Towards Application Driven Networking

Francklin Simo Tegueu[a], Slim Abdellatif[a], Thierry Villemur[a], Pascal Berthou[a], Thierry Plesse[b]

[a] LAAS-CNRS, Université de Toulouse, CNRS, INSA, UPS, Toulouse, France

[b] Directorate General of Armaments (DGA), La Roche Marguerite, 35998 Rennes, France

Abstract—In order to address the performance problems that many business applications are experiencing, network vendors and Network Service Providers are reconsidering the integration of some form of application awareness in the way their networks forward user traffic. Their ultimate goal is to devise new network service models that are dedicated and customized to applications. This trend is mainly enabled by the emergence of software-Defined networking (SDN), which allows for flexible flow-based forwarding.

This paper proposes an SDN-based Application Driven Network (ADN) that deploys customized data paths on an application basis, that the ADN is able to adapt to meet the, potentially, varying Quality of Service (QoS) needs of applications. These needs are, either, explicitly expressed and submitted by the middleware or by an application agent, or inferred by the proposed ADN with DPI (Deep Packet Inspection) based traffic classification techniques. This paper presents the general architecture of our proposed ADN by describing its main components, their requirements as well as their main algorithms. The proposed ADN has been implemented, evaluated and applied to the OMG Data Distribution Service (DDS) based distributed applications in an enterprise network context.

Keywords—Application Driven Networking; Software Defined networking; Quality od Service; Network Virtualization

I. Introduction

Despite all the advances on network Quality of Service (QoS) provisioning that we witnessed during the last decades, the performance of business critical applications on an enterprise network is still an issue that concerns an increasing number of organizations [1]. The study of [1] reveals that most organizations do not have any knowledge on the QoS requirements of most of their critical applications; Some of the reasons are that some QoS parameters are hard to specify especially for applications that exhibit some form of dynamicity (i.e. data flows exchanged between application processes and their associated QoS requirements vary over time). Also, they have a poor visibility on the performance that applications get from the network. As a consequence, these organizations resort to network resource over-provisioning coupled with network and QoS planning adjustments based on user complaint and feedback.

One way to address these problems is to introduce some form of application awareness into the forwarding behavior of computer networks. This means that the network is able to provide customized data paths to applications, i.e. data paths that are assigned on a per application basis (and not on a destination basis) with the required characteristics (in terms of assigned network resources) to meet application QoS needs. The network must also cope with changing needs. It must be able to derive current application needs and then compute and mobilize on the fly the required network resources along the data path(s). This is what we are proposing in this paper and what we have referred to as an Application Driven Network (ADN). ADN enables the provision of network services with guaranteed dynamic QoS and efficient network utilization.

One crucial aspect to ADN functioning is how to identify the traffic belonging to an application and what are its QoS needs. For some applications these needs are explicitly expressed either from the middleware on top of which they are built, or from an external agent with usually a human in the loop (e.g. [2][3]). In both cases, application level QoS requirements are mapped to QoS requirements on the network service (here, the ADN service). For the applications that do not provide any information on their needs, the proposed ADN relies on Deep Packet Inspection (DPI) to detect their presence and infer their short-term requirements.

In fact, the general idea of directing network behavior to meet application demands with efficient resource utilization is not new and has already been considered in the dense literature related to QoS provisioning. All these works were applied to (and hence constrained by) legacy computer networks with distributed control and destination based forwarding (potentially, with a complementary priority tag). There have been some recent proposals on leveraging SDN to meet the above-cited goal. Most were centered on taking benefit from the centralized nature of the control and continued to assume a destination-based forwarding [4][5][6][7][8]. In contrast to previous works, our proposed ADN relies on a flow-based forwarding at the SDN substrate that combines layer 2 to layer 4 packet headers. We argue that this is the condition to provide services (data paths) that exactly or closely match a fine-grained description of applications' traffic and needs. For completeness, some works [9][10] have advocated the use of flow-based forwarding, but in contrast to our work, they were not focused on providing strict QoS guarantees to applications.

This paper is organized as follows. Section II describes the design principles of our proposal. Next, we present the general architecture of our proposed ADN as well as some of its main algorithms. Section V presents the ADN prototype that we implemented and some experimental results. Finally, section VI concludes the paper.

