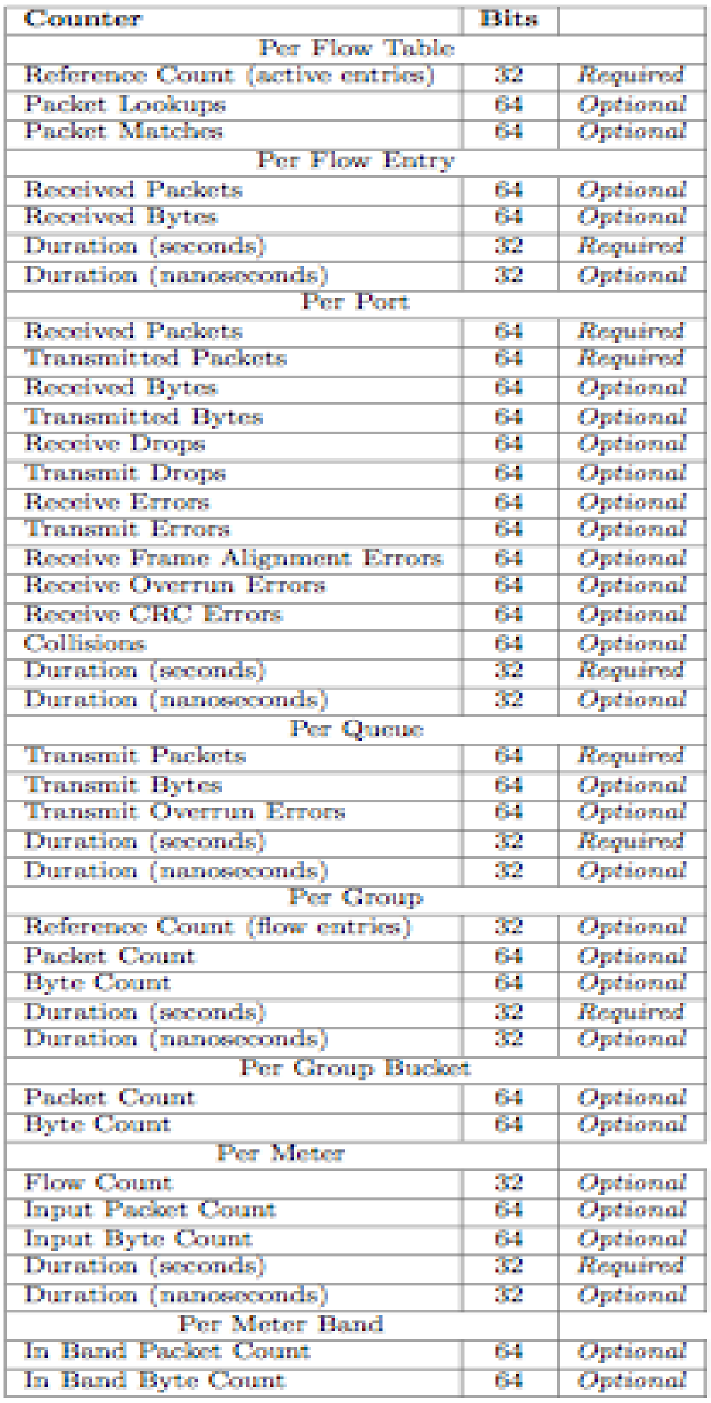
may contain one or more wildcarded fields. In this case, a wildcarded field matches against all the possible values of that field.

Counters are supported by each OpenFlow-enabled switch and are maintained for each flow table, flow entry, switch port, queue, group and group bucket, meter and meter band. OpenFlow-compliant counters can be implemented in software and maintained by polling hardware counters. The set of counters defined by the OpenFlow specification is presented in Table 3.5.1-3. It should be noted that an OpenFlow-enabled switch is not required to support all counters, but just those marked "Required" in the mentioned table.

Each flow entry contains a set of instructions that are executed when a packet matches the entry. Such instructions result in action set, changes to the incoming packet and/or pipeline processing. An OpenFlow-enabled switch is not required to support all possible instruction types, just those marked as "Required Instruction" below. Theses instructions are considered as absolutely necessary. It should be noted that a switch must reject a flow entry, if it is unable to execute the instructions associated with this flow entry.

- *Optional Instruction*: **Meter meter_id**: Directs packet to the specified meter.
- *Optional Instruction*: **Apply-Actions action(s)**: This instruction may be used for the modification of the packet between two tables or for the execution of multiple actions of the same type. It applies the specific action(s) immediately to the packet, without changing the Action Set. Such actions are described as an action list.
- *Optional Instruction*: **Clear-Actions**: Clears all the actions in the action set immediately.

- *Required Instruction*: **Write-Actions action(s)**: Merges the specified action(s) into the current action set.
- *Optional Instruction*: **Write-Metadata metadata / mask**: Writes the masked metadata value into the metadata field.
- *Required Instruction*: **Goto-Table next-table-id:** Indicates the next table in the processing pipeline. The next table-id must be greater than the current table-id.



Counter	Bits	
Per Flow Table		
Reference Count (active entries)	32	*Required*
Packet Lookups	64	*Optional*
Packet Matches	64	*Optional*
Per Flow Entry		
Received Packets	64	*Optional*
Received Bytes	64	*Optional*
Duration (seconds)	32	*Required*
Duration (nanoseconds)	32	*Optional*
Per Port		
Received Packets	64	*Required*
Transmitted Packets	64	*Required*
Received Bytes	64	*Optional*
Transmitted Bytes	64	*Optional*
Receive Drops	64	*Optional*
Transmit Drops	64	*Optional*
Receive Errors	64	*Optional*
Transmit Errors	64	*Optional*
Receive Frame Alignment Errors	64	*Optional*
Receive Overrun Errors	64	*Optional*
Receive CRC Errors	64	*Optional*
Collisions	64	*Optional*
Duration (seconds)	32	*Required*
Duration (nanoseconds)	32	*Optional*
Per Queue		
Transmit Packets	64	*Required*
Transmit Bytes	64	*Optional*
Transmit Overrun Errors	64	*Optional*
Duration (seconds)	32	*Required*
Duration (nanoseconds)	32	*Optional*
Per Group		
Reference Count (flow entries)	32	*Optional*
Packet Count	64	*Optional*
Byte Count	64	*Optional*
Duration (seconds)	32	*Required*
Duration (nanoseconds)	32	*Optional*
Per Group Bucket		
Packet Count	64	*Optional*
Byte Count	64	*Optional*
Per Meter		
Flow Count	32	*Optional*
Input Packet Count	64	*Optional*
Input Byte Count	64	*Optional*
Duration (seconds)	32	*Required*
Duration (nanoseconds)	32	*Optional*
Per Meter Band		
In Band Packet Count	64	*Optional*
In Band Byte Count	64	*Optional*

Table 3.5.1-3: List of the available OpenFlow-compliant counters [Source: OpenFlow Switch Specification version 1.4.0]

An action set is associated with each packet. By default, this set is empty. An action set can be modified by a flow entry using a Write-Action or a Clear-Action instruction associated with a specific match. Each action set is carried among flow tables. The actions in the action set of the packet are executed and the pipeline processing stops when a Goto-Table instruction is not included in the instruction set of a flow entry.

An action set contains a maximum of one action of each type. Regardless of the order that they were added to the set, the actions in an action set are applied in the order specified below. However, an OpenFlow-enabled switch may support arbitrary action execution order through the action list of the Apply-Actions instruction:

1. **Copy TTL inwards:** apply copy TTL inward actions to the packet.
2. **Pop:** apply all tag pop actions to the packet.
3. **Push-MPLS:** apply MPLS tag push action to the packet.
4. **Push-PBB:** apply PBB tag push action to the packet.
5. **Push-VLAN:** apply VLAN tag push action to the packet.
6. **Copy TTL outwards:** apply copy TTL outwards action to the packet.
7. **Decrement TTL:** apply decrement TTL action to the packet.
8. **Set:** apply all set-field actions to the packet.
9. **QoS:** apply all QoS actions to the packet.
10. **Group:** if a group action is specified, apply the actions of the relevant group bucket(s) in the order specified by this list.
11. **Output:** forward the packet on the port specified by the output action unless a group action is specified.

The output action in the action set is executed last. An output action is ignored only in the case that both an output action and a group action are specified in an action set because the group action takes precedence. The packet is dropped unless an output or a group action (or both) was specified in an action set.

The Apply-Actions instruction includes an action list. The actions of an action list are executed in the order specified by the list and are applied immediately to the packet. Each action is executed on the packet in sequence and that execution starts with the first action in the list.

However, a switch is not required to support all action types, but just those marked as “Required Action” below. Moreover, the controller can query the switch about which of the “Optional Actions” it supports.

- *Required Action*: **Output**. According to this action, a packet is forwarded to a specified OpenFlow port. OpenFlow switches must support forwarding to physical ports, switch-defined logical ports and the required reserved ports.
- *Optional Action*: **Set-Queue.** It sets the queue id for an incoming packet. When the packet is forwarded to a port using the output action, the queue id specifies which queue, attached to this port, is used for scheduling and forwarding the packet. More specifically, the forwarding behavior is determined by the configuration of the queue and is used for the basic QoS support.
- *Required Action:* **Drop.** This result can come from empty instruction sets or empty action buckets in the processing pipeline, or after the execution of a Clear-Actions instruction. In

other words, there is no explicit action to represent drop, but packets whose action sets have no output actions should be dropped.

- *Required Action:* **Group.** Process the packet through the specified group.
- *Optional Action:* **Push-Tag/Pop-Tag.** Switches may support the ability to push and pop the tags shown in Table 3.5.1-4. For instance, the ability to push/pop VLAN tags is suggested to be supported.
- *Optional Action:* **Set-Field.** The Set-Field actions modify the values of respective header fields in the packet. Such actions are identified by their field type.
- *Optional Action:* **Change-TTL.** Such actions result in the modification of the values of the IPv4 TTL, IPv6 Hop Limit or MPLS TTL in the packet.

Action	**Associated Data**	**Description**
Push VLAN header	Ethertype	Push a new VLAN header onto the packet. The Ethertype is used as the Ethertype for the tag. Only Ethertype 0x8100 and 0x88a8 should be used.
Pop VLAN header	-	Pop the outer-most VLAN header from the packet.

Push MPLS header	Ethertype	Push a new MPLS shim header onto the packet. The Ethertype is used as the Ethertype for the tag. Only Ethertype 0x8847 and 0x8848 should be used.
Pop MPLS header	Ethertype	Pop the outer-most MPLS tag or shim header from the packet. The Ethertype is used as the Ethertype for the resulting packet (Ethertype for the MPLS payload).
Push PBB header	Ethertype	Push a new PBB service instance header (I-TAG TCI) onto the packet. The Ethertype is used as the Ethertype for the tag. Only Ethertype 0x88E7 should be used.
Pop PBB header	-	Pop the outer-most PBB service instance header (I-TAG TCI) from the packet.

Table 3.5.1-4: Push/pop tag actions [Source: OpenFlow Switch Specification version 1.4.0]

When executing a push action, values for all the fields listed in Table 3.5.1-5 should be copied from existing outer headers to new outer headers. “New Fields”, specified in Table 3.5.1-5, without corresponding “Existing Field(s)”, should be set to zero.

New Fields		Existing Field(s)
VLAN ID	$\leftarrow$	VLAN ID
VLAN priority	$\leftarrow$	VLAN priority
MPLS label	$\leftarrow$	MPLS label
MPLS traffic class	$\leftarrow$	MPLS traffic class
MPLS TTL	$\leftarrow$	$\left\{\begin{array}{l}\text{MPLS TTL}\\\text{IP TTL}\end{array}\right.$
PBB I-SID	$\leftarrow$	PBB I-SID
PBB I-PCP	$\leftarrow$	VLAN PCP
PBB C-DA	$\leftarrow$	ETH DST
PBB C-SA	$\leftarrow$	ETH SRC

Table 3.5.1-5: Existing fields that can be copied into new fields on a push action [Source: OpenFlow Switch Specification version 1.4.0]

3.5.2. Matching a packet with the corresponding flow entry

On arrival of a packet, an OpenFlow-enabled switch starts by performing a table lookup in the first flow table, and according to the pipeline processing, may perform table lookups in other flow tables as well.

