An Automated Negotiation Framework for Application-Aware Transport Network Services

Antonio Marsico[a,*], Marco Savi[b], Domenico Siracusa[b], Elio Salvadori[b]

[a] *British Telecom, Adastral Park, Ipswich, UK*
[b] *Fondazione Bruno Kessler, Trento, Italy*

Abstract

Application-centric networking proposes a novel approach to provision connectivity services in transport networks based on application-specific requirements. By exploiting this paradigm, an application can request a service with several requirements to a generic control and management plane, which can leverage this information to differentiate a service on the overall network layers. However, an application-centric provisioning can lead to higher blocking probability when satisfying multiple requirements at the same time, thus impacting negatively both network operators and applications. This paper proposes a framework for the *negotiation* of connectivity services between a transport network and the applications running on top. In particular, an application can communicate its requirements to it and, in case of resource scarcity (either in term of bandwidth, latency, availability, etc.), the network can offer alternative solutions with a *degraded* service quality. The effectiveness of the proposed approach is proved by the lower blocking probability experienced by the service requests against a standard approach lacking negotiation capabilities, without in turn causing significant service degradation to the applications. In addition, the paper describes the software architecture behind the negotiation framework and

*Corresponding author
Email address: antonio.marsico@bt.com (Antonio Marsico)

[1] The research leading to these results has received funding from the European H2020 Framework Programme under grant agreement no. 645127 ACINO project.

[2] A preliminary version of this paper appeared in 2018 Optical Fiber Communication Conference (OFC) [1].

its implementation on top of the ONOS SDN controller.

Keywords: Network Optimization; Service Negotiation; Software-Defined Networking; Transport Networks.

---

1. Introduction

Applications for industry verticals with diverse and stringent requirements are characterizing modern Internet traffic. Banking systems, smart cities and grids are a few examples of this plethora of applications [2]. Their internet traffic is extremely heterogeneous and characterized by diverse requirements in terms of bandwidth, latency, availability, etc., defining the type of connectivity service that a network should be able to provide.

In this context, transport networks are required to offer ad-hoc connectivity services and adapt their configuration based on specific application requirements. To this end, the application-centric networking [3] is an emerging paradigm that aims at catering to multiple application requirements during the provisioning of a connectivity service. The applications can communicate their requirements to a generic *control and management plane* (i.e. a Software-Defined Networking (SDN) network controller), which offers provisioning algorithms for satisfying the application requirements over the different network layers (i.e. IP/MPLS and optical). The communication between the applications and the SDN controller can be performed by means of a well-defined North-bound Interface (NBI) (i.e. an *Intent-based* interface [4]).

Recent studies (e.g. [5, 3, 6, 7, 8, 9]) 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, [6] shows the advantages of considering a number of application requirements in addition to simple bandwidth when applications' *service requests* must be provisioned. This work proves that it is possible to achieve service blocking probabilities similar to an application-unaware scheme in low network load, while also en-

suring 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 utilization, the service acceptance ratio experiences a reduction that negatively impacts on the revenues of both network operators, which cannot accommodate new service requests while meeting all the application requirements, and customers, which have their service blocked.

In this work, we introduce a novel interaction between applications and networks that allows the reduction of the service blocking probability based on application feedback. We define this interaction as *application-aware service negotiation*. The *negotiation* scheme offers the possibility to find an agreement between applications and networks for the provisioning of a downgraded service with *looser* requirements. In particular, the considered application requirements are the bandwidth (b), the latency (l), and the availability (a), but the scheme could be easily extended to consider additional ones. The enhanced provisioning 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, an application can receive a predictable service degradation and avoid a block of its request.

The contributions of this paper can be summarized as follows:

- **Definition** of a intent-based negotiation scheme between applications and networks. We propose a negotiation scheme based on the degradation of application requirements. We extend an algorithm for the provisioning of *Application-Aware services* in order to support the negotiation scheme and to search for alternative provisioning solutions for applications. In addition, we propose an algorithm to let the applications automatically select or reject the alternative solutions offered by the network.

- **Implementation** of the negotiation scheme on top of the ONOS SDN network controller [10]. We rely on the ONOS Intent Framework [11] REST APIs to support the negotiation scheme. We evaluate the processing time of our implementation and we demonstrate a limited overhead.

Being a fully working proof of concept, we have released its code as open source [12], publicly available for researchers and experimenters that are willing to test it or further enrich it. This code is part of the software outcome of the EU H2020 ACINO project [13].

- **Evaluation** of the *application-aware negotiation* scheme performance on both the network and the application sides in a simulation environment. We show that (i) our negotiation scheme is able to significantly reduce network service blocking probability while leading to predictable (application-endorsed) downgrades in service quality and (ii) jointly relaxing multiple application requirements can lead to counter intuitive effects on how each of them is impacted by the relaxation of the others.

This paper is structured as follows: the state of the art is described in Section 2, while Section 3 proposes the network and application algorithms and the negotiation scheme. The software architecture is proposed in Section 4. Then, Section 5 shows the performance evaluation of our negotiation mechanism on top of a real Internet Service Provider (ISP) transport network by means of simulations and the ONOS implementation. Section 6 closes the paper.

2. Related Work

The negotiation of application-centric services requires (i) a strategy to reduce the service blocking probability in a network by means of *service (or application) requirements degradation*, (ii) a *negotiation model* to find an agreement between applications and networks on such requirements, and (iii) a communication system that provides a proper *interaction* between the involved parties. In the state of the art, there are several works that independently discuss all these aspects. In the following paragraphs, we discuss the main related work for each one of them.

2.1. Degradation of service requirements

This strategy has been proposed to increase the number of service request that can be accommodated in the network. The application requirements can be degraded to looser values than the initial request. Different works [14, 15, 16, 17] propose some service admission strategies based on the degradation of the sole bandwidth requirement in the case of network failures and/or network congestion. However, all these works propose a degradation that is unilaterally decided by the network without any feedback from applications. In addition, evaluating the degradation of bandwidth without considering other application-specific requirements is a common limitation of these works. In this paper, we try to fill both these gaps.

2.2. Negotiation models

In the state of the art, there are a few examples of negotiation mechanisms, with strong focus on cloud computing. In [18], the authors propose a bandwidth negotiation mechanism based on a *price/service trade-off* depending on network congestion. An application chooses an alternative solution (in terms of guaranteed bandwidth) based on a utility function. However, this work does not consider any other requirement rather than the bandwidth (e.g., availability, latency, etc.). Another type of negotiation model that can be found in literature is the *auction* [19, 20]. The users can create economical offers to request the provisioning e.g. 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 work proposes SNAP [21], a Service Level Agreement (SLA) negotiation protocol to allocate computational resources (e.g., CPU, RAM, etc.) and execute tasks on a cloud computing environment. Yet another type of negotiation mechanism is presented in [22]. In this work, the authors propose a scheme for negotiating computational resources based on the *Alternate Offers Protocol* [23], where a requester can make a counter offer to a resource manager system. The negotiation finishes when the application and the resource manager find an agreement

for the execution of the requested task.

The negotiation of application-centric services proposed in our paper differs from these mechanisms for two main reasons: (i) it is specifically tailored to tranport networking (instead of cloud computing) and (ii) our proposed negotiation scheme is triggered only in the case of resource scarcity.

2.3. Interaction between applications and networks

In the SDN ecosystem, many works discussed how applications and networks can interact. Specifically, they focus on how it possible to improve the Quality of Service (QoS) of application-generated traffic by making the network aware of what are the requested connectivity requirements and by enabling a feedback mechanism with respect to the experienced traffic treatment. Then, a network control and management plane (i.e., an SDN controller) can exploit this information to change the network configuration in real time to increase the QoS experienced by applications/users. For instance, paper [24] shows that a video streaming application experiences a significant increase in the throughput if the application provides feedback about the left amount of its video buffer. Paper [25] proposes a framework to ensure high QoS of real-time applications, such as online gaming. Each real-time application interacts with an SDN controller when it requires high traffic priority, and the controller is able to modify the network configuration accordingly. In [26], the authors present a framework where a user can directly interact with a browser-based Graphical User Interface (GUI) and choose which application traffic, within several pre-defined ones, they want to prioritize in the network. The SDN controller receives the requests from the users and translates them into forwarding rules for the network. Finally, paper [27] proposes a framework to to let the users request specific services from the network, such as bandwidth limitation or access control (e.g. firewalling), and the possibility to schedule a request for a certain amount of time. However, even though such previous work paves the way towards a tight interaction between applications and networks, all these papers only exploit this interaction to improve the application QoS, without any focus on network performance.