II. Key principles of the proposed ADN

One key principle of our proposed ADN is that the service provided to applications is typically (but not exclusively) based on a fine-grained knowledge of applications' flows and QoS needs (i.e. application communication profile). The rationale is: a precise knowledge of applications' needs allows deploying the right service (that meets exactly the needs) with the optimal set of network resources. In comparison to existing works, this is clearly one key characteristic of our proposal. The communication profile can be derived from the explicit

description of the application's requirements, optionally, complemented with some traffic estimations performed by the ADN. With the DDS Publish/Subscribe middleware [14] that we consider in our implemented prototype, this knowledge goes up to identifying the flows of data that are exchanged between each data-publisher and its associated data-subscribers (and their associated QoS). Also, DPIs fed with relevant traffic patterns are deployed at network node edges to classify the traffic belonging to the applications that are supposed to use the ADN services, and derive their current needs. In case of no explicit expression of needs, DPI based statistical traffic estimation techniques are used.
Another principle is that the network service provided to applications is expressed as a Virtual NETwork (VNET) composed of a set of logical (virtual) end-to-end links (from end host to end host). Each virtual link is either point-to-point or point-to-multipoint and is characterized by a bandwidth requirement and a maximum transfer delay requirement.
Our proposal assumes an SDN/OpenFlow enabled network infrastructure. This latter can be completely or partially dedicated to our ADN. In this latter case, some form of network virtualization applies and pre-defined slices on network elements are exclusively dedicated to our ADN. An SDN network control application, that we refer as "ADN service provisioning", implements the ADN logic and is in charge of provisioning the ADN services. It is based on a low-level northbound interface (i.e. OpenFlow like).

The last key principle is to build an autonomic "ADN service provisioning" network function. More precisely, we primarily target the self-configuring property and approach the self-healing and self-optimizing properties.

III. Reference Architecture of the Proposed ADN

Figure 1 depicts the functional architecture of the network control function "ADN service provisioning" that implements our ADN approach. Its components are presented hereafter.

A. Request handler

It acts as a front-end and orchestrates the execution of the different components involved in servicing a VNET addition, update or deletion request. Upon a VNET addition request, for scalability purposes, it triggers the "flow aggregator" component to check whether some aggregation applies to the virtual links that compose the VNET. Then, it launches the "resource allocator" to compute the data paths supporting the VNET. Finally, it triggers the "VNET deployer" to install the VNET on the SDN substrate.

B. Flow Aggregator

The proposed ADN has the advantage of providing services with guaranteed QoS while efficiently using network resources/ But, it clearly raises scalability issues. One important aspect is the number of flow table entries that are installed on OpenFlow (OF) switches to support the service. Indeed, current flow-tables, which are often based on fast TCAMs, have a size limited to a few thousands of entries. The "*Flow Aggregator*" component tackles this problem. It is in charge of computing the final set of flows that describes the expected service by grouping, when feasible, some flows together. The aggregations have generally a cost in terms of network resource utilization. This component optimizes this tradeoff.

C. Virtual Network Resource Allocator

On a virtual network request, it is in charge of computing the optimal set of physical paths (with the necessary resources) to use in order to support the virtual links with their QoS characteristics. Many optimization criteria can be considered. Amongst, the ones considered in this work, namely minimizing network resource utilization and minimizing network elements' load disparities (which contribute to improve the admissibility of subsequent virtual network requests). For the same purpose, *path splitting* (multiple paths support a virtual link) can be enabled for some requests. Two types of network resources are taken into account: classically, the bandwidth of links but also the switching resources of nodes, i.e. the number of OpenFlow flow-table entries, group-table entries and meters.

It is worth to note that this component also performs the reallocation or de-allocation of resources in case of an update or a cancellation request.