First of all, packet match fields are extracted from the OpenFlow packet. Packet match fields used for table lookups typically include various Layer 2 to Layer 4 header fields and usually depend on the packet type. Apart from the header fields, matches can be performed against ingress switch port and metadata fields.

A packet matches a flow table entry, if the values in the packet match fields, used for the lookup, match those specified in the flow entry. As it was mentioned in the previous section, if a flow table entry field is wildcarded, it matches all possible values in the packet header. Each packet is matched against the table and only the highest priority entry that matches the packet must be selected. The counters associated with this particular flow entry must be increased and the instruction set included in the selected flow entry must be applied. In case of multiple matching flow entries with the same highest priority, the chosen flow entry is undefined.

All the aforementioned functions, which are performed by an OpenFlow switch, are shown in the following figure:

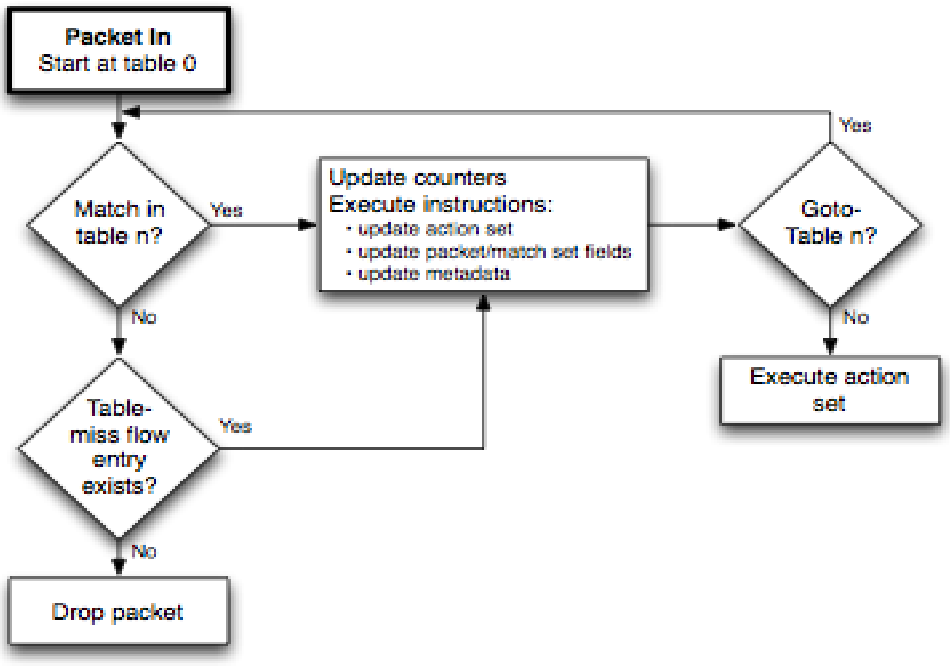


Figure 3.5.2-1: Flowchart illustrating packet flow through an OpenFlow switch [Source: OpenFlow Switch Specification version 1.4.0]

Moreover, every flow table must support a table-miss flow entry to process table misses. This flow entry defines how to handle packets that are not matched against other flow entries in the flow table. As a result, such packets may be sent to the controller, be dropped or be directed to a subsequent table. If such a table-miss entry does not exist, by default, packets unmatched by flow entries are discarded.

Flow entries are removed from flow tables in three ways:

- at a request of a controller
- via the switch flow expiry mechanism
- via the optional switch eviction mechanism

The controller may actively dictate the deletion of a flow entry from a flow table by sending **delete** flow table modification messages.

The switch **flow expiry** mechanism is run by the switch independently of the controller and is based on the state and the configuration of flow entries. Every flow entry has an *idle timeout* and a *hard timeout* indicator associated with it. A non-zero *hard timeout* field causes a flow entry to be deleted after the given number of seconds, regardless of the number of packets that it has matched. A non-zero *idle timeout* field causes the flow entry to be removed when it has matched no packets in the given amount of seconds. A switch must implement both the aforementioned features.

Flow entries may be **evicted** from flow tables when the switch needs to reclaim resources. That is an optional feature, and the mechanism used to select which flow entries to evict is switch defined and may depend on flow entry parameters, resource mappings in the switch and other internal switch constraints.

4. Design principles of the proposed implementation

Generally speaking, there are two popular multi-tenant SDN architectures. The first one deploys an OpenFlow transparent proxy controller, such as FlowVisor, that enables tenants to share or "slice" the control plane and develop their own arbitrary forwarding logic within their slice [4]. The second one assumes a network hypervisor that supports various network abstractions towards network virtualization [6].

In order to achieve multi-tenancy across any SDN environment (as well as in both of the SDN architectures mentioned above), the classification of packets into flows is required. A typical way to achieve this classification is via logical separators within packet headers. In this context, three "**network control plane slicing methods**" are proposed and analyzed in section 4.1. A network control plane slicing method is an algorithm that ensures the creation of non-overlapping flowspace rules. Each slicing method takes into account different fields of the packet header.

Regardless of the selected slicing method, isolation among slices (tenants) should be enforced across a shared physical infrastructure. To that end, in section 4.2, the implications of isolation policy enforcement for each slicing method are presented.

In order to enforce isolation policy among tenants, a certain number of non-overlapping flowspace rules should be generated. In some cases, this number may be quite large resulting in extreme overheads. In order to keep flowspace rules to a reasonable number and, thus, avoid such overheads, a flowspace rule reduction approach is described in section 4.3.

4.1. Network control plane slicing methods

As described above, packet classification into flows via logical separators within a packet header (packet ID) is required to achieve multi-tenancy. Such a packet ID can also be considered by an OpenFlow controller as the identifier of a network slice. For instance, the VLAN IDs or the MPLS tags can be considered as packet IDs.

However, using a single separator as a slice identifier (e.g. the VLAN IDs are restricted to 4096 per domain) could result in limited scalability. In order to overcome these limitations, multiple separators could be considered within an SDN controller and, thus, a network slice can be identified via a set of tuples.

The proposed slicing methods are the following: **i) domain-wide slicing method, ii) switch-wide slicing method and iii) port-wide slicing method.** Examples of a separator tuple for each slicing method are presented in Table 4.1.3-1.

4.1.1. Domain-wide slicing method

In the domain-wide slicing method, each network slice is strictly associated (identified) with a unique value of the packet ID. This is achieved by using a single separator per domain, e.g. <MPLS tag> or <VLAN ID>. This slicing method could be referred to as the "naïve" way to classify packets and "slice" flowspace.

4.1.2. Switch-wide slicing method

In the switch-wide slicing method, each slice is identified (associated) via multiple separators, which form tuples. In addition to a single or multiple packet header fields, these tuples also include the identification of the switching elements that this slice spans. Such tuples can be specified as follows: <MPLS tag, switch ID> or <VLAN ID, switch ID>. This method is more sophisticated than the "naïve" domain-wide slicing and, thus, it is expected to result in a more efficient flowspace segmentation.

4.1.3. Port-wide slicing method

In the port-wide slicing method, each slice is identified (associated) via multiple separators along with specific switch ports and switch identification. In other words, a slice is further identified (compared to the switch-wide slicing method) using specific switch ports.

For instance, a tuple regarding this particular slicing method could be defined as follows: <MPLS tag, switch ID, switch port ID>. One could think of this method as the most complex one, which, however, provides the greatest network programmability to the administrator or the infrastructure provider.

Slicing method	Separator tuple
Domain-wide	<MPLS tag>
Switch-wide	<MPLS tag, switch ID>
Port-wide	<MPLS tag, switch ID, switch port ID>

Table 4.1.3-1: Example of separator tuple for each slicing method

4.2. Isolation policy enforcement in multi-tenant virtualized environments

As stated above, regardless of the selected slicing method and the underlying physical network topology, strong isolation should be enforced among network slices. That is to say, actions of one slice should be prevented from affecting other slices allowing tenants to safely coexist across a common physical network infrastructure.

The overall concept of network virtualization may break down if one slice conflicts with others and exhausts their resources. In order to enforce such strong isolation among network slices, non-overlapping flowspace rules should be created.

Enforcing isolation in the domain-wide slicing method is a trivial process. Tenants should use each instance of the selected separator only once across a network domain. For example, each reserved MPLS tag or VLAN ID should not be reused across the same domain.

However, isolation enforcement in the switch-wide slicing method is more complex because a specific instance of a selected separator can be reused across a network domain. As a consequence, multiple network slices may select the same separator instance across a common physical network substrate risking the isolation of network slices. At the worst-case scenario, the control plane could be poisoned by the data plane traffic harshly violating the isolation among network slices and thus exhausting their network resources.

In Figure 4.2-1, a scenario of two, potential conflicting, network slices is illustrated. Slices of *Tenant K and L* are allocated within two separate *switching elements A and B*. These switches are interconnected via port 3 and port 1 respectively. Moreover, within these switches, the same separator instance of *MPLS tag i* has been reserved by each tenant. If tenant L selects *port 1 of switch B* as the egress port of its traffic, packets will be forwarded to *switch A.* However, in *switch A*, *MPLS tag i* has been delegated to tenant K and, thus, tenant's K OpenFlow controller would be overloaded by alien OpenFlow control messages.

Such an outcome can be avoided by not assigning *port 3 of switch A* and *port 1 of switch B* for the specific separator instance to any of the tenants. In order to fulfill this requirement, the required non-overlapping flowspace rules regarding *tenants K and L and switches A and B* are presented in Table 4.2-1. It is worth noting that one rule per delegated switch port should be defined.

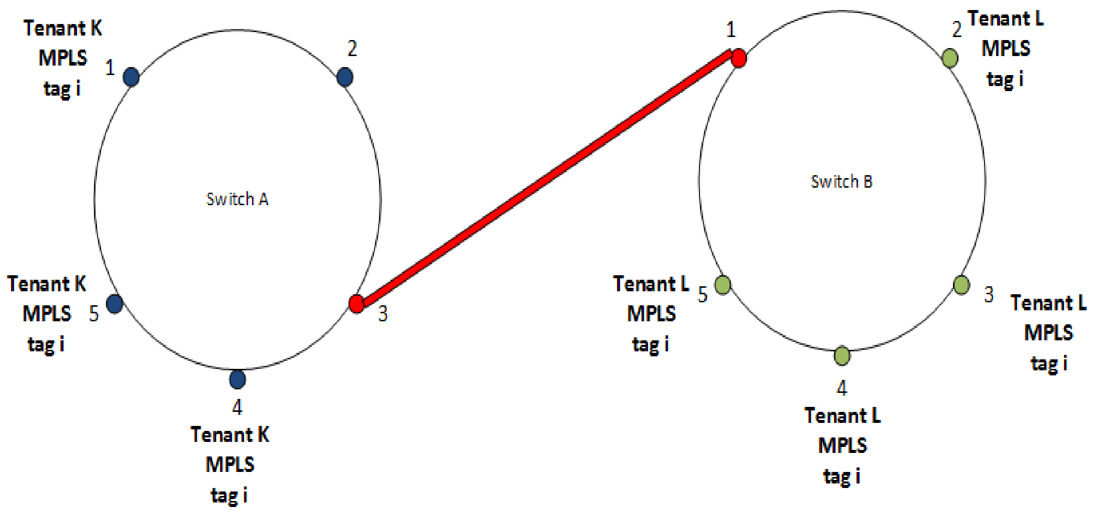


Figure 4.2-1: Scenario of two potential conflicting network slices in the switch-wide slicing method

Rule identification	Tenant identification	Rule priority	Egress/Ingress switch port	Datapath identification	Separator instance
Rule 1	tenant L	priority=1	port=2	datapath=switch B	MPLS tag=i
Rule 2	tenant L	priority=1	port=3	datapath=switch B	MPLS tag=i
Rule 3	tenant L	priority=1	port=4	datapath=switch B	MPLS tag=i
Rule 4	tenant L	priority=1	port=5	datapath=switch B	MPLS tag=i
Rule 5	tenant K	priority=1	port=1	datapath=switch A	MPLS tag=i
Rule 6	tenant K	priority=1	port=2	datapath=switch A	MPLS tag=i
Rule 7	tenant K	priority=1	port=4	datapath=switch A	MPLS tag=i
Rule 8	tenant K	priority=1	port=5	datapath=switch A	MPLS tag=i

Table 4.2-1: Required flowspace rules in case of isolation policy enforcement in switch-wide slicing

In the port-wide slicing method, non-overlapping flowspace rules should be created in the same manner as above. Figure 4.2-2 illustrates a potential scenario of violating tenant isolation in port-wide slicing. *Ports 1 and 5 of switch A* have been assigned to *tenant L*, while *ports 3 and 4 of the same switching element* have been assigned to *tenant K* and the *switch port 2* has not been assigned to any of the tenants.