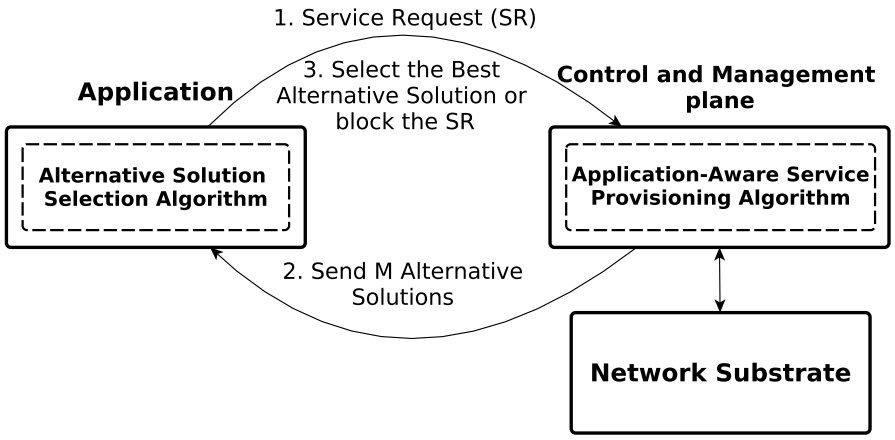

Figure 1: Application/network negotiation scheme and related algorithms.

In our work, we instead propose a scheme where a proper application/network interaction leads to service blocking probability reduction without substantially compromising the QoS of applications.

3. Service Negotiation Scheme

This section presents our scheme for the negotiation of application-aware services. As general model, the negotiation is based on a controlled degradation of the requested application requirements. The application requests a connectivity service to the network and, in case the service cannot be provisioned, it gets back several possible alternatives (in terms of guaranteed requirements), analyzes them and provides a feedback on the best one (according to its preferences) to be provisioned.

The negotiation scheme is divided into two algorithmic blocks (Fig. 1). On the network side, we rely on an extended version of the *Application-Aware Service Provisioning Algorithm* [6], which includes the negotiation features. The algorithm is implemented as part of a generic *Control and Management plane* that controls the substrate network, e.g. a hierarchical SDN controller. On the application side, an algorithm is designed to autonomously decide among the

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[4].

As already mentioned, the applications require connectivity services to the network in the form of *service requests* (SR, i.e., intents). We adopt a similar formalism as the one presented in [6]. It defines every service request as a tuple $SR = \{s, d, b, l, a\}$, in which s and d represent the source/destination node identifiers (i.e., endpoints), b the minimum required bandwidth, l the maximum allowed latency and a the minimum tolerable path availability. 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*. b, l and a are the *application requirements*. In [6], the Application-Aware Service Provisioning Algorithm evaluates the application requirements for each service request, and aims at finding a path meeting all the application requirements (i.e., an *application-aware path*). If an application-aware path can be provisioned, the service request is *accepted*. Otherwise, the service request is *blocked*.

Our negotiation scheme extends the algorithm presented in [6] by defining a new possible outcome of a service request between applications and networks: when it is not possible to provision an application-aware path for a service request, meaning that some of the requested application requirements cannot be guaranteed, a *negotiation* phase is started. This happens when the limited network resources have already been provisioned to meet other service requests with strict requirements.

For example, a service request generated by an application with a stringent latency requirement may find the shortest paths busy because of traffic generated by already-provisioned service requests with a similar requirement, but may be willing to negotiate a looser guaranteed latency instead of having its request blocked. In this way, the application can achieve a controlled service degradation and can modify its behavior based to the new guaranteed application requirements. In order to inform the network that an application is willing

to negotiate, the service request tuple reported above has been extended to carry also the information on which application requirements are *negotiable*: 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 Application-Aware Service Provisioning Algorithm of which requirements can be negotiated.

The algorithm thus offers a set of *alternative solutions* (AS) to the application, in which the negotiable application requirements can have looser values than the ones in the original service request and that the network can meet. The tuple $\overline{AS} = \{AS_1, AS_2, \cdots, AS_M\}$ represents all the possible alternative solutions where, in our specific case, $AS_j = \{b_{n,j}, l_{n,j}, a_{n,j}\}$ includes the values that the network can guarantee for each considered application requirement. For example, if a service request requires a specific value of l with $n_l = true$, the algorithm can offer a new value $l_n > l$, meaning that it can provide an application-aware path meeting such looser latency requirement. For more details on how the Application-Aware Service Provisioning Algorithm works and how the alternative solutions are generated see Section 3.1)

On the application side, the alternative solutions need to be analyzed to find which is the best for the application. This task is performed by the Alternative Solution Selection Algorithm. In our model, every application maintains 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 would like to have guaranteed from the network (which are thus included in the original service request SR, meaning that $b = b_p$, $l = l_p$ and $a = a_p$), while the second represents the *threshold values* that the application is willing to accept in case the negotiation phase is started, and are not disclosed to the network.

For instance, consider a company (e.g. a bank) that requires a seamless virtual machine migration between two end-points. Standard virtual machine migration techniques require a maximum latency of $l = l_p = 10$ ms. However the virtual machine can be migrated, with reduced performance, also in case of

network delays up to $l_t = 150$ ms.

As already explained, an application requirement is negotiable only if the related flag $n_{b,l,a} = true$. In that case, the threshold value is looser than preferable one (i.e., $b_p > b_t$, $l_p < l_t$, $a_p > a_t$). Conversely, if a requirement is not negotiable, i.e., $n_{b,l,a} = false$, the preferable and threshold values are the same (i.e., $b_p = b_t$, $l_p = l_t$, $a_p = a_t$). Starting from the information on preferable and least acceptable values for the application requirements, the Alternative Solution Selection Algorithm takes its choice on the *best* alternative solution (for more details on how the algorithm works see Section 3.2), or reject all of them. If all the alternative solutions are rejected, the service request is blocked.

Note that, since the network does not have any knowledge on the least-acceptable values for the application requirements, it cannot bias its choice to provide, as unique alternative solution, exactly such looser values to the application. In this way, the application can potentially experience a lower service degradation than the least-acceptable one.

In the following subsection, we report the detailed description on how the Application-Aware Service Provisioning Algorithm and the Alternative Solution Selection Algorithm work.

3.1. Application-Aware Service Provisioning Algorithm

The Application-Aware Service Provisioning Algorithm presented in this paper extends the algorithm designed in [6] to deal with the negotiation features proposed above. Such algorithm offers the provisioning of application-aware services on top of *multi-layer IP/optical networks*. However, note that our proposed negotiation scheme is not strictly related to multi-layer networks, indeed it could be adopted to work on top of any different network technology, as long as a well-designed service provisioning algorithm is provided.

As substrate network, the Application-Aware Service Provisioning Algorithm adopted in this paper considers 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 wavelengths. At the IP/MPLS layer, the nodes (i.e., IP/MPLS routers) are interconnected by IP adjacencies (i.e., links) that are realized through *lightpaths*, i.e., transparent optical connections.

As general concept, the algorithm first attempts to find an application-aware path for every service request by only considering the *existing lightpaths* (i.e., already-established IP adjacencies). Then, if no application-aware path is found, it includes in the investigation multiple *potential lightpaths* (i.e., lightpaths that have not been established yet, but that could be established if needed, since enough optical resources are available): this second stage increases the chance of finding a suitable solution at the expense of establishing new optical resources. For path computation, the algorithm exploits an auxiliary graph [28], including as nodes the IP/MPLS nodes. Such graph allows to represent the 2-layer network (including its state) on a single-layer topology. The algorithm works as follows, and is executed every time a service request $SR = \{s, d, b, l, a, n_b, n_l, n_a\}$ needs to be provisioned (for more details please refer to [6]):

1. All the *existing lightpaths* are added as edges to the auxiliary graph. The edges not meeting the bandwidth requirements b of the service request are pruned from the graph.

