UNIVERSITÀ DEGLI STUDI DI TRENTO

Dipartimento di Ingegneria e Scienza dell'Informazione

Ph.D. Dissertation

Enabling Novel Interactions between Applications and Software-Defined Networks

Ph.D. Candidate:
Antonio Marsico

Advisor:
Domenico Siracusa, Ph.D.

Abstract

Over the last few decades the pervasive diffusion of software has greatly simplified the introduction of new functionalities: updates that used to require complex and expensive re-engineering of physical devices can now be accomplished almost at the push of a button. In the context of telecommunications networks, recently modernized by the emergence of the Software-Defined Networking (SDN) paradigm, software has manifested in the form of self-contained *applications* driving the behavior of the *control plane* of the network. Such SDN controller applications can introduce profound changes and novel functionalities to large deployed networks without requiring downtime or any changes to deployed boxes, a revolutionary approach compared to current best practices, and which greatly simplifies, perhaps even enables, solving the challenges in the provisioning of network resources imposed by modern distributed business applications consuming a network's services (e.g., bank communication systems, smart cities, remote surgery, etc.).

This thesis studies three types of interaction between business applications, SDN controller applications and networks with the aim of optimizing the network response to a consumer's needs.

First, a novel interaction paradigm between SDN controller applications and networks is proposed in order to solve a potential configuration problem of SDN networks, which is caused by the limited memory capacity of SDN devices. An algorithm that offers a virtual memory to the network devices is designed and implemented in a SDN application. This interaction shows an increase of the amount of traffic that a SDN device can process in the case of memory overflows.

Second, an interaction between business applications and SDN networks shows how it is possible to reduce the blocking probability of service requests in application-centric networks. A negotiation scheme based on an Intent paradigm is presented. Business applications can request connectivity service, receive several alternative solutions from the network based on a degradation of requirements and provide a feedback.

Last, an interaction between business applications, SDN controller applications and networks is defined in order to increase the number of ad-hoc connectivity services offered by network operators to customers. Several service providers can implement a connectivity service in the form of SDN applications and offer them via a SDN App Store on top of a SDN network controller. The App Store demonstrates a lower overhead for the introduction of customized connectivity services.

Acknowledgments

I would like to thank my colleagues for their collaboration in projects and scientific discussions during the last four years. They helped me to find new ideas and learn new technical skills. A special thanks goes to my advisor Domenico, who helped me to grow professionally and personally. Finally, I would like to thank my family, in particular Marianna and Matteo, for providing me the motivation to continue in this scientific career.

Contents

List of Figures

Chapter 1

Introduction

In the last years, software has driven innovation in many different fields with a direct impact on society [1]. Cloud, Internet of Things, and Networking are important examples of how software has rapidly brought novel functionalities, in tight cooperation with hardware advancements.

A particular type of software, the self-contained application package, can be seen as a key enabler for efficiently introducing novel functionalities, even among different technological domains. Applications are computer programs designed to perform different types of functions and activities. An application can receive inputs from users, other applications or hardware, and elaborate an answer (i.e. output) through specific algorithms. They can exploit a modular architecture, which offers the possibility to develop applications as independent and interchangeable modules, and executed on top of an Operating System (OS), which provides a consistent, abstracted view of the underlying hardware. In this way, applications can be easily modified and updated with novel functions when required.

In the context of Software-Defined Networking, which foresees the decoupling between the *control* and *infrastructure* layers, the network can take advantage of several software characteristics, such as programmability, simplified development and modularity. In particular, the *control layer*, which represents the traffic forwarding intelligence of a network, has been subjected to a deep change towards an application-based architecture. Indeed, novel SDN network controllers offer features similar to OSs. They provide abstraction with respect to the infrastructure layer by offering many different Application Programming Interfaces (APIs), which can be exploited by external software modules to interact with network devices. These modules are defined as *SDN controller applications.* Every SDN application can provide specific functionalities to SDN networks, such as firewalling, load balancing, customized traffic routing, etc.

On top of the *control layer*, novel business applications for industry verticals, such as trading platforms for banks, e-commerce, video streaming services, etc., pose specific challenges on the network management and service provisioning. They generate a multitude of diverse traffic patterns, which are characterized by many different requirements, such as bandwidth, latency, availability, etc. These multitude of traffic patterns are usually ignored during the provisioning of a connectivity service. Thus, this implies that traffic with diverse requirements is eventually groomed in same network connections. For instance, a distributed database application, based on a strict consistency model, requires low traffic latency. If the network cannot provide this requirement, the database application may experience synchronization issues.

The main objective of this thesis is to study how business applications and SDN controller applications can improve the performance of SDN networks by means of interactions between them. Every interaction tackles a specific network problem and shows one possible solution. Three different type of interactions are analyzed:

- **Interaction between SDN controller applications and networks:** This first type of interaction aims at showing how the SDN applications can be used to modify the network behavior in the case of potential issues, such as misconfigurations. SDN applications receive network events, such as traffic statics, topology information, etc. and they perform actions into the network. In particular, this interaction shows how SDN applications can be exploited to mitigate the limited memory of SDN network devices. This issues may potentially cause delays in accessing distributed services, since a device with a full memory cannot accept anymore the installation of new forwarding rules [2].

- **Interaction between business applications and SDN networks:** Business applications can interact with the network by exploiting an *application-centric paradigm*, which proposes the provisioning of connectivity services based on application-specific requirements. An application can communicate to a SDN controller its requirements and, whether the network can satisfy all of them, the application request can be provisioned otherwise blocked. In order to reduce the request blocking probability, we propose a novel interaction between business applications and SDN networks, defined as the *application-aware service negotiation*. In the case of the network cannot provide all the requirements, it calculates a set of alternative solutions based on the *degradation* of requirements. The applications analyze the alternatives and provide a feedback to the network.

- **Interaction between business applications, SDN controller applications and networks:** This last type of interaction shows how business and SDN controller applications can enable a novel model for the selection of connectivity services. We propose a SDN App Store, which offers a multi-service selection model for business applications and users. Several providers can develop a customized version of a connectivity service (e.g., SD-WAN, VPN, etc.) in the form of SDN controller applications.

1.1 Structure of the thesis

The thesis structure is organized as follows:

Chapter 2 proposes a general description of two paradigms that are exploited in this thesis: the SDN and *application-centric* paradigms. In particular, the former section presents an overview of the SDN technologies, with a focus on the OpenFlow control protocol [3]. While the latter describes an approach for the provisioning of connectivity services based on application-specific requirements.

Chapter 3 analyzes the first form of interaction proposed in this thesis, the *interactions between SDN controller applications and networks.* In particular, this Chapter is based on the following joint work:

- A. Marsico, R. Doriguzzi-Corin e D. Siracusa. An effective swapping mechanism to overcome the memory limitation of SDN devices. In: 2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM), mag. 2017.

Chapter 4 describes the *interactions between business applications and SDN networks.* This Chapter collects contributions from the following joint work:

- A. Marsico, M. Savi, D. Siracusa, An Automated Service-downgrade Negotiation Scheme for Application-centric Networks. In press: 2018 The Optical Networking and Communication Conference & Exhibition (OFC).

Chapter 5 offers an overview on the *interactions between business applications, SDN controller applications and networks.* This Chapter is based on the following joint work:

- A. Marsico, M. Chamania, D. Siracusa, Empowering Users with Multi-service Selection: Towards an SDN App Store for Network Services. Submitted to IFIP Networking 2018.

Chapter 2

Background

This section provides an overview of the Software-Defined Networking (SDN) paradigm exploited in this thesis. In particular, we propose an overview of modern SDN controller architectures, which provides functionalities similar to computer OSs.

2.1 SDN generic architecture

Inside every network device, such as switches or routers, two different logical parts coexist: the *control plane* and the *data plane*. The *control plane* is the intelligence of the system. It is responsible for taking decisions on how packets should be forwarded between the different input/output ports. On the other hand, the *data plane* represents the hardware part of the device, where decision on packets from the *control plane* are physically applied (e.g. send the packet out of a specific hardware port).

The SDN paradigm proposes the decoupling between the *control plane* and the *data plane* of the network devices. The Open Networking Foundation (ONF) [4], a standardization body for open source networking, defines the general SDN architecture based on layers in order to guarantee interoperability between the different components through well-known APIs. It is

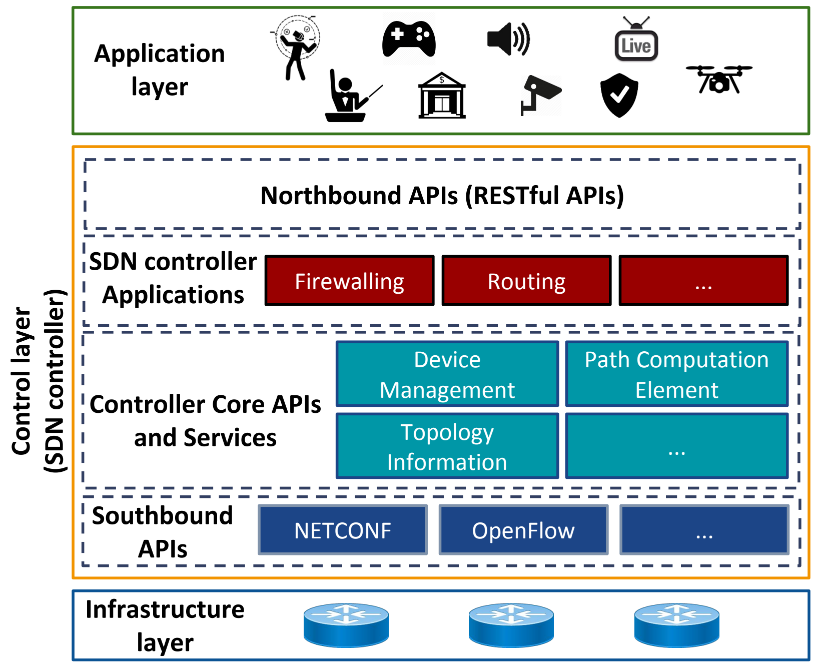


Figure 2.1: SDN General Architecture.

composed of three main parts: *Application layer*, *Control layer* and *Infrastructure layer* (Fig. 2.1).

The *Application layer* represents all the applications that exploit a SDN network infrastructure to transmit data traffic. In particular, this thesis considers novel business applications for industry verticals, such as live streaming video, security systems, on-line trading platforms, etc. The second layer, the *Control layer*, is the main part of the architecture, where all the management and provisioning decisions for a network are made. This layer is represented by *SDN network controllers*. They are compound software packages that configure networks' behavior by exploiting a centralized view and several Southbound APIs to configure network devices. Finally, the *Infrastrucure layer* represents all the physical network devices. They have limited intelligence and only apply the decisions coming from a SDN controller.

2.1.1 Modern SDN network controllers

In the last years, SDN network controllers have evolved to provide abstraction with respect to the *Infrastructure layer* similar to computer OSs. The configurations that should be performed on a device, the network events, such as failures, topology changes, etc., can be managed and processed by exploiting several APIs of a SDN controller, which define a generic and high level language to interact with a network infrastructure.

In general, SDN controller architectures are designed based on a layered structure. Higher layers provide higher abstraction with respect to a network infrastructure, while the bottom layers implement the specific communication protocols required to interact with a network infrastructure. Starting from the top, SDN controllers provide *Northbound APIs* to let applications, such as external software for network provisioning, to gather network information and to interact with them. Usually, these APIs are implemented by exploiting HTTP RESTful APIs, such as Swagger [5], an editor for defining REST APIs and translating them to a programming language. In order to increase the abstraction of Northbound APIs, a new paradigm, called *Intent-based networking* [6], has been proposed. It offers a higher level of abstraction than specific network commands (e.g., send a packet out of port 2). This paradigm can be used to express network requests in the form of high level intentions, which specify *what* it is required by a network rather than *how* this should be accomplished. For instance, an application can request "I want to connect host A to host B with 10 Gbps of bandwidth". Then, a SDN controller is responsible to translate this *intent* into commands understandable by network devices.

Beyond the Northbound APIs, a SDN controller can host several *SDN controller applications*, each one providing a particular network functionality. For example, one application can offer traffic forwarding, another one firewalling, etc. SDN applications are based on the modular architecture of self-contained software applications. They can be installed, updated and uninstalled as computer applications without requiring major modifications to a SDN controller.

SDN applications exploit the *Controller Core APIs and Services* to interact with a network infrastructure. Every API represents a particular network service, which offers a set of functionalities to SDN controller applications. The most important generic services that a SDN controller offers are the *Device Management*, the *Path Computation Element* and the *Topology Information.* The first service provides all the operations required to interact with a network device (e.g. installation of forwarding rules, forwarding table status, etc.). The second service, the *Path Computation Element*, can be exploited to calculate the shortest paths between different endpoints in a network. Finally, the *Topology Information* service exploits the centralized view of SDN controllers to provide all the data about the network topology, such as devices, links, etc., and the network events, such as failures.

The bottom layer is represented by the Southbound APIs. These APIs offer a communication interface between a SDN controller and network devices. Every API implements a specific communication protocol and the commands required to interact with a network device. Since the SDN environment is composed by many different communication protocols, the modern SDN controllers implement the APIs as software modules, which can be added or removed at runtime, like the SDN applications.

On of the most well-known Southbound API for SDN is *OpenFlow* [3]. It is an API that uses an encrypted connection, based on *Secure Socket Layer/Transport Layer Secure* (SSL/TLS), to exchange messages between the SDN controller and every network device. The most important command is the `FLOW_MOD`, which defines the installation, the modification, and the deletion of forwarding rules in a network device. A forwarding rule (i.e., a Flow Rule) is composed of three main parts: the *Flow Match*, the *Actions* and the *Stats*. The *Flow Match* represents the *matching* structure, thus which are the header field (and their values) of a packet where the *Actions* should be applied. The concept of *Flow* is borrowed from the TCP protocol, which creates virtual and reliable connections between hosts for a data transfer in the network. *OpenFlow* extends and generalizes this concept to all the traffic inside the network. A traffic *Flow* is a set of packets between a source and a destination that shares the same characteristics with respect to the transport connection (i.e. TCP/UDP), the carried information, etc. For example, a *Flow* can be defined by a set of packets sharing the same IP source or destination and TCP/UDP source and destination ports. The *Action* field represents what the network device has to perform on the data traffic (e.g. exit from port 2, drop a packet, etc.). The *Stats* are the statistics, i.e., number of packets that have matched a particular *Flow*.

Another important Southbound API is NETCONF [7]. Different from OpenFlow, it defines a set of operations to configure network devices based on the YANG data model [8]. The NETCONF protocol can be exploited to edit device configurations and receive events from network devices. In particular, this protocol implements transactions based on the ACID properties: *Atomicy*, *Consistency*, *Independence* and *Durability*. In this way, every transaction provides a predictable outcome. Indeed, this language is mainly exploited into devices for transport networks, such as routers.

In the state of the art, the most significant examples of SDN controllers providing high levels of abstractions and modularity are ONOS [9] and OpenDayLight [10]. They are open source controllers which are supported by important vendors, such as Cisco and Huawei.

Chapter 3

Interactions between SDN controller applications and networks

In this chapter we analyze the effect of the interaction between SDN controller applications and networks. In particular, we show how this interaction can overcome the memory limitation of SDN network devices that are usually based on Ternary Content Addressable Memory (TCAM). Unfortunately, in many network scenarios, TCAMs can quickly fill due to their limited memory size, thus preventing the installation of new flow-rules and leading to inefficient traffic forwarding. This issue has already been addressed in computer programming, where *Virtual Memory* is offered to applications to mimic a much larger physical memory, by swapping memory pages to disk.

We propose a memory swapping mechanism for SDN controllers, which gives SDN applications the illusion of unlimited memory space in the forwarding devices, without requiring any hardware modification or changes in the control protocol. This algorithm is implemented in the Memory Management System (MMS), a SDN controller application for optimizing the memory usage of network devices. We discuss the memory swapping mechanism design, its implementation; furthermore, we prove its quality using real traffic traces, demonstrating lower TCAM memory utilization and potentially increased network performance in terms of end-to-end throughput. A prototype of the MMS is available for testing as an open source project on GitHub [11].

3.1 A SDN application for the Memory Management of SDN devices

In SDN, network applications rely on SDN controllers to handle the incoming traffic, via instructions that are executed by network devices. This piece of information is commonly saved in the memory of the devices and it is called *flow entry*. In the OpenFlow protocol, a widely used standard for SDN [3], flow entries can be of variable length/size according to the specificity of the action that is requested to the networking device (e.g. L3 routing, L4 firewalling, etc.). With respect to traditional L2 and L3 forwarding, OpenFlow allows a much finer control of the network traffic, but, at the same time, it requires more memory space for each flow entry. For instance, recent versions of OpenFlow require up to 773 bits for each entry[1], while 60 bits are sufficient to identify a flow for L2 forwarding (destination MAC address plus VLAN identifier).

Modern network switches are usually equipped with two different types of memory in which forwarding instructions are stored: *Binary* and *Ternary* Content Addressable Memory (BCAM and TCAM). Both memory types can do lookups in one clock cycle and in parallel fashion, therefore they are very efficient when matching the incoming traffic with the forwarding rules they

[1]OpenFlow Switch specification v1.5.1 [12], including optional fields.

store. However, BCAMs only provide binary lookups, so they can store only information represented by 0 or 1 bits. They are most useful for building tables that search on exact-matches such as MAC address tables. On the other hand, TCAMs can store three bit states (0, 1, and "don't care") and are the most commonly used mainly because they can store IP prefixes. TCAMs work very well in conjunction with OpenFlow, where the flow table entries foresee a wildcard bit, used to inform the switch to ignore the value of the specified header field. Unfortunately, TCAMs are very expensive, power hungry and have a considerable footprint size with respect to the number of supported flow entries. Therefore, vendors tend to install TCAMs with very limited capacity, which can quickly get full, leading to inefficient forwarding operations.

This problem is partially mitigated in the switches' firmware, which combine Random Access Memories (SRAM, DRAM, etc.) and TCAM to build the flow table: RAM for exact-match entries, the TCAM for wildcard entries [13]. However, this strategy is not suitable for SDN-enabled switches, where exact-match rules are rarely used.