Both tenants K and L use the *instance i* of *separator MPLS tag*. However, if *tenant L* selects *port 3 or 4* as the egress port of its traffic, the isolation policy will be violated and the tenant slices will conflict. In order to avoid this violation, the required flowspace rules for both tenants are presented in Table 4.2-2. It is deduced that one rule per delegated switch port should be defined in this slicing method as well. Finally, it should be noted that in any of the aforementioned scenarios, if a packet does not match any flowspace rule, it will be discarded.

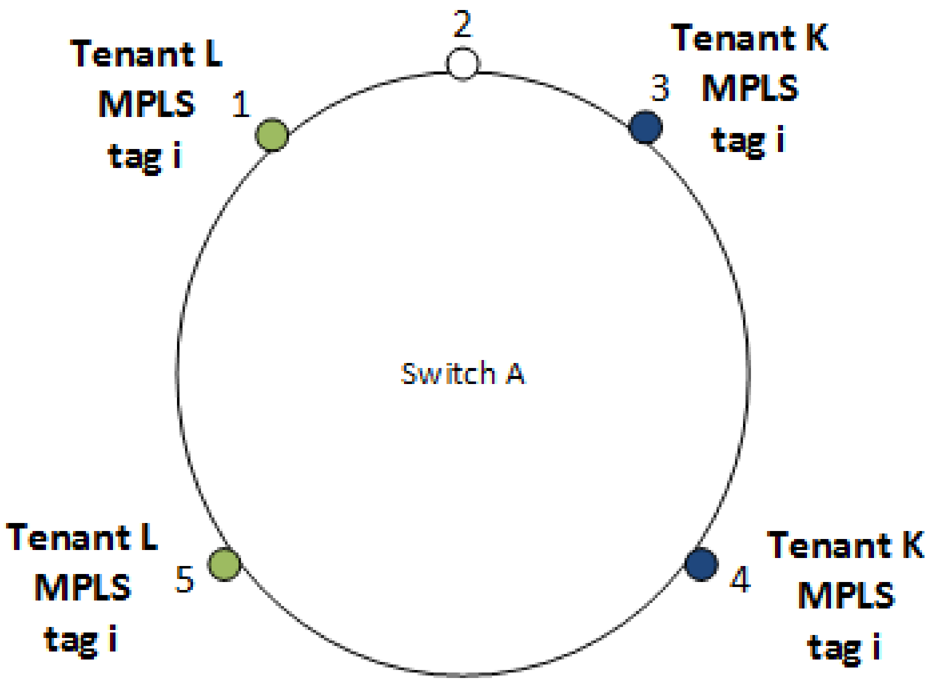


Figure 4.2-2: Scenario of two potential conflicting network slices in the port-wide slicing method

Rule identification	**Tenant identification**	**Rule priority**	**Egress/Ingress switch port**	**Datapath identification**	**Separator instance**
Rule 1	tenant L	priority=1	port=1	datapath=switch A	MPLS tag=i
Rule 2	tenant L	priority=1	port=5	datapath=switch A	MPLS tag=i
Rule 3	tenant K	priority=1	port=3	datapath=switch A	MPLS tag=i
Rule 4	tenant K	priority=1	port=4	datapath=switch A	MPLS tag=i

Table 4.2-2: Required flowspace rules in case of isolation policy enforcement in port-wide slicing

4.3. Flowspace rule reduction approach in multi-tenant virtualized environments

As mentioned in the previous section, the number of required flowspace rules in switch-wide and port-wide slicing is equal to the number of switch ports that have been assigned to tenants, so that isolation among network slices (tenants) is safeguarded. Moreover, the ports that interconnect the switching elements of a network topology should not be delegated to any of the tenants unless both of the interconnected switches are delegated to the same tenant.

However, this isolation policy could result in large numbers of generated flowspace rules and, as a consequence, its enforcement could cause severe performance overheads and large memory consumption, thus becoming the bottleneck of the entire network infrastructure. In this way, it is overt that an approach towards the reduction of the required flowspace rule number should be made. This approach is applicable to the switch-wide slicing method.

When the number *a of ports assigned to tenants per topology switch* is greater than the number *u of unassigned switch ports,* the overall number of flowspace policy rules can be reduced by defining high (higher than normal) priority drop rules for the interconnection ports of the topology switches. These rules would be handled by a special-purpose (administrative) OpenFlow controller that has a global view of the network topology. In this way, if the data traffic of a specific tenant was forwarded to a topology switch delegated to another tenant, the corresponding packets would be dropped (discarded). In addition to the aforementioned packet-dropping rules, low (lower than normal) priority wildcard entries should be used for the typical process of flowspace delegation to the existing tenants.

Keeping up with the scenario presented in the previous section, the required flowspace policy rules, after applying the rule reduction approach, are presented in Table 4.3-1 (denoting that Priority 1 is greater than Priority 2). It is worth noting that the required rules are now reduced to half compared to the required rules defined in Table 4.2-1.

Flowspace Implementation				
Slice id	**Priority**	**Port**	**Datapath id**	**MPLS tag**
Reserved slice 1	Priority 1	Port=3	Switch A	MPLS tag i
Tenant K slice	Priority 2	Port=*	Switch A	MPLS tag i
Reserved slice 1	Priority 1	Port=1	Switch B	MPLS tag i
Tenant L slice	Priority 2	Port=*	Switch B	MPLS tag i

Table 4.3-1: Rule reduction approach in switch-wide slicing when a>u

On the other hand, when the number *a of ports assigned to tenants per topology switch* is smaller than the number *u of unassigned switch ports*, a flowspace rule reduction cannot be achieved because the number of required flowspace policy rules for each topology switch is equal to the number of reserved switch ports. Thus, the number of required flowspace rules is equal to the number of policy rules mentioned in Table 4.2-1.

5. Implementation analysis

In the previous chapter (sections 4.1, 4.2, 4.3), the design principles of the proposed implementation were discussed. In this chapter, implementation specific features are described in detail.

5.1. Implementation structure

In order to evaluate the efficiency and the feasibility of the aforementioned slicing methods and flowspace isolation policy, a flowspace rule engine was implemented in software using Python [18].

This engine takes as inputs tenant requests for virtual network topologies (simple, disjoint and star topologies - all of them are defined below) and physical substrate topologies and, based on the selected slicing method, it generates the required flowspace rules, so that isolation is enforced among tenants. It is noteworthy that this engine takes as input real (WAN, medium and small sized) physical network topologies. As a result, the selected slicing methods are applied to real network substrate topologies providing a fertile environment for reliable evaluation. The structure of this implementation is illustrated in Figure 5.1-1. Specifically, it consists of the following scripts:

- **run_engine.sh:** This bash script is used to initialize and start the engine by specifying all the necessary parameters of its execution. These parameters include the desired physical network topologies, the selected weight type (propagation delays or link bandwidths), the interconnection points between the selected network domains (reserved for future development) and the number of virtual network topologies of each type.

- **generate.py:** This script generates the selected real network topologies (specified in run_engine.sh) as topology graphs using the dataset of "The Internet Topology Zoo" project [19]. Moreover, it computes the selected type of weights for each graph link and attaches these weights to the topology graph.
- **metrics.py:** This script initializes the generation of virtual network topologies and the computation of evaluation metrics.
- **paths.py:** Given a source and a destination node of a network topology graph, this script searches for a disjoint path set between these nodes.
- **evaluation.py:** It calculates various performance indicators and generates the required flowspace rules for the domain-wide, the switch-wide and the port-wide slicing method. Simple paths, star topologies and sets of disjoint paths along with the network topology graph are needed for this computation.
- **graph_util.py:** This script contains functions for the interconnection of two or more network graphs. The interconnection points are specified in run_engine.sh. This script is useful for the future development of the implemented engine, so that multi-domain support is added.

- **lookup_process.py:** This script takes as input the lookup query and initializes the flowspace lookup process by creating various data structures.
- **lookup.py:** It contains the linear search, the single hashing and the open addressing with double hashing lookup algorithms. The elapsed time of the lookup process for each algorithm is also computed. The average and the worst-case time complexity for each algorithm were described in chapter 3.
- **kdtree.py:** It contains functions for the construction of a k-dimensional binary search tree (kd tree) and the implementation of the lookup process in it. If the desired element belongs to the kd tree, then the corresponding node is returned. If there is not such a node in the kd tree, the nearest neighbor of the requested element is returned. The elapsed time is also computed.
- **util_lookup.py:** It includes various helpful functions for the lookup process implementation.

The computed evaluation metrics are defined as follows:

- **Acceptance ratio:** the fraction of total requests issued by tenants that were accepted by each slicing method. This indicator shows the efficiency of a proposed slicing method.
- **Number of required flowspace rules:** the flowspace rules that are generated by the rule engine and are required to be established within an OpenFlow transparent proxy controller (such as FlowVisor) in order for isolation policy to be enforced among network slices (tenants)

- **Proxy controller time overhead:** the time overhead added to the networked system by the OpenFlow proxy controller when it handles the generated flowspace rules
- **Proxy controller memory consumption:** The memory needed by the OpenFlow proxy controller to create, manipulate and update the generated flowspace.