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

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

4. The algorithm prunes all the candidates paths not meeting l and a returns the first in the list (i.e., the shortest) and allocates resources. A path does not meet l (a) if the sum (product) of latency (availability) contributions on the path is greater than l (a) [28].

5. If the list of candidate paths is empty after step 4, the algorithm augments the auxiliary graph by including a multiple *potential lightpaths* computed

at the optical layer, and steps 2, 3 and 4 are executed again. If a path is found, the algorithm allocates resources. Otherwise, in the original version of the algorithm, the request is blocked.

We extended the Application-Aware Service Provisioning Algorithm described above to support the *negotiation* phase in case no application-aware path can be found on the augmented auxiliary graph, instead of blocking the service request. The extended version is able to compute a set of M alternative solutions, in which the negotiable application requirements can have looser and network-achievable values than the ones specified in the service request. In particular, in the negotiation phase the algorithm works as follows:

6. A copy of the service request is created and all the negotiable parameters are neglected.

7. Such modified service request is used as input for steps 1-5 described above.

8. In this phase, if the algorithm does not find any path, the service request is blocked. Otherwise, in steps 4, instead of returning the first path in the list, the algorithm stores the guaranteed requirements for each of the (up to) K_{ip} computed alternative candidate paths (i.e., the guaranteed bandwidth, the guaranteed latency, and the guaranteed availability).

9. 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 all of b, l and a are negotiable, the algorithm stores a maximum of three *alternative solutions*: the ones corresponding to the paths 1) with maximum residual bandwidth ($AS_1 = \{b_{n,1}, l_{n,1}, a_{n,1}\}$), 2) minimum guaranteed latency ($AS_2 = \{b_{n,2}, l_{n,2}, a_{n,2}\}$), and 3) maximum guaranteed latency ($AS_3 = \{b_{n,3}, l_{n,3}, a_{n,3}\}$).

10. The M alternative solutions computed in step 9 are sent to the application,

that can run the Alternative Solution Selection Algorithm described in the next subsection.

3.2. Alternative Solution Selection Algorithm

The Alternative Solution Selection Algorithm is in charge of automatically selecting the best alternative solution for each application, according to its needs. The algorithm works as follows:

1. The application receives M alternative solutions, as computed by the service provisioning algorithm in the negotiation phase.

2. The algorithm prunes all the solutions that have at least one requirement looser 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_t, the service request is blocked. Conversely, the algorithm calculates the weighted normalized Euclidean distance d between SR_p and every AS_j, as defined in Eq. 1:

$$d(SR_p, AS_j) = \sqrt{\sum_{i=1}^{N} w_i \left(\frac{SR_{p,i} - AS_{j,i}}{SR_{p,i} - SR_{t,i}} \right)^2} \tag{1}$$

   where the index i refers to each one of the N application requirements included in SR_p, SR_t and AS_j, while w_i represent a weight for the application requirements i. By properly tuning the weights w_i, the application can specify a different prioritization for any application requirement. For example, if latency is the most important requirement among the negotiable ones, the weight w_i related to latency will be set bigger than the weights for the other requirements. Clearly, $\sum_{i=1}^{N} w_i = 1$.

4. After evaluating every weighted normalized Euclidean distance, the algorithm selects the *best* alternative solution, i.e., the one leading to the

minimum one:

$$AS_{best} = AS_j : AS_j = \underset{j}{\operatorname{argmin}} \left[d(SR_p, AS_j),\ j \in M \right] \tag{2}$$

5. AS_{best} is sent to the control and management plane, which allocates resources on the associated path.

4. Software Architecture

This section describes the implementation of the negotiation scheme presented in Section 3 by exploiting a real SDN controller, which performs all the needed control and management plane functions (see Fig. 1). We discuss the functional requirements of the SDN controller components/modules and how the negotiation scheme is implemented.

From a high level view, the negotiation scheme demands some mandatory components from an SDN controller: (i) a *RESTful API* to simplify the submission of the service requests (in the form of intents) from the applications, (ii) an *intent compiler*, which perform the translation of such intents into forwarding rules, and (iii) a network *resource manager*, which keeps track of the available resources in the network.

We chose to implement our negotiation scheme by relying on the ONOS SDN controller [10]. ONOS offers a modular architecture in which external software modules, i.e., OSGi bundles [29], can be added and removed at runtime. The modules can exploit many *Services*, which use Java APIs, to interact with the *ONOS Core* software and the underlying substrate network. The modular architecture allowed us to implement the negotiation scheme by leveraging such *Services*, without the need of modifying the inner parts of the ONOS software (i.e., the ONOS Core).

Fig. 2 depicts the overall software architecture, including all the relevant ONOS *Services* and *APIs*. In the following subsections, we describe in details such *Services* and *APIs*, how they have been modified to support the negotiation scheme, and how the negotiation workflow is implemented in ONOS.

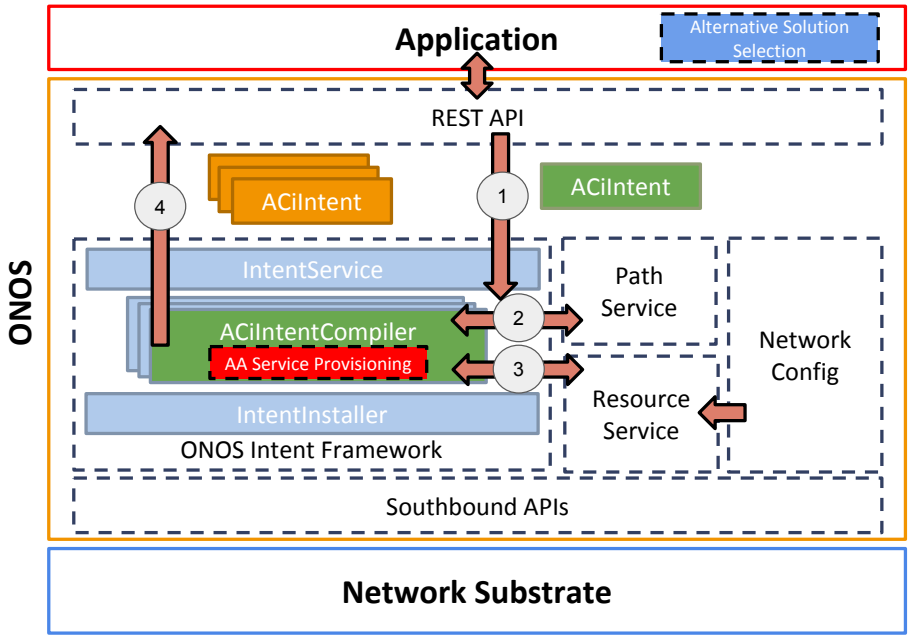

Figure 2: Negotiation scheme implemented in the ONOS controller.

4.1. ONOS Services and APIs

The *negotiation scheme*, as shown in Fig. 2, exploits the following *Services* and *APIs* from the ONOS controller.

- **ONOS Intent Framework** [11]. 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 (e.g. switches). The intents can be submitted via the *IntentService*, which exposes appropriate Java APIs for intent submission. The ONOS intent framework adopts several *intent compilers*, each one in charge of the compilation for a specific type of intent. The intent compilers interact with the other ONOS *Services* to obtain the required information for intent translation into specific forwarding rules. For instance, ONOS provides a set of pre-defined intents (and respective compilers) representing how to provide network connectivity, such as the *HostToHostIntent* [30]. Specifically, this in-

tent represents the request for a point-to-point connection between two hosts in a network. When a compiler finishes the computation of the forwarding rules, it requests to the *IntentInstaller* their installation.

In order to support the negotiation scheme in the ONOS intent framework, we added a new type of intent and its compiler: the *Application-Centric Intent* (ACiIntent, encoding a service request) and the *ACiIntentCompiler*. The former represents a request for a point-to-point connection between two network endpoints including the specific set of application requirements described in Section 3 (i.e., b, l, a), while the latter implements the path pruning logic of the Application-Aware Service Provisioning Algorithm, and all the logic required to manage the installation of an *ACiIntent* in the network.