Starting from specification 1.4.0, OpenFlow introduces two mechanisms to allow the SDN developer to handle the lack of memory space available on network devices: *eviction* and *vacancy events*. The first one enables the switch to automatically delete the flow entries with lower importance. The degree of importance of each single flow entry is set by the SDN applications. The second mechanism enables the controller to get an early warning based on a capacity threshold set by the SDN application. However, such approaches force the SDN developers (and the SDN applications) to take care of the memory utilization.

In literature, several works have been proposed to tackle the problem of the limited TCAM memory space, all with very different approaches. However, as reported in Section 3.2, they impose significant constraints on the network architecture, changes of the switches' software or hardware, modifications of the control protocol, or the usage of exact-match rules in BCAMs to save TCAM space.

In our work, we advocate that SDN controllers shall grant their applications a reliable access to network devices' memory. To this purpose, we propose a Memory Management System (MMS) for SDN that aims at improving the TCAM usage and, consequently, the robustness of the network. Compared to the existing approaches, the MMS transparently optimizes the usage of the available memory by exploiting two different functions: (i) the *memory de-allocation* and the *memory swapping.*

The first one offers automatically removal of the flow entries installed by applications that are no longer running. We found that most of network controllers do not automatically remove the flow entries associated to a deactivated application [14].

The second one, the *memory swapping*, recalls a technique used in computer OSs to exchange *memory pages* between the fast but limited in size Random Access Memory (RAM) and the slow but much larger hard disk. Especially in the past, when the RAM capacity was often not sufficient for multi-tasking environments, *memory swapping* was the only way to make the applications seamlessly run even in low memory space conditions. In this context, the OS moves (*swaps out*) the least used memory pages to a pre-configured space on the hard disk called *paging file*, *swap file* or *swap partition*, and makes the memory available to those applications that need it in that particular moment. *Swap in* is the opposite operation executed by the OS to restore the memory pages back to the RAM, when they are required by the applications. Even though the swapping technique permits software application to use more memory than the physically available, moving memory pages back and forth from RAM to hard disk generally slows down the system execution.

In the SDN context, the fast/size-limited memory is represented by the B/TCAMs of the network elements and the slow/large memory can be implemented as a database maintained by the SDN controller in the RAM memory of the computer where it is running. Our rationale is to maintain the most matched flow rules in the switches' memory and move the other rules to the database, in a completely transparent way for the network applications running on top of the controller.

We can summarize the contributions of this work as follows:

- **Design** of a platform-independent memory swapping mechanism for SDN controllers, as part of a more powerful Memory Management System, whose aim is to optimize the usage of switches' memory, transparently to SDN network applications.

- **Implementation** of the swapping mechanism for the ONOS platform [9]. ONOS is an advanced SDN controller that provides the required services and APIs for the implementation of the swapping mechanism, as identified in [14]. Unlike other controllers like Ryu [15], where the table full condition must be handled by the SDN applications, ONOS hides these low-level details and holds the flow rules that cannot be installed in a *pending add* state until there is enough memory space in the switches. We compare the ONOS mechanism against our memory swapping.

- **Validation** of the memory swapping mechanism using real traffic traces. We demonstrate that our prototype optimizes the usage of switches' TCAM memory in terms of free space available for new flow wildcard entries and, therefore, it improves the performance of the network in terms of throughput.

3.2 Related Work

There is a vast array of work related to the TCAM memory utilization and optimization in SDN. We classify the most relevant techniques in five categories:

Flow rule Caching: Like the MMS, CacheFlow [16] is a *Virtual Memory* mechanism that gives SDN applications the illusion of an arbitrarily large switch memory. In CacheFlow, the additional memory is provided by software SDN switches which are attached to the datapath and implemented as software agents in commodity server-class hardware or directly in the hardware switches. However, while the latter approach requires modifications of the switches' firmware, the first imposes strict constraints to the network, as CacheMaster, the component that hosts the software switches, must communicate with the hardware switches via either single-hop Layer-1 connectivity or Layer-2 tunnels. Moreover, CacheFlow does not tackle the *table full* case, i.e. the critical situation when the TCAM of one or more hardware switches is full, which the MMS covers by design.

Exact match rules: Authors of DomainFlow [17] leverage exact match rules to overcome the limited capacities of the TCAM memories, as exact match rules are saved in binary memories (like briefly explained in the Introduction). DomainFlow keys ideas are: (i) use exact matching where possible and (ii) split the network into sections to allow exact matches to be used more often. DevoFlow [18] propose a modification of the OpenFlow model by introducing a new action type in form of the *clone* flag. If the flag is set, the switch clones wildcard rules with exact match rules that are saved in the exact match flow table, i.e. saving TCAM memory space. However, DevoFlow imposes modifications of both OpenFlow protocol and switches' firmware and, like DomainFlow, it does not address the limitations imposed by the size of the TCAM.

Idle timeouts: SmartTime [19] uses adaptive heuristic to compute idle timeouts for the flow rules which results in optimal utilization of the TCAM memory. SmartTime pro-actively evicts flow rules with finite timeout in a random manner when the TCAM utilization crosses a pre-defined threshold. However, SmartTime does not cover the common case in which pro-active rules with infinite timeouts are used to control the traffic (e.g. cloud orchestrators like Open-Stack/Neutron pro-actively install all the rules to create the virtual network topologies).

Flow table compression: Tag-in-Tag [20] proposes a technique to replace the OpenFlow entries stored in the TCAM memories with two layers of simpler and shorter tags. However, Tag-in-Tag requires changes in the packet header and in the switches' firmware to correctly handle the tags. Authors of [21] leverage OpenFlow's wildcards to reduce the memory utilization in the specific scenario of Border Gateway Protocol (BGP) routing tables.

Flow rule placement optimization: Several recent works propose algorithms or mechanisms for an optimal rule placement across the network, with the aim of saving TCAM memory space ([22], [23], [24], [25]). However, none of them tackles the *table full* case.

3.3 Memory swapping design

The memory swapping mechanism automatically frees the TCAM memory of the switches from the least used wildcard flow entries by temporarily moving them to a slower memory. The swapping process consists of two different operations. The first, called *swap out*, is performed when the SDN controller detects that the flow tables are full. In this case the mechanism swaps out the least used wildcard rules to free up TCAM memory space for new entries. Vice-versa, the *swap in* operation restores the swapped out rules when they are needed again by the network device to forward the traffic.

3.3.1 Swap out

By default, the *swap out* operation is executed when the SDN controller detects that one or more switches are operating in *table full* condition. In case of OpenFlow-enabled switches, that condition is notified to the SDN controller via a `TABLE_FULL` error message. From OpenFlow 1.4.0 or higher, the swapping mechanism can be also configured to react to `TABLE_STATUS` events with reason `VACANCY_DOWN`, meaning that the remaining space in the flow table has decreased to less than a pre-defined threshold.

Unlike a traditional computer OS virtual memory system, where a page is either allowed to be swapped out or not, the removal of a flow rule from the TCAM may change the semantic of the network. This problem may happen because high priority rules may be dependent on low priority ones. As a trivial example, consider a rule set where the default rule drops any packet and any allowed communication is handled with higher priority rules. When the MMS swaps out an entry of this rule set because of resource pressure, any further packet matching this rule will be then caught by the default rule and dropped, which is an undesirable result. So, whenever we consider a rule as a candidate to swap out, we have to build the (transitive) set of (lower priority) rules which are also affected. The analysis of the rule-dependency problem, which is outside the scope of this work, has already been tackled by other works such as CacheFlow [16].
Authors of CacheFlow propose an algorithm to incrementally analyze and maintaining rule dependencies. In such a work, a dependency between a child rule C and a parent rule R is defined as follows: if C is removed from the flow table, packets that are supposed to hit C will hit rule P.
The current design of the MMS leverages on CacheFlow's algorithms to correctly compute the dependencies between rules and, consequently, to *swap out/in* the wildcard entries between TCAM and MMS databases. Please note that, the memory swapping mechanism focuses on wildcard entries to

free TCAM memory space. Thus, it does not take into consideration the rarely used exact match rules (which are likely stored in other memories such as DRAM or SRAM), to avoid the risk freeing the wrong memory.

```
1  function installFlowRule(flow_rule fr):
2    if check_wildcards(fr) is True:
3      dep_chain=compute_dependencies(fr,flow_db)
4      fr.addDependencies(dep_chain)
5      flow_db.add(fr)
6      swap_chain=get_dependencies(fr,swap_db)
7
8  while install_rules(fr,swap_chain) is False:
9    swapOut()
10
11 function swapOut():
12   least_used_fr=get_least_used_fr(flow_db,quota)
13     for fr in least_used_fr:
14       dep_chain=get_dependencies(fr,flow_db)
15         for dep_fr in dep_chain:
16           if dep_fr.timeout is 0:
17             swap_db.add(dep_fr)
18             remove dep_fr
```

Listing 3.1: Flow rule installation process.

The pseudo-code in Listing 3.1 illustrates how the *swap out* process is executed. We assume that the SDN/OpenFlow switch is set up to generate *new flow* messages (e.g. an OpenFlow `PACKET_IN`), in the case of an unknown destination of a packet, and SDN applications may request the installation of flow rules both reactively and proactively (i.e. with/without a new flow message).

When an SDN application requests for a flow rule installation, the MMS intercepts the rule, it checks whether the rule contains any wildcard and computes the dependencies with the other wildcard rules already installed in the network and saved in the `flow_db` (lines 2-4 in the Listing). Then, the rule is stored in the `flow_db` database (line 5) along with the dependency chain. Before installing the flow rule, we retrieve the dependency chain of the new rule from the `swap_db` (which stores the rules that have been previously swapped out), to avoid network inconsistencies (line 6). The actual installation of the new rule onto the device is performed at line 8. If the operation returns either a `TABLE_FULL` error or a `VACANCY_DOWN` event, the `swapOut` function is called to free up some space in the switch flow table (line 9) and then the rule is finally installed.

The `swapOut` function gets the least used flow entries from the `flow_db` (line 12), based on traffic statistics collected periodically (as described below in this section). `quota` is the percentage of the installed rules to be swapped out. By default, its value is 20% for all the switches, but this threshold can be tuned dynamically based on the performance of the network. Swapped out flow entries are removed from the TCAM of the devices and from the `flow_db`. Entries with infinite timeout are saved in the `swap_db` (line 17) and automatically restored by the MMS when necessary. Entries with finite timeout are just dropped, based on the assumption that network applications can autonomously restore them, exactly how they would do if those flow entries were naturally expired due to the lack of matching traffic.

A network application may need to uninstall a flow rule. In this case, the MMS automatically deletes the corresponding entry from the `flow_db` and from the `swap_db` (if necessary) and updates the dependency chains of the child entries in those databases.

The ranking of the least used rules is determined by periodically collecting traffic statistics, such as packet and byte counters, for each wildcard rule. For each rule, the complete history of the collected samples is kept in the `flow_db` until the rule is deleted. The classification of the rules is computed with the Exponential Weighted Moving Average (EWMA) algorithm [26], which weights the statistics in geometrically decreasing order so that the most recent samples are weighted more than the oldest samples. This approach avoids erroneous classifications where, for instance, a flow entry periodically matched by a micro-flow has more chances to be swapped out than an entry matched by an elephant flow far in the past, i.e. no more active.

3.3.2 Swap in

Swapped out flow rules are automatically re-installed by the MMS onto the network when the switches need them again to forward the traffic, transparently to the SDN applications running atop the SDN controller.
The mechanism is fairly simple: when a switch does not find a match for an incoming flow inside its flow tables, it sends a *new flow* message (e.g. an OpenFlow `PACKET_IN`) to the SDN controller. The MMS intercepts the message before it arrives to the applications and checks whether the flow matches any of the swapped out rules in the `swap_db` database (line 2 in Listing 3.2). If the MMS finds a match, it automatically reinstalls the rule along with its dependency chain into the switch's memory (lines 3-5), otherwise the *new flow* message is released to the other processes of the SDN controller, which eventually relays it to all the listening applications.

```
1 function swapIn(packet pkt):
2   fr=swap_db.get_rule(pkt)
3   if fr is True:
4     swap_chain=get_dependencies(fr,swap_db)
5   install_rules(fr, swap_chain)
```

Listing 3.2: Swap in process.

3.4 Software architecture

In [14] we presented the concepts behind the MMS and we listed the requirements for its implementation as a component for a generic SDN controller.

Specifically, the MMS requires a number of services and interfaces to interact with the network devices and to accomplish the memory management operations. Since not all the SDN controllers meet the requirements, we started our development work by implementing the *memory deallocation* function for ONOS [27]. In this section, we recall the building blocks of the MMS architecture and we map them into ONOS with specific focus on the requirements for the *memory swapping*.

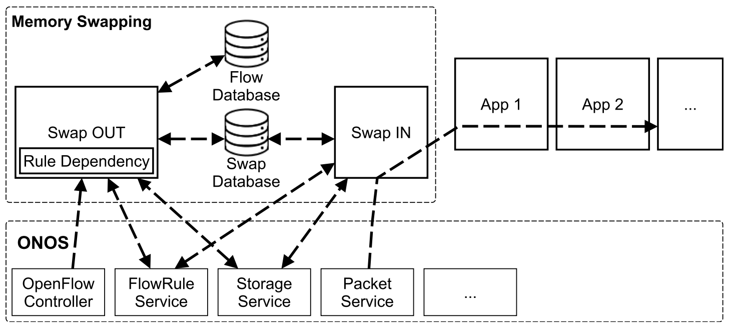


Figure 3.1: The memory swapping in the context of the ONOS platform.

3.4.1 Interaction with the controller

The *memory swapping* requires read/write access to the flow tables of the switches. It must keep track of all the flow entries installed in the network along with their statistic counters. Moreover, it must be able to intercept some events generated by the network, such as the `TABLE_FULL` error and the `TABLE_STATUS` event with reason `VACANCY_DOWN`, used to trigger the *swap out* process, and the notification of new flows (i.e., `PACKET_IN` in OpenFlow), used by the *swap in* process to re-install any previously swapped out entry matching the new flow with all its dependencies (as explained in Section 3.3.2).

As shown in Figure 3.1, in ONOS such functions are accomplished by four different interfaces called: *OpenFlowController*, *FlowRuleservice*, *StorageService* and *PacketService*. Hereafter, a description is provided of such interfaces and how they are used by the memory swapping mechanism.

FlowRuleService [28]. Interface for installing/removing flow rules into/from the network and for obtaining updates on those already installed. The MMS is also registered as a listener to this interface to: (i) intercept all the flow rules installed by the SDN applications and (ii) get the statistic counters of the installed flow entries, as soon as such statistics are made available by ONOS which collects them every 5 seconds. This information is used by the memory swapping mechanism to recognize the least matched entries which are swapped out in case of full TCAM.

OpenFlowController [29]. Abstraction of the OpenFlow controller. It is used for obtaining OpenFlow devices, for sending OpenFlow messages to them and to register/unregister listeners on OpenFlow events. Specifically, in the current version the MMS registers as a listener to this interface to get the `TABLE_FULL` error message. We plan to add the support for `TABLE_-STATUS` events in the next releases.

StorageService [30]. ONOS is a distributed SDN controller platform. An ONOS cluster comprises one or more ONOS instances, running the same set of modules and sharing network state with each other. In that respect, MMS internal databases are based on the *StorageService* interface which ensures a consistent state of the databases across all the instances of a ONOS cluster.

PacketService [31]. Service for intercepting the control messages generated by the switches in case of table miss events. As part of the registration process to this service, listeners (SDN applications as well as the MMS) specify a priority value which determines the order for processing the event. Lowest is the value, earliest the listener receives the message. The MMS registers with priority value 0 (the lowest), as required for the implementation of the *swap in* function.

3.4.2 Building blocks

Flow Database: The MMS implements an internal *Flow Database* to replicate the information contained in the flow tables of network devices. To do so, the MMS is registered as a listener to the *FlowRuleService* to intercept and collect the new rules installed by the network applications. The database is implemented using the *EventuallyConsistentMap* [32] distributed primitive, a data structure provided by ONOS *StorageService* which provides high read/write performance. The structure of the data stored in the database extends the ONOS *FlowRule* to contain: (i) current statistics counters of each flows rule, (ii) the whole history of statistic counters and (iii) the list of parent rules, based on the computation of the CacheFlow algorithm [16].

Swap Database: It is implemented as an *EventuallyConsistentMap* containing all the swapped out flow rules. In the first prototypes, the *Swap Database* was obtained with just a flag in the *Flow Database* indicating whether the flow rule was swapped out from the switches' memory. However, as the number of swapped out rules is low compared to the total amount of entries in the *Flow Database*, we realized that a dedicated database for the swapped out rules was a better idea to minimize the lookup time and to reduce the latency introduced by the *swap in* process when inspecting the *new flow* messages.

Blocks **Swap OUT** and **Swap IN** in Figure 3.1 represent the two main processes of the memory swapping mechanism.
The *swap out* process is in charge of freeing the switch's TCAM by moving the least matched flow entries to the RAM memory of the machine where the SDN controller is running. This process is configured to react to `TABLE_FULL` errors received via the *OpenFlowController* interface. In our ongoing work, we plan to add support for `VACANCY_DOWN` table status notifications, which will allow the MMS to get an early warning and to execute the swapping process before getting the table full.

The *swap out* process is divided into the following steps: (i) the MMS retrieves from the *Flow Database* all the wildcard entries associated to the switch that generated the `TABLE_FULL` alarm, (ii) the flow entries are sorted based on the average number of matching flows as computed by the EWMA algorithm, (iii) finally the least matched rules are removed from the TCAM with all their dependent rules. The copies of such rules maintained in the *Flow Database* are moved to the *Swap Database*.

Based on our experiments (cf. Section 3.5), we swap out the 20% of the whole TCAM content. However, this value can be tuned switch by switch at runtime based on the performance of the network, i.e. it should be increased in case of frequent `TABLE_FULL` events or decreased in case of too many rules re-installed into the network by the *swap in* process after being swapped out.