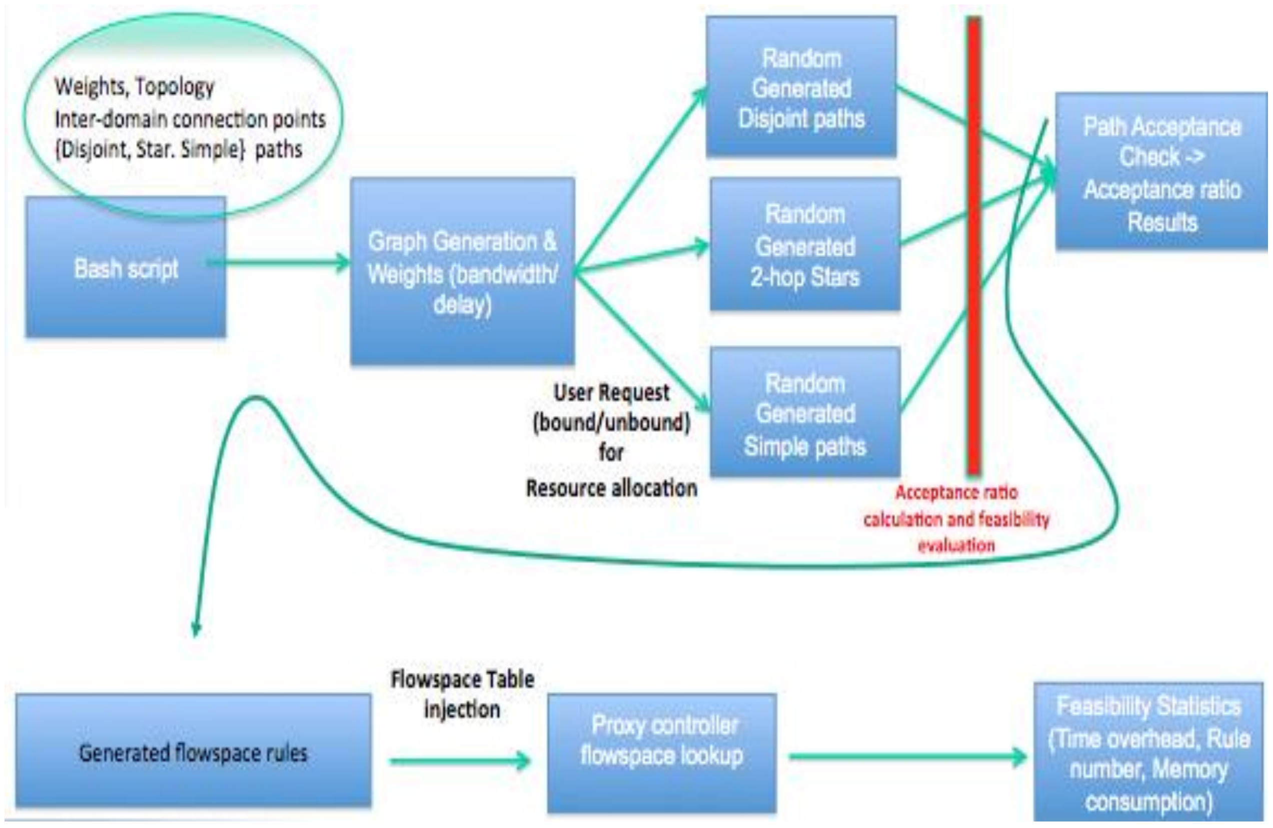

Weights, Topology
Inter-domain connection points
{Disjoint, Star, Simple} paths
Bash script
Graph Generation & Weights (bandwidth/ delay)
Random Generated Disjoint paths
Random Generated 2-hop Stars
Random Generated Simple paths
User Request (bound/unbound) for Resource allocation
Path Acceptance Check -> Acceptance ratio Results
Acceptance ratio calculation and feasibility evaluation
Generated flowspace rules
Flowspace Table injection
Proxy controller flowspace lookup
Feasibility Statistics (Time overhead, Rule number, Memory consumption)

Figure 5.1-1: Structure of the software implementation

5.2. Detailed implementation analysis

In this section, a detailed description of the scripts, mentioned in section 5.1, is presented.

5.2.1. Script run_engine.sh

As mentioned above, this bash script initializes and starts the flowspace rule engine by defining all the necessary execution parameters. Such parameters include the desired real network topologies (**topo_list**), the desired number of virtual network topologies (**simple_paths, disjoint, star_paths**), the percentage of bound and unbound requests (**unbound**) (these requests are defined in detail below), the type of desired graph weights (**bandwidth**), an argument for writing output in a file (**w**), an argument specifying whether the evaluation metrics will be computed (**reuse**), the graph interconnection points (**connection_points**), an argument specifying whether the graph weigths will be computed or imported from a source file (**weights_from_file**) and, if that is the case, the name of the file that includes the graph weights (**weights_file**). All of these parameters are described in detail below:

- **topo_list:** This argument specifies the desired real network topologies, which constitute one of the inputs given to the engine. The structure of each network topology is included in "The Internet Topology Zoo" project. Such network topologies include **Internet2/OS3E [20]**, **GÉANT [21], ULAKNET [22]** and **PSiNET [23].**

- **simple_paths:** The desired number of simple paths. These are multi-hop (point-to-point) paths with no repeated vertices. Source and destination nodes are randomly chosen.
- **disjoint:** The desired number of disjoint path sets. These are sets of paths between a source and a destination node having no vertex and edge in common. Source and destination nodes are randomly selected.
- **star_paths:** The selected number of star topologies. A star topology is defined as a tree with one internal vertex and k leaves. Internal vertex and leaves are randomly selected.
- **unbound:** A parameter that takes a numerical value specifying the percentage of unbound tenant requests. The percentage of bound requests is computed as **(100-$UNBOUND)%** These two types of requests are defined as follows:
    - **Bound request:** a tenant requests a specific instance of the selected logical separator across a path (e.g. a specific VLAN ID or MPLS label).
    - **Unbound request:** the allocation of any available instance of the selected logical separator across a path is acceptable (e.g. any available VLAN ID or MPLS label).
- **bandwidth:** A parameter that takes a boolean value. If that value is TRUE, graphs weights represent link bandwidths otherwise they represent propagation delays. The default value of this parameter is FALSE.
- **w:** This argument determines that the evaluation results will be printed in an output file.
- **reuse:** A parameter that takes a boolean value. If that value is TRUE, the computation of the evaluation metrics is executed.

Otherwise, none of the metrics are computed. Its default value is FALSE.

- **connection_points:** The interconnection points of the specified network topologies (network domains) are defined as a list of strings. This parameter is reserved for the future development of the rule engine execution in multi-domain environments.
- **weights_from_file:** A parameter that takes a boolean value. If that value is TRUE, the graph weights are imported from a source file otherwise they are computed during the engine execution. The default value of this parameter is FALSE.
- **weights_file:** In case that the previous parameter has a TRUE value, this parameter specifies the name of the .json file [24] that includes the graph weights.

After the initialization of these parameters, the script **generate.py** is called and takes them as input arguments.

5.2.2. Script generate.py

This script generates each of the selected real network topologies as a topology graph by parsing its attributes from the dataset of "The Internet Topology Zoo" project (described in detail in section 5.3). For this purpose, the **get_topo_graph** function is called. This function is contained in the **topo_lib.py** script, which is described in the next section. Moreover, the selected type of graph weights is generated and these weights are appended to a NetworkX [25] Graph data structure named **g**. In case of a successful topology parsing from the dataset, this successful parsing is recorded and the function **do_metrics** is called for the initialization of the evaluation metric computation otherwise the topology is ignored.

In addition to the above features, there is an option of unifying two or more network topology graphs. However, this option is noted as a comment because it is reserved for the future development of the engine regarding multi-domain environments.

All the aforementioned process is described in Snippet 5.2.2-1.

```
for i, topo in enumerate(topos):
    print "topo %s of %s: %s" % (i + 1, t, topo)
    g, usable, note = get_topo_graph(topo)
        #g_unified = nx.union(g,topo_test)#unify graphs
    exp_filename = metrics.get_filename(topo, options)

    if not g:
      raise Exception("WTF?!  null graph: %s" % topo)


    elif not options.force and os.path.exists(exp_filename + '.json'):
print "skipping already-analyzed topo: %s" % topo
      ignored.append(topo)
    elif not has_weights(g):
      ignored.append(topo)
      print "no weights for %s, skipping" % topo
    else:
      do_all(topo, g, 1, 1, None, mylist)
      successes.append(topo)

  print "successes: %s of %s: %s" % (len(successes), t, successes)
  print "ignored: %s of %s: %s" % (len(ignored), t, ignored)
```

Snippet 5.2.2-1: Snippet of script generate.py that determines whether the attributes of a network topology were parsed successfully from the dataset.

5.2.3. Script topo_lib.py

The most important function included in this script is **get_topo_graph** (Snippet 5.2.3-1). It takes as input a network topology name and parses the corresponding network graph attributes from the dataset of "The Internet Topology Zoo" project. To that end, it either returns an error message or the entire topology graph. If the attribute parsing process is successful, the selected type of weights and each node name are attached to the NetworkX Graph data structure named **g**.

```
def get_topo_graph(topo):
  if topo == 'os3e':
      g = OS3EWeightedGraph()
      return g, True, None
  elif topo == 'Geant2012':
    g, note, note2 = import_zoo_graph(topo)
      attr = nx.get_node_attributes(g,'Country')
      for node in g.nodes():
        temp = str(attr[node])
        mapping[node] = temp
      g = nx.relabel_nodes(g,mapping)
      return g, True, False
  else:
      g, note, note2 = import_zoo_graph(topo)
    attr = nx.get_node_attributes(g,'label')
    for node in g.nodes():
        temp = str(attr[node])
        mapping[node] = temp
    g = nx.relabel_nodes(g,mapping)
    return g, True, False
```

Snippet 5.2.3-1: the **get_topo_graph** function

5.2.4. Script metrics.py

For each successful parsing of a network topology, the function **do_metrics** is called. This function initializes the computation of acceptance ratio and the generation of flowspace rule tables (one for each slicing method) for a single network domain.

Firstly, the flowspace rule tables are initialized as empty lists and the input arguments are parsed. Secondly, if the input argument **weights_from_file** (defined in section 5.2.1) has a TRUE value, the graph weights are imported from a source file. If that is the case, the function **read_weights_from_file** (Snippet 5.2.4-1) is called. This function takes as inputs the topology graph and the name of the file that includes the graph weights on an appropriate format. The function **read_json_file** (Snippet 5.2.4-2) is used for the weight parsing from the given .json file. Finally, the parsed graph weights are appended to the NetworkX Graph data structure named **g**.

```
def read_weights_from_file(g,filename):
        weights = {}
        weights = read_json_file(filename)
        for src,dst in g.edges():
           tuples = [weights.get(src)]
           if tuples[0]!=None:
                      try:
                                 index = tuples[0].index(dst)
                      except ValueError:
                                 continue
                      else:
                                 g[src][dst]['weight'] = tuples[0][index+1]
           tuples = [weights.get(dst)]
           if tuples[0]!=None:
                  try:
                                 index = tuples[0].index(src)
                      except ValueError:
                                 continue
                  g[src][dst]['weight'] = tuples[0][index+1]
           return
```

Snippet 5.2.4-1: the **read_weights_from_file** function

```
def read_json_file(filename):
   input_file = open(filename, 'r')
   return json.load(input_file)
```

Snippet 5.2.4-2: the **read_json_file** function

The process of the optimal path finding from each source to each destination node is executed according to the Dijkstra algorithm using the built-in functions of NetworkX **all_pairs_dijkstra_path_length** and **all_pairs_dijsktra_path.**

After that, using the value of the argument **disjoint,** the required number of disjoint path sets is searched. Specifically, the disjoint path finding process is executed by calling the function **vertex_disjoint_shortest_pair** of script **paths.py**, which is described in the next section. If there are less disjoint path sets in the given graph than the requested number, the maximum number of existing disjoint paths is returned. In any case, an appropriate message is printed to standard output. This process is described in Snippet 5.2.4-3.

```
for node in g.nodes():
        if dis_counter >= disjoints:
                break
        src = node
        counter = 0
        for i in range(len(dst))
        temp1,temp2  = paths.vertex_disjoint_shortest_pair(g, src, dst[i])
        if temp1!=None and temp2!=None:
                length1 = get_length(apsp,temp1)
                        if length1 == -1:
                                break
                paths_temp.append((temp1,length1,dst[i]))
                length2 = get_length(apsp,temp2)
                        if length2== -1:
                                break
                paths_temp.append((temp2,length2,dst[i]))
                counter = counter+2
        elif temp1!=None and temp2==None:
                length = get_length(apsp,temp1)
                        if length == -1:
                                break
                paths_temp.append((temp1,length,dst[i]))
                counter=counter+1
        if counter == 0 or counter==1:
                continue
        paths_temp = sorted(paths_temp, key=itemgetter(1))
      path1,path2 = get_disjoint(g,paths_temp)
        if path1!=None and path2!=None:
                dis_counter = dis_counter +2
                dis_paths.append(path1[0])
                dis_paths.append(path2[0])

      if dis_counter == disjoints:
        print("-------Found %d disjoint paths" % dis_counter)
      else:
                print("-------Found %d disjoint paths out of %d that was
requested" % (dis_counter,disjoints))
```

Snippet 5.2.4-3: Disjoint path finding

After the disjoint path finding process, the evaluation metric computation takes place by calling the **compute_metrics** function of script **evaluation.py**, described in section 5.2.6. Finally, the output

.json files, which will contain the generated flowspace rule tables and the resulting acceptance ratios, are created and the computed metrics are copied to them. The functions get_filename and get_tablefilename that are presented in Snippet 5.2.4-4 create the appropriate file names.

```
def get_filename(topo, options):
   number_of_requests = options.star_paths + options.disjoint +
options.simple_paths
   type_of_requests = options.unbound
   mix = options.mix
   filename = "acceptance_ratio/" + topo + str(number_of_requests) +
mix  + "("+ str(type_of_requests) +"% unbound)" +"/"
   return filename

def get_tablefilename(topo,options):
        number_of_requests = options.star_paths + options.disjoint +
options.simple_paths
        type_of_requests = options.unbound
        mix = options.mix
        filename_domain = "tables/" + "domain-wide"  + topo +
str(number_of_requests) + mix+ "(" + str(type_of_requests) + "%
unbound)"+ "/"
        filename_switch = "tables/" + "switch-wide"  + topo +
str(number_of_requests) + mix+ "(" + str(type_of_requests) + "%
unbound)"+ "/"
        filename_port = "tables/" + "port-wide"  + topo +
str(number_of_requests) + mix+ "(" + str(type_of_requests) + "%
unbound)"+ "/"
           return filename_domain, filename_switch, filename_port
```

Snippet 5.2.4-4: The **get_filename** and the **get_tablefilename** functions

5.2.5. Script paths.py

This script contains the Python implementation of the algorithm described in section 3.4.2. Given a source and a target graph node, the included functions search whether a set of optimal paths, which are both edge and vertex disjoint, exists between these end nodes.