- **PathService** [31]. This service represents the Path Computation Element (PCE) of ONOS. All the modules can query this service to obtain the shortest paths between two endpoints of a network. By default, the *PathService* returns all the shortest paths that have the same cost between a source and a destination. The *ACiIntentCompiler* exploits this service to discover the shortest paths between the two network endpoints included in the *ACiIntent*. Then, it provides this information as input for the path pruning logic of the Application-Aware Service Provisioning Algorithm, implemented in the same module.

- **ResourceService** [32]. 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., capacity, latency, availability, etc.) and consumption. All the ONOS modules can use this service-specific APIs to interact with the service both to query or update the consumption of resources. Only particular types of resources can be consumed, such as the capacity of a device port, an IP address (e.g. in the case of a DHCP server), etc. For instance, consider an intent requiring a bandwidth requirement of 100 Mbps between two endpoints. Its intent compiler first queries the *PathService* and receives the shortest path(s) connecting the two endpoints. Then, it checks whether all the devices'

ports associated to the path can support the requested capacity. Finally, if so, it records the intent in the *ResourceService* as a *resource consumer* of those ports' capacity. Thus, as en example, the capacity left for a port of 1000 Mbps is 900 Mbps.

The Application-Aware Service Provisioning Algorithm logic implemented in the *ACiIntentCompiler* uses the information provided by the *ResourceService* to check whether a path can guarantee the application requirements and, when negotiation is triggered, to compute the guaranteed requirements for any alternative solution.

- **NetworkConfig** [33]. This service allows to set or configure network devices and connected links specific parameters with custom values. For example, it can be used to configure the capacity of a device port or to annotate on a link its physical properties, such as latency or availability. This information is then provided to the ONOS *ResourceService*, so that it can be queried by other modules.

- **Southbound APIs**. One or more *Southbound APIs* are needed to communicate with the devices in the *Network Substrate*. These APIs can be OpenFlow [34], NETCONF [35], etc. Specifically, in this paper, we use OpenFlow.

4.2. Service Negotiation in ONOS

This subsection provides an overview on how we have defined the workflow implementing the negotiation scheme in the ONOS controller. We refer to the arrows depicted in Fig. 2 to describe the overall process, which is roughly divided in three main phases: *Intent Submission*, *Intent Compilation* and *Intent Negotiation*.

- **Intent Submission.** An application submits an *ACiIntent* including the preferable application requirements (i.e., SR_p, as defined in Section 3) via a *REST API* interface. The ONOS controller receives the *ACiIntent* and submits it to the *IntentService* for the compilation (arrow 1). The *IntentService* assigns an identifier to the *ACiIntent*, defined as *key*.

- **Intent Compilation.** The *Intent Framework* sends the submitted intent to the *ACiIntentCompiler*, which starts the compilation phase. First, the compiler requests to the *PathService* all the possible shortest paths between the two intent endpoints (arrow 2). Then, for every found path, the compiler queries the *ResourceService* and sends the information to the *Application-Aware Service Provisioning Algorithm* path pruning logic. The algorithm prunes the paths not meeting the application requirements and checks whether the intent can be satisfied by at least one path (arrow 3). If such a path is found, it is converted into forwarding rules and sent to the *IntentInstaller*. Otherwise, the *negotiation* phase starts, and the application is notified that its original submitted *AciIntent* is in a *negotiation required* status.

- **Intent Negotiation.** The paths that were previously found by the *PathService* are analyzed again. The *Application-Aware Service Provisioning Algorithm* queries the *ResourceService* to find the guaranteed requirements (i.e., b_n, l_n, a_n) for each of the computed paths. For each path, an *ACiIntent* (including the computed guaranteed requirements) is generated by the *ACiIntentCompiler*, submitted to the *REST API* and delivered to the application, to let it choose an alternative solution among all the generated *ACiIntents* (arrow 4), by means of the *Alternative Solution Selection Algorithm*.

  Note that the application, to make the communication of the alternative solutions from the ONOS controller happen, once notified about the *negotiation required* status, queries a specific URL of the *REST API* to receive the alternative solutions in JSON format.

5. Performance Evaluation

In this section, we present simulation and experimental results for a comprehensive evaluation of the negotiation scheme. They are divided into two different types: (i) *analysis of network sensitivity*, performed on a network simulator, and (ii) *evaluation of the ONOS controller implementation* in an emulated environment.

5.1. Network Sensitivity Analysis via Simulations

The sensitivity tests allows us to show what is the large-scale impact of negotiation on both networks and applications in different scenarios. Network sensitivity tests are performed on Net2Plan [36], an open source tool for network planning and simulation. Both the *Application-Aware Service Provisioning Algorithm* and the *Alternative Solution Selection Algorithm* have been developed on Net2Plan, exploiting its event-driven simulation features.

5.1.1. Simulation Setup

The sensitivity tests are based on a real multi-layer network topology and non-uniform traffic matrix provided by the Telefónica Spain ISP. The topology is a multi-layer network composed of 30 ROADMs 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, and we consider as node/link availability realistic random values ranging from 99.9% to 99.999%. At the IP layer, 14 IP/MPLS routers can be interconnected by lightpaths provisioned at the optical layer [37]. The traffic generation is performed by using the provided non-uniform traffic matrix, in which most of traffic is routed to/from the capital city.

Net2Plan offers a discrete event simulator composed of an event *generator* and an event *processor*. The event *generator* generates service requests according to a Poisson process, with exponentially-distributed inter-arrival times and holding times. After the expiration of the holding time, the allocated network resources (i.e., bandwidth on the links) for the considered service request are released. In our simulations, we simulate 5×10^5 service requests, while the statistics start to be collected after 2×10^4 events, to exclude the initial transitory phase.

We consider as application requirements, for each service request, the bandwidth (b), the latency (l) and the availability (a). The application requirements are randomly chosen from the following sets: $b = \{1, 2, 5, 10\}$ Gbps, $l = \{10\}$ ms, and $a = \{99.6\}$ %, and the values generated this way are the *preferred val-*

ues for the service request. The b values provide a reasonable set of bandwidth connectivity services that may be provided by an ISP. l and a were chosen as constraining values based on the network topology and the traffic matrix under test. The s and d parameters are instead generated accordingly to the non-uniform traffic matrix.

5.1.2. Sensitivity Test Methodology

We performed three experiments to evaluate how the negotiation scheme influences the network behavior:

- *Experiment 1)* All the application requirements can be negotiated at the same time for all the service requests;
- *Experiment 2)* All the application requirements can be negotiated at the same time, but only part of the service requests are willing to negotiate;
- *Experiment 3)* Only one specific application requirement can be negotiate for all the service requests.

These tests cover a comprehensive performance sensitivity analysis of our negotiation scheme. However, additional tests could be performed, also considering different types of application classes.

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

5.1.3. Discussion

Experiment 1). In this experiment, the service requests are always willing to negotiate b, l and a with the network. We define multiple Negotiation Levels (NLs), representing the maximum allowed degradation for each service request. The threshold values b_t, l_t and a_t are set as reported in Table 1. According

Table 1: Negotiation Level values

	$(b - b_t)/b \cdot 100$ (%)	l_t (ms)	a_t (%)
NL_1	10%	15	99.5
NL_2	20%	20	99.4
NL_3	30%	25	99.2
NL_4	40%	30	99

to the defined Negotiation Levels, each service request can always tolerate a relaxation of b, l and a: the maximum bandwidth tolerated degradation is set in terms of degradation 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 %). A higher Negotiation Level subscript is always associated to higher tolerance to requirement degradation.

We perform a simulation for every Negotiation Level, in which all the service requests belong to it.

Fig. 3 and Fig. 4 show an overview on the trade-off between the gain in service request acceptance (in terms of blocking probability reduction) and service request average bandwidth, latency and availability degradation, for the negotiated service requests, as a function of network load and Negotiation Level. In Fig. 4, the degradation values are normalized between 0% and 100%, in which 0% means no degradation, while 100% corresponds to the maximum allowed degradation specified by the threshold values, for the considered application requirement, as specified for each Negotiation Level. 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 (Fig. 5a), while the bandwidth (Fig. 4a), the latency (Fig. 4b), and availability (Fig. 4c) experience the 35%, 2% and 1% of their maximum allowed degradation, respectively. Thus, the applications experience much less than the maximum tolerated degradation, while the network significantly increases the number of service requests provisioned.