3.5 Performance Evaluation

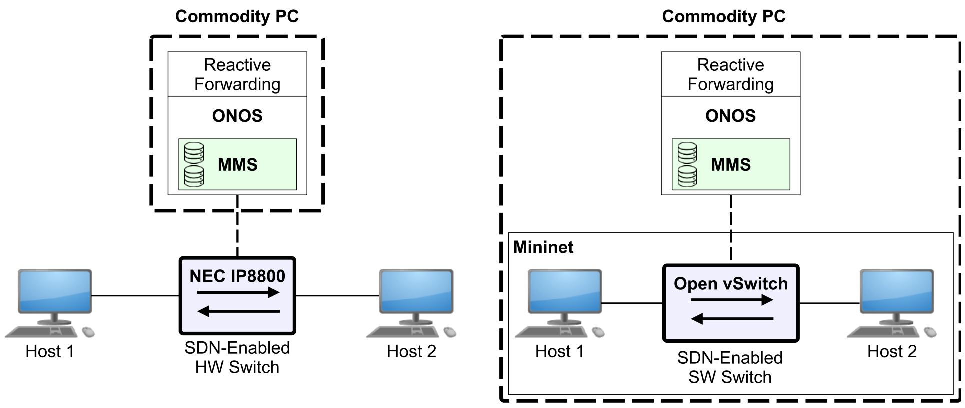
We evaluate the memory swapping mechanism by considering two metrics: (i) average end-to-end throughput and (ii) TCAM space available for new flow entries. The effectiveness of the proposed approach is measured by comparing the results observed with and without the memory swapping using flow table of different sizes.

3.5.1 Test methodology

Experimental setup. For the experiments, we use both hardware and software OpenFlow-enabled switches. In the first case we use the NEC IP8800 [33] (Figure 3.2a), while in the second, Mininet [34] and Open vSwitch (OVS) [35] switches (Figure 3.2b).

We run ONOS and the MMS on a commodity PC equipped with a Intel i7-5600U quad-core CPU running at 2.60GHz and 16GB of DDR3 memory working at 1600Mhz. This machine is connected to the NEC switch via Gigabit Ethernet for the OpenFlow control channel. Two physical hosts are connected to the switch via Gigabit Ethernet to inject network traffic during the evaluation. For the test with software switches, the commodity PC also hosts Mininet configured with a single OVS-based switch and two virtual hosts attached to it.

We use publicly available SMTP and HTTP traffic traces from [36]. The first one produces 23 new flows per second on average when considering Layer 3 fields, while the second trace produces 65 new flows per second on average.



(a) NEC IP8800 setup. (b) Open vSwitch setup.

Figure 3.2: Experiment setups for the evaluation.

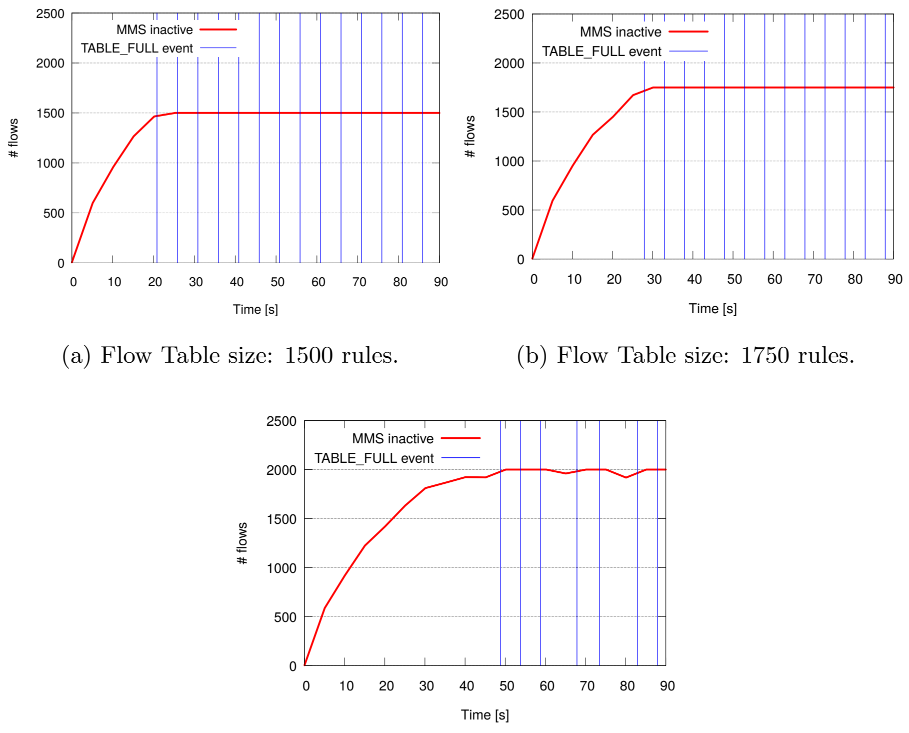
Context. The result of the evaluation depends on three main aspects: (i) the flow table size of the switch, (ii) the flow rate of the traffic trace, i.e., number of new flows per second, and (iii) how fast the corresponding flow entries expire. Thus, we expect our memory swapping mechanism to be more effective with small flow tables and high rate of flows controlled using flow entries with high expiry timeouts.

To stress the memory swapping, we started with the HTTP trace, but unfortunately we experienced many disconnections of the OpenFlow channel due to CPU overload of our NEC switch. Due to this, we were forced to

limit the evaluation with the hardware switch to just the SMTP trace.
The TCAM of our NEC IP8800 can host up to 1500 flow entries, but we are interested to understand the effectiveness of the memory swapping mechanism when varying the size of the flow table. For this reason, we move to the OVS switch, since it can be set up to have different flow table sizes. We configured it from 1500 entries (like our NEC), to 2000 entries like a HP 8200/5400 [37]. In this case, we use the HTTP traffic trace.

Methodology. We compare the memory swapping implementation for ONOS described in Section 3.4 with the default ONOS memory management system. By default, ONOS holds in *pending add* state the flow rules that cannot be installed until there is enough memory space in the switches. We demonstrate that our mechanism is better in terms of (i) memory space available in the switches for the new entries, and (ii) performance of the network measured in terms of end-to-end throughput.
The experiments are executed under the following conditions:

- At time 0, one of the hosts (e.g., `Host1` in Figure 3.2) starts injecting the traffic trace into the switch (either hardware or software). At this point in time, the flow table of the switch is empty.
- The switch is controlled by ONOS via the *ReactiveForwarding* application [38]. This application reactively installs a flow entry in the switch for each incoming new flow.
- The application is configured to generate wildcard flow entries with the only IP source and destination addresses specified. The application also randomly assigns either infinite or 10 seconds idle timeouts to the flow entries.
- Dependencies between flow table entries are synthetically generated based on pairs of random IP subnet masks and priorities. Thus, we



(a) Flow Table size: 1500 rules. (b) Flow Table size: 1750 rules.

(c) Flow Table size: 2000 rules.

Figure 3.3: Number of installed rules without memory swapping at different flow table sizes.

configured the forwarding application to randomly apply different levels of priorities combined with different submasks by following the longest prefix match principle, where the longer the subnet mask, the higher the priority.

3.5.2 Results and discussion

When a network device runs out of memory, it starts refusing the installation of new forwarding rules. Eventually, the buffer of the network device, which holds the packets waiting for forwarding instructions, becomes full and starts dropping the buffered packets. This will lead to a degradation of the user's quality of the experience in terms of low throughput and high delays For example, when accessing an online service provided by a data center, such as on-demand video streaming, the service performance may drastically drop down [2].

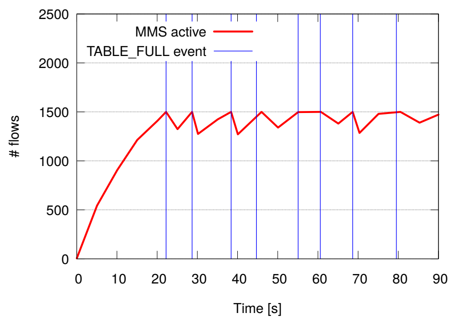
One of the possible criteria to evaluate the loss of performance is the end-to-end throughput measured at the destination host (e.g., `Host2` in Figure 3.2). In this way, we demonstrate the effects of a device with a full flow table on the network traffic. Table 3.1 summarizes the results obtained with the NEC hardware switch using the SMTP traffic trace (23 new flows per second), and with the OVS software switch using the HTTP trace (65 new flows per second).

Table 3.1: Average throughput (90 sec. test).

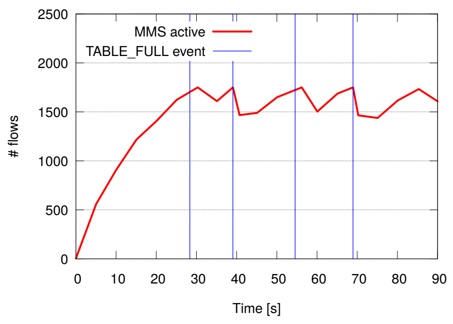
Flow Table Size	**Traffic trace**	**Throughput MMS Active [Kbps]**	**Throughput MMS Inactive [Kbps]**
1500 (NEC)	SMTP	19.85	15.81
1500 (OVS)	HTTP	80.17	58.97
1750 (OVS)	HTTP	85.63	72.38
2000 (OVS)	HTTP	86.54	81.13

Specifically, when using the memory swapping we measure a throughput increase from 15.81 Kbps to 19.85 Kbps (21% on average) with the NEC switch and the SMTP trace. When using the software switch and the HTTP trace, we observe different performance depending on the size of the flow table. We measure throughput increases of 26%, 15% and 6% with flow table sizes of 1500, 1750 and 2000 flow entries respectively.

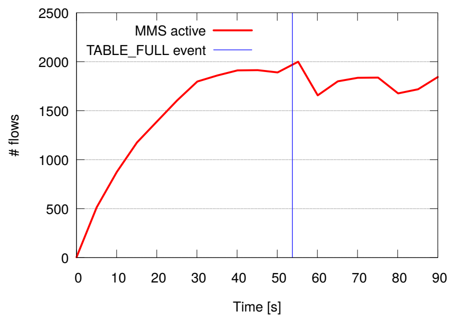
Results obtained with the OVS-based software switch are also reported in Figures 3.3, 3.4 and 3.5. Vertical blue lines in Figures 3.3 and 3.4 represent the `TABLE_FULL` error messages sent by the switch operating in full table condition to ONOS, when ONOS tries to install the pending rules. By default, this installation process is automatically performed every 5 seconds if the *pending add* queue is not empty. Please note that, for the sake of



(a) Flow Table size: 1500 rules.



(b) Flow Table size: 1750 rules.

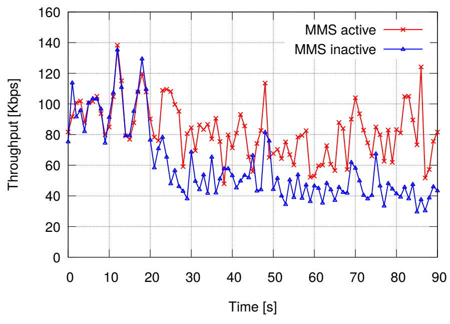


(c) Flow Table size: 2000 rules.

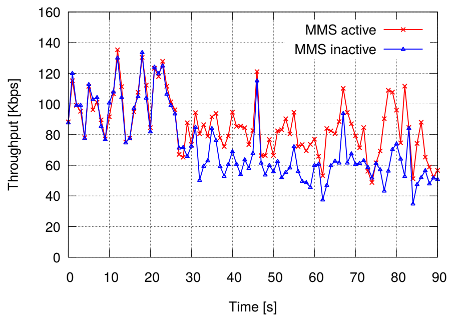
Figure 3.4: Number of installed rules with memory swapping at different flow table sizes.

readability, we represent only one error message for each attempt.

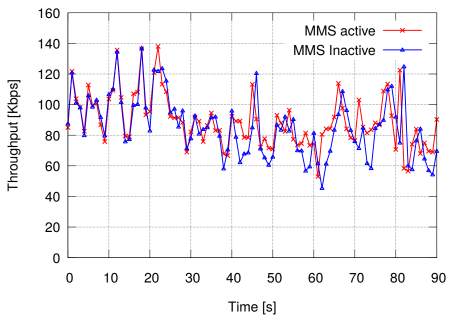
Without the memory swapping, the flow table is constantly full and every time ONOS tries to empty the *pending add* queue, it gets a `TABLE_FULL` error (Figures 3.3a and 3.3b). The problem is partially mitigated when the software switch is configured with a larger flow table (Figure 3.3c). In this case, the number of flow rules which are evicted for expiring timeout is often sufficient to make space for the rules waiting in the *pending add* queue. Conversely, the memory swapping process frees the 20% of the TCAM capacity in reaction to `TABLE_FULL` errors (Figure 3.4). More precisely, the least matched entries are moved to the *Swap Database* and the free space in the TCAM is used by ONOS to install the rules in *pending add* state and, possibly, the new rules the *ReactiveForwarding* application generates to con-



(a) Flow Table size: 1500 rules.



(b) Flow Table size: 1750 rules.



(c) Flow Table size: 2000 rules.

Figure 3.5: Throughput comparison with and without memory swapping at different flow table sizes.

trol new flows. The benefits of the memory swapping are demonstrated by the reduced number of `TABLE_FULL` errors, as shown in Figure 3.4, and by the increased performance in terms of end-to-end throughput at different flow table sizes, as shown in Figure 3.5 and also summarized in Table 3.1.

Finally, we observe that the effectiveness of our mechanism increases when the ratio between the flow rate and the flow table size increases. In this respect, recall that SDN platforms like ONOS are designed to scale to large networks, where the flow arrival at the switches can be in the order of thousands flows/sec [18]. Thus, we conclude that a memory management mechanism like the one presented and validated in this section, can help such SDN environments to operate efficiently without requiring network devices mounting large, expensive and power hungry TCAMs.

Chapter 4

Interactions between business applications and SDN networks

Novel business applications for industry verticals with specific and stringent service requirements are expected to represent one of the key challenges for the future transport networks, forcing network operators to design multiple services to cope with these requirements. By exploiting an application-centric paradigm, the operators can provide connectivity service differentiation to business applications. However, maintaining all the requirements is not always possible and application requests can experience higher service blocking probability.

This chapter discusses a novel interaction between business application and SDN networks. We propose a model for the *negotiation* of application-aware connectivity services by extending an existing application-aware provisioning algorithm. An application can communicate its connectivity requirements and, whether the network is not able to respect all of them, can offer several alternative solutions based on a *degradation* of service requirements. The application can provide a feedback to the network, based on an autonomous interaction. We prove the effectiveness of our approach by showing how it leads to a lower blocking probability of service requests with respect to the case no negotiation scheme is adopted, without in turn causing significant service degradation to the applications.

4.1 An Intent-based Negotiation Scheme for Application-centric Networks

Today's transport networks are composed by a three layer architecture in which on top sits the *application layer*, a *grooming layer* and the *transport layer* (Figure 4.1). The *application layer* is composed by a multitude of diverse applications, characterized by their own requirements in terms of bandwidth, latency, availability, etc. These parameters define the type of connectivity service that an application requires from the network. For instance, distributed databases running between several data centers require low latency, guaranteed bandwidth and minimum packet losses in order to avoid synchronization issues [39]. The *grooming layer*, usually IP/MPLS, aggregates all the application flows into large optical connections at the *transport layer*.

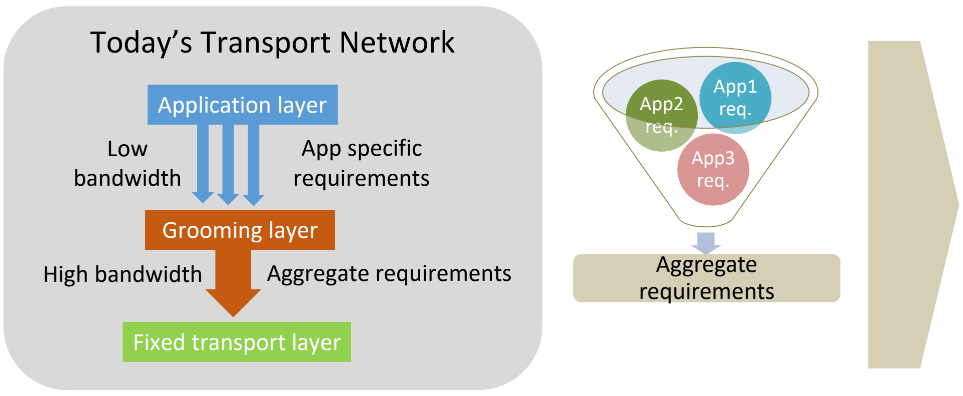


Figure 4.1: Current network traffic aggregation.

However, the existence of a *grooming layer* implies that applications with different requirements are always subjected to the same treatment in the transport layer. All the applications are aggregated in the same optical connections, thus, this implies an inefficient use of network resources and a static model for traffic treatment. Indeed, the network traffic generated by applications with diverse requirements is always groomed in the same large optical connection pipes.

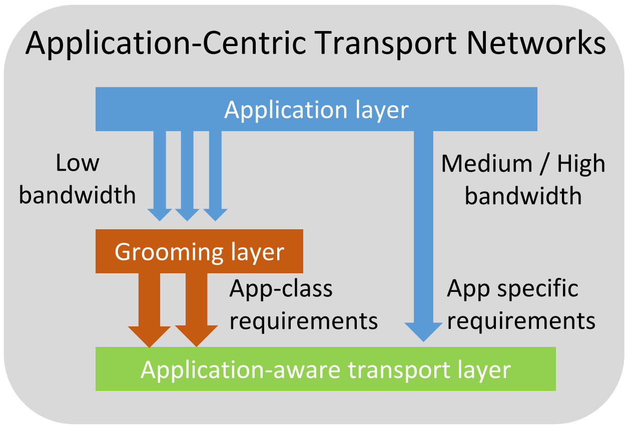

Application-Centric Transport Networks
Application layer
Low bandwidth
Medium / High bandwidth
Grooming layer
App-class requirements
App specific requirements
Application-aware transport layer

Figure 4.2: Application-centric network.

To this end, the *Application-centric* (or application-aware) *networking* is an emerging paradigm that aims at catering to multiple requirements (e.g. bandwidth, latency, availability, security, etc.) when serving the traffic generated by business applications (e.g. live video streaming, financial transactions, remote control of drones, etc.). This paradigm enables a novel network model, which offers a more fine-grained traffic control for the provisioning of connectivity services. The application-specific requirements are mapped into the lower network layers, either into pre-defined traffic classes or directly into the transport layer (Figure 4.2). A generic *Network Provisioning and Management* plane, based on a SDN paradigm, is in charge of receiving the application requirements and the modification of the network configuration accordingly.