As stated in section 3.4.2, in order to check whether the conditions of edge-disjointness and vertex-disjointness are fulfilled, a number of runs of the modified Dijkstra Algorithm for Shortest Path finding are required. If the requested disjoint path set exists, the paths are returned along with their costs (sum of their edge weights) to the user. Otherwise, an error message is returned.

5.2.6. Script evaluation.py

This script computes the evaluation metrics and generates the required flowspace rules for the domain-wide, the switch-wide and the port-wide slicing methods. The tenant requests for virtual network topologies along with the topology graph are needed for this computation.

The main function of this script is **compute_metrics.** Firstly, the number of bound and unbound requests for each virtual network topology type is computed. After that, the number of each switch interconnection ports is computed based on the number of edges that are attached to each switch (Snippet 5.2.6-1).

```
def precompute_ports(g,mylist):
      for node in g.nodes():
        neighboors = g.neighbors(node)
        num_of_neigh = len(neighboors)
        for port in range(0,num_of_neigh):
                mylist.append((node,neighboors[port],port+1))
        return mylist
```

Snippet 5.2.6-1: the **precompute_ports** function

The tenant request acceptance ratio is computed by separating bound and unbound requests. The type of each tenant request (bound or unbound) and the selected slicing method differentiate the way that a tenant request gets accepted or rejected by the rule engine. Specifically:

- In case of an unbound request using the:
    - **Domain-wide slicing method:** if any instance of the selected logical separator is available across the entire physical network, the request is accepted otherwise it is rejected.
    - **Switch-wide slicing method:** if any instance of the selected logical separator is available within each switch of the generated path, the request is accepted otherwise it is rejected.
    - **Port-wide slicing method:** if any instance of the selected logical separator is available on the appropriate ingress and egress port of each switch across the generated path, the request is accepted otherwise it is rejected.

- In case of a bound request using the:
    - **Domain-wide slicing method:** if the selected instance of the logical separator is available across the entire network, the request is accepted otherwise it is rejected.
    - **Switch-wide slicing method:** if the selected instance of the logical separator is available within each switch of the generated path, the request is accepted otherwise it is rejected.
    - **Port-wide slicing method:** if the selected instance of the logical separator is available on the appropriate ingress and egress port of each switch across the generated path, the request is accepted otherwise it is rejected.

In this context, this script contains functions for accepting or rejecting a tenant request based on the request type in case of each virtual network topology type (simple path, star topology or disjoint path set). These functions also generate the required flowspace rules, in case of an accepted tenant request, based on the isolation policy and the flowspace rule reduction approach described in chapter 4. In Snippet 5.2.6-2 and Snippet 5.2.6-3, the functions regarding bound requests for simple paths using the port-wide slicing method and unbound requests for simple paths using the domain-wide slicing method are presented respectively. Finally, it is worth noting that each star topology used for the metric computation includes two neighboring nodes across each star radius.

```
def
reusability_perswitch_perport(FlowSpace_port,user_id,mylist,vlan,paths,port_list,numb
er_of_rules):
        query = []
        query2 = []
        f = itemgetter(0,1)
        temp_no=0
        rules_to_append = []
        length = len(paths)
        for i in range(length-1):
           temp_no += 2
          src_node = paths[i]
           dst_node = paths[i+1]
           index = map(f,port_list).index((src_node,dst_node))
           port1 = port_list[index][2]
           index = map(f,port_list).index((dst_node,src_node))
           port2 = port_list[index][2]
          temp = (src_node,port1,dst_node,port2,vlan)
           index = map(f,port_list).index((src_node,dst_node))
          rules_to_append.append((user_id,3000,port1,src_node,vlan))
           rules_to_append.append((user_id,3000,port2,dst_node,vlan))
          query.append(temp)
           temp2 = (dst_node,port2,src_node,port1,vlan)
           query2.append(temp2)
        rules_to_append.append((user_id,3000,port2,dst_node,vlan))
        temp_no +=1
        if len(mylist)==0:
           for i in range(len(query)):
                      temp1 = query[i]
                      temp2 = query2[i]
                      mylist.append(temp1)
                      mylist.append(temp2)
           for i in range(len(rules_to_append)):
                      FlowSpace_port.append(rules_to_append[i])
           user_id +=1
           number_of_rules[2] = temp_no
           return True
        for i in range(len(query)):
           temp1 = query[i]
           temp2 = query2[i]
           if ((temp1 in mylist) or (temp2 in mylist)):
                      return False
        for i in range(len(query)):
           temp1 = query[i]
          temp2 = query2[i]
           mylist.append(temp1)
           mylist.append(temp2)
        for i in range(len(rules_to_append)):
        FlowSpace_port.append(rules_to_append[i])
     user_id +=1
        number_of_rules[2] = temp_no
        return True
```

Snippet 5.2.6-2: Computation of acceptance ratio and rule generation in case of bound requests for simple paths using the port-wide slicing method

```
def
reusability_per_domain_unbound(FlowSpace_domain,user_id,mylist,sh
ow_vlan_unbound,number_of_rules):
        temp = show_vlan_unbound[0]+1
      while temp <= 4096:
           rules_to_append = []
        if not temp in mylist:
                                 mylist.append(temp)
                      show_vlan_unbound[0]=temp
                      number_of_rules[0] = 1

        FlowSpace_domain.append((user_id,3000,'*','*',show_vlan_unbou
nd))
                      user_id +=1
                      return True
           else:
                      temp=temp+1
      return False
```

Snippet 5.2.6-3: Computation of acceptance ratio and rule generation in case of unbound requests for simple paths using the domain-wide slicing method

5.2.7. Script graph_util.py

This script contains functions regarding the interconnection of two (or more) given network graphs. For each desired pair of graphs to be interconnected, the following actions are performed: Firstly, the interconnection points are parsed and, secondly, these points are connected with each other by a bidirectional link. This link weight is also computed and attached to the network graph.

This script is about to be used for the future enhancement of the rule engine, so that to support multi-domain environments. The function **parse_points,** which parses the given interconnection graph points, is presented in Snippet 5.2.7-1.

```
def parse_points(g,dst,connection_points):
    counter = 1
    char = ''
    src= ''
    i=0
    enough = 0
    length= len(connection_points)
      while counter<length-1:
        enough=0
        while enough<1:
            char = connection_points[counter]
            if char == '@' and enough == 0:
                char=' '
                src = src + char
                counter = counter +1
            elif char!= ',' and char!=']' and enough == 0:
                src = src + char
                counter = counter +1
            elif char == ',' and enough == 0:
                enough = enough +1
                counter = counter +1
            else:
                        enough = enough +1
                counter = counter +1
        if enough == 1:
                if g.__contains__(src):
                dst.append(src)
                src=''
        return dst
```

Snippet 5.2.7-1: the **parse_points** function

5.2.8. Script lookup_process.py

This script takes as input a user query and initializes the flowspace lookup process by creating various data structures. Specifically, the required parameters of the k-dimensional binary search tree algorithm are initialized, the hash tables are created and the functions implementing the various lookup algorithms are called.

5.2.9. Script lookup.py

This script contains the functions implementing the various lookup algorithms noted in section 3.4.1. The average and worst-case time complexity of these algorithms have been mentioned in section 3.4.1 as well. The first function implements the linear search algorithm and computes the elapsed time of the lookup process. The linear search or "naïve" search algorithm is used as a point of reference for the elapsed time of the lookup process.

The second function implements the search algorithm of open addressing with double hashing. As noted in section 3.4.1, this algorithm is expected to result in a quite efficient lookup process. However, because of the double hashing that takes place, this algorithm is expected to result in slower lookups than the single hashing algorithm.

The third function implements the single hashing search algorithm. This algorithm is expected to result in the fastest lookups because of its simple hashing approach. The software implementation of this algorithm is presented in Snippet 5.2.9-1.

```
def single_hashing(array,element,crc32_values):
   start = time.time()
   key = 
zlib.crc32('{}{}{}{}'.format(element[0],element[1],element[2],element[
3]))
   key_existing = crc32_values.has_key(key)
   if key_existing :
      if crc32_values[key] == 
['{}{}{}{}'.format(element[0],element[1],element[2],element[3])]:
         end=time.time()
         print("Simpe hashing found the requested element after %s ms" 
% ((end-start)*1000))
         return
   else:
      end=time.time()
      print("Simple hashing did not find the requested element after %s 
ms" % ((end-start)*1000))
      return
```

Snippet 5.2.9-1: Software implementation of the single hashing search algorithm

5.2.10. Script kdtree.py

This script constructs a k-dimensional binary search tree (kd tree) and implements the lookup process in it for a requested element. The average and the worst-case time complexity of this algorithm were noted in section 3.4.1. In case that the requested element is not a part of the kd tree, the nearest neighbor of this element is returned. If the lookup process is successful (the requested element is actually a part of the tree), the corresponding tree node is returned.

In either case, along with the lookup process outcome, the distance of the returned node from the root of the tree and the nodes visited during the lookup process are returned. In Snippet 5.2.10-1, the class of a k-dimensional binary search tree node is described.

```
class Kd_node(object):
    __slots__ = ["dom_elt", "split", "left", "right"]
    def __init__(self, dom_elt_, split_, left_, right_):
        self.dom_elt = dom_elt_
        self.split = split_
        self.left = left_
        self.right = right_
```

Snippet 5.2.10-1: The class of a k-dimensional binary search tree node

5.2.11. Script util_lookup.py

This script contains some quite simple, but useful functions for handling various data types during the execution of the lookup algorithms. Such functions convert the elements of a data structure from one type to another and copy a multi-dimensional array to another. For instance, there is a function that converts an ascii string to an integer number and a function that copies any four dimensional array to another.

5.3. The Internet Topology Zoo project

The dataset of this project includes the structure of diverse real network topologies. Specifically, it includes their nodes and edges. Each topology node consists of the following attributes: *id*, *label*, *Country*, *Longitude, Internal* and *Latitude.* Moreover, each topology edge consists of a certain group of the following attributes: *source, target, LinkType, LinkLabel, LinkSpeed, LinkSpeedUnits, LinkSpeedRaw* and *LinkNote.* However, for certain network topologies, the available dataset was incomplete. To that end, it was updated and enhanced by the completion of the missing attributes. An example of a graph node and a graph edge is shown in Snippet 5.3-1. The available attributes are described in detail below:

- **id**: this attribute refers to the sequence number of a particular node.
- **label**: it refers to the label (name) of a particular graph node.
- **Country**: it refers to the name of the country where a particular graph node is located.
- **Longitude**: this attribute refers to the longitude of a graph node, so that precise propagation delays among nodes are computed. These delays are used as the link weights when that is determined by the input arguments.
- **Internal**: reserved attribute for internal graph functionalities.
- **Latitude**: this attribute refers to the latitude of a graph node. It is used along with the **Longitude** attribute.