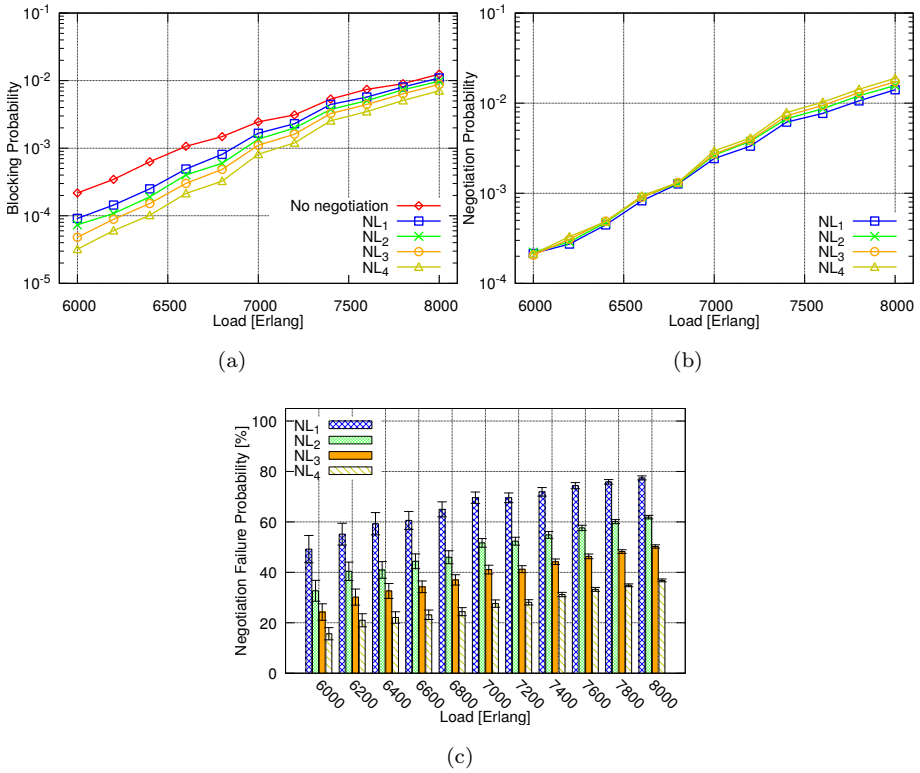

Figure 3: *(Experiment 1)* Evaluation of service request blocking probability (a), negotiation probability (b) and negotiation failure probability (c).

The bandwidth degradation (Fig. 4a) increases both with respect to network load and Negotiation Level. In fact, with higher loads, the network is only able to offer alternative solutions with more degraded bandwidth on average, since the average network utilization is higher. As opposed to bandwidth degradation, latency degradation (Fig. 4b) decreases with respect to network load and Negotiation Level. The reason is that, in our assumptions, each service request can have b, l and a degraded at the same time: thus, the higher b degradation is, both as a function of load and Negotiation Level, the easier finding spare resources on shortest paths is. Furthermore, a degradation is constant with respect to the network load, while decreases as the Negotiation Level increases: this happens because availability does not strictly depend on the length of the

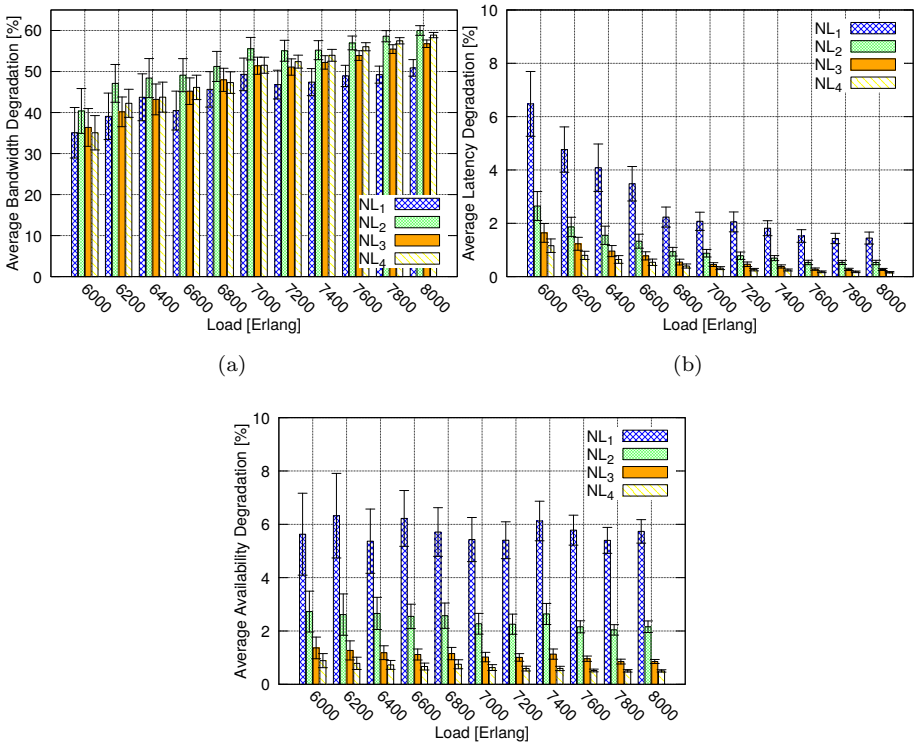

Figure 4: *(Experiment 1)* Evaluation of the average degradation of bandwidth (a), latency (b) and availability (c) experienced by service requests.

paths, as latency does. This behavior importantly points out how multiple application requirements experience different degradation trends when they can be relaxed at the same time, and how they mutually influence their trends.

Figs. 3b-3c show the service request negotiation probability (i.e., the probability that the network starts the negotiation phase for a service request) and the negotiation failure probability (i.e., the probability that the negotiation fails because no alternative solution suits the least-acceptable values for the application requirements) as a function of network load and Negotiation Level. Fig. 3b shows that the service request 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 the Negotiation Level. As expected,

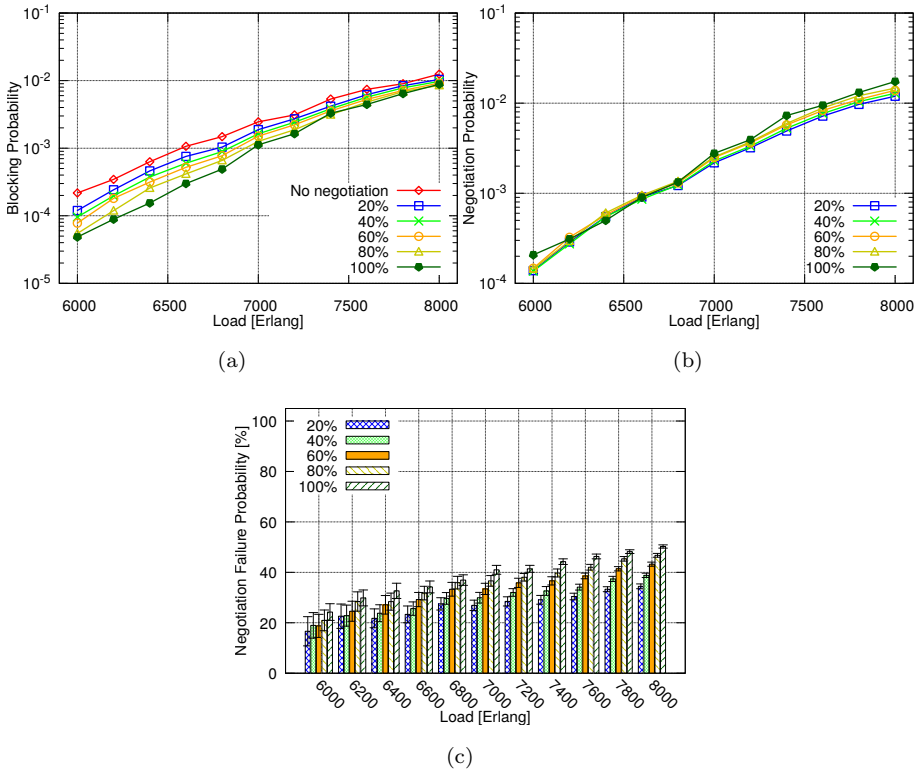

Figure 5: *(Experiment 2)* Evaluation of service request blocking probability (a), negotiation probability (b) and negotiation failure probability (c).

the negotiation failure probability (Fig. 3c) is higher (i) when the Negotiation Level is lower and (ii) as the load increases, i.e., in all the cases where network utilization is higher.