Recent studies (e.g. [40, 41]) show how application-centric networking can be pursued in multi-layer transport networks by means of joint configuration/optimization of IP/MPLS and optical layers, in order to provide a tailored service throughout the network stack. Specifically, in [41] we showed the advantages of considering a number of application requirements (ARs) in addition to simple bandwidth when applications' service requests (SRs) must be provisioned. We proved that it is possible to achieve service blocking probabilities similar to an application-unaware scheme, while also ensuring that application needs are met. In this way, the network can ultimately deliver added-value services to customers at roughly the same cost. However, when facing high network utilizations, the service acceptance ratio experiences a reduction that negatively impacts on the revenues of both network operators, which cannot accommodate new SRs while meeting all the ARs, and customers, which have their service blocked.

In this section, we propose the concept of *application-aware service negotiation*. The *negotiation* offers the possibility to find an agreement between applications and networks for the provisioning of a service with *looser* requirements. The algorithm offers several alternative solutions to the application, based on the current status of the network. The application can analyze the solutions and provide a feedback to the network. In this way, the application can receive a predictable service degradation and avoid a block of its request. We propose an architecture and a fully working proof of concept on top of the ONOS network controller [9]. The communication between the applications and the SDN orchestrator is performed by means of a Northbound Interface (NBI) based on an *Intent* paradigm.

The contributions of this chapter can be summarize as follows:

- **Extension** of a multi-layer provisioning algorithm for *Application-Aware (AA) services.* We extend this algorithm in order to support the negotiation mechanism and to search for alternative provisioning solutions for applications.

- **Definition** of a negotiation scheme between applications and networks. We propose a negotiation scheme based on the degradation of application requirements. In addition, we propose an algorithm to let the applications automatically select or reject the alternative solutions offered by the network.
- **Implementation** of the intent-based negotiation scheme on top of the ONOS network controller. We modified the ONOS Intent Framework [42] and adapt the REST API to support the negotiation. The software has been released as open source project [43].
- **Validation** of the performance of the *application-aware negotiation* system on both the network and the application sides. On the network side, we demonstrate a lower service blocking probability with respect to the AA provisioning algorithm without the negotiation. On the application side, we show that the service degradation with respect bandwidth and latency constraints can be kept below a certain threshold.

4.2 Related Work

The negotiation of application-centric service requests requires (i) an interaction between applications and control planes, (ii) a negotiation scheme and (iii) a technique to offer the alternative solutions when the provisioning cannot be performed. In the state of the art several works have been proposed on these topics and we divide them into three different categories:

Interaction between applications and SDN controllers: It offers the possibility to modify in real time the SDN network configuration in to improve the Quality of Service (QOS) of application traffic by exploiting an interaction between applications and SDN network controllers. An application (or a user) may inform the network of either what are the connectivity requirements or provide feedbacks on the traffic treatment experienced. For instance, Jarschel et al. [44] shows that a video streaming application benefits of a high performance improvement, in the case of network congestion, if the application provides feedbacks about the amount left of the video buffer. In [45], the authors presented a system where the users can directly interact with a browser-based Graphical User Interface (GUI) and choose which application, within several pre-configured ones, they want to prioritize. The SDN controller receives the requests from the user and it translates them into forwarding rules for the network. Ferguson et al. [46] proposes a framework to request network policies by the users, such as bandwidth limitation or access control (e.g., firewalling) on traffic, and the possibility to schedule a request for a certain amount of time. However, these works do not consider a possible solution when the application request cannot be satisfied.

Negotiation models: In the state of the art there are a few examples of negotiation mechanisms between a generic control and management plane and a user or an application. These models are not only related to networks but also to cloud computing. In [47], the authors propose a negotiation mechanism based on a *price/service trade-off* based on the network congestion. The application chooses the alternative based on an utility function. However, this work does not provide an evaluation of other parameters rather than the bandwidth. Another type of negotiation is the *auction* [48, 49]. The users can create economical offers to request the provisioning of computational resources in public clouds. When the time for bidding is concluded, the cloud controller decides which are the best offers to be provisioned based on the current resource status. Another type of negotiation mechanism is presented in [50]. In this work, the authors propose a mechanism for negotiating computational resources based on the *Alternate*

Offers Protocol [51]. Specifically, the requesters can make a counter offer to the resource manager. The negotiation finishes when the application and the resource manager find an agreement for the execution of the requested task.

Degradation of service requirements: This technique has been proposed to increase the number of service request that can be accommodated in a network. The application requirements can be degraded to looser values than the initial request. For example, [52] and [53] propose the admission of services with a bandwidth requirement degradation in the case of network failures and/or network congestions. However, all these works propose a degradation that is unilaterally decided by the network without any feedback from applications. In addition, they evaluate only the degradation of the bandwidth without considering other application specific requirements.

4.3 Service Negotiation

This section presents our scheme for the application-aware negotiation of network services. As general model, the negotiation is based on the degradation of the service requirements. The application requests a connectivity service to the network and, in the case of the service cannot be provisioned, the application receives several alternatives from the network. The application analyzes the alternatives and provides a feedback based on an algorithm that selects the best alternative solution for the service provisioning.

As general networking model, we consider a 2-layer physical network composed of a transparent Dense Wavelength-Division Multiplexing (DWDM) optical layer and an IP/MPLS packet layer. The optical layer is composed of ROADM nodes (i.e., reconfigurable optical add-drop multiplexers) that are interconnected by fiber links supporting multiple wavelength. At the IP/MPLS layer, the nodes (i.e., IP/MPLS routers) are interconnected by IP adjacencies that are realized through *lightpaths*, i.e., transparent optical connections.

4.3.1 Negotiation Algorithm

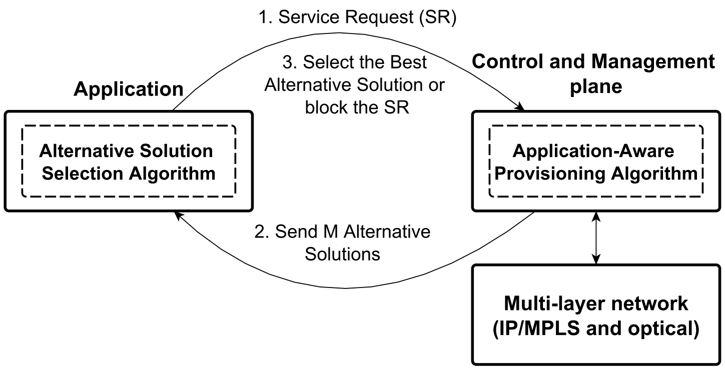

1. Service Request (SR)
3. Select the Best Alternative Solution or block the SR
Application
Alternative Solution Selection Algorithm
Control and Management plane
Application-Aware Provisioning Algorithm
2. Send M Alternative Solutions
Multi-layer network (IP/MPLS and optical)

Figure 4.3: Negotiation interaction between orchestrator and application negotiation-specific algorithms.

The negotiation system is divided into two algorithmic blocks (Fig. 4.3). On the network side, we rely on an extended version of the *Application-Aware Service Provisioning Algorithm* [41], including the negotiation features. The algorithm is implemented as part of a generic Control and Management plane that controls the multi-layer network, e.g. a hierarchical SDN controller. On the application side, an algorithm is designed to autonomously decide among multiple solutions with degraded service offered by the network in the negotiation process (*Alternative Solution Selection Algorithm*). The communication between the two blocks can be provided by a well-designed intent-based northbound interface[54].

We define the application requests to the network as Service Requests, SR (i.e., intents). Every SR is represented as a tuple $SR = \{s, d, b, l, a\}$, in which s and d represent the source node and the destination node IDs, b is the minimum required bandwidth, l the maximum allowed latency and a is the minimum tolerable path availability. b, l and a are the service requirements. The Application-Aware Service Provisioning Algorithm evaluates the service requirements for each SR on the network side, i.e., in the network orchestrator. The provisioning algorithm aims at finding a path meeting all the service requirements, i.e. an application-aware path. If an application-aware path can be provisioned without requirement violations, the SR is *accepted*. This happens when the network has enough resources to provision an application-aware path. Otherwise the SR is *blocked*.

In this section, we extend the features of [41]. In fact, in our new scheme there is a new outcome in this interaction between the application and the network. When it is not possible to provision an application-aware path, since some of the requested service requirements cannot be guaranteed, a *negotiation* phase for the SR is started. This can happen because a network has limited resources that have already been provisioned to meet other SRs with strict requirements. After the negotiation phase, a new negotiated SR can be *accepted*, or the negotiation is aborted and the original SR *blocked*.

For example, a SR generated by an application with stringent latency requirements may find the shortest paths busy because of traffic generated by other applications, but may be willing to negotiate a lower requirement instead of having its request blocked. In this way, the application can achieve a predefined service degradation and it can modify its behavior based to the new service requirements. In order to inform the network that an application is willing to negotiate, the *SR* tuple reported above is extended to carry also the information on which requirements of a SR are negotiable. In particular, the new considered tuple is $SR = \{s, d, b, l, a, n_b, n_l, n_a\}$, where the flags n_b, n_l, n_a are associated to b, l and a, respectively. They can be set to *true* or *false* to inform the provisioning algorithm of which service constraint can be negotiated.

The provisioning algorithm offers a set of *Alternative Solutions (AS)* to the application, in which the negotiable requirements have looser values than the original SR. For example, if a SR requires a particular value of l with $n_l = true$, the provisioning algorithm can offer a new value $l_n > l$, meaning that it can provide an application-aware path meeting such looser latency requirement, as in the example above. The application evaluates the new solutions and accepts one of them or rejects all. If all the alternative solutions are rejected, the SR is blocked.

Note that, since the Provisioning Algorithm does not have any knowledge on the threshold values for the service constraints, it cannot bias its choice to provide the least acceptable constraints to the application. In this way, the application can potentially experience a lower downgrade than the least-acceptable one, if there are enough available resources in the network. This assumption is valid if there is no a pricing model applied in the system. The application can have a more complex decision process in which evaluates all the alternatives based on an utility function. On the other side, also the network can have an utility function which can try to maximize both the resource occupation and the revenues.

4.3.2 Application-Aware Service Provisioning Algorithm

The Application-Aware Service Provisioning Algorithm presented in this section extends the algorithm presented in [41]. Such algorithm offers the provisioning of application-aware services on top of multi-layer IP/optical networks. For every SR, the algorithm attempts to find an application-aware path between the existing *lightpaths*, i.e., IP adjacenices, first and then, if no application-aware path is found, by considering also the *potential lightpaths* (i.e., lightpaths that have not been established yet, but that could be established if needed). This increases the chance of finding a solution at the expense of resorting to new optical resources. The algorithm exploits an Auxiliary Graph (AG) model [55]. The algorithm works as follows, and is executed for every SR:

1. All the *existing lightpaths* (i.e., edges) are added to an AG, where the added nodes are IP/MPLS nodes. The edges not meeting the bandwidth

requirements of the SR are pruned from the AG.

2. The K_{ip}-Shortest Path (SP) algorithm is executed on the AG between s and d of the SR. The weight for each edge is the physical length of the corresponding *lightpath*.

3. Up to K_{ip} *candidate paths* are returned, all meeting the b requirement.

4. Then, the algorithm prunes all the *candidates paths* not meeting l and a, and returns the first in the list (i.e., the shortest).

5. If the list of candidate paths is empty after step 4, the algorithm augments the AG by including the *potential lightpaths* and steps 2, 3 and 4 are executed again. In the original version of the algorithm, if no cadidate path is found at this point, the request is blocked. For more details on the algorithm implementation see [41].

Then, we extended the Application-Aware Service Provisioning Algorithm described above to support the *negotiation* phase for the SRs in case no application-aware path can be found. The Provisioning Algorithm thus computes a set of M ASs, in which the negotiable service requirements can have looser and network-achievable values than the ones specified in the SR. In particular, the Provisioning Algorithm in the negotiation phase works as follows:

1. A copy of the initial SR is created and all the negotiable parameters are neglected.

2. Such modified SR is used as input for the Application-Aware Service Provisioning Algorithm described above.

3. If the algorithm does not output any path, the SR is blocked. Otherwise, it stores the guaranteed requirements for each computed alternative

candidate path (e.g., the minimum available bandwidth, the maximum latency, etc.).

4. The number of candidate paths can be high, if there are many negotiable parameters and a high value of K_{ip} is used. The algorithm thus keeps only the best path for each of the negotiable requirements. For example, if b and l are negotiable, the provisioning algorithm offers two *alternative solutions* to the application: the one corresponding to the path with maximum residual bandwidth and the one corresponding to the path with minimum guaranteed latency.

5. The alternative solutions selected in step 4 are sent to the application, that can thus run the Alternative Solution Selection Algorithm, which is described in the next section.

4.3.3 Alternative Solution Selection Algorithm

On the application side, the AS needs to be analyzed in order to find which is the best one for the application. In our model, every application has a set of preferable values $SR_p = (b_p, l_p, a_p)$ and a set of least acceptable values $SR_t = (b_t, l_t, a_t)$, represented as tuples. The former indicates the preferable values that the application wants to obtain from the network, while the second represents the *threshold* values that the application is willing to accept in the case of a negotiation is needed. If the preferable and minimum value for a service requirement is the same, it means that such requirement is not negotiable. The application communicates only the values included in SR_p when it sends the SR to the orchestrator. For example, a VoIP application, according to the ITU-T G.114, requires a maximum latency of 150 ms. However, this is not a preferable value but the maximum allowed. A VoIP call with 150 ms of latency have a really bad quality. A preferable value could be less than 50 ms. Potentially, the VoIP application can thus request a SR where SR_p has $l_p = 50$ ms and set a SR_{min} with $l = 150$ ms. The application can is thus flexible in accepting an alternative solution with *reduced* quality but with a predictable degradation.

The Alternative Solution Selection Algorithm is in charge of automatically selecting the best AS for each application. The algorithm works as follows:

1. The application receives M ASs in the form of reduced SR tuples (i.e., without negotiation flags). Each tuple is defined as $AS_j = \{b_{n_j}, l_{n_j}, a_{n_j}\}$, in which $b_{n_j}, l_{n_j}, a_{n_j}$ represent the values, for each AS j, as computed by the orchestrator in the negotiation phase.

2. The algorithm prunes all the solutions that have requirement lower than the values specified in the SR_t tuple (e.g., if $b_{n_j} < b_t$, prune the solution j).

3. If there is no alternative solution meeting all the requirements specified in SR_{min}, the SR is blocked. Conversely, the algorithm calculates the weighted Euclidean distance d_j between SR_p and every AS_j, as defined in Eq. 4.1.

$$d_j(SR_p, AS_j) = \sqrt{\sum_{i=1}^{N} w_i(SR_{p_i} - AS_{j_i})^2} \tag{4.1}$$

   where the index i refers to each one of the N service requirements included in SR_p and AS_j and w_i represent the weight for the service requirements i. By properly tuning the weights w_i, the application can specify a different preference for every service requirement. For example, in the case of an application has a more stringent latency requirement among the negotiable ones, the weight w_l will be higher than the weights of other requirements.

4. After evaluating every Euclidean distance, the algorithm selects the alternative solution leading to the minimum Euclidean distance to SR_p:

$$AS_{best} = AS_j : j = \arg\min(d_1, \cdots, d_M) \tag{4.2}$$

5. AS_{best} is sent to the orchestrator, which allocate resources on the associated path.

4.4 Software Architecture

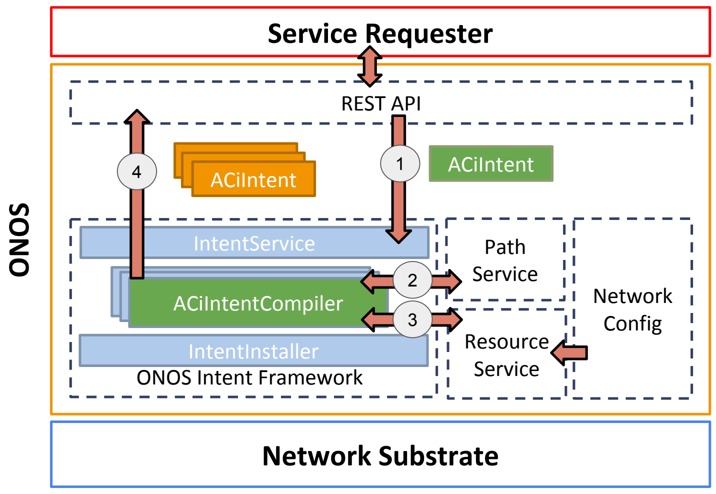


Figure 4.4: Negotiation architecture of the ONOS controller.

The negotiation algorithm requires several services from a SDN controller: (i) a RESTful API to simplify the submission of the intents from the users or applications, (ii) the translation of the intents in forwarding rules, i.e. the intent compilers, and, (iii) a network *resource manager*, which keeps track of the available resources in the network.

To this end, we chose ONOS [9] as SDN controller for the implementation of the negotiation algorithm. It offers a modular architecture in which external software modules, i.e. OSGi bundles [56], can be added and removed at runtime. The modules can exploit many *Services*, based on Java APIs, to interact with the controller core and the underlying network substrate. The negotiation algorithm is implemented as a ONOS module, which exploits several *Services* and relative APIs to work.

In this section, we describe the implementation of the *negotiation module* for ONOS. We describe the ONOS *Services* used, how they are mapped to the negotiation module and how the negotiation works in ONOS.

4.4.1 ONOS Services and APIs

The negotiation module exploits the following *Services* from the ONOS controller:

ONOS Intent Framework. The ONOS controller provides a comprehensive *intent framework*, which manages the intent submission, compilation and the installation of the corresponding forwarding rules in the network devices. The intents can be submitted via the *IntentService*, which exposes the Java APIs for interacting with the service. It is composed of several intent compilers, each one compiling a specific type of intent. The intent compilers interact with the other ONOS *Services* in order to obtain the information required for the intent translation in specific forwarding rules. For example, ONOS provides a set of pre-defined intents representing a network connectivity action, such as the *HostToHostIntent*. This intent represents a point-to-point connection between two host in the network. The only compiler responsible for the translation in forwarding rules is the *HostToHostIntentCompiler*, which interacts with the other ONOS *Services* to define the associated forwarding rules. When a compiler defines the forwarding rules, it requests to the *IntentInstaller* the installation. The negotiation module exploits the ONOS intent framework since it implements the *ACiIntentCompiler* for the *Application-Centric Intents* (ACiIntents). This type of intent represents the request for an Point-To-Point connection between two endpoints in a network with a specific set of application-centric constraints, such as bandwidth, latency, availability, etc.