- **source**: the **id** attribute of the source node of a graph link.
- **target**: the **id** attribute of the target node of a graph link.
- **LinkType:** this attribute refers to the type of a link. Nowadays, the most prevalent choice is optical fiber.
- **LinkLabel:** a string attribute that refers to a link speed along with the speed unit of measurement, for instance "*10 Gbps*". This attribute represents the link bandwidth used as the link weight in case that this is determined by the input arguments.
- **LinkSpeed:** this attribute refers to a link speed as a numerical string.
- **LinkSpeedUnits:** the unit of measurement of a link speed.
- **LinkSpeedRaw:** it refers to a link speed as a floating-point number.
- **LinkNote:** a note regarding a graph link.

```
node [
	id 15
	label "GR"
	Country "Greece"
	Longitude 23.71622
	Internal 1
	Latitude 37.97945
]

edge [
	source 4
	target 31
	LinkSpeed "10"
	LinkLabel "10 Gbps"
	LinkSpeedUnits "G"
	LinkSpeedRaw 10000000000.0
]

edge [
	source 5
	target 23
	LinkType "Fibre"
	LinkLabel "10 Gbps"
	LinkNote "Lit "
]
```

Snippet 5.3-1: Examples of a graph node and a graph edge (2 cases) included in "The Internet Topology Zoo" project

6. Evaluation of the proposed implementation

In this chapter, the evaluation results of the implemented slicing methods are presented. The computed evaluation metrics were discussed in section 5.1.

The main part of the experimental setup was tenant requests. Each of these requests was associated with a randomly generated virtual network topology that belongs to one out of three different categories: **i) simple paths, ii)star topologies and iii)disjoint path sets**. For the performed experiments, the following mixture scenarios were generated:

- **Mix1:** this mixture consisted of 17 requests for disjoint path sets. Half of the remaining requests regarded star topologies and the rest of these requests regarded simple paths. All of these requests were bound.
- **Mix2:** this mixture consisted of 17 requests for disjoint path sets, while 70% of the remaining requests regarded star topologies and 30% regarded simple paths. All of these requests were bound.
- **Mix3:** it consisted of 17 requests for disjoint path sets, while 70% of the remaining requests regarded star topologies and 30% regarded simple paths. 20% of the requests for each category were unbound.
- **Mix4:** it consisted of 17 requests for disjoint path sets, while 70% of the remaining requests regarded star topologies and 30% regarded simple paths. All of these

requests were unbound.

It should be noted that the number of disjoint paths, which was selected for the performed experimental evaluation, was based on the maximum number of existing disjoint paths per physical network topology.

6.1. Tenant request acceptance ratio

In this experiment, the slicing method implementation runs on top of diverse real network topologies for various numbers of randomly generated tenant requests (up to 16,000) that are consistent with the aforementioned mixture scenarios (mix1, mix2, mix3 and mix4**)** and the resulting acceptance ratio is presented. The real network topologies used for the evaluation process consisted of 6 up to 81 network nodes (WAN, medium and small sized topologies).

In Figure 6.1-1 and Figure 6.1-2, the resulting acceptance ratio, in case of the Internet2/OS3E topology (34 nodes) for 8,000 and 16,000 randomly generated tenant requests using the aforementioned slicing methods and mixture scenarios, is presented. Link bandwidths parsed from the dataset of " The Internet Topology Zoo" project constituted the graph weights.

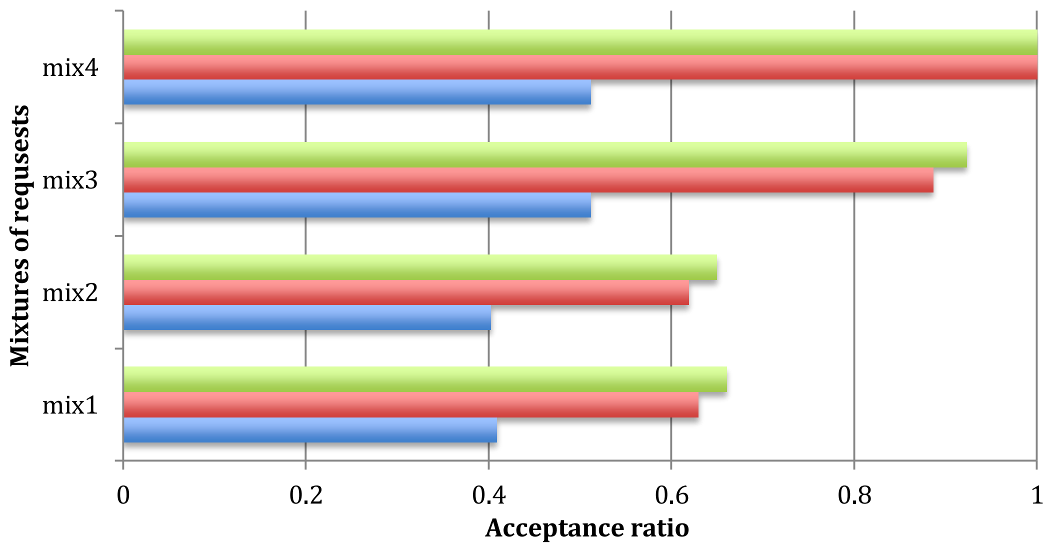


Figure 6.1-1: Acceptance ratio in Internet2/OS3E for 8K requests

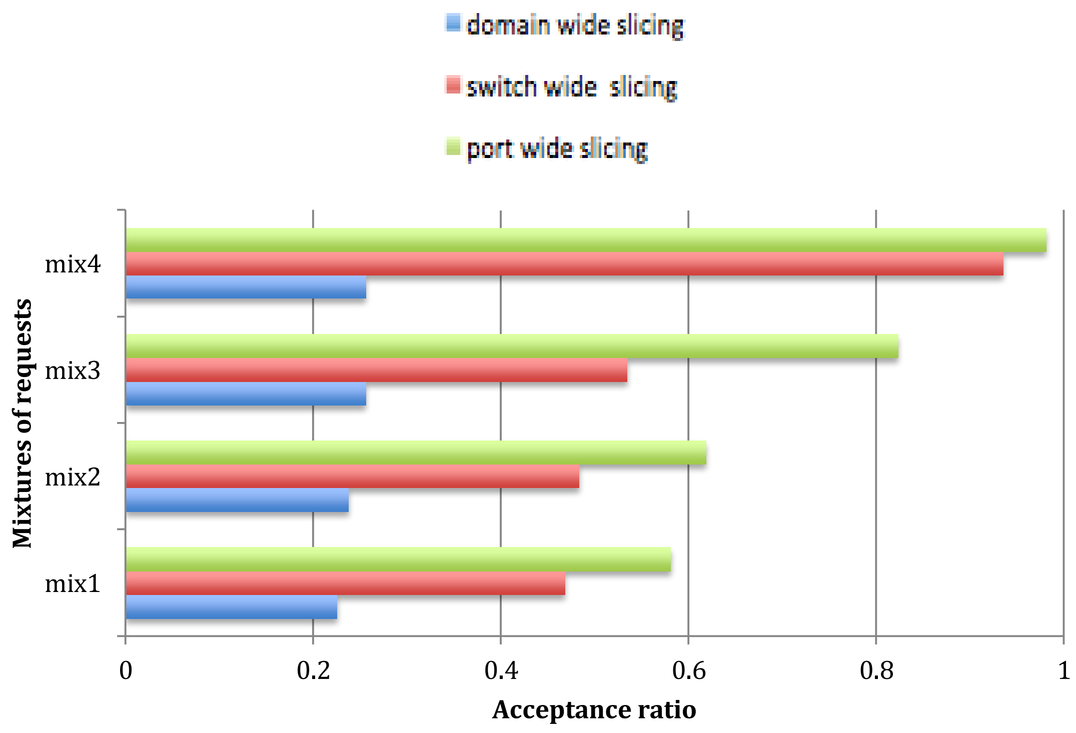


Figure 6.1-2: Acceptance ratio in Internet2/OS3E for 16K requests

In the aforementioned figures, it is shown that the port-wide slicing method scales better as the number of tenant requests increase. The acceptance ratio of this method is quite large in scenarios that involve a small percentage of unbound tenant requests (i.e. mix3) providing almost perfect resource utilization for exclusively unbound requests (i.e. mix4). In this context, the port-wide slicing method accepts more than 80% of the tenant requests for a total of 16,000 requests (reaching 97-98% for the mix4 scenario). In scenarios involving exclusively bound tenant requests (i.e. mix1, mix2), the efficiency of this method is quite satisfactory as well.

In Figure 6.1-3, the resulting acceptance ratio, in case of the GÉANT backbone topology (39 nodes) for 16,000 tenant requests and mixtures 1-4, is presented. For this experiment, the propagation delays among the topology nodes were computed and attached as graph weights.

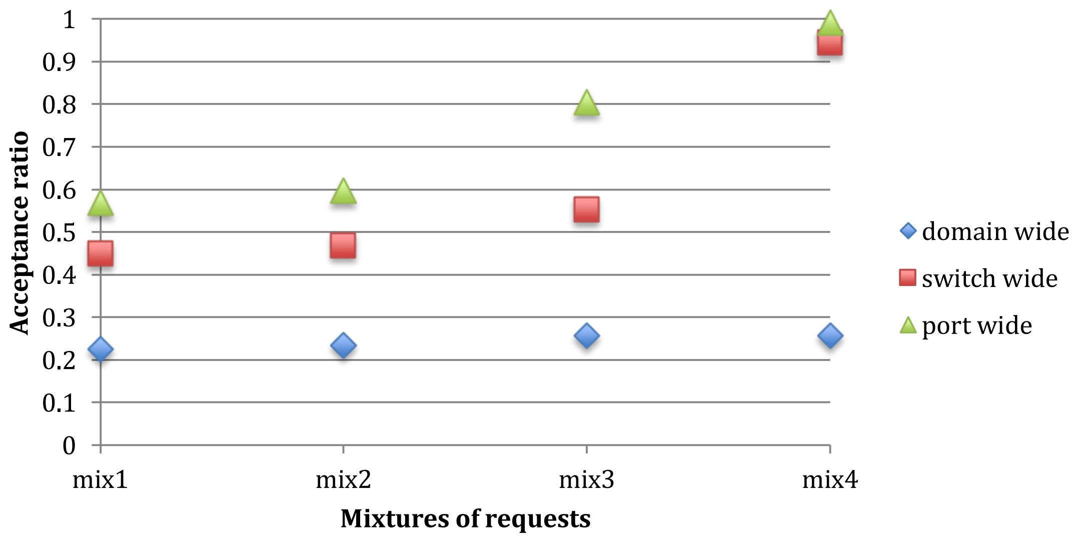


Figure 6.1-3: Acceptance ratio scaling as the percentage of unbound requests increase

The above figure illustrates that as the percentage of unbound requests (for a certain total number of tenant requests) is increased, the port-wide slicing method scales much better than the switch-wide and the domain-wide slicing methods ideally accepting all tenant requests for a total of exclusively unbound requests.

In order to evaluate the efficiency of the implemented slicing methods in very large real network topologies, experiments involving the ULAKNET network topology (81 network nodes) were performed. The propagation delays among the topology nodes were computed and attached as graph weights. The resulting acceptance ratio for all the mixture scenarios in case of 16,000 requests is illustrated in Figure 6.1-4.