Experiment 2). This experiment aims at showing the impact of negotiation when a different amount of service requests are willing to negotiate all their application requirements. The amount of negotiable service requests is expressed as a percentage, which represents the ratio between the negotiable and the total number of service requests in the simulation. The maximum allowed degradation is fixed for b, l and a to NL_3, since it roughly represents the average value of degradation within all the performed experiments.

Fig. 5 offers an overview of the negotiation performance in terms of service

Table 2: *(Experiment 2)* Degradation of the Application Requirements

	Low Load (6000 Erlang)			*High Load* (8000 Erlang)		
	b (%)	l (%)	a (%)	b (%)	l (%)	a (%)
20%	30	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
100%	36	1.64	1.35	57	0.27	0.85

request blocking probability (Fig. 5a), negotiation probability (Fig. 5b) and negotiation failure probability (Fig. 5c) as a function of the network load and the percentage of negotiable requests. As expected, the service request blocking probability considerably reduces as the number of negotiable service requests increases. Moreover, all the three metrics increase with respect the network load, and negotiation/negotiation failure probabilities slightly increase as the percentage of negotiable service requests increases. This is mainly due to fact that, at higher loads, a negotiable service request has a higher probability to enter the negotiation phase and to fail it, since less resources are available. In Table 2, we report the experienced average application requirement (b, l, a) degradation (in percentage) at *low* and *high* network loads (i.e., 6000 and 8000 Erlang). Any application requirement degradation is more affected by an increase in the network load than in the average number of negotiable service requests. This means that having more applications that are willing to negotiate gives benefits to both the network and applications, since more service requests can be accepted without any substantial additional application requirement degradation.

Experiment 3). In this experiment, the applications are willing to negotiate only one application requirement at a time, while the others cannot be negotiated. This allows to evaluate how the negotiation of a single application requirement influences the service requests blocking probability and the

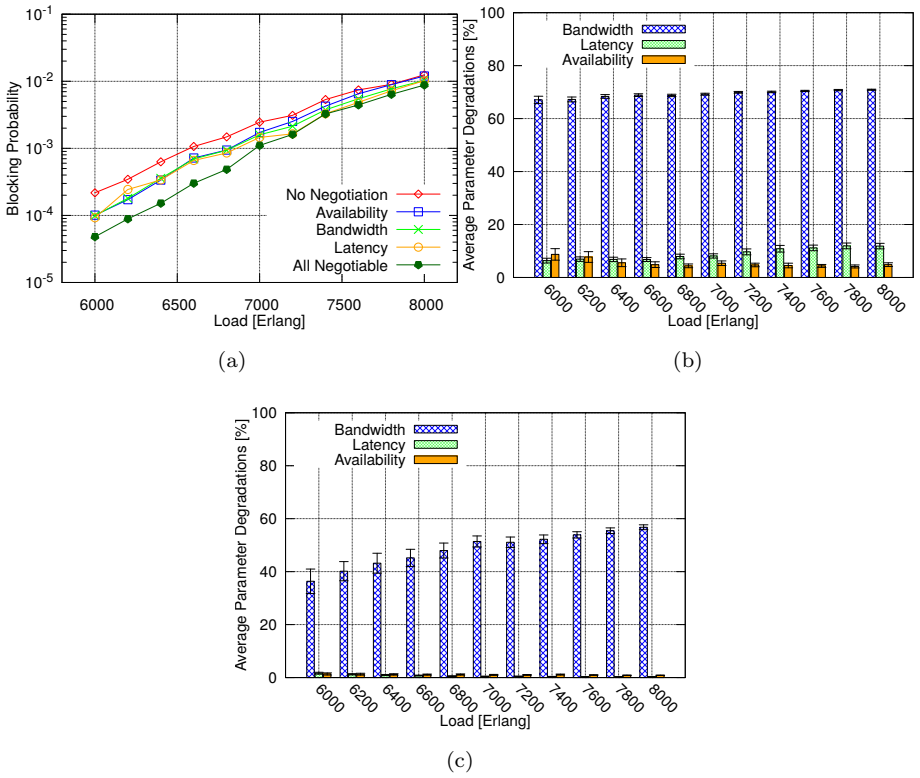

Figure 6: *(Experiment 3)* Evaluation of service request blocking probability (a), average degradation for a single negotiable (b) and all negotiable requirements cases (c).

experienced degradation. The maximum allowed degradation of the negotiable application requirement (either b, or l, or a) is set to the one of NL_3 for that specific requirement. We compare the results with the ones of Experiment 1 where all the application requirements can be negotiated with the maximum degradation of NL_3 (called *All Negotiable* in the figures).

The service request blocking probability (Fig. 6a) shows almost overlapping values between b, l and a cases. This shows that there is not a predominant application requirement that can lead to a substantial requests blocking probability improvement with respect to the others. In addition, the blocking probability for all the analyzed cases is between the *No Negotiation* and *All negotiable* values. This means that by negotiating only one application requirement at a

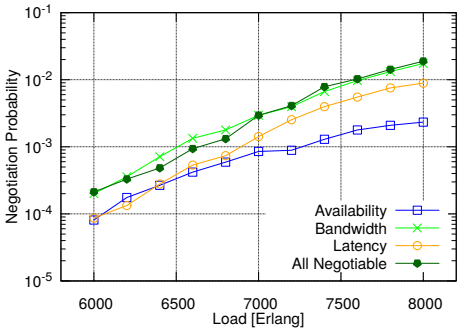

Figure 7: *(Experiment 3)* Negotiation probability.

Table 3: *(Experiment 3)* Negotiation Failure Probability

	b (%)	l (%)	a (%)
Low Load (6000 Erlang)	47.67	0	0
Medium Load (7000 Erlang)	52.38	0	0
High Load (8000 Erlang)	58.87	0	0

time, the negotiation still shows some improvement with respect to the case in which no negotiation scheme is adopted, but it becomes more difficult for the network to find alternative solutions that satisfy the other two non-negotiable application requirements at the same time.

Additionally, in the case of only l or a negotiable, we found that the 100% of blocked service requests is caused by the impossibility to find any alternative solution by the Application-Aware Service Provisioning Algorithm. This is different from the case of only b negotiable, in which the algorithm always finds an alternative solution. This is also supported by Fig. 7, which depicts the negotiation probability as a function of the network load. The negotiation probability is similar to the *All negotiable* case when b is negotiable, while lower in the case of l or a negotiable since, if no alternative solution can be found, the service request is blocked and no negotiation phase is started.

On the application side, in the case of l or a negotiable and that at least one alternative solution has been found by the network, the negotiation never fails (see Table 3). It means that, among the proposed alternative solutions, the application can always find one better than either l_t or a_t. This is instead not true in the case of b negotiable, as shown in the same Table, where the negotiation failure probability is close to 60% in case of high load.

The application requirement degradation, normalized between 0% and 100%, is depicted in Fig. 6b as a function of the network load. The figure report results in which the degradation can be experienced only by the application requirement chosen as negotiable. In the case of b negotiable, the degradation is almost constant between 6000 and 8000 Erlang. Fig. 6c show instead the degradation experienced by the application requirements when all of them can be negotiated at the same time (as per Experiment 1). In this case, b experiences an increasing degradation as the network load increases, but it is always lower than the case in which only b can be negotiated (Fig. 6b). This because a degradation of also l and a influences the discovery of the alternative solutions, making it easier to find spare resources on shortest paths. Also in the case of l or a negotiable (Fig. 6b), these requirements present a higher degradation with respect to the case of all the application requirements negotiable (Fig. 6c). This means that an application willing to negotiate multiple requirements will experience, in average, a lower degradation for each one of the negotiable requirements with respect to an application only willing to negotiate one single requirement.