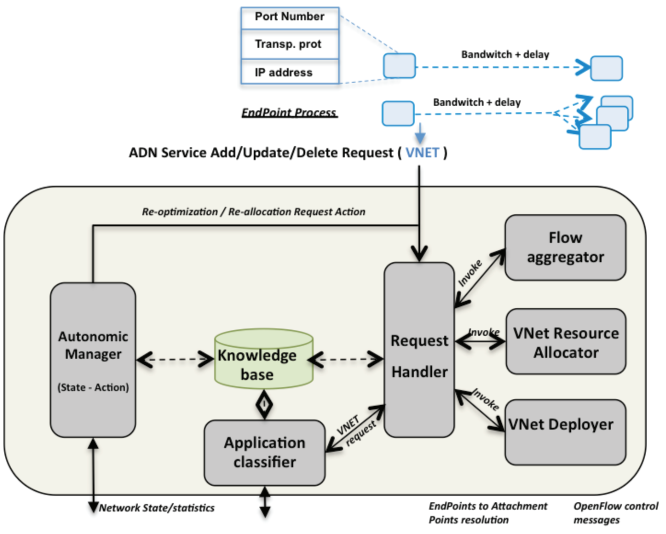

Figure 1 ADN's functional components

D. Virtual Network Deployer

The ultimate goal of this component is the effective deployment of the virtual network on the OpenFlow network infrastructure. It takes as input the data paths and the associated resources computed by the "VNET Resource allocator", it generates the OpenFlow rules to apply on each OpenFlow switch, and it submits them to the OpenFlow controller via the northbound interface. The controller is instructed to install the rules on the identified switches.

E. Application Classifier

The proposed ADN targets, the applications that explicitly express their will of using the service by providing their QoS needs, as well as those that do not. Through network traffic analysis performed by DPIs deployed on the network and under its control, this component is in charge of identifying in real-time the applications that are allowed to use the ADN service, estimating their current needs and then issuing the corresponding VNET request.

F. Autonomic Manager

The ultimate goal of the "*Autonomic manager*" is to instill some of the autonomic properties to the "*ADN service provisioning*" network function. It manages the components described previously by implementing the MAPE (Monitoring, Analysis, Planning and Execution) loop (based on the frameself framework [11]).
Without being exhaustive, some of the important identified situations that the "Autonomic manager" has to react to are

described below. On a network topology change (detected by monitoring the network), it decides whether network resource re-allocations must be triggered (self-repairing). According to the available network resources and the virtual network request, it tunes the resource allocation (e.g. enable/disable path splitting) and/or the flow aggregation algorithms. Similarly, after virtual network cancellations, it decides to re-compute the allocated resources in order to better distribute the load of network elements (self-optimizing). From its network monitoring, it detects that the rate allocated to a virtual link is not adequate and decides to enable traffic estimation related to the associated application (self-configuring).

IV. MAIN ALGORITHMS

In this section, we present the internal algorithms of the proposed "ADN service provisioning" network control function, more precisely, those of the "Virtual Network Deployer" and the "Virtual Network Resource Allocator".

The SDN substrate network is modeled as a bidirectional graph $G = (V, E)$ where $V(|V|)$ is the set of SDN nodes and $E(|E|, E \subseteq V \times V)$ the set of physical links which operate in full-duplex mode. To each node $i \in V,$ is associated a switching capacity U_i, which is the maximum number of entries (i.e. size limit) of its flow table. The current size of node i flow table is denoted by U_i'. Similarly, N_i and N_i' denote respectively the maximum and the current size of the group table of switch i. Each Link $(i,j), i,j \in V$ is weighted by its bandwidth B_{ij} and its propagation delay D_{ij}. Links are assumed to have the same characteristics in both directions, i.e. $B_{ij}=B_{ji}$ and $D_{ij}=D_{ji}$. The bandwidth that is currently assigned at link *(i,j)* by already admitted virtual links is denoted by B_{ij}'.

A virtual network request is composed of a set of K virtual links. Each virtual link k is characterised by:

- a source node $s_k \in V$, and a set of destination nodes $T_k \subseteq V - \{s_k\}$ (when $|T_k| = 1$, the virtual link is point-to-point, otherwise it is point-to-multipoint);
- a bandwidth requirement of $b_k \in \mathbb{N}$, a maximum transfer delay of $d_k \in \mathbb{N}$ and a maximum packet size of p_k.

We denote as $f_k^t(i,j)$ the bandwidth allocated at link (i,j) to the packets of virtual link k that are flowing from the source node s_k to a destination node t. Also, $f_k(i,j)$ refers to the amount of bandwidth used on link (i,j) by the virtual link k. It is set to the maximum of $f_k^t(i,j)$ for all $k \in K$.

A. The "Virtual Network Deployer" algorithm

The "ADN service provisioning" function is implemented on top of a low-level northbound interface. First, we briefly present some prerequisites on the OpenFlow protocol before detailing the algorithm. For simplicity, the presented algorithm does not address the deployment of point-to-multipoint link with path splitting enabled.