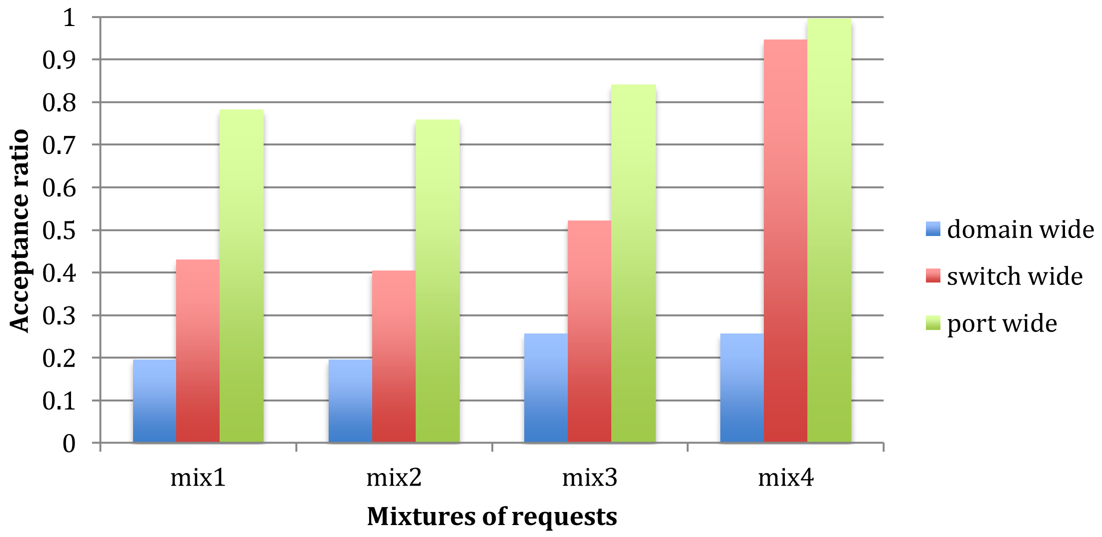


Figure 6.1-4: Acceptance ratio in the ULAKNET network topology for 16K requests

Moreover, experiments were performed in small sized real network topologies. A typical example is the PSiNET network topology (23 nodes). This experiment was based on propagation delays. The resulting acceptance ratio in case of 16,000 requests for mixtures 1-4 is illustrated in Figure 6.1-5.

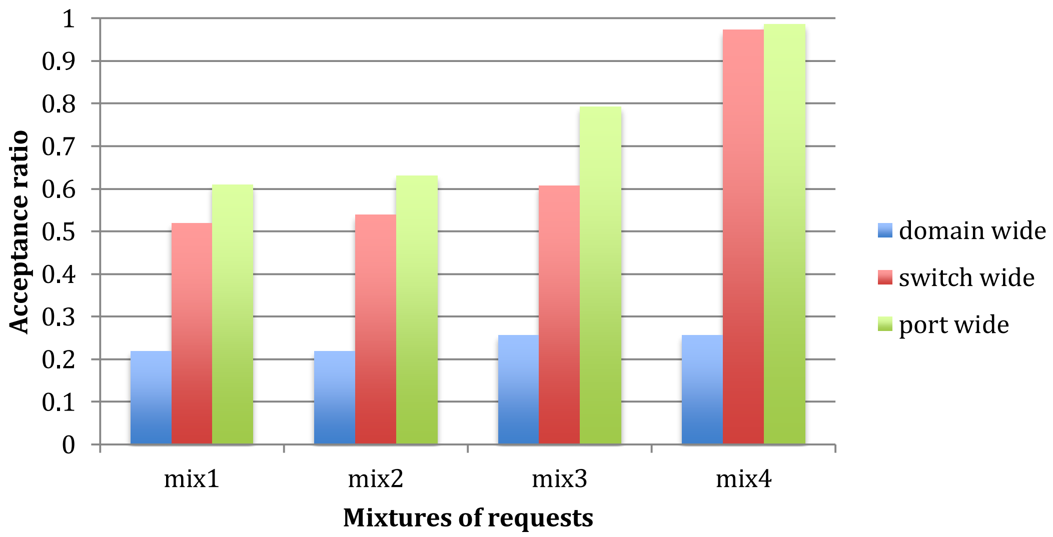


Figure 6.1-5: Acceptance ratio in the PSiNET network topology for 16K requests

As illustrated in Figure 6.1-1, Figure 6.1-2, Figure 6.1-3, Figure 6.1-4 and Figure 6.1-5, the port-wide slicing method results in the largest acceptance ratios (scaling up to 16,000 tenant requests). The performed experiments also showed that this deduction is accurate for real network topologies of various sizes (small, medium and WAN sized network topologies).

6.2. Generated flowspace rule tables

The implemented rule engine, except for computing the acceptance ratio for each slicing method and for given real network topologies and tenant requests, generates the required flowspace rules based on the isolation policy described in section 4.2 and the rule reduction approach described in section 4.3.

As mentioned in section 6.1, the port-wide slicing method results in the highest efficiency in terms of acceptance ratio. However, the high efficiency comes at the cost of the increased number of flowspace rules required to be generated and established within an OpenFlow proxy controller, so that isolation is enforced.

A large number of established flowspace rules could result in a great performance overhead added to the networked system by the OpenFlow proxy controller. In such a case, the proxy controller would handle extremely large flowspace rule tables resulting in a slow manipulation process of its flowspace.

For this evaluation experiment, the selected real network topologies were divided into three groups. The first group consisted of small sized network topologies (up to 25 topology nodes), the second group of medium sized network topologies (26-45 topology nodes) and the last group of WAN sized topologies (46-81 topology nodes).

In the figures below, the number of required rules for the port-wide slicing method, normalized to the number of required rules for the domain-wide slicing method (Figure 6.2-1) and the switch-wide slicing method (Figure 6.2-2), is illustrated. These comparisons were performed for each topology group and tenant request mixture scenario (for a total of 6,000 requests).

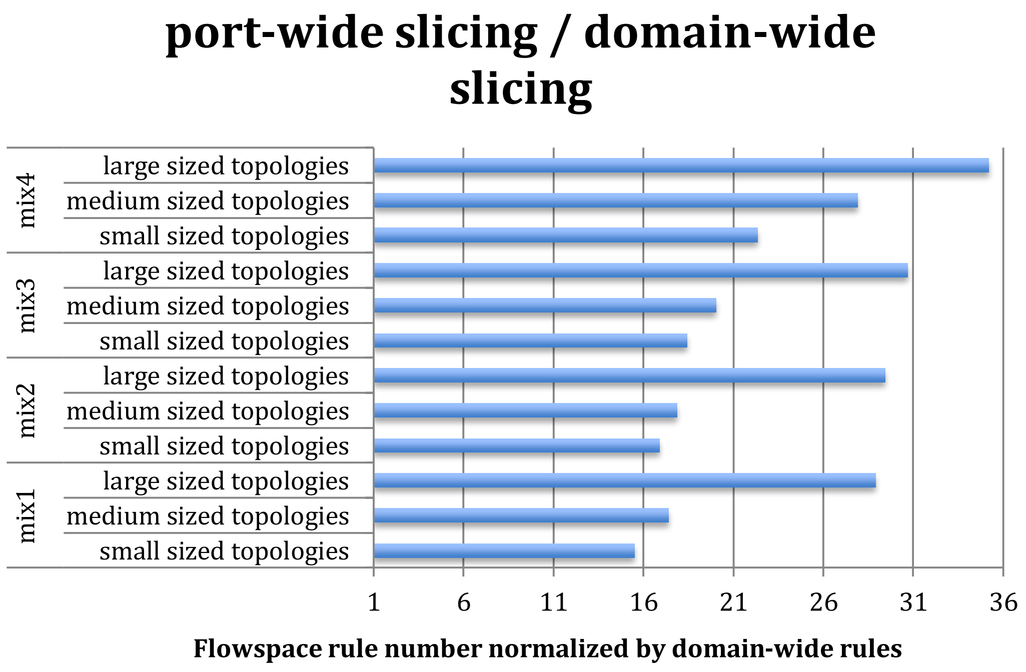


Figure 6.2-1: Flowspace rule number for port-wide method normalized by domain-wide rules for small, medium and large sized topologies (6K requests)

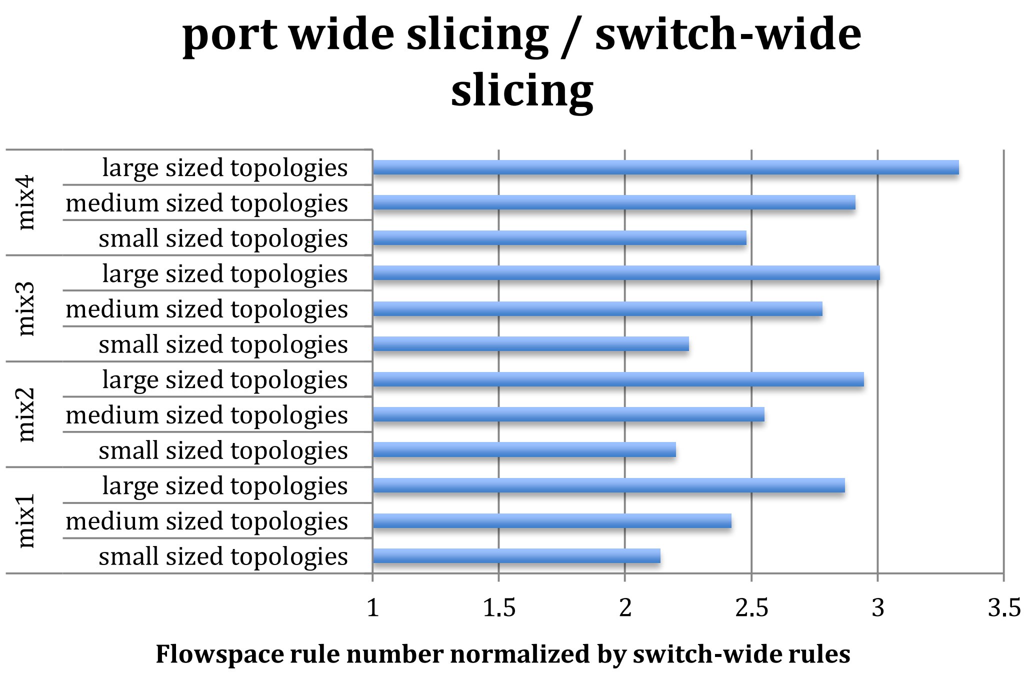


Figure 6.2-2: Flowspace rule number for port-wide method normalized by switch-wide rules for small, medium and large sized topologies (6K requests)

Based on the results illustrated in the aforementioned figures, the initial case was validated. Indeed, the port-wide slicing method results in the highest acceptance ratios, but also in the largest numbers of required flowspace rules. By carefully observing Figure 6.2-1 and Figure 6.2-2, the following conclusions are drawn:

I. The port-wide slicing method results in larger required flowspace rule tables compared to domain-wide and switch-wide slicing. That is to say, in port-wide slicing, a tenant request is translated into more flowspace rules than in domain-wide and switch-wide slicing.

II. Given a mixture scenario, the normalized rule number is greater for large (WAN) sized real network topologies than for medium and small sized topologies.

III. Given a topology group, the normalized rule number is the greatest in case of exclusively unbound requests (i.e. mix 4).

This is due to the fact that, in such a mixture scenario, a greater percentage of the overall tenant requests gets accepted and thus more flowspace rules are generated.

6.3. Feasibility evaluation of the implemented slicing methods

In order to evaluate the feasibility of the implemented slicing methods within a networked system, the software implementation was associated with a popular OpenFlow proxy controller, FlowVisor.

Generally speaking, an OpenFlow proxy controller, such as FlowVisor, adds performance overhead to actions that cross between the control and data plane layers of a SDN system. This is due to the fact that an additional layer between these planes has been added.

As mentioned in section 6.2, the port-wide slicing method results in high efficiency at the cost of large generated flowspace rule tables. Despite their large number, these rules should be handled efficiently by the proxy controller, so that the isolation policy among tenants is enforced.

Specifically, in this experiment, the generated non-overlapping flowspace rules were injected into FlowVisor (version 1.4) in order to measure the introduced performance (time) overhead and its memory consumption. It should be noted that the used FlowVisor version provides high performance of flowspace lookups due to the advanced implemented hashing algorithms instead of the "naïve" linear search algorithm implemented in the early FlowVisor releases.