PathService. It represents the Path Computation Element (PCE) of ONOS. All the modules can query this service to obtain the shortest path (i.e. a set of links) between two endpoints of a network (e.g., devices, hosts, etc.). The *ACiIntentCompiler* exploits this service to get the shortest path between the two network endpoints defined in the *ACiIntent*.

ResourceService. This service provides a database to keep track of all the available network resources (e.g., links, the ports of a device, etc.), their characteristics (e.g., latency, capacity, etc.) and consumption. The ONOS modules can use the APIs to interact with the service both to query or update the consumption of a resource. Only particular type of resources can be consumed, such as the capacity of a device port or the number of a VLAN. For example, an intent requires a bandwidth constraint of 100 Mbps between two host in a network. The *IntentCompiler* queries the *PathService* and receives the shortest path connecting the two hosts. Then, the *IntentCompiler* checks if all the devices' ports associated to the path can support the requested capacity. Finally, if allowed, it associates the intent as a resource consumer of all the ports' capacity in the *ResourceService.* Thus, on a port of 1000 Mbps, the capacity left is 900 Mbps. The *ACiIntent* compiler queries this service in order to check if a path can provide the specific application-centric constraints and calculate the possible alternative solutions.

NetworkConfig. ONOS provides a service for configuring the characteristics of the network device and links with custom values. For example, it can be used to configure the capacity of a device port or to annotate on a link its features, such as the latency or the availability. These information are then provided to the ONOS *ResourceService* to be queried by other modules.

4.4.2 Negotiation in ONOS

The negotiation algorithm in ONOS is implemented in the following way:

Service Request submission. The *Service Requester*, either an application or a user, submits an *ACiIntent* with several constraints via the *REST API*. The request is converted to an *ACiIntent* Java class and it is submitted to the *IntentService* for the compilation (arrow 1).

Intent Compilation. The *ACiIntentCompiler* starts the compilation of the *ACiIntent*. First, it requests to the *PathService* all the possible shortest path between the two intent Endpoints (arrow 2). Then, for every path found, the compiler queries the *ResourceService* to check if the intent constraints can be satisfied (arrow 4). The paths that do not satisfy the constraints are excluded from the list. If at least one path is found, it is converted in forwarding rules and they are sent to the *IntentInstaller*. Otherwise, if all the paths are excluded, the negotiation phase starts.

Intent Negotiation. The paths that were previously found by the *PathService* are analyzed again. The *ACiIntentCompiler* queries the *ResourceService* in order to find the minimum constraints that the paths can allow. For example, if the bandwidth constraint is too high with respect to the current path provisioning, the *ACiIntentCompiler* calculates the maximum allowed constraint for the path. Then, all the new constraints associated to the paths are associated to a set of alternative *ACiIntents*. Finally, they are submitted to the *REST API* to let the *Service Requester* to choose an alternative solution.

Solution Selection Algorithm Implementation. After a service request to the network with several negotiable constraints, an application can receive back several alternative solution. The solutions received from the SDN controller should be analyzed in order to understand which is the best one. Listing 4.1 shows the algorithm for the selection of the best solution proposed by the SDN controller. First, the *Intent Adapter* creates a *Solu-*

tionVector from the *Service Requirements Model.* The *SolutionVector* is a data structure composed of all the *constraints*, the preferable and the acceptable values. The same is performed for the solutions proposed by the SDN controller. In this case, the minimum/maximum acceptable values are set to zero and ignored. Second, from line 8, the solutions coming from the network are iterated in order to check if they are within the minimum/maximum acceptable values. Every solution that does not respect the minimum value of a constraint is removed from the possible solutions. Then, in line 25 is calculated the Euclidean distance between the preferable solution and the current solution and this value is updated inside the solution proposed by the SDN controller. Finally, from the *solutionList* is chosen the solution with the mimimum Euclidean distance, thus the one closer to the *preferable* solution (lines 31-32). Then, the answer is notified to the SDN controller. In case of an empty list, the negotiation is aborted (line 34), since there are no solutions that can be accepted by the application.

```
1
2  SolutionVector preferableVector =
3                 buildPreferableVectorFromModel();
4  List<SolutionVector> solutionList =
5                 buildSolutionListFromJson();
6  Long intentID = getIntentIDFromJson();
7
8  for(SolutionVector solution : solutionList){
9
10   if(solution.latency >
11            preferableVector.maxAllowedLatency){
12     solutionList.remove(solution);
13     continue;
14   }
15   if(solution.bandwidth <
16          preferableVector.minAllowedBandwidth){
17     solutionList.remove(solution);
```

```
      continue;
    }

    ...

    //Update the Euclidean distance
    //for the proposed solution
    calculateEuclideanDistance(preferableVector,
                               solution)
}

if (!solutionList.empty()) {
     SolutionVector bestSolution =
                    Collections.min(solutionList);
     sendAcceptedMessageToController(bestSolution);
} else {
    sendNotAcceptedMessageToController(intentID);
}
```

Listing 4.1: Algorithm for solution selection on the application side

4.5 Performance Evaluation

In this section, we present the experimental results on the negotiation algorithm. The network sensitivity tests are performed on Net2Plan [57], an open source tool for network planning and simulation. Both the *Application-Aware Provisioning Algorithm* and the *Solution Selection Algorithm* are implemented on Net2Plan.

The sensitivity tests allowed us to demonstrate the influence of the negotiation on both the network and the applications in different scenarios. In particular, the sensitivity tests evaluate the SR blocking probability, i.e. the amount of blocked SRs caused by constraint violations, and the average SR degradation experienced by applications with respect to the (b_p, l_p, a_p)

constraints of every SR negotiated.

Although the sensitivity tests are performed on a network simulator, they are based on a real network topology and a traffic matrix provided by the ISP Telefónica Spain. This offers the possibility to evaluate the negotiation algorithm on a network environment comparable to a real one.

4.5.1 Simulation Setup

The Telefónica's topology is a multilayer network composed of 30 ROADMs/OXCs and 56 bi-directional fiber links carrying up to 80 wavelengths, with a capacity of 100 Gbps each at the optical layer. We doubled the propagation delay of each fiber to simulate a larger network. On the IP layer, there are 14 IP/MPLS routers that can be interconnected by lightpaths provisioned at the WDM layer [58]. The traffic generation is performed by using the non-uniform traffic matrix of the same ISP, in which the majority of traffic is routed to/from the capital city Madrid.

Net2Plan offers a discrete event simulator composed of an event *generator* and an event *processor*. The event *generator* produces a new SR based on a Poisson process, i.e., with exponentially-distributed inter-arrival times and holding times. After the expiration of the holding time, the allocated network resources (e.g., the links and the capacity) to the SR are released. The number of requests generated is 5×10^5 and the statistics are collected after 2×10^4 events, in order to exclude the values in low network utilization. The provisioning algorithm has $K_{ip} = 50$ and $K_{wdm} = 5$.

We consider as service requirements for each SR the bandwidth (b), the latency (l) and the availability (a). a is expressed as $(MTBF/(MTBF + MTTR)) \cdot 100$, where $MTBF$ is the *Mean Time Between Failures* and $MTTR$ is the *Mean Time To Repair*. The availability values provided on the topology are different on every link and comparable to real ones, thus the availability is not strictly dependent on the length of paths. All the parame-

Table 4.1: Negotiation Level values

	$(b_p - b_t)/b_p \cdot 100$ (%)	l_t (ms)	a_t (%)
NL1	10%	15	99.5
NL2	20%	20	99.4
NL3	30%	25	99.2
NL4	40%	35	99

ters are randomly chosen based on the following sets: $b = \{1, 2, 5, 10\}$ Gbps, $l = \{10\}$ ms, and $a = \{99.6\}$ %. The s and d parameters are generated accordingly to the non-uniform traffic matrix.

4.5.2 Sensitivity Test Methodology

We performed three experiments in which we evaluate the impact of the negotiation scheme on the network behavior: (i) all the SR constraints can be negotiated at the same time and can experience a degradation; (ii) different ratios of SRs are willing to negotiate with respect to the total amount of SRs, and (iii) the SRs can negotiate only one constraint at time.

In each experiment, we study the trade-off that the negotiation offers in terms of the gain on the number of SRs accepted and the degradation of parameters experienced by the applications. All the different scenarios are compared with the case of the negotiation scheme is not adopted.

1) The SRs can negotiate all the constraints. In this experiment, the SRs are always willing to negotiate b, l and a with the network by accepting looser values of parameter degradations. We define multiple Negotiation Levels (NLs), representing the maximum allowed degradation for application SRs. The threshold values b_t, l_t and a_t are set as reported in Fig. 4.1. According to the defined NLs, each SR can always tolerate a relaxation of b, l and a: the maximum bandwidth tolerated degradation is set in terms of degradation

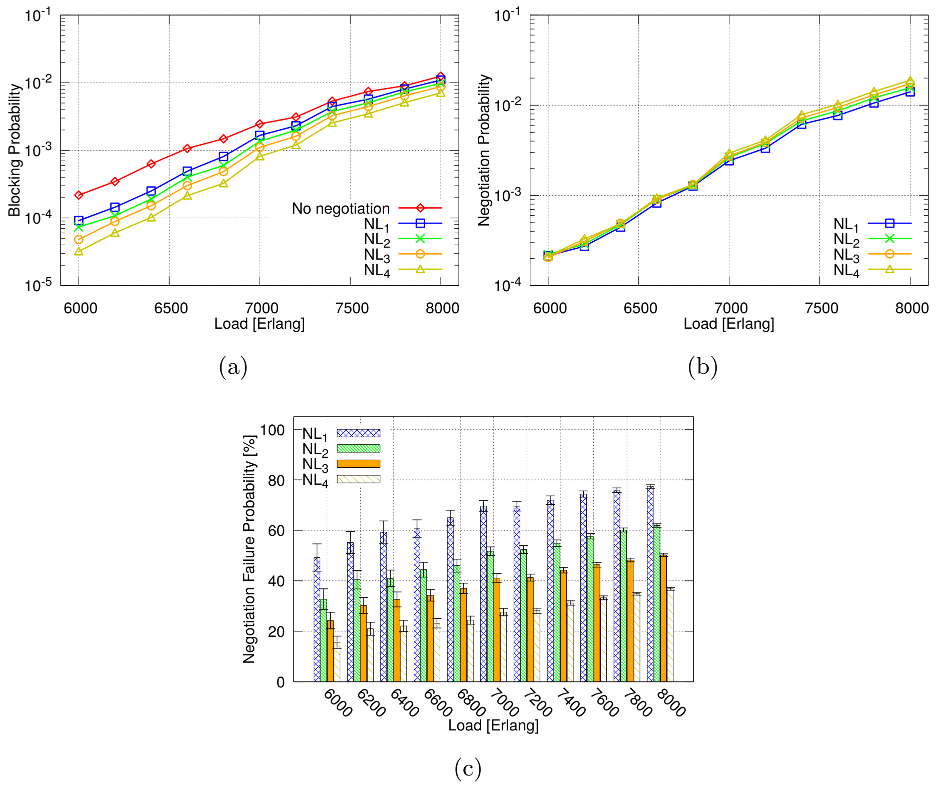
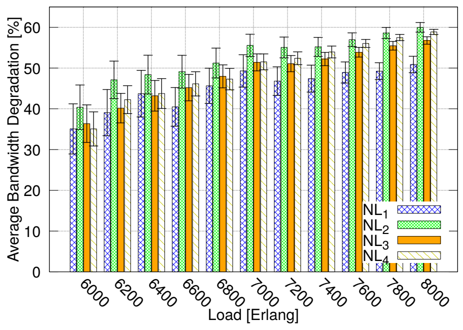


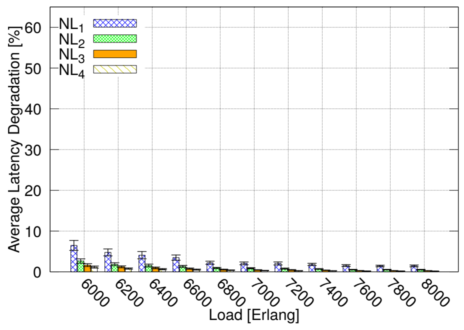
Figure 4.5: (Experiment 1) Evaluation of SR blocking probability (a), negotiation probability (b) and negotiation failure probability (c).

percentage, the maximum latency tolerated degradation is set in terms of a higher delay (in ms), while the maximum availability degradation is set in terms of a lower value (in %). Higher NL subscript is always associated to higher tolerance to service downgrade. For each simulation, we made all SRs belong to the same NL.

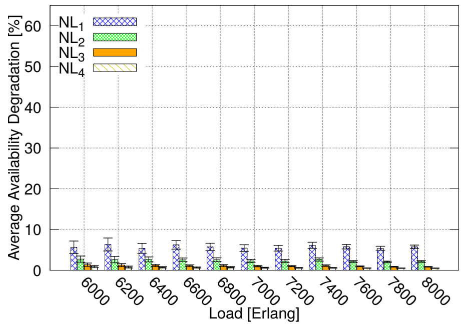
Fig. 4.5a and Fig. 4.6 offer an overview on the trade-off between the gain in SR acceptance (in terms of blocking probability reduction) and SR average bandwidth, latency and availability degradation, for the negotiated SRs, as a function of network load and NL. To be noticed in Fig. 4.6, the values are normalized between 0% and 100%, in which 0 represents no degradation, while 100 corresponds to the maximum allowed degradation for the parameter at the selected NL. For instance, in the bandwidth degradation



(a)



(b)



(c)

Figure 4.6: (Experiment 1) Evaluation of the average degradation of bandwidth (a), latency (b) and availability (c) experienced by SRs.

(Fig. 4.6a), at NL_4, 100% represents $(b_p + 40\%)$. In the case of a network load of 6000 Erlang, the blocking probability decreases of about an order of magnitude between the *No negotiation* and NL_4 cases, while the bandwidth (Fig. 4.6a), the latency (Fig. 4.6b), and availability (Fig. 4.6c) experience the 35%, 2% and 1%, respectively, of their maximum allowed degradation. Thus, the applications experience much less than the maximum tolerated degradation, while the network increases much more the number of SRs provisioned.

The bandwidth degradation increases both with respect to (i) network load and (ii) NL. In fact, with higher loads, the network is only able to offer ASs with in average more degraded bandwidth, since the average network utilization is higher. Moreover, a higher NL makes SRs more tolerant

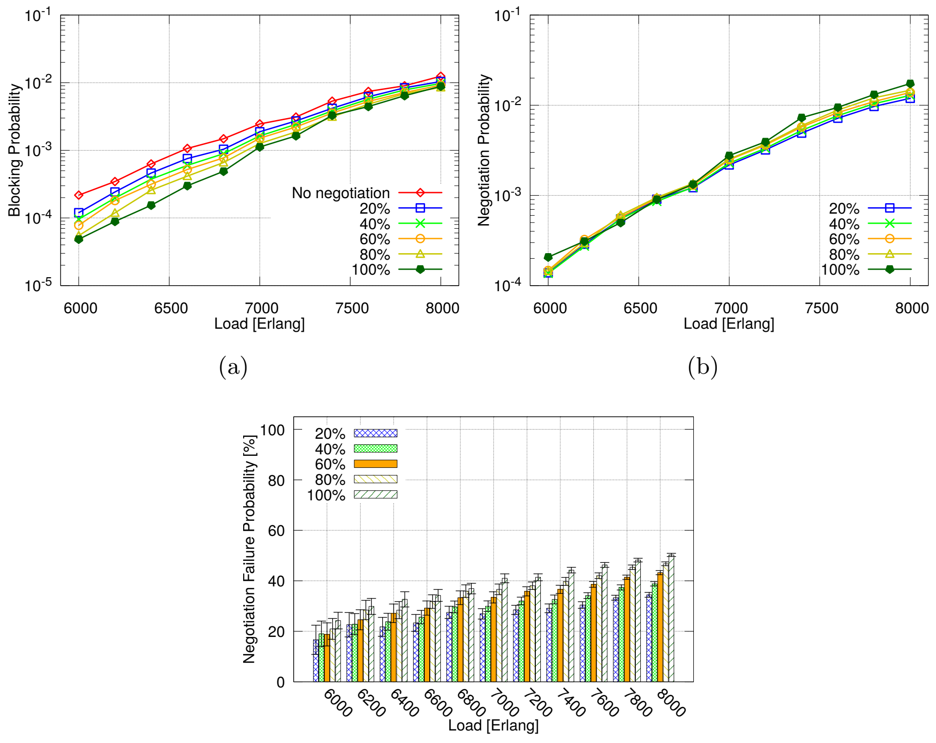


Figure 4.7: (Experiment 2) Evaluation of SR blocking probability (a), negotiation probability (b) and negotiation failure probability (c).

to bandwidth degradations, and the SRs bandwidth is thus in average degraded more. As opposed to bandwidth degradation, latency degradation decreases with respect to NL and network load. The reason is that, in our assumptions, each SR can have b, l and a degraded at the same time. The higher b degradation is, both as a function of load and NL, the easier finding spare resources on shortest paths is. Higher b degradations are thus always associated to lower l and a degradations. This behavior points out how multiple ARs experience different degradation trends when they can be relaxed at the same time, and how they mutually influence their trends.