5.2. Evaluation of the ONOS-based Scheme in an Emulated Scenario

The tests performed on our ONOS implementation of the negotiation scheme provide an overview of the processing times required to handle the negotiation process on a real implementation. We evaluate both the ONOS SDN controller and applications processin times for the different phases of the negotiation scheme, when a different number of service requests has been already provisioned in the network.

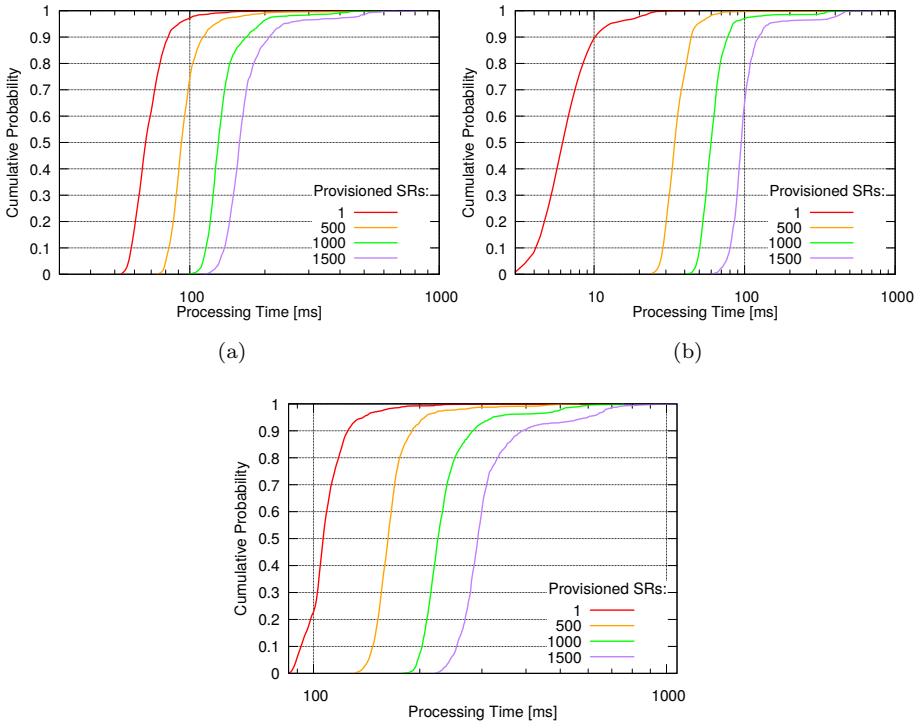

Figure 8: Cumulative distribution of processing times for *Solution generation* (a), *Compilation* (b), and *Overall negotiation* (c).

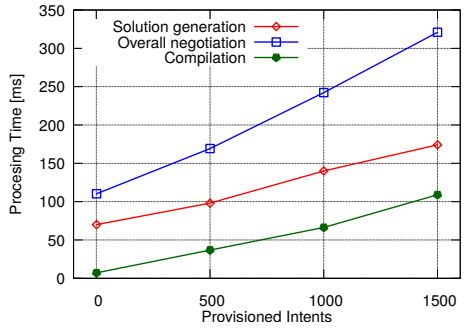

Figure 9: Average processing times of the different negotiation processes.

5.2.1. Test Setup

The ONOS implementation of the SDN controller is tested on top of Mininet [38], a well-know network emulator. This software offers the possibility to generate

big network topologies and control them by exploiting SDN interfaces. For these performance tests, we reproduced the Telefónica's IP network topology, composed of 14 IP/MPLS nodes, which is described in detail in [37] and already used for the sensitivity tests of the previous subsection. The ONOS controller manages the overall network devices by exploiting the OpenFlow [34] protocol. The capacity of the device ports and the information about the links (i.e., latency, availability) are annotated in a JSON configuration file and loaded into the *NetworkConfig* module.

We implemented a Java software artifact that emulates the application behavior. It generates the service requests in the form of *ACiIntents* and submits them through the REST API interface of the SDN controller. It also handles their negotiation status and includes the *Alternative Solution Selection Algorithm*.

The ONOS implementation and the Mininet instance run on an Intel Quad Core i7-5600U CPU @ 2.60GHz with 16 GB of RAM.

5.2.2. Test Methodology

We evaluate the time required by the SDN controller (i) to generate the alternative solutions (*Solution generation*), (ii) to compile a service request (i.e., translating an ACiIntent into commands for the network devices, *Compilation*) and (iii) the overall time required by the negotiation, from a service request submission to the end of the compilation phase (*Overall negotiation*), thus excluding any time required by the network devices to process the commands, since it strongly depends on the substrate network characteristics. We evaluate these processing times as a function of the number of already-provisioned service requests in the network, which gives an indication on the processing load on the SDN controller. The amount of already-provisioned service requests varies from 1 to 1500.

The application generates all the service requests between two fixed endpoints in the network, which are the farthest in terms of number of hops on their shortest path. This poses us in the worst-case scenario, allowing us to evaluate

the time required to process any negotiation request by considering the highest amount of crossed network devices and links in a nearly-congested scenario. In order to emulate a scenario where negotiation is needed and triggered, we generate and provision different service requests by appropriately varying their bandwidth requirement to saturate the network resources after our considered number of already-provisioned requests (i.e., from 1 to 1500) has been reached, and let the SDN controller trigger the negotiation scheme for the subsequent ones. The already-provisioned service requests leave enough resources to allow the SDN controller finding minimum two alternative solutions for every service request under negotiation. The tests are executed 1000 times for every SDN controller processing load.

5.2.3. Discussion

Fig. 9 provides an overview of the considered processing times as a function of the number of already-provisioned service requests in the network, i.e., *Solution generation*, *Compilation* and *Overall negotiation*. All the processing times linearly increase as the number of already-provisioned service request requests increases, since the SDN controller has to manage a higher processing load for the service request to be provisioned.

The *Solution generation* time is mainly affected by the *ResourceService* database query, since the database keeps track of all the provisioned resources associated to every service request (e.g. crossed links and their consumed bandwidth). A higher number of provisioned service requests means a higher number of entries in this database and thus more time is required to retrieve the amount of free resources for the execution of the Application-Aware Service Provisioning Algorithm. Fig. 8a shows the cumulative distribution of processing times for the *Solution generation* process.

The *Compilation* time is affected by both the *ResourceService* and the overall controller load. In fact, during the compilation process, the compilation algorithm checks again the availability of resources to avoid that another concurrent request has modified it. Then, the compilation algorithm saves the

resources consumed by the service request in elaboration into the *ResourceService* database and translates them into the specific forwarding rules required by the network devices. In these tests, the number of rules varies based on the selected path, but no more than 8 rules are generated for a specific service request. The cumulative distribution of compilation times is depicted in Fig. 8b, showing how the compilation times are much shorter for 1 already-provisioned service request than for 500, 1000 and 1500. In fact, when the SDN controller is unloaded, the *ACiIntentCompiler* does not experience any delay caused by accessing the *ResourceService* database, since it is almost empty and the time to query it is negligible.

The *Overall negotiation* curve is steeper than the curve obtained by the sum of the *Solution generation* and *Compilation* ones. This happens because the overall negotiation time includes also other time contributions, such as (i) the communication time between the application emulator and the SDN controller (in this case negligible, $\sim$1 ms), (ii) the conversion between JSON and Java classes and (iii) the alternative solution selection processing time. This last time contribution has also been evaluated, but it is always much less than 1 ms (and thus negligible) on average, so results are not reported for the sake of conciseness. Specifically, Fig. 9 clearly shows how the *Solution generation* and *Compilation* times dominate in the *Overall negotiation* time with respect to all the other aforementioned time contributions. Finally, Fig. 8c shows the cumulative distribution of the overall negotiation time. Note also that in the case that the negotiation scheme is not implemented (i.e., *No negotiation*), the overall service request processing time is comparable to the sole *Compilation* time.

In general, from the obtained results, we can state that our implementation of the negotiation scheme in the ONOS controller shows some overhead in terms of processing times. with respect to the case in which the negotiation scheme is not implemented. Indeed, the highest overall negotiation times happen in the case of 1500 already-provisioned service requests in the network, and they have an average value of 320 ms. However, this time is still some order of magnitude

lower than the time required by the hardware (e.g. optical transponders), which requires from tens of seconds to minutes. This time should be taken into account anyway in the deployment of a service request (even in the case our negotiation scheme was not adopted).