1) Some prerequisites on Openflow

An OpenFlow switch embeds at least one flow table, a group table and a meter table. The OpenFlow controller relies on OpenFlow modification messages to fill these tables. Three types of messages are distinguished by the OpenFlow protocol. They are described hereafter.

OpenFlow <u>Flow</u> Modification Messages (ofp_flow_mod) are used by the controller to insert, delete or update one flow entry into a flow table of a switch. Each flow entry contains a match field and a set of instructions that are executed when a packet matches the entry. These instructions result in changes to the packet action set. There are multiple types of instructions, among which, the "write-actions" and "meter", which are used in the proposed algorithm. "Write-Actions" instruction gathers a list of actions to add to the current "Action-Set" of the matching packet. The Meter instruction directs the matching packet to the specified meter.

OpenFlow <u>Group</u> Modification Messages (ofp_group_mod) are sent by the controller to insert, delete or update a group entry into the group table of a switch. Each group entry has a group id; it is typically used by a flow table entry as a reference to the group. A group entry has a list of action buckets. Depending on the group type, the actions in one or more action buckets are applied to packets sent to the group. Two types are of interest to this work:

- "Select": Each bucket has a weight, which is used to choose the bucket that applies to an arriving packet.
- "All": used to perform multicast or broadcast forwarding. The packet is effectively cloned for each bucket.

OpenFlow <u>Meter</u> Modification Messages (ofp_meter_mod) are sent by the controller to insert, delete or update a meter entry into the meter table of a switch. A meter entry is identified by a meter id and composed of one or more meter bands. Each meter band specifies a target rate for that band and the way packets are treated when that rate is exceeded: it is either dropped (for a meter band of type "Drop") or remarked (for a meter band of type "DSCP remark").

A last point concerns the order with which OpenFlow Flow Modification Messages are sent to a switch. Obviously, when referring from a flow table entry to a meter or group, these must have been already created. This means that the processing of the OpenFlow meter (or group) modification message at the switch must precede the OpenFlow flow modification message. As described below, our algorithm schedules the transmission of modification messages to meet this constraint.

2) The proposed algorithm

The goal of the algorithm is to build the list of OpenFlow Modification Messages to deliver to each network node involved in the support of the virtual links that compose the Virtual Network. In fact, three lists of messages are computed for each node: 1) the list of OF Meter Modification Messages, 2) the list of OF Group Modification Messages and 3) the list of OF Flow Modification Messages. Once computed, the algorithm informs the controller to convey them to node in the order specified above. Our algorithm uses (without being a necessity) the bundle mechanism introduced in OF version 1.4, for message grouping and ordering as well as, to store and pre-validate them on each node before a global confirmation across multiple nodes. The main steps of the algorithm are:

For each node *i* crossed by the virtual network [line 2], and for each virtual link *k* [line 3]:

- If the node *i* is the source node of virtual link *k*, then it inserts an OpenFlow meter into meterModMessageList and keeps the meterID for later use (when building the flow modification message of the flow table entry that refers to the meter). [line 4 – line 8]
- If the virtual link *k* is split at node *i*, then it inserts a group into groupModMessageList and it inserts a flow rule into

flowModMessageList with that group as action of "Write-Actions" instruction. If the node i is the source of virtual link k, a meter instruction is also added [line 20 – line 34]. If k is not split at node i, then it inserts a flow into flowModMessageList with a simple output port action in the "Write-Actions" instruction, and eventually, a meter instruction if the node is the source of virtual link k [line 10 – line 19].

- It transmits successively the meterModMessageList, groupModMessageList, flowModMessageList to node *i's* bundle. This latter is configured with the ordered flag set, to request that the messages of the bundle are processed in the order of arrivals. [line 37-End].

VNET Deployer Algorithm

Inputs: V,E,K, f_k^t, $f_k(i,j)$, V': the set of nodes crossed by at least one virtual link.