Figs. 4.5b-4.5c show the SR negotiation probability (i.e., the probability that the network starts the negotiation phase for a SR) and the negotiation failure probability (i.e., the probability that the negotiation fails because no

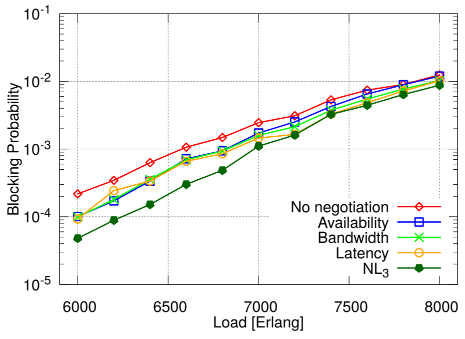
AS suits the least-acceptable values for the ARs) as a function of network load and NL. Fig. 4.5b shows that the SR negotiation probability is, as expected, similar to the blocking probability of *No negotiation*: it increases as the network load increases and it is only slightly dependent on NL. As expected, the negotiation failure probability (Fig. 4.5c) is higher (i) when the NL is lower and (ii) as the load increases, i.e., in all the cases where network utilization is higher.

Table 4.2: Degradation of the parameter values

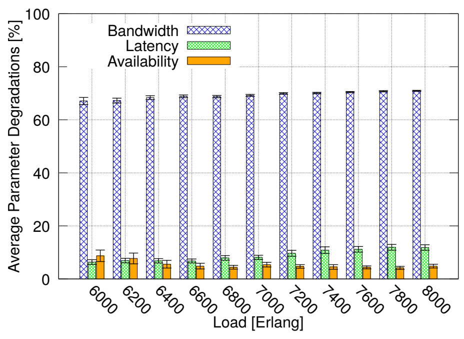
	Low Load			High Load		
	b (%)	l (%)	a (%)	b (%)	l (%)	a (%)
20%	29	1.86	1.45	54	0.27	0.91
40%	31	1.77	1.35	56	0.24	0.92
60%	33	1.68	1.00	57	0.2	0.92
80%	34	1.77	1.21	58	0.2	0.87

2) A percentage of the total number of SRs can negotiate. This experiment aims at showing the impact of negotiation when a different amount of SRs are willing to negotiate all their constraints. The amount of negotiable SRs is expressed as a percentage. It represents the ratio between the negotiable and the total number of SRs in the simulation. We defined the percentage as a value between 0% and 100% that is increased by steps of 20% between each simulation. The maximum allowed degradation is fixed for b, l and a to NL_3 since it represents an average value of degradation within all the experiments performed.

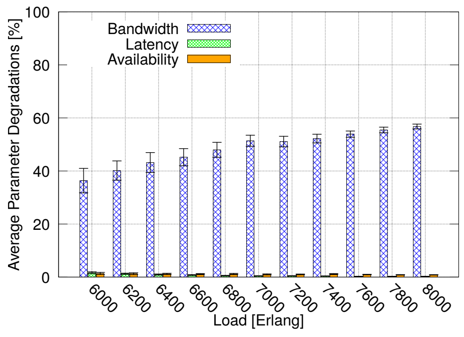
Fig. 4.7 offers an overview of the negotiation performance in terms of SR blocking probability (Fig. 4.7a), negotiation probability (Fig. 4.7b) and negotiation failure probability (Fig. 4.7c) as a function of the network load and the percentage of negotiable requests. As expected, the SR blocking probability is reducing as the number of negotiable SRs increases. In Ta-



(a)



(b)



(c)

Figure 4.8: (Experiment 3) Evaluation of SR blocking probability (a), average parameter degradation (b) and average parameter degradation at NL_3 (c).

ble 4.2, we report the experienced parameter degradation at low and high network loads (i.e. 6000 and 8000 Erlang). The degradations are mostly dependent on the network load than the average number of negotiable SRs.

3) Single negotiable constraint. In this test, the SRs are willing to negotiate only one constraint at time while the others are kept at the preferable value. We aim at evaluating how the negotiation of a single SR constraint can influence the blocking probability and the experienced degradation and if there is a predominant constraint. Between each simulation run, we set as negotiable b, l or a. The maximum allowed degradation of the negotiable parameter is set to NL_3. We compared the results to the ones of the first experiment performed, in which all the constraints can be negotiated, with the maximum degradation of NL_3.

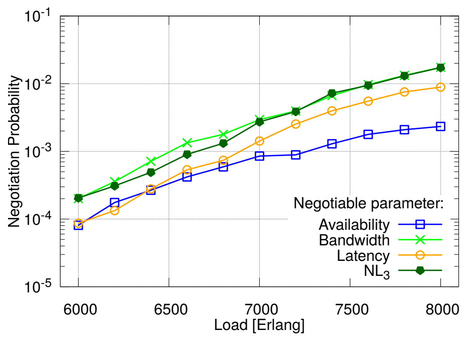


Figure 4.9: (Experiment 3) Negotiation probability.

The SR blocking probability (Fig. 4.8a) presents a slight reduction at 6000 Erlang with respect the *No negotiation* case. From 7000 Erlang, the negotiation and *No negotiation* cases have the same values. Different from the NL_3 line, which shows a decrease of an order of magnitude at 6000 Erlang and the negotiation effects are still present at 8000 Erlang. Thus, the negotiation of all the constraints demonstrates better performances than the negotiation of a single constraint. The blocking probability shows almost overlapping values between b, l and a cases on all the network load analyzed. This implies that there is not a predominant constraint and both the negotiable and non-negotiable ones influence each other in the SR evaluation on the network and application sides. In particular, on the network side, in the case of l and a are negotiable, we found that the 100% of blocked SRs is caused by the impossibility to find any AS by the Application-Aware Algorithm. Different from the case of b is negotiable, in which the algorithm always manages to find an AS. On the application side, in the case of l or a are negotiable, the application can always find an AS fitting in the range of either l_t or a_t, while in the case of b negotiable, the 100% of the ASs excluded by the Alternative Solution Algorithm are caused by a violation of b_t. Indeed, as reported in Table 4.3, the probability that the negotiation fails depends only on b between all the analyzed network loads.

The constraint degradations, normalized between 0% and 100%, are depicted in Fig. 4.8b. In this experiment, the degradation can be experienced only by the parameter chose as negotiable for the simulation. In the case of b negotiable, the degradation of b is constant and does not present significant variations between 6000 and 8000 Erlang. Different from when all the constraints are negotiable (Fig. 4.8c), in which b experiences an increasing degradation from the lowest to the highest network loads. To be noticed, in the case of l or a are negotiable, they present higher degradations with respect to the case of all the constraints are negotiable. For instance, at 6000 Erlang l experiences the 6.20% and 1.66%, respectively. In addition, the degradation of l increases with respect to the network load in this experiment, unlike the initial one. This occurs due to the parameter b that is not negotiable and the SRs are forced to use always longest paths since the shortest ones are the first to fill up the capacity.

Table 4.3: Negotiation Failure Probability

	b (%)	l (%)	a (%)
Low load	47.67%	0	0
Medium load	52.38%	0	0
High load	58.87%	0	0

Fig. 4.9 shows the negotiation probability as a function of the network load and the negotiable constraint. The negotiation probability in the case of b is negotiable is almost overlapped to the NL_3 case, thus, they present a negotiation probability that is similar to the *No negotiation* case. The latency and availability negotiation probabilities present similar values between 6000 and 6800 Erlang, while they are different at higher network loads. The availability presents lower values at higher loads, thus the number of SRs that enter in the negotiation phase are less than the other cases. This implies that the network has more difficulties to find an AS when both b and l are not negotiable.

Chapter 5

Interactions between business applications, SDN controller applications and networks

Network operators are looking forward to novel business models and services for their customers in order to increase the revenues while reducing CAPEX and OPEX. However, offering a customized service to every customer poses many challenges and overheads to network providers. The services developed on networks are often imperative and have to consider integrations with a number of subsystems in an operator's ecosystem. Consequently, network operators typically limit commercial offers to a fixed pool with limited configuration options that novel business applications with stringent requirements could not exploit.

In this chapter, we show how the interaction between three different entities, business applications, SDN controller applications and networks can offer a multi-service selection system to increase the number of ad-hoc connectivity services offered by a network operator. We exploit the flexibility of SDN networks to propose an SDN App Store, which leverages *intents* to offer a multi-service selection model for network connectivity services, implemented as applications on top of an SDN controller. The user or the business application can request an SDN controller application (SD-WAN, VPN, etc.) and the associated constraints in the form of an intent, and receive offers from multiple SDN application providers, each one delivering a customized implementation of the application. Performance evaluation of the implementation demonstrates the capability to integrate multiple providers with very low overhead while enabling users to choose from multiple offers for a given application.

5.1 An SDN App Store for Network Connectivity Services

A primary revenue stream for network operators comes from offering services for specific types of applications and customers, such as data center interconnection, SD-WAN, etc., with specific requirements and constraints. Demands for such services is expected to show significant increasing, driven by a heterogeneity in terms of customer requirements. This implies that network operators are forced to look forward to novel market models and technologies while increasing the revenues and reducing the operating costs.

However, the logistical overhead in deploying and maintaining a wide set of services, which consists of specific configurations is non-trivial. The high number of permutations to configure and maintain a service on vendor-specific devices is one of the primary reason why operators only offer a limited portfolio of services to their customers.

The advent of software control on network platforms provides the potential to deliver an enhanced flexibility to network operators. Software control simplifies service provisioning and Operations, Administration and Management (OAM) in a vendor-agnostic fashion. The SDN controllers, based on application requirements, can deploy specific software-based network functions on generic white-box hardware in the network.

The architectural modularity of modern SDN controllers, such as ONOS [9] and OpenDayLight [10], can be exploited to develop single software modules, executed on top of them, that can offer all the logic for computing, provisioning and maintaining services required by a given connectivity application. It is envisioned that an SDN controller should be able to host multiple software modules (or SDN controller applications) that can deliver distinct solution flavors for a given request for a connectivity application. This solution may increase the competition between multiple service providers, which can share the same network infrastructure to provide their own solution for a connectivity service to a customer.

This work presents the SDN App Store architecture that proposes a system for multi-service selection built on top of an SDN controller. The architecture enables operators to host multiple SDN applications offering the same network connectivity services (e.g., SD-WAN, CDN, etc.) and enables the users to select between multiple offers as computed by these applications. Multiple algorithms are incorporated as SDN applications to compute service offerings and an intent-based NBI, named DISMI [59], offers the possibility to define a generic, technology-agnostic grammar to communicate with the users. We extended DISMI to support a negotiation mechanism, enabling a user to choose from multiple offers in a seamless, application-agnostic fashion.

The SDN App Store and its intent-based interface are released as open source project [43].

5.2 Related Work

In the state of the art, the possibility to select a service from multiple providers has been proposed under the concept of service *brokers*. They aim at simplifying the service selection and provisioning in a multi-provider setting. Service brokers are middlewares that offer an abstraction between the users/applications and service providers. *Brokers* receive a service request from the users, and query multiple providers for offers. Finally, the brokers orchestrate provisioning for offers that are either selected automatically or by the user.

Service *brokers* can be divided into two main categories:

Cloud Service Brokers. The cloud paradigm has always been considered as a commodity service for computational resource, which can be scaled up and down on-demand. The number of cloud providers is continuously growing, making it difficult for users to find the best service in terms of price and performance guarantees. To overcome this issue, several works proposed the concept of *Cloud Service Brokers.* In [60], authors proposed a cloud broker that automatically deploys an application by selecting resources between multiple cloud providers. User can describe application requirements (storage, CPUs etc.) and the broker selects the best service among the registered cloud providers in terms of price and performance. Another work [61] presents a cloud broker, based on standard APIs and model descriptors for application requirements. This work proposes a solution to automatically load balance applications between multiple cloud providers. The work in [62] propose service selection by offering several alternative prices offered by the registered cloud providers to the user.

Connectivity Brokers. Network operators are exploring solutions to dynamically provision on-demand connectivity services in order to enable new market models [63], which in turn has led researchers to explore the use of brokers to choose between multiple services. In [64], the authors propose

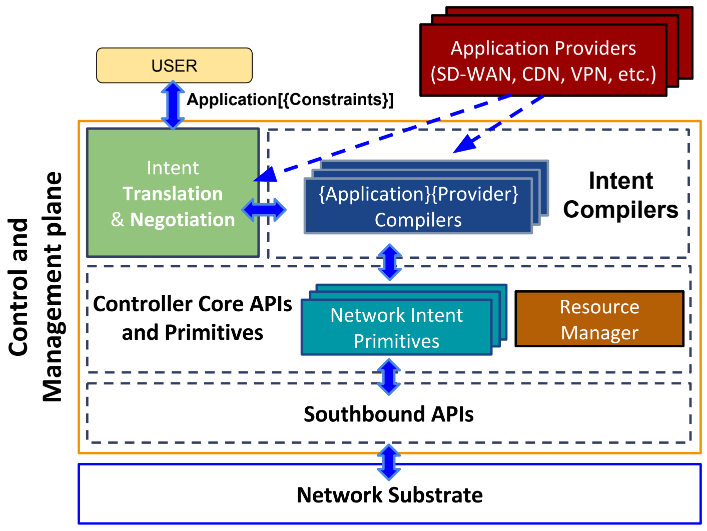

USER
Application[{Constraints}]
Application Providers (SD-WAN, CDN, VPN, etc.)
Control and Management plane
Intent Translation & Negotiation
{Application}{Provider} Compilers
Intent Compilers
Controller Core APIs and Primitives
Network Intent Primitives
Resource Manager
Southbound APIs
Network Substrate

Figure 5.1: App Store generic model.

a market model in which a broker queries the service providers and attempts to find the best solution that optimizes performance and costs for the user. The selection of a custom end-to-end path traversing multiple domains has been proposed in [65], which is able to query multiple network exchange points in order to formulate an offer for the user.

To date, brokers are designed for specific applications, and cannot be easily extended to other applications. For example, connectivity brokers do not inherently provide the possibility to request advanced connectivity service, such as SD-WAN, and limit the service requirements that can be requested are based on the broker's specifications. This work presents a general framework which uses multiple SDN applications to compute offers for specific network application requests, and facilitates offer selection by the user.

5.3 The App Store Model

This section presents our model for the App Store. The App Store is designed as a platform where users can request a specific application and multiple SDN application providers (*apps*) can compute service offerings for the request. Computed offers from the apps are sent to the user, and a selection from these offers is used to initiate provisioning actions. For the scope of this work, the application under consideration are traditional business connectivity applications like SD-WANs, CDNs, VPNs, etc. The App Store offers *applications* on top of single Internet Service Provider (ISP) management infrastructure with networking resources shared between multiple applications. Apps can target implementations for a given *application* to optimize specific metrics: For instance, applications offering a CDN application can optimize selection of caching points based on different metrics (i.e. total capacity reserved, or on maximum geographical distance from caching point) which leads to a different offer to the user.

Fig. 5.1 shows the detailed App Store architectural model. The App Store is based on a SDN framework, in which a generic network control and management plane (i.e. a SDN network controller) can control the network substrate in a vendor-agnostic fashion. The App Store also exposes an Intent-based NBI, which offers the possibility to state *what* is the required by the application instead of *how* it should be implemented. The intent API is used by the users to request an application and to interact with the SDN controller. The user intent is composed of an *application* request with several *constraints* describing the application requirements. The grammar defining the intent is defined as {*application[contraints]*}. For instance, the intent grammar for a SD-WAN intent can be broadly specified as defined as {*SD-WAN[endpoints=[DataCenter1, DataCenter2], bw=1 Gbps]*}, where *endpoints[], bw* represent the list of network endpoints, and the capacity, respectively.

After the intent submission, a specific module named *Intent Translation and Negotiation* is responsible for managing the intent lifecycle in the SDN controller. It provides the abstraction between an *application* request and the implementation of all the *application providers.* This module maintains a list with all the *providers* that implement a particular *application.* On receiving an *application* request, the *Intent Translation and Negotiation* module validates the request, and queries the provider list to find all the possible *providers* that offer the *application.* Finally, this module generates the requests for all the *providers* implementing the *application*, which are made by generating specific intents, defined as {*Application*}{*Provider*}*Intents.*

{*Application*}{*Provider*}*Intents* are high level instructions that require a translation into network actions and commands for the SDN orchestrator, which are computed inside intent *compilers*, called {*Application*}{*Provider*}*Compilers.* These compilers implement the logic of computing resources for a specific *application* based on an *application* request and its *constraints*, check whether the request can be accepted, and they convert the *application* intent into a set of configurations that can be installed in the network substrate.

An {*Application*}{*Provider*}*Compiler* is in charge of compiling only a specific {*Application*}{*Provider*}*Intent.* For instance, the App Store can have two providers for a SD-WAN application, defined as Provider1 and Provider2. When the *Intent Translation and Negotiation* module receives a SD-WAN request, it generates two intent requests, namely *SDWANProvider1Intent* and *SDWANProvider2Intent.* They can be compiled only by the *SDWAN-Provider1Compiler* and the *SDWANProvider2Compiler*, respectively.

The {*Application*}{*Provider*}*Compilers* can interact with several services offered by a SDN controller. In particular, the recent SDN controllers, such as ONOS [9] or OpenDayLight [10], provide several services (i.e. APIs), which can be exploited by external software modules. Specifically, the {*Application*}{*Provider*}*Compilers* exploit both the SDN controller APIs, such as path computation, forwarding rule installation, event notifications, etc., and other services that were specifically designed for the App Store, namely the *Network Intent Primitives* and the *Resource*

Manager. The first ones represent a set of intent compilers for basic connectivity services, such as path provisioning, access control, optical provisioning, etc., which can be developed directly from the hosting ISP. The *{Application}{Provider}Compilers* can reuse these services without the need for re-develop them, reducing overhead in developing new application providers. The second service is the *Resource Manager*, which maintains network resource inventory and current allocations. The *{Application}{Provider}Compilers* rely on this component in order to check if an application request can be provisioned in the network based on its requirements.

5.3.1 Offer generation and negotiation

The App Store can generate multiple offers for an *application* request based on the number of registered *application providers*. This phase is defined as *offer generation and negotiation* and it is managed by the *Intent Translation and Negotiation* module. Specifically, this phase works as follows:

1. The *Intent Translation and Negotiation* module generates the specific intents for all *{Application}{Provider}Compilers* registered for a specific *application*. Then, it submits these intents to the associated compilers.