For the quantification of the performance overhead, the method described in [4] was used. Specifically, the time between receiving a control packet from an OpenFlow switch and sending this packet to a tenant OpenFlow controller was measured. For this measurement, the *libpcap [26]* was used. For the measurement of the required memory, operating system specific RAM metrics were used.

In the following tables, the obtained measurements are presented. Specifically, these measurements regard the entire Internet2/OS3E topology (Table 6.3-1) as well as the Internet2/OS3E node of Chicago (Table 6.3-2). This particular node was selected because it has one of the highest node degrees across the entire network topology and thus is heavily used.

	Internet2/OS3E topology		
Tenant requests	**Generated flowspace rules**	**Performance overhead (ms)**	**Memory consumption (Mbytes)**
1K	15K	0.042	622
2K	43K	0.044	1643
4K	95K	0.050	3951
6K	145K	0.053	5910
7K	175K	0.056	7400

Table 6.3-1: Performance overhead and memory consumption of FlowVisor regarding the entire Internet2/OS3E topology

	Internet2/OS3E Chicago Node		
Tenant requests	**Generated flowspace rules**	**Performance overhead (ms)**	**Memory consumption (Mbytes)**
1K	1.1K	0.038	125
2K	2.5K	0.040	151
4K	4.6K	0.041	202
6K	7.5K	0.0414	250
7K	11K	0.0425	427

Table 6.3-2: Performance overhead and memory consumption of FlowVisor regarding the Internet2/OS3E node of Chicago

The aforementioned results demonstrate that an OpenFlow proxy controller, such as FlowVisor, adds a minor performance overhead to the network, even for a very large number of established flowspace rules (up to 175,000). However, in case of a real network topology that consists of 34 network nodes deployed across the United States of America (such as Internet 2/OS3E), the proxy controller memory consumption can be quite large, but not prohibitive for a generic purpose hardware hosting the OpenFlow proxy controller (such as a hosting server).

As a conclusion, the implemented slicing methods can be associated with an OpenFlow proxy controller, such as FlowVisor, as the obtained measurements showed that the proxy controller can efficiently handle the generated flowspace.

6.4. An approach towards a more efficient flowspace lookup process

In section 3.4.1, a detailed analysis of various search algorithms was presented. Moreover, in sections 5.2.8, 5.2.9, 5.2.10 and 5.2.11, the software implementation of those algorithms was described.

In case of a large amount of tenant requests or a WAN sized network topology, the generated flowspace is quite large. As a consequence, high performance flowspace lookups should be introduced in order to reduce the networked system time overhead.

In this context, the elapsed lookup time of the most efficient implemented search algorithms (single hashing and open addressing with double hashing) are compared with the FlowVisor time overhead, presented in Table 6.3-1 (for the same generated flowspace). In Figure 6.4-1, the resulting elapsed times are illustrated.

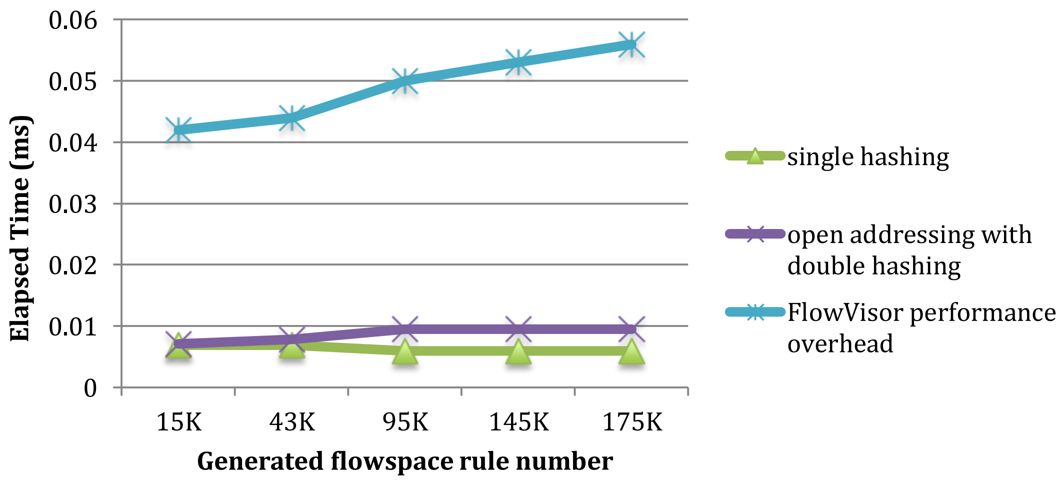


Figure 6.4-1: Elapsed time of the most efficient implemented search algorithms compared with the FlowVisor performance overhead

Based on this figure, it is deduced that, despite the advanced search algorithms used in version 1.4 of FlowVisor, the implemented lookup algorithms result in more efficient flowspace lookups. However, it should be noted that the FlowVisor time overhead consists of operating system specific overheads (e.g. the time needed by the FlowVisor process to interrupt the operating system) plus the elapsed flowspace lookup time. As a consequence, a part of the measured FlowVisor overhead does not regard the flowspace lookup process. Generally speaking, though, this part is minor (its typical value is a few useconds in modern computer systems) compared to the flowspace lookup overhead and, thus, it can be ignored.

7. Conclusion and Future Work

Material based on the aforementioned work was submitted for publication [27].

7.1. Conclusion

Multi-tenancy, as a feature of SDN, constitutes a typical case study of network virtualization. In this context, network virtualization aims at providing to each tenant the perception that it uses the available network resources exclusively on its own, without being aware of other tenant existence or the physical network substrate and topology. As a consequence, a basic principle of multi-tenancy is the isolation policy enforcement among network slices (tenants). As a result of this policy, potential conflicts among the existing network slices are prevented.

A typical way to achieve multi-tenancy is to apply one of the proposed **network control plane slicing methods** across a physical substrate network. The proposed slicing methods, defined and analyzed in section 4.1, are the following: **(i) domain-wide slicing**, **(ii) switch-wide slicing** and **(iii) port-wide slicing.**

In order to enforce isolation among tenants based on a network slicing method, a number of non-overlapping flowspace rules should be created. For instance, in case of a deployed OpenFlow transparent intermediate controller (e.g. FlowVisor), the created non-overlapping flowspace rules should be established and handled by it.

In case of a large number of issued tenant requests or a WAN sized real network topology, the number of required flowspace rules could be quite large. Based on these rules, the isolation policy enforcement could result in severe overheads. Thus, a rule reduction approach was proposed, resulting, at the worst-case scenario, in equal numbers of required flowspace rules for the switch-wide and the port-wide slicing methods.

An experimental evaluation of the proposed network slicing methods was performed via the association of these methods with real network topologies (e.g. ULAKNET and PSiNET). Based on this evaluation, the following conclusions were drawn:

- The port-wide slicing method results in the greatest efficiency and scales better for large amounts of tenant requests (up to 16,000). This conclusion is independent of the network topology size.
- The great efficiency of the port-wide slicing method comes at the cost of the large number of flowspace rules that should be established within an OpenFlow proxy controller (e.g. FlowVisor), so that isolation is safeguarded among tenants.
- In case of port-wide slicing, each accepted tenant request is translated into a larger number of flowspace rules compared to the domain-wide and the switch-wide slicing methods.
- Given a mixture scenario, the rule number for port-wide slicing, normalized by the domain-wide and switch-wide slicing rule number, is greater for WAN sized network topologies than for medium and small sized topologies.

Finally, a feasibility evaluation of the proposed slicing methods was performed via the injection of the generated flowspace rules into FlowVisor. This experiment showed that the proposed flowspace policies resulted in a minor proxy controller performance (time) overhead. However, the required rules resulted in large memory consumption by the proxy controller. This memory consumption, though, is not considered a severe limitation in modern generic purpose servers validating that the proposed implementation is robust enough to run on top of real network topologies.

7.2. Future Work

The proposed network slicing methods, and especially the port-wide slicing method, were proved to constitute a handy way to achieve multi-tenancy across a physical substrate network. However, these slicing methods could be enriched and enhanced in order to result in higher tenant request acceptance ratios. For instance, each physical switch port could be mapped to multiple virtual ports thus enabling the port-wide slicing method to perform much better scaling up to even larger numbers of tenant requests.

In case of an unbound request, the lookup, performed for an available instance of the selected separator tuple, is slow. Advanced hashing algorithms could be implemented for this lookup process resulting in greater performance and lower overheads.

Another quite useful and interesting development would be to add full integration of the proposed software implementation with the API of an OpenFlow proxy controller, such as FlowVisor. In this way, the software implementation would deal with real-time tenant requests and, as a consequence, would generate and inject flowspace rules into the proxy controller in real time.

8. References

[1] Open Networking Foundation https://www.opennetworking.org

[2] OpenFlow protocol http://archive.openflow.org

[3] McKeown, N., T. Anderson, et al. (2008). "OpenFlow: Enable Innovation in Campus Networks."

[4] R. Sherwood, G. Gibb, K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Can the production network be the testbed?" in *Proceedings of USENIX OSDI*, Vancouver, Canada, October 2010

[5] Balchunas, A. (2007). "Switching Tables v1.01."

[6] Martin Casado, Teemu Koponen, Rajiv Ramanathan, and Scott Shenker, "Virtualizing the network forwarding plane", in Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow (PRESTO '10), ACM , New York, USA, Article 8, 2010

[7] The FCAPS management Framework: ITU-T Rec. M. 3400

[8] N. Feamster, J. Rexford, and E. Zegura, "The Road to SDN: An Intellectual History of Programmable Networks," 2013.

[9] M. Yu, Y. Yi, J. Rexford, and M. Chiang, "Rethinking virtual network embedding: Substrate support for path splitting and migration," ACM SIGCOMM Computer Communication Review, vol. 38, no. 2, pp. 17–29, April 2008.

[10] PlanetLab http://www.planet-lab.org

[11] OVX (OpenVirtex) project http://www.sdncentral.com/projects/ovx-openvirtex

[12] Flowspace Firewall project http://globalnoc.iu.edu/software/sdn.html

[13] D. Knuth, *"The Art of Computer Programming, Volume 3: Sorting and Searching, Second Edition"*, Chapter 6.3, page 492. Addison Wesley, 1997.

[14] Bentley, J. L. 1975 Multidimensional binary search trees used for associative searching. Communications of the ACM, 18, 9 (Sept.), 509 517.

[15] R. Bhandari, "Optimal physical diversity algorithms and survivable networks", Proc. Second IEEE Symposium on Computers and Communications 1997, Alexandria, Egypt, July 1997, pp. 433-441.

[16] E.W. Dijkstra, "A Note on Two Problems in Connexion with Graphs", Numer. Math. 1 (1959) 269-271.

[17] OpenFlow Protocol version 1.4.0 specification https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf

[18] "Python Programming Language. *http://www.python.org*"

[19] " The Internet Topology Zoo" project http://www.topology-zoo.org

[20] "Internet2/OS3E http://www.internet2.org/news/detail/4865/"

[21] GÉANT, the pan-European research and education network that interconnects Europe's National Research and Education Networks (NRENs).
http://www.geant.net/Resources/Media_Library/Pages/Maps.aspx

[22] ULAKNET, the Academic network of Turkey http://www.ulakbim.gov.tr/ulaknet/

[23] PSiNET research group of Internet Interdisciplinary Institute, Open University of Catalonia (UOC-IN3) http://psinet.uoc.edu/index.php/en/

[24] Crockford, D. (2009). "Introducing JSON."

[25] NetworkX Python software package for the manipulation of complex networks http://networkx.github.io

[26] Tcpdump/Libpcap, http://www.tcpdump.org

[27] C. Argyropoulos, S. Mastorakis, G. Androulidakis, D. Kalogeras, V. Maglaris, “Network Virtualization in Software Defined Networking”, May 2014 (Submitted for publication).

9. Appendix

The software implementation is available in a public GitHub repository at the following url:

https://github.com/spirosmastorakis/FSP_Engine