Variables : match: ofp_match; group_mod: ofp_group_mod; nextHops : set of nodes; flowModMessageList: ofp_flow_mod[]; groupModMessageList: ofp_group_mod[]; meterModMessageList: ofp_meter_mod[]; groupID, meterID, bundleID : integer;

```
1   begin
2   for each i in V' do
3       for each unicast link k in K
4         if(i = s_k) then
5           meterID ← get_meter_id(i,k)
6           insert_into (meterModMessageList, {id =meterID, bands [0] ={rate =
7   b_k, type = drop}})
8         end if
9         nextHops ← computeNextHops(i, k,V', E, f)
10        if (|nextHops| = 1) then
11          for each j in nextHops do
12            if(i = s_k) then
13              insert_into (flowModMessageList, { match=create_match(),
14              instructions ={meter: meterID, write-actions: output.port=Pij})
15            else
16              insert_into (flowModMessageList, { match=create_match(),
17              instructions ={write-actions: output.port =Pij}})
18            end if
19          end for
20        else
21          groupID ← get_group_id(i,k)
22          group_mod.id ← groupID; group_mod.type ← select
23          for each j in nextHops do
24            group_mod.buckets[j] ← {weight =f_k(i,j), actions ={ouput.port=Pij} }
25          end for
26          insert_into (groupModList, group_mod)
27          if (i = s_k) then
28            insert_into (flowModMessageList, {match=create_match(),
29            instructions = {meter: meterID, write-actions: group.id=groupID}})
30          else
31            insert_into (flowModMessageList, match =create_match(),
32            instructions ={write-actions: group.id =groupID}})
33          end if
34        end if
35      end for
36      bundleID ← get_bundle_id(i)
37      OFPBCT_OPEN_REQUEST {id =bundleID, flags =ordered}
38      for each msg in meterModMessageList do
39        OFPT_BUNDLE_ADD_MESSAGE {id = bundleID, message = msg}
40      end for
41      for each msg in groupModMessageList do
42        OFPT_BUNDLE_ADD_MESSAGE {id = bundleID, message = msg}
43      end for
44      for each message in flowModMessageList do
45        OFPT_BUNDLE_ADD_MESSAGE {id =bundleID, message = msg}
46      end for
47      reset (meterModMessageList); reset (groupModMessageList); reset
48      (flowModMessageList)
49  end for
50  for each i in V' do
51      OFPBCT_CLOSE_REQUEST {id = get_bundle_id(i)}
52      OFPBCT_COMMIT_REQUEST {id =get_bundle_id(i)}
53  end for
54  end
```

B. VNET Resource Allocator

This section describes the Integer Linear Programming (ILP) formulation that we propose to solve the resource allocations for the virtual network. Virtual network Requests arrive and are processed in sequence with no information on future requests. For each request, the output is the set of routes (with the bandwidth allocations at each supporting physical link and the number of flow table, meter table and group table entries at each crossed node) that support each of the virtual links composing the request. This algorithm extends our previous work [8] by considering meter tables and group tables (in addition to the flow tables) as network resources to assign. For meter tables, things are quite simple since OpenFlow meters are only activated on the source nodes of the virtual links. The algorithm simply checks that meter table entries are still available at source nodes. If so, the algorithm assigns a meter table entry on each of these nodes and proceeds with the other resources as describes below. If not, the VNET request cannot be accepted.

1) Resource-related assignment variables

In addition to $f_k^t(i,j)$ and $f_k(i,j)$, our model considers the following variables:

- $l_k(i)$: is a boolean variable that specifies the number of entries that are installed in node i flow table to support virtual link k with the assumption that all entries consume the same amount of resources regardless of the complexity of the match operation and the related instructions to perform. A flow table entry is added if at least one node i port is supporting traffic from k (equations 1).

$$\forall j \in V, \forall (j,i) \in E : g_k(j,i) \le l_k(i) \qquad (1.b)$$

$$l_k(i) \le \sum_{\substack{j \in V \\ (j,i) \in E}} g_k(j,i) \qquad (1.c)$$

  where $g_k(j,i)$ is an intermediate boolean variable that indicates whether some bandwidth from link (j,i) is assigned to virtual link k. It is derived from another set of more focused intermediate variables $g_k^t(j,i)$ that reflects whether the flow of packets of virtual link k destined to t is supported by the physical link (j,i) (i.e.$g_k^t(j,i) = 0$ if $f_k^t(j,i) = 0$ and $g_k^t(j,i) = 1$ otherwise).

- $n_k(i)$: is an integer variable that counts the number of group table entries assigned to k at node i. As described in section V.A, a group table entry is added when splitting or when duplicating packets (for point-to-multipoint links).

- f_{max}: is the maximum link utilization (when considering all network links) after request acceptance.

- u_{max}: is the maximum flow table utilization (when considering all network nodes) after VNET acceptance.