2. The *{Application}{Provider}Compilers* generate an offer for the application based on their own implementation and the available network resources. To understand the available resources, the compiler queries the *Resource Manager* and checks whether the resources may allow the *application* and its *constraints*. In the case of the resources are not enough, the compiler may propose an alternative solution based on a degradation of the requirements. For instance, an application requires 1 Gbps of bandwidth between two network endpoints. The compiler checks whether the path between the endpoints allows the requested capacity. In the case of a negative response, the compiler may calculate the amount of spare

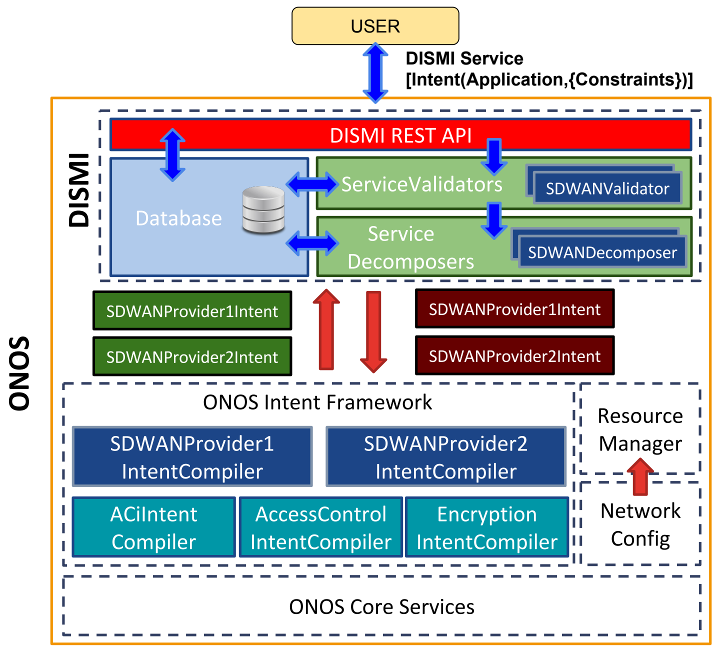


Figure 5.2: Software architecture for the ONOS controller.

capacity on the path and offer it as an alternative solution to the user.

3. The *Intent Translation and Negotiation* module collects all the offers from the *application compilers* and starts the *negotiation* phase with the user. It provides all offers to the user, who can select one of the provided offers or withdraw the *application* request.

4. If a user selects an offer, the *Intent Translation and Negotiation* module receives the selected offer, converts it to the associated $\{Application\}\{Provider\}Intent$, and forwards this to the $\{Application\}\{Provider\}Compiler$.

5. Finally, the compiler translates the intent to the configuration operations and allocates the network resources in the *Resource Manager*.

5.4 Software Architecture

As indicated in the reference architecture, the AppStore is built on top of an SDN controller. Several requirements imposed by the reference architecture are key in selecting the underlying SDN orchestrator, namely: (i) a RESTful API to receive application intents from the users; (ii) a module that can negotiate the requests for applications with the users; (iii) the translation of the intents into configuration operations, i.e. the intent compilers, and (iv) a network *resource manager*, which keeps track of the available resources in the network.

To this end, we rely on ONOS [9] as SDN controller for the implementation of the App Store prototype. This controller offers a modular architecture in which external software modules, i.e. OSGi [56] bundles, can be added and removed at runtime. ONOS provides a comprehensive intent framework [42], which offers the possibility to handle all the intent life cycle from the submission to the installation of the corresponding configurations in a network. In addition, new intent compilers can be inserted at runtime as part of the software modules. The NBI is based on a modified version of DISMI [59], an intent-based interface for requesting application-centric connectivity services. The overall software architecture is depicted in Fig. 5.2.

In this section, we discuss App Store implementation over ONOS. We also describe the modifications to the DISMI architecture, and demonstrate how the ONOS *Intent Framework* and *Core Services* are employed in the architecture.

5.4.1 Extending the DISMI intent interface

The DISMI intent interface provides an abstraction layer where application-centric connectivity service requests are translated into simpler intents understandable by a specific SDN controller implementation. The DISMI intents are composed of an *action* and several *constraints*. The *action* specifies the type of connectivity between endpoints (e.g., connection between two endpoints, multicast, multipoint, etc.) and the *constraints* represent its requirements, such as minimum bandwidth, maximum latency, minimum availability, etc.

DISMI provides a REST API based on Swagger [5] to receive application requests in the JSON format [66]. A Graphical User Interface (GUI) is provided in order to simplify the intent generation and management by the users.

The intents are processed by several *action Validators* and *Decomposers*. Every *action* is associated to a particular *Validator* and *Decomposer*, e.g., the *Mesh* action relies on the *MeshActionValidator* and *MeshActionDecomposer* classes. The *Validators* are in charge of checking the DISMI intent and reject it in the case of composition errors of the *actions* and the *constraints*. For instance, the *Mesh* action requires at least two *connectionPoints*. In the case of the mesh intent does not provide this information, it is rejected by DISMI. On the other side, the *Decomposers* are responsible for the generation of low-level intents from other high-level intents. They are in charge of converting the *actions* into intents understandable by the ONOS implementation. For instance, a user requires a mesh connection between several data centers distributed in different geographical locations. By exploiting the DISMI interface, the user can require a unique high level intent in the form of *Mesh(connectionPoint1, connectionPoint2,* $\cdots$ *)[bw=1 Gbps]*, in which the *connectionPoints* represent the data center endpoints (i.e. the IP address) and *bw* the bandwidth required. Then, this intent is validate and decomposed into simpler ones understandable by ONOS and, finally, they are submitted to the ONOS Intent Framework for the compilation and installation. For instance, a mesh connection is decomposed into a list of ONOS PointToPoint intents [67].

New *Validators* and *Decomposers* can be added in DISMI by exploiting its modular architecture. We use this feature to implement the *Intent Translation and Negotiation* module of the App Store. The DISMI software architecture has been modified and extended as shown in Fig. 5.2. In order to support new App Store applications and negotiation of the offers from the providers, the *action Validators* and *Decomposers* have been modified in order to become *application Validators* and *Decomposers*. We implemented the *SDWANValidator* and *SDWANDecomposer* which are responsible to manage the SD-WAN application intents in DISMI.

Unlike the original DISMI model, an *application* can be offered by several *providers*. The *application Decomposer* maintains the list of the *providers* to contact in the negotiation phase and the type of {*Intent*}{*Provider*}*Intents* that should be submitted. In this implementation, the *SDWANDecomposer* maintains a list of two providers to contact, Provider1 and Provider2. On receiving a request, it generates two intents, *SDWANProvider1Intent* and *SDWANProvider2Intent*, with the requested application constraints. These intents are submitted to the ONOS Intent Framework and DISMI waits for the generation of the offer from the two providers.

5.4.2 ONOS Intent Framework and Services

The ONOS Intent Framework manages the intent submission, compilation and the installation of the configurations on the network devices. In general, it is composed of several intent compilers, each one compiling a specific type of intent. ONOS provides a set of pre-defined intents representing a network connectivity action, such as the *HostToHostIntent* [68]. This intent represents a point-to-point connection between two hosts in the network. The compilation of this intent is associated only to the *HostToHostIntent-Compiler* for the translation to configurations.

The intent compilers interact with the other ONOS *Core Services* in order to obtain the information required for translating an intent into requisite configurations. Specifically, ONOS provides the *Path Service* [69] and the *Resource Service* [70], which represent the ONOS path computation element and the implementation of the App Store *Resource Manager*, respectively. The ONOS *Resource Service* provides a database to keep track of all available network resources (e.g., links, the ports of a device, etc.), their characteristics (e.g., latency, capacity, etc.) and consumptions. The intent compilers may exploit these service to retrieve information about the shortest path between two network endpoints and the status of the network resources to check whether an intent constraint can be satisfied. For example, consider an intent with a bandwidth constraint of 100 Mbps between two hosts in a network. The associated intent compiler queries the *Path-Service* and receives the shortest path connecting the two hosts. Then, the compiler checks if the path can support the requested capacity. Finally, in the case of a positive response, it allocates the requested capacity in the *Resource Service.*

The *Resource Service* must be initialized with the initial network configuration to identify the complete set of resources available, which is performed by exploiting the *Network Config* subsystem. This sybsystem provides an interface to load the network configuration via a REST API. For example, it can be used to configure the capacity of a device port or to annotate on a link a particular information, such as the latency, the availability or if it supports the traffic encryption.

By exploiting the current architecture of the ONOS Intent Framework, the *application compilers* of the App Store can be developed as ONOS intent compilers. We developed two *application compilers* for the SD-WAN use case, representing the service implementation of two providers, i.e. the *SD-WANProvider1Compiler* and the *SDWANProvider2Compiler*. They exploit several services and *Network Intent Primitives* from the ONOS controller.

5.4.3 Implementation of the Network Intent Primitives

In ONOS, the intent compilation may rely on more than one compiler in order to reduce the development complexity. ONOS provides many basic compilers for simplifying the generation of configurations from the intents. For example, the *LinkCollectionIntentCompiler* [71] can be used to translate a set of links into a single adjacency, and every compiler can use it to avoid the re-implementation of the same feature. We exploited this property in order to create a set of *Network Intent Primitives* for the App Store implementation. They are defined as follow:

AciIntentCompiler. This intent compiler can be used to request application-centric connectivity in the network. It translates and analyzes the *Application-Centric Intents* (ACiIntents). This type of intent represents the request for a connection between two endpoints in a network with a specific set of application-centric constraints, such as bandwidth, latency, availability, etc. This compiler searches for the shortest paths between the network endpoints and uses the ONOS *Resource Service* to evaluate if all intent constraints are satisfied on this path. In case no path can satisfy all the constraints, the compiler calculates a set of alternative solutions, i.e. the constraints that the possible shortest paths are able to satisfy.

AccessControlIntentCompiler. This intent compiler offers the generation of rules for network access control. We create a new constraint, defined as *DenyAccessContraint*. This constraint can be exploited by the user to create a blocking policy for a particular IP address or a subnet. The compiler generates a set of forwarding rules that drop the traffic.

EncryptionIntentCompiler. It checks if two network endpoints can be connected by an encrypted path and generates the forwarding rules accordingly. It exploits the *Resource Service* to query the information about whether a link supports an encryption scheme. We provide a simplified implementation, in which we annotate on the links only a value (i.e. encryption) to state whether an encryption scheme is supported. This implementation can be extend also to support optical encryption schemes and generate the commands to install an optical encrypted path.

5.4.4 The negotiation phase

The implementation of the negotiation phase in ONOS works as follows:

1. The user requests a SDWAN application between two network endpoints with several constraints (e.g. bw=1 Gbps).

2. DISMI validates and decomposes the request into two intents, defined as *SDWANProvider1* and *SDWANProvider2*, since the *SDWANDecomposer* has two registered providers. Then, it submits the intents to the ONOS Intent Framework.

3. The *SDWANProvider1* and *SDWANProvider2* compilers calculate the initial offer by querying the *Path Service* and the ONOS *Resource Manager* to check whether the application can be installed while respecting the constraints. After that, they generate a notification for the initial intents, defined as NEGOTIATION_REQUIRED, and generate a new set of {*Application*}{*Provider*}*Intents* that are saved in the ONOS Intent Framework database.

4. DISMI receives these notifications and gets the offers from the providers by querying the API of the ONOS Intent Framework. It sends the intents to the GUI, where the user can select one offer and submit it to DISMI, which forwards the request directly to the ONOS Intent Framework.

5. Finally, the selected *application compiler* translates the intent into configurations and allocates the network resources in the *Resource Manager*.

5.5 Performance Evaluation

The preliminary performance evaluation of the App Store has been performed by considering the processing time required for: (i) the generation of an application offer from the providers, (ii) the intent compilation, and (iii) the overall installation of an *application*. These times are evaluated by varying the number of *application* requests already provisioned in the network. In this way, we can evaluate the scalability of the App Store implementation.

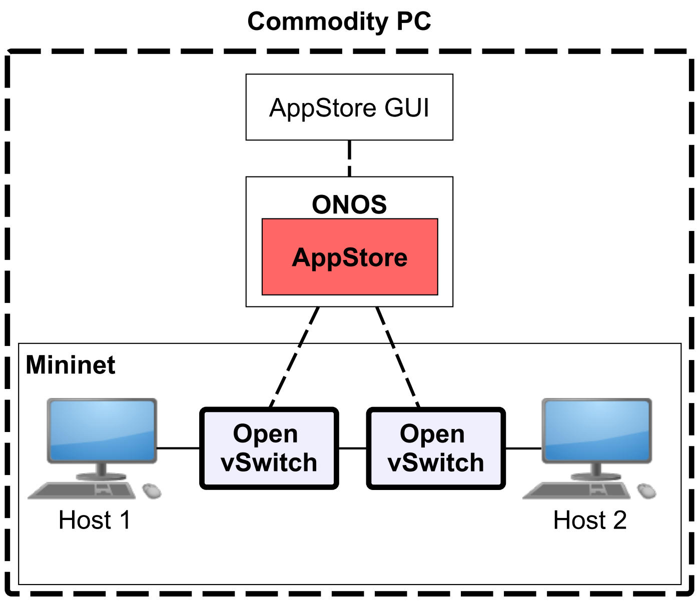


Figure 5.3: Experiment setup for the evaluation.

5.5.1 Test methodology

Experimental setup. For the experiments, we use an emulated network environment based on Mininet [34]. The test setup is depicted in Fig. 5.3. The network is controlled by a modified version of the ONOS controller to host the App Store. We run the App Store and Mininet on a commodity PC equipped with a Intel i7-5600U quad-core CPU running at 2.60GHz and 16GB of DDR3 memory working at 1600Mhz.

Methodology. The App Store implementation adds several new modules to the ONOS controller, which require an evaluation of their processing time. In particular, we evaluate the processing time for the DISMI framework to generate the requests for the providers, the generation of an offer from one of the *SD-WAN compilers*, the compilation time, and the overall *application* installation time.

Since the App Store is part of the ONOS controller, we can easily retrieve the processing time by exploiting the logging system. We generate the log messages at the beginning and the end of the methods inside the modules that we want to test. Then, we evaluate the time delta between the first and the second log message. The results are based on an average of ten requests.

The experiments are executed by submitting SD-WAN application requests without time expiration. We start from an empty state of the App Store, in which there are no application requests provisioned. Then, we increase the number of provisioned applications in the App Store by steps of 500 up to 2000.

5.5.2 Results and discussion

The processing time required to decompose the application requests into the providers specific intents is shown in Fig. 5.4. These tests consider the time required from the receipt of an user request to the intent generation for the application providers. The results show a slight increase in the processing time with respect to the number of provisioned application by the controller. This is mainly caused by the overall controller load. Indeed, the DISMI application decomposition mainly depends on the number of provider that should be contacted. In these tests, the number of application providers is fixed to two.

After the application decomposition, DISMI requests an offer from the registered application providers. Fig. 5.5 shows the average time required

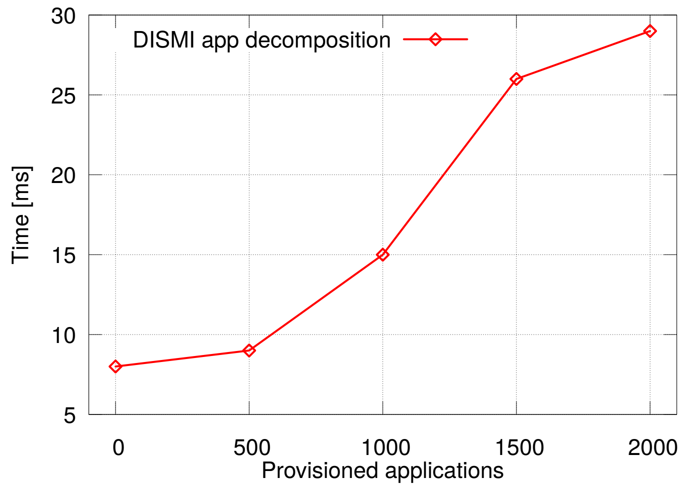


Figure 5.4: DISMI processing time for decomposing the SD-WAN application request.

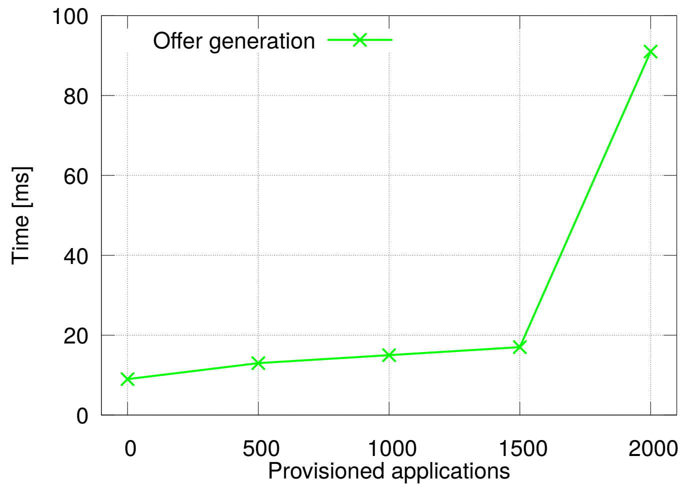


Figure 5.5: Processing time for generating an offer from the SD-WAN compilers.

by the *SDWANProvider1Compiler* to generate the initial application offer. We do not provide the *SDWANProvider2Compiler* results since they show overlapping values. Indeed, both the *SDWANProvider1Compiler* and the *SDWANProvider2Compiler* have the same compilation logic at the time of writing. The processing time for the offer generation by the SD-WAN compilers slowly increases between 0 and 1500 provisioned applications. After

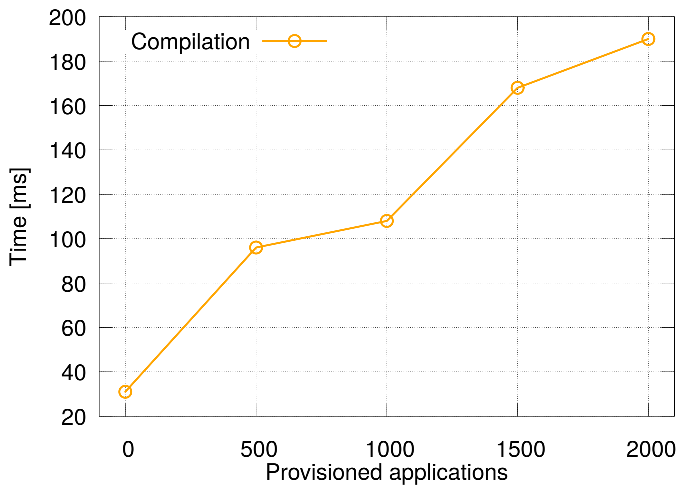


Figure 5.6: Processing time required by the compiler to generate the forwarding rules.