6. Conclusion

In this work, we proposed an interaction between applications and networks that enables the *negotiation* of application-aware connectivity services. The application can request a connectivity service with several constraints, such as bandwidth, latency and availability between two transport network endpoints. The network, when the request cannot be fully satisfied, proposes several alternative solutions with *looser* requirements. Finally, applications can automatically provide a feedback on which is the best solution to accept by exploiting the *Solution Selection Algorithm*.

Both applications and networks can take advantage of a negotiation mechanism. Specifically, the applications that negotiate accept a degradation of their service requests, while (i) avoiding a complete block of their request and (ii) obtaining a predictable degradation of a service. Moreover, an application can use the degradation information to modify its traffic behavior within the network. On the other side, network operators can accommodate a higher number of services on their transport network while increasing the revenues. A fully working proof of concept of the *application-aware negotiation framework* has been implemented on top of the ONOS SDN controller. The related code has been released as open source and made publicly available [12] to researchers and experimenters for further testing and enhancements.

The effectiveness of the solution proposed has been demonstrated by means of simulations. The performance evaluation has shown a lower blocking probability of service requests on the network side and a limited degradation of service constraints on the application side. Moreover, the implementation on top of the ONOS controller has demonstrated a limited overhead.

In the next future, we plan to extend the negotiation framework with a cost/pricing model. Such model is under definition, and we plan to evaluate the trade-off between the experienced service degradation and service cost/price.

References

[1] A. Marsico, M. Savi, D. Siracusa, E. Salvadori, An automated service-downgrade negotiation scheme for application-centric networks, in: 2018 Optical Fiber Communications Conference and Exposition (OFC), 2018.

[2] S. Borkar, H. Pande, Application of 5G next generation network to Internet of Things, in: 2016 International Conference on Internet of Things and Applications (IOTA), Pune, India, 2016.

[3] O. Gerstel, V. Lopez, D. Siracusa, Multi-layer orchestration for application-centric networking, in: 2015 International Conference on Photonics in Switching (PS), Florence, IT, 2015.

[4] M. Pham, D. B. Hoang, SDN applications - The intent-based Northbound Interface realisation for extended applications, in: 2016 IEEE NetSoft Conference and Workshops (NetSoft), 2016.

[5] V. Lopez, D. Konidis, D. Siracusa, C. Rozic, I. Tomkos, J. P. Fernandez-Palacios, On the Benefits of Multilayer Optimization and Application Awareness, Journal of Lightwave Technology 35 (6) (2017) 1274–1279.

[6] M. Savi, D. Siracusa, Application-aware service provisioning and restoration in SDN-based multi-layer transport networks, Journal of Optical Switching and Networking 30 (2018) 71 – 84.

[7] L. Velasco, A. Asensio, J. Berral, A. Castro, V. López, Towards a carrier sdn: an example for elastic inter-datacenter connectivity, Optical Express 22 (1) (2014) 55–61.
URL `http://www.opticsexpress.org/abstract.cfm?URI=oe-22-1-55`

[8] A. Dixit, B. Lannoo, G. Das, D. Colle, M. Pickavet, P. Demeester, Dynamic bandwidth allocation with sla awareness for qos in ethernet passive optical networks, IEEE/OSA Journal of Optical Communications and Networking 5 (3) (2013) 240–253.

[9] S. Choi, J. Park, Sla-aware dynamic bandwidth allocation for qos in epons, IEEE/OSA Journal of Optical Communications and Networking 2 (9) (2010) 773–781. doi:10.1364/JOCN.2.000773.

[10] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, 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, 2014.

[11] ONOS Intent Framework, last access 30-08-2018.
URL https://wiki.onosproject.org/display/ONOS/Intent+Framework

[12] ACINO Multi-layer Network Orchestrator.
URL https://github.com/ACINO-H2020/network-orchestrator

[13] EU H2020 ACINO, last access 30-11-2018.
URL https://cordis.europa.eu/project/rcn/194286_en.html

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

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

[16] R. Roy, B. Mukherjee, Degraded-Service-Aware Multipath Provisioning in Telecom Mesh Networks, in: OFC/NFOEC 2008 - 2008 Conference on

Optical Fiber Communication/National Fiber Optic Engineers Conference, 2008, pp. 1–3.

[17] H. Y. Chang, A Multipath Routing Algorithm for Degraded-Bandwidth Services under Availability Constraint in WDM Networks, in: 2012 26th International Conference on Advanced Information Networking and Applications Workshops, 2012, pp. 881–884.

[18] X. Wang, H. Schulzrinne, Integrated resource negotiation, pricing, and QoS adaptation framework for multimedia applications, IEEE Journal on Selected Areas in Communications 18 (12) (2000) 2514–2529.

[19] Amazon EC2 Spot Instances (2017).
URL `https://aws.amazon.com/ec2/spot/`

[20] Google Preemptible Virtual Machines (2017).
URL `https://cloud.google.com/preemptible-vms/`

[21] K. Czajkowski, I. Foster, C. Kesselman, V. Sander, S. Tuecke, SNAP: A Protocol for Negotiating Service Level Agreements and Coordinating Resource Management in Distributed Systems, in: D. G. Feitelson, L. Rudolph, U. Schwiegelshohn (Eds.), Job Scheduling Strategies for Parallel Processing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 153–183.

[22] S. Venugopal, X. Chu, R. Buyya, A negotiation mechanism for advance resource reservations using the alternate offers protocol, Enschede, NL, 2008.

[23] A. Rubinstein, Perfect equilibrium in a bargaining model, Econometrica 50 (1) (1982) 97–109.

[24] M. Jarschel, F. Wamser, T. Hohn, T. Zinner, 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, 2013.

[25] S. Gorlatch, T. Humernbrum, F. Glinka, Improving QoS in real-time internet applications: from best-effort to Software-Defined Networks, in: 2014 International Conference on Computing, Networking and Communications (ICNC), Honolulu, HI, USA, 2014.

[26] Y. Yiakoumis, S. Katti, T. Huang, N. McKeown, K. Yap, R. Johari, Putting home users in charge of their network, in: Proceedings of the 2012 ACM Conference on Ubiquitous Computing, Pittsburgh, PA, USA, 2012.

[27] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, S. Krishnamurthi, Participatory networking: An api for application control of sdns, SIGCOMM Comput. Commun. Rev. 43 (4) (2013) 327–338.

[28] H. Zhu, H. Zang, K. Zhu, B. Mukherjee, Dynamic traffic grooming in WDM mesh networks using a novel graph model, in: Global Telecommunications Conference (IEEE GLOBECOM), Tapei, Taiwan, 2002.

[29] OSGi Alliance.
URL `https://www.osgi.org/`

[30] ONOS PointToPoint Intent, last access 30-08-2018.
URL `http://api.onosproject.org/1.11.0/org/onosproject/net/intent/HostToHostIntent.html`

[31] ONOS Interface PathService, last access 30-08-2018.
URL `http://api.onosproject.org/1.11.0/org/onosproject/net/topology/PathService.html`

[32] ONOS ResourceService, last access 30-08-2018.
URL `http://api.onosproject.org/1.11.0/org/onosproject/net/resource/ResourceService.html`

[33] ONOS Network Configuration, last access 30-08-2018.
URL `https://wiki.onosproject.org/display/ONOS/The+Network+Configuration+Service`

[34] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner, OpenFlow: Enabling Innovation in Campus Networks, SIGCOMM Comput. Commun. Rev. 38 (2) (2008) 69–74.

[35] J. Schönwälder, M. Björklund, P. Shafer, Network Configuration Management Using NETCONF and YANG, Comm. Mag. 48 (9) (2010) 166–173.

[36] P. Pavon-Marino, J.-L. Izquierdo-Zaragoza, Net2plan: An open source network planning tool for bridging the gap between academia and industry, IEEE Network 29 (5) (2015) 90–96.

[37] F. Rambach, B. Konrad, L. Dembeck, U. Gebhard, M. Gunkel, M. Quagliotti, L. Serra, V. Lopez, A multilayer cost model for metro/core networks, IEEE/OSA Journal of Optical Communications and Networking 5 (3) (2013) 210–225.

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