2) Problem Constraints

The constraints on bandwidth allocations are described in equations 2 to 8. Equation 2 reflects the linearization of the Max operator applied to variables $f_k^t(i,j)$ to get $f_k(i,j)$. Equations 3 and 4 have a similar purpose and focus respectively on f_{max} and u_{max}, which are minimized by the objective function (as explained below).

$$\forall k \in K, \forall (i,j) \in E, \forall t \in T_k: \quad f_k^t(i,j) \le f_k(i,j) \qquad (2)$$

$$\forall(i,j)\in E:\ \ \frac{1}{B_{ij}} * \left(B'_{ij} + \sum_{k\in K} f_k(i,j)\right) \leq f_{max} \qquad (3)$$

$$\forall i\in V:\ \ \frac{1}{U_i} * \left(U'_i + \sum_{k\in K} l_k(i)\right) \leq u_{max} \qquad (4)$$

Equation 5 ensures that the bandwidth assigned to each virtual link k at link (i,j) does not exceed the remaining bandwidth. Equation 6 represents the usual flow conservation constraints.

$$\forall(i,j)\in E:\ \sum_{k\in K} f_k(i,j) \leq\ B_{ij} -\ B'_{ij} \qquad (5)$$

$\forall k \in K, \forall t \in T_k, \forall i\ \in V:$

$$\sum_{k\in K}\left(f_k^t(i,j)\ -\ f_k^t(j,i)\right) = \begin{cases} b_k & if\ i = s_k \\ -b_k & if\ i = t \\ 0 & else \end{cases} \qquad (6)$$

Equation 7 is a channeling constraint between integer and boolean variables: $f_k(i,j)$ and $g_k(i,j)$. It also constrains the virtual link k's bandwidth assignment at a physical link to the requested bandwidth $\mathrm{b_k}$. Equation 8 constrains the bandwidth that is assigned to the flow of packets destined to a specific virtual link's end-point. The inequality on the right side ensures that the bandwidth requirement of the virtual link is never exceeded. The inequality on the left side directs path-splitting and avoids the multiplication of splits with low bandwidth allocations. Indeed, if active, path splitting is feasible only if the bandwidth allocated to the splits respects a minimum threshold b_k^{min}. In practice, $\mathrm{b_k^{min}}$ is a ratio of b_k, $b_k^{min} = PS_{ratio} * b_k$, with $PS_{ratio} \in [0,1]$.

$\forall k \in K, \forall(i,j) \in E:$

$$g_k(i,j) \leq f_k(i,j)\ and\ f_k(i,j) \leq b_k\ *\ g_k(i,j) \qquad (7)$$

$\forall k \in K, \forall(i,j) \in E:$

$$b_k^{min}\ *\ g_k^t(i,j)\ \leq f_k^t(i,j)\ \ and\ \ f_k^t(i,j)\ \leq\ b_k\ * g_k^t(i,j) \quad (8)$$

The constraints related to switching resource allocations are described in equations 9 and 10. Equation 9 simply ensures that with the addition of flow table entries needed by the virtual links composing the VNET, the size of network nodes' flow tables remains below their maximum size.

$$\forall i\in V: \sum_{k\in K} l_k(i) \leq\ U_i\ - U'_i \qquad (9)$$

Equations 10 constrain the allocations of group table entries. Equation 10.b applies when no group entries are needed for the virtual link k at node i (it neither crosses i nor requires a flow split or packet duplication). Equation 10.c applies when a group entry is needed. Finally, equation 10.d simply ensures that the addition of group entries that are needed by the virtual links respect the maximum size of all the group tables.

$$\forall k \in K, \forall i \in V:\ 0 \leq n_k(i) \leq 1 \qquad (10.a)$$

$\forall k \in K, \forall(i,j) \in E:$

$$n_k(i) \leq \left(\sum_{\substack{k\in K\\(i,j)\in E}} g_k(i,j)\right) - g_k(i,j) \qquad (10.b)$$

$\forall k \in K, \forall\ (i,j_1),(i,j_2) \in E, j_1 \neq j_2:$

$$n_k(i) \geq g_k(i,j_1) + g_k(i,j_2) - 1 \qquad (10.c)$$

$$\forall i \in V:\ \sum_{k\in K} n_k(i)\ \leq N_i - N'_i \qquad (10.d)$$

3) Objective function

The objective function aims at minimizing link and node *r*esource consumption but also at distributing the consumed resources among nodes and links in order to reduce the creation of bottlenecks. Both contribute to improve the admissibility of forth coming requests. As shown in expression 11, it consists of four components, each weighted with a parameter that controls the impact of the component on the resolution process. The first two concern bandwidth allocations and the last two are their analogues for flow table entries allocations.