1500, the processing time shows a huge increase (337%). This happens because the offer generation depends on the lookup of the *resource manager*. Increasing the number of provisioned applications leads to a higher number of entries registered in the *resource manager* database. Thus, the lookup time required by the compilers increases with respect to the number of provisioned applications. In addition, a higher number of previsioned applications leads to more forwarding rules installed into the network, thus increasing the overall SDN controller load for maintaining the traffic statistics.

The compilation time required by the *SDWANProvider1Compiler* is presented in 5.6. It represents the processing time required for the conversion of an application intent into configurations. For these tests, we manually select the *Provider1* offer on the GUI, ensuring that the compilation is always performed by the *SDWANProvider1Compiler*. Different from the offer generation test, the compilation phase includes the time required by the compiler to lookup and also write the network resources in the *resource manager*. As expected, the compilation time constantly increases with respect to the number of provisioned applications due to an increased overhead

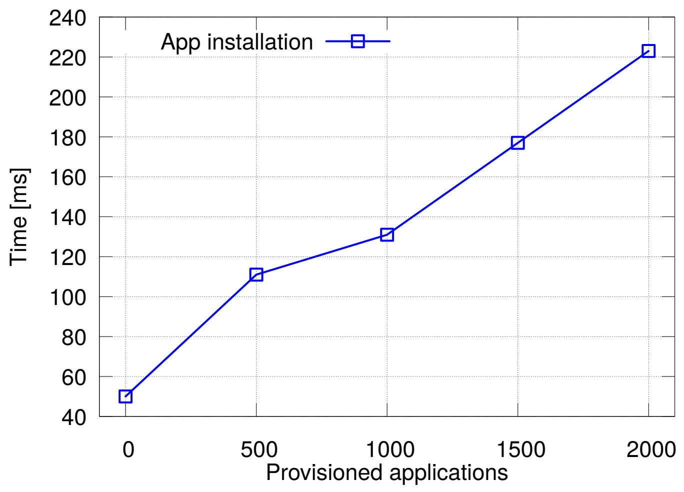


Figure 5.7: Overall time required to install an application in the network.

for operations with the *resource manager*. Compilation time also requires interactions with other systems outside of ONOS, and is larger with respect to the sum of decomposition and offer generation times.

Fig. 5.7 shows the overall installation time from the receipt of an application request to the submission of the forwarding rules in the network. Mainly, the installation phase depends on the decomposition and compilation times, while processing times have very little variation at across the load spectrum analyzed.

Chapter 6

Conclusions

This thesis studied three types of interaction between business applications, such as trading platforms for banks, e-commerce, video streaming services, etc., applications for SDN controllers, and SDN networks. In particular, the following interactions are analyzed (i) *interactions between SDN controller applications and networks*, (ii) *interactions between business applications and SDN networks*, and *interactions between business applications, SDN controller applications and networks.*

In the first interaction, this work showed how an application for SDN controllers can interact with a SDN network in order to overcome the limited memory capacity of SDN devices. We presented a novel *memory swapping* mechanism for SDN controllers that aims at improving the performance and the reliability of the network. The mechanism, which is part of a SDN controller application, ensures that the flow entries needed to smoothly manage the network, i.e., the most frequently matched and the most recent ones, are stored in the fast TCAM memory of the switches even when the space on such memory reach its limit. Performance evaluations showed an increase in the throughput of network devices in the case of memory overflows. This interaction demonstrates that a SDN application is able to intercept potential network issues, i.e. an alarm from a network device, and provide the right decisions to resolve it.

In the second interaction, this thesis analyzed how business applications and SDN controllers can cooperate together to reduce the connectivity service blocking probability in an application-centric network. The application-centric paradigm proposes the provisioning of connectivity services by respecting all the business application requirements, such as bandwidth, latency, availability, etc. The application communicates its requirements to a SDN controller, which blocks the service request in the case of the network cannot provide all of them. We defined an *application-centric negotiation* scheme, which offers the possibility to the SDN controller to provide a set of alternative solutions, in the case of network resource scarcity, and the possibility for applications to provide a feedback for the provisioning. We demonstrate a lower blocking probability of service requests with limited degradation of the application requirements.

The last interaction shows how business applications and SDN controller applications can cooperate together to enable a multi-service selection model on a SDN network. We proposed an SDN App Store, which provides a collection of SDN applications, developed by different providers, for a specific connectivity service (e.g. SD-WAN, VPN, etc.). The business applications or the users can request a connectivity service and receive several offers from the SDN controller applications implementing the service.

Bibliography

[1] (2015) Deep Shift 21 Ways Software Will Transform Global Society. Last access 14-12-2017. [Online]. Available: http://www3.weforum.org/docs/WEF_GAC15_Deep_Shift_Software_Transform_Society.pdf

[2] A. Marsico, R. Doriguzzi-Corin, and D. Siracusa, "Overcoming the Memory Limits of Network Devices in SDN-enabled Data Centers," *to appear in IEEE/IFIP IM 2017.*

[3] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[4] (2017) Open Networking Foundation. [Online]. Available: http://www.opennetworking.org

[5] Swagger RESTful API framework. Last access 14-12-2017. [Online]. Available: http://swagger.io/

[6] (2015) Intent As The Common Interface to Network Resources. [Online]. Available: http://www.ietf.org/mail-archive/web/i2nsf/current/pdfEhAfL7kT9F.pdf

[7] J. Schönwälder, M. Björklund, and P. Shafer, "Network configuration management using netconf and yang," *Comm. Mag.*, vol. 48, no. 9, pp. 166–173, Sep. 2010. [Online]. Available: http://dx.doi.org/10.1109/MCOM.2010.5560601

[8] RFC 6020. Last access 24-01-2018. [Online]. Available: https://tools.ietf.org/html/rfc6020

[9] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar, "ONOS: Towards an Open, Distributed SDN OS," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, Chicago, Illinois, USA, 22 Aug. 2014.

[10] (2017) The OpenDayLight Project. [Online]. Available: http://www.opendaylight.org/

[11] "MMS GitHub repo." [Online]. Available: https://github.com/fp7-netide/Tools.git

[12] "OpenFlow Switch Specification 1.5.1." [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.1.pdf

[13] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (CAM) circuits and architectures: A tutorial and survey," *IEEE Journal of Solid-State Circuits*, vol. 41, no. 3, pp. 712–727, March 2006.

[14] R. Doriguzzi-Corin, D. Siracusa, E. Salvadori, and A. Schwabe, "Empowering Network Operating Systems with Memory Management Techniques," in *IEEE/IFIP Network Operations and Management Symposium*, Istanbul, Turkey, 25-29 Apr. 2016.

[15] Ryu SDN Framework. [Online]. Available: http://osrg.github.io/ryu/

[16] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "CacheFlow: Dependency-Aware Rule-Caching for Software-Defined Networks," in *Symposium on SDN Research*, Santa Clara, CA, USA, 14-15 Mar. 2016.

[17] Y. Nakagawa, K. Hyoudou, C. Lee, S. Kobayashi, O. Shiraki, and T. Shimizu, "DomainFlow: Practical Flow Management Method Using Multiple Flow Tables in Commodity Switches," in *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, Santa Barbara, California, USA, 9-12 Dec. 2013.

[18] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling Flow Management for High-performance Networks," in *Proceedings of the ACM SIGCOMM 2011 Conference*, Toronto, Ontario, Canada, 15-19 Aug. 2011.

[19] A. Vishnoi, R. Poddar, V. Mann, and S. Bhattacharya, "Effective Switch Memory Management in OpenFlow Networks," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, Mumbai, India, 26-29 May 2014.

[20] S. Banerjee and K. Kannan, "Tag-In-Tag: Efficient Flow Table Management in SDN Switches," in *10th International Conference on Network and Service Management (CNSM) and Workshop*, Rio de Janeiro, BR, 17-21 Nov. 2014.

[21] W. Braun and M. Menth, "Wildcard Compression of Inter-Domain Routing Tables for OpenFlow-Based Software-Defined Networking," in *2014 Third European Workshop on Software Defined Networks*, 2014.

[22] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with DIFANE," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4, p. 351, 2010.

[23] M. Rifai, N. Huin, C. Caillouet, F. Giroire, D. Lopez-Pacheco, J. Moulierac, and G. Urvoy-Keller, "Too Many SDN Rules? Compress Them with MINNIE," in *2015 IEEE Global Communications Conference (GLOBECOM)*, San Diego, CA, USA, 6-10 Dec. 2015.

[24] X.-N. Nguyen, D. Saucez, C. Barakat, and T. Turletti, "Optimizing rules placement in openflow networks: Trading routing for better efficiency," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, Chicago, Illinois, USA, Aug. 2014.

[25] Y. Kanizo, D. Hay, and I. Keslassy, "Palette: Distributing tables in software-defined networks," in *IEEE INFOCOM 2013 - IEEE Conference on Computer Communications*, Torino, Italy, 14-19 Apr. 2013.

[26] J. M. Lucas, M. S. Saccucci, R. V. Baxley, Jr., W. H. Woodall, H. D. Maragh, F. W. Faltin, G. J. Hahn, W. T. Tucker, J. S. Hunter, J. F. MacGregor, and T. J. Harris, "Exponentially weighted moving average control schemes: Properties and enhancements," *Technometrics*, vol. 32, no. 1, pp. 1–29, Jan. 1990.

[27] A. Marsico, R. Doriguzzi-Corin, M. Gerola, D. Siracusa, and A. Schwabe, "A Non-disruptive Automated Approach to Update SDN Applications at Runtime," in *IEEE/IFIP Network Operations and Management Symposium*, Istanbul, Turkey, 25-29 Apr. 2016.

[28] "ONOS Interface FlowRuleService." [Online]. Available: http://api.onosproject.org/1.6.0/org/onosproject/net/flow/FlowRuleService.html

[29] "ONOS Interface OpenFlowController." [Online]. Available: https://github.com/opennetworkinglab/onos/blob/onos-1.6/protocols/openflow/api/src/main/java/org/onosproject/openflow/controller/OpenFlowController.java

[30] "ONOS Interface StorageService." [Online]. Available: http://api.onosproject.org/1.6.0/org/onosproject/store/service/StorageService.html

[31] "ONOS Interface PacketService." [Online]. Available: http://api.onosproject.org/1.6.0/org/onosproject/net/packet/PacketService.html

[32] "ONOS Interface EventuallyConsistentMap." [Online]. Available: http://api.onosproject.org/1.6.0/org/onosproject/store/service/EventuallyConsistentMap.html

[33] "NEC IP8800 OpenFlow Networking." [Online]. Available: https://support.necam.com/SDN/ip8800/

[34] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible Network Experiments Using Container-based Emulation," in *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, Nice, France, 10-13 Dec. 2012.

[35] The Open vSwitch Project. [Online]. Available: http://openvswitch.org/

[36] A. Dainotti, A. Pescapé, P. S. Rossi, F. Palmieri, and G. Ventre, "Internet traffic modeling by means of Hidden Markov Models," *Computer Networks*, vol. 52, no. 14, pp. 2645 – 2662, 2008.

[37] "HP OpenFlow 1.3 Administrator Guide," Hewlett-Packard, p. 15, Jun. 2015, rev. 2.

[38] "ONOS ReactiveForwarding." [Online]. Available: https://github.com/opennetworkinglab/onos/blob/onos-1.6/apps/fwd/src/main/java/org/onosproject/fwd/ReactiveForwarding.java

[39] H. Perros, *Networking Services: QoS, Signaling, Processes.* CreateSpace Independent Publishing Platform, 2014.

[40] V. Lopez, D. Konidis, D. Siracusa, C. Rozic, I. Tomkos, and J. P. Fernandez-Palacios, “On the benefits of multilayer optimization and application awareness,” *Journal of Lightwave Technology*, vol. 35, no. 6, pp. 1274–1279, March 2017.

[41] M. Savi, F. Pederzolli, and D. Siracusa, “An Application-Aware Multi-Layer Service Provisioning Algorithm based on Auxiliary Graphs,” *Proceedings of the OFC 2017 Conference*, 2017, to appear.

[42] ONOS Intent Framework. Last access 24-01-2018. [Online]. Available: https://wiki.onosproject.org/display/ONOS/Intent+Framework

[43] ACINO Multi-layer Network Orchestrator. [Online]. Available: https://github.com/ACINO-H2020/network-orchestrator

[44] M. Jarschel, F. Wamser, T. Hohn, T. Zinner, and P. Tran-Gia, “SDN-Based Application-Aware Networking on the Example of YouTube Video Streaming,” in *2013 Second European Workshop on Software Defined Networks*, Berlin, DE, 10-11 Oct. 2013.

[45] Y. Yiakoumis, S. Katti, T. Huang, N. McKeown, K. Yap, and R. Johari, “Putting home users in charge of their network,” in *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, Pittsburgh, PA, USA, 5-8 Sep. 2012.

[46] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi, “Participatory networking: An api for application control of sdns,” *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 327–338, Aug. 2013. [Online]. Available: http://doi.acm.org/10.1145/2534169.2486003

[47] X. Wang and H. Schulzrinne, “Integrated resource negotiation, pricing, and QoS adaptation framework for multimedia applications,” *IEEE*

Journal on Selected Areas in Communications, vol. 18, no. 12, pp. 2514–2529, 2000.

[48] (2017) Amazon EC2 Spot Instances. [Online]. Available: https://aws.amazon.com/ec2/spot/

[49] (2017) Google Preemptible Virtual Machines. [Online]. Available: https://cloud.google.com/preemptible-vms/

[50] S. Venugopal, X. Chu, and R. Buyya, "A negotiation mechanism for advance resource reservations using the alternate offers protocol," Enschede, NL, 2-4 Jun. 2008.

[51] A. Rubinstein, "Perfect equilibrium in a bargaining model," *Econometrica*, vol. 50, no. 1, pp. 97–109, 1982.

[52] S. S. Savas, M. F. Habib, M. Tornatore, and B. Mukherjee, "Exploiting degraded-service tolerance to improve performance of telecom networks," in *OFC 2014*, San Francisco, CA, USA, 9-14 Mar. 2014, pp. 1–3.

[53] Z. Zhong, J. Li, N. Hua, G. B. Figueiredo, Y. Li, X. Zheng, and B. Mukherjee, "On qos-assured degraded provisioning in service-differentiated multi-layer elastic optical networks," in *2016 IEEE Global Communications Conference (GLOBECOM)*, 4-8 Dec. 2016, pp. 1–5.

[54] M. Pham and D. B. Hoang, "Sdn applications - the intent-based northbound interface realisation for extended applications," in *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, June 2016, pp. 372–377.

[55] H. Zhu, H. Zang, K. Zhu, and B. Mukherjee, "Dynamic traffic grooming in WDM mesh networks using a novel graph model," in *Global Telecommunications Conference (IEEE GLOBECOM)*, Tapei, Taiwan, 17-21 Nov. 2002.

[56] "OSGi Alliance." [Online]. Available: https://www.osgi.org/

[57] P. Pavon-Marino and J.-L. Izquierdo-Zaragoza, "Net2plan: An open source network planning tool for bridging the gap between academia and industry," *IEEE Network*, vol. 29, no. 5, pp. 90–96, September-October 2015.

[58] F. Rambach, B. Konrad, L. Dembeck, U. Gebhard, M. Gunkel, M. Quagliotti, L. Serra, and V. Lopez, "A multilayer cost model for metro/core networks," *IEEE/OSA Journal of Optical Communications and Networking*, vol. 5, no. 3, pp. 210–225, March 2013.

[59] P. Sköldström, S. Junique, A. Ghafoor, A. Marsico, and D. Siracusa, "Dismi - an intent interface for application-centric transport network services," in *19th International Conference on Transparent Optical Networks (ICTON)*, Girona, Spain, 2-6 Jul. 2017.

[60] P. Pawluk, B. Simmons, M. Smit, M. Litoiu, and S. Mankovski, "Introducing STRATOS: A Cloud Broker Service," in *IEEE Fifth International Conference on Cloud Computing.* Honolulu, HI, USA: IEEE, 24-29 Jun. 2012.

[61] S. Yangui, I. Marshall, J. Laisne, and S. Tata, "CompatibleOne: The Open Source Cloud Broker," *Journal of Grid Computing*, vol. 12, pp. 93–109, 2014.

[62] S. Sundareswaran, A. Squicciarini, and D. Lin, "A brokerage-based approach for cloud service selection," in *2012 IEEE 5th International Conference on Cloud Computing*, Honolulu, HI, USA, 24-29 Jun. 2012.

[63] Why you should consider networking as a service. Last access 14-12-2017. [Online]. Available: https://www.cisco.com/c/en/us/solutions/enterprise-networks/network-as-service-naas.html

[64] D. Di Sorte and G. Reali, "Pricing and brokering services over interconnected IP networks," *Journal of Network and Computer Applications*, vol. 28, no. 4, pp. 249–283, nov 2005.

[65] J. R. Lane and A. Nakao, "Path brokering for end-host path selection," in *Proceedings of the ACM CoNEXT Conference*, Madrid, Spain, 9-12 Dec. 2008.

[66] JavaScript Object Notation. Last access 14-12-2017. [Online]. Available: http://www.json.org/

[67] ONOS PointToPoint Intent. Last access 14-12-2017. [Online]. Available: http://api.onosproject.org/1.11.0/org/onosproject/net/intent/PointToPointIntent.html

[68] ONOS PointToPoint Intent. Last access 14-12-2017. [Online]. Available: http://api.onosproject.org/1.11.0/org/onosproject/net/intent/HostToHostIntent.html

[69] ONOS Interface PathService. Last access 14-12-2017. [Online]. Available: http://api.onosproject.org/1.11.0/org/onosproject/net/topology/PathService.html

[70] ONOS ResourceService. Last access 14-12-2017. [Online]. Available: http://api.onosproject.org/1.11.0/org/onosproject/net/resource/ResourceService.html

[71] ONOS LinkCollection Intent. Last access 14-12-2017. [Online]. Available: http://api.onosproject.org/1.11.0/org/onosproject/net/intent/LinkCollectionIntent.html