$$\textbf{Minimize}\ \alpha_1 * \frac{1}{|E|} * \sum_{(i,j)\in E}\left(\frac{1}{B_{ij}} * \left(B'_{ij} + \sum_{k\in K} f_k(i,j)\right)\right) + \alpha_2 * f_{max} + \beta_1 * \frac{1}{|V|} * \sum_{i\in V}\left(\frac{1}{U_i} * \left(U'_i + \sum_{k\in K} l_k(i)\right)\right) + \beta_2 * u_{max} \qquad (11)$$

V. Implementation & performance analysis

A proof-of-concept prototype of the proposed ADN was implemented and applied to provide ADN services to DDS-based distributed interactive simulation applications for vehicle driver training. These latters involve networked driving simulators that evolve in a shared virtual world. The mobility of the simulators (in the virtual world) brings dynamicity in the data flows that are delivered/consumed by each simulator and on their associated QoS. The topology of a real campus network with 31 nodes and 55 links (with 100Mbps and 1Gbps) was considered as the SDN substrate. Apart the "autonomic manager" which was partially implemented, all the components of section III were implemented. The "VNET Resource allocator" implements the algorithm presented in section IV.B using concert technologies C++ as the modeling layer and IBM CPLEX 12.6 as solver. The "Virtual Network Deployer" implements the algorithm of section IV.A on top of the Floodlight SDN controller platform. The SDN substrate is emulated with mininet with OFSoftSwitch13 network nodes. One of the interesting features of Floodlight V.1.0 and OFSoftSwitch13 is their full support for OpenFlow (OF) 1.3. This allows the use of meters, and groups of type "all" and "select". Moreover, they offer an experimental support for OF 1.4, including bundles with atomic modification features. An online "application classifier" was also implemented for DDS; it inspects for "DDS subscription request" packets (or QoS change or cancellation) to identify the new application flows and their QoS needs.

Evaluations of the resource allocation algorithm were also realized to assess its general performance and benefits in comparison to some Shortest Path (SP) heuristics. The results presented below were based on the hierarchical real campus network topology cited above with the flow table and group table sizes respectively set to 2000 and 512 entries. VNET requests are assumed to arrive following a Poisson process with an arrival rate that is varied from 4 requests to 10 requests per 100 Unit of Time (UT). Each VNET request is made of a number of virtual links that is randomly chosen from 1 to 4. Each virtual link has a number of destinations that is randomly chosen from 1 to 4 and has a bandwidth requirement randomly chosen from 1 to 3 Mbps. Once a VNET request is accepted it lasts till the end of the experiment, which is set to 10000 UT. Path Splitting (PS) is activated with a PS_{ratio} set to 0,3.

The considered SP heuristic is defined as follows. A cost function assigns a cost to each physical link that is inversely proportional to its current available capacity. For each couple of end-points that belong to a virtual link, the physical path with the minimum cost is chosen. If the bandwidth available on the chosen path is below the bandwidth required by the virtual link, the corresponding request is not admitted.

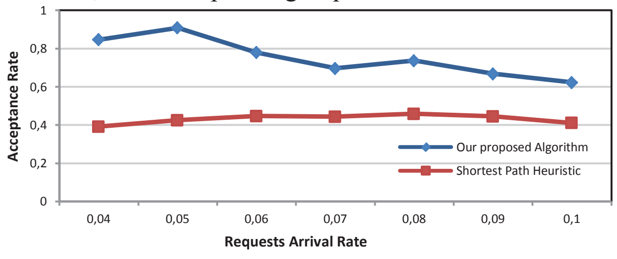

Figure 2 – Requests Acceptance Rate

Figure 2 describes the requests acceptance rate as a function of the requests arrival rate. Under this high load, it clearly shows that our algorithm achieves an acceptance rate significantly greater than the heuristic. Our experiments show that with our algorithm the average link utilization is between 60 and 80% at some backbone links, we observe, at the end of the experiments, that more than 95% of their capacity has been allocated. They are the main reason for VNET request rejection.

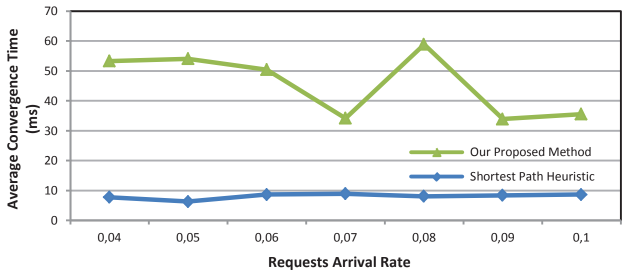

Figure 3 – Convergence times

Figure 3 presents the average time needed by our algorithm and the SP heuristic to compute the optimal allocations associated to a VNET request. For our algorithm, the convergence times remain at acceptable levels: on average below 60ms and a longest convergence time of 400ms.

VI. Conclusion and Future Work

This works proposes an SDN Based Application Driven Network that is able to provide QoS enabled data-paths on an application flow basis. This allows providing tailored network services to applications while using efficiently network resources (even when application requirements are dynamic). This detailed consideration of applications' communication profile has a cost in terms of scalability. Clearly, the intention is not to apply the proposed ADN to any application and in any context. It rather targets real-time or business critical applications in an enterprise or campus networks, where scalability is not the primary concern. The proposed ADN was implemented and applied to prove its feasibility.

Perspectives to this work mainly concern pursuing the evaluation of our proposal by considering other types of applications and other contexts. Extending the "application classifier" to other applications and evaluating its accuracy and reaction time.

Acknowledgment

This work is partially funded by the French National Research Agency (ANR), the French Defense Agency (DGA) under the project ANR DGA ADN (ANR-13-ASTR-0024) and by European Union's Horizon 2020 research and innovation programme under the ENDEAVOUR project (grant agreement 644960).

References

[1] Justin Fielder and Thierry Grenot, "Killer Apps 2012" research study, Easynet Global services and Ipanema Technologies, 2012.

[2] S. Gorlatch, T. Humernbrum, and F. Glinka, "Improving QoS in real-time internet applications: from best-effort to Software-Defined Networks," in *International Conference on Computing, Networking and Communications, 2014.*

[3] H. E. Egilmez, S. T. Dane, K. T. Bagci, and A. M. Tekalp, "OpenQoS: An OpenFlow controller design for multimedia delivery with end-to-end Quality of Service over Software-Defined Networks," in *APSIPA ASC, 2012.*

[4] I. Bueno, J. I. Aznar, E. Escalona, J. Ferrer, and J. Antoni Garcia-Espin, "An opennaas based sdn framework for dynamic qos control," in *IEEE SDN for Future Networks and Services,* 2013

[5] P. Sharma, S. Banerjee, S. Tandel, R. Aguiar, R. Amorim, and D. Pinheiro, "Enhancing network management frameworks with SDN-like control," in *IFIP/IEEE International Symposium on Integrated Network Management, 2013.*

[6] S. Tomovic, N. Prasad, and I. Radusinovic, "SDN control framework for QoS provisioning," in *22nd Telecommunications Forum Telfor, 2014.*

[7] M. F. Bari, S. R. Chowdhury, R. Ahmed, and R. Boutaba, "PolicyCop: an autonomic QoS policy enforcement framework for software defined networks," in *IEEE SDN for Future Networks and Services*, 2013

[8] T. Zinner, M. Jarschel, A. Blenk, F. Wamser, and W. Kellerer, "Dynamic application-aware resource management using Software-Defined Networking: implementation prospects and challenges," in IEEE *Network Operations and Management Symposium, 2014*

[9] A. Georgi, R. G. Budich, Y. Meeres, R. Sperber, and H. Hérenger, "An integrated SDN architecture for application driven networking," International Journal on Advances in Systems and Measurements, vol. 7, pp. 103–114, 2014.

[10] MRV, "Application-Aware Networking at A Glance." white paper, 2013

[11] Open Networking Foundation, "OpenFlow Notification Framework*"*, ONF TS 014, Version 1.0, October 2013.

[12] M. Capelle, S. Abdellatif, M.J. Huguet, P. Berthou, "Online Virtual Links Resource Allocation in Software-Defined Networks", IFIP Networking , 2015.

[13] Ben Alaya, M.; Monteil, T., "FRAMESELF: A Generic Context-Aware Autonomic Framework for Self-Management of Distributed Systems," in 21st IEEE WETICE, 2012.

[14] Object Management Group, "Data-Distributed Service for Real-Time Systems, "OMG, version 1.4. Sept. 2014.