A Generic Framework to Support Application-Level Flow Management in Software-Defined Networks

Lautaro Dolberg*, Jérôme François*†, Shihabur Rahman Chowdhury‡, Reaz Ahmed‡,
Raouf Boutaba‡ and Thomas Engel*

*SnT (Interdisciplinary Centre for Security, Reliability and Trust), University of Luxembourg, Luxembourg
Email: lautaro.dolberg@uni.lu, thomas.engel@uni.lu

†INRIA Nancy Grand Est, France, email: jerome.francois@inria.fr

‡David R. Cheriton School of Computer Science, University of Waterloo, Canada
Email: sr2chowdhury@uwaterloo.ca, r5ahmed@uwaterloo.ca, rboutaba@uwaterloo.ca

***Abstract*—Software-Defined Networking (SDN) provides a highly flexible flow management platform through a logically centralized controller that exposes network capabilities to the applications. However, most applications do not natively use SDN. An external entity is thus responsible for defining the corresponding flow management policies. Usually network operators prefer to control the flow management policies, rather than granting full control to the applications. Although IP addresses and port numbers can suffice to identify users and applications in ISP networks and determine the policies applicable to their flows, such an assumption does not hold strongly in cloud environments. IP addresses are allocated dynamically to the users, while port numbers can be freely chosen by users or cloud-based applications. These applications, like computing or storage framework, use diverse port numbers which amplifies this phenomenon. This paper introduces higher-level abstractions for defining user- and application-specific policies. These policies are then automatically mapped to OpenFlow rules by retrieving flow-based information of active users and applications in real-time. We implemented this framework and evaluated its practicality by measuring the underlying overhead.**

I. Introduction

Traditional networks rely on distributed routing protocols to decide the route of a packet. Typically, routing protocols have very little interaction with the applications. A stronger coupling between the network and the application can enable the network to more efficiently adapt to continuously changing application demand for bandwidth.

Recently, Software Defined Networking (SDN) [1] has been proposed to decouple a network's control plane from the data plane with the use of a logically centralized which can expose an interface to applications. However, OpenFlow [2], the de facto standard for SDN does not yet have the capability to inject application level context into the network. In this paper, we propose an application level flow management framework for SDN.

A major challenge in designing an application level flow management framework is to identify the applications and even the users associated with a flow. One approach can be to rely on IP addresses and transport layer port numbers. Such an hypothesis can hold for some classes of applications (*e.g.* Web). However, with the popularity of cloud computing, a wide variety of applications are appearing and their flows are hard to track just using IP address and port numbers. There is so a strong need to monitor flows from the applications in a flexible way under such dynamic conditions [3]. Finally, due to privacy concerns and the growing usage of encryption techniques [4], deep packet inspection is not a viable approach for identifying applications.

In this scope, our framework transparently maps application-level policies (involving application and user names) to OpenFlow rules (IP addresses, protocols and port numbers), which alleviates the necessity for the control applications (those interacting with the Northbound interface of the controller) to keep track of the network characteristics (like location) of users and applications themselves. To achieve this end, application-level information is retrieved in real-time through local remote system agents, which can be easily deployed in a cloud platform where both network and computational infrastructure are hosted by the same entity. Our work enables the association of flows with applications and users. In the rest of the paper, application-level information refer to both user- and application-specific information, namely their names.

The rest of the paper is organized as follows. Related works are reviewed in Section II. Based on small examples, Section III refines the problem description whereas the solution is introduced in Section IV along with the framework design. Its implementation is described in Section V. Section VI is dedicated to the evaluation of the framework under different scenarios. Finally, we conclude with possible future directions in Section VII.

II. Related work

In this section, we discuss the traffic management techniques for traditional networks as well as for SDN. Common approaches to handle application-level traffic flows assume that

978-1-4673-9486-4/16/$31.00

Ingress port	Mac Src Addr	Mac Dst Addr	Ip Src Address	Ip Dst Address	Protocol	Src port	Dst port	Instructions
*	*	*	*	1.2.3.*	*	*	*	Set Mac src addr=AB:CD:EF:00:11:33, Mac dst addr = AB:CD:EF:00:11:44, forward to port 5
*	*	*	1.2.3.4	*	TCP	*	22	Drop

TABLE I: Examples of OpenFLow flow table entries (* represents a wildcard to match any value)

all packets in a flow embed specific attributes, for example IPv4 Type-of-Service (TOS) field or the IPv6 Traffic Class (TC) field. However, this kind of approaches lack flexibility (need for dedicated routers, fixed number of service classes, no uniform interface to update all router configurations, etc.) and are of limited use as claimed in [5].

Authors in [6], [7], have exercised the concept of application-aware network by relying on deep packet inspection (DPI). However, DPI does not scale well with large traffic volumes and is not robust against strong encryption mechanisms. Another approach consists in delegating applications to raise their network-awareness through active network monitoring, especially to adapt their configuration through a control loop [8]. However, end hosts inevitably have a narrow view of the network and cannot make globally optimized decisions unlike the network operator.

Chowdhury *et al.* proposed the concept of Coflow [9], where flows from the same application are grouped together for scheduling purposes.

With the introduction of SDN, it becomes possible for the applications to specify their requirements. PANE [10] introduces a SDN controller with an interface for applications to express their requirements. Some of the early works tried to increase application awareness in the network by focusing on data analytics using software agents running at the end-hosts [11] or based fixed port numbers or traffic classes [12]. Similarly, authors in [13] relies on port numbers for tracking applications. Authors in [14] have modified PostgreSQL server to pro-actively request SDN controllers for bandwidth reservation during SQL query execution.

Most of these work are well summarized in [15], however, none of them are mature in the sense that they are focused on a specific application. Only user applications for which interfaces are defined can be used.

III. Examples and problem description

Before describing the problem, let us consider the following scenario where we want to block SSH access for a user connected at host 1.2.3.4. One way to implement this policy is to use OpenFlow rules on the IP address of the host as shown in Table I. However, if the user moves to a different host or port, this rule no longer implements the policy, i.e., block SSH access of that particular user.

To circumvent such an issue, we need to extend the definition of a rule and include the user and application context into the flows along with other fields. For instance, it should be possible to specify from which application a flow originates from, regardless of the IP addresses.

Although the previous scenario assumes a unique and well identified port number, those can be changed by users. Hence, application-based policies would require that users inform the cloud operator about the port numbers they use. Second, large applications or frameworks may use numerous ports. In contrast, using a single application name would be easier.

In this paper, we address the problem of extending OpenFlow by allowing the definition of rules giving application level context to the network. However, it does not prevent to use other features offered by OpenFlow to match flows.

IV. Application-Level Flow Management

A. System design

Our framework is composed of two components: the augmented controller (AC) and the system probe (SP).

The SP is instantiated at all hosts hosting users and applications to be referred to when defining policies. Such an assumption is realistic since our framework is envisioned to be leveraged in data-centers.

It keeps track of application-level information about traffic flows at these hosts. This information is stored in the database of the AC. Then the AC takes as input application-level policies composed of rules. These rules are very similar to OpenFlow rules with additional fields to match application-level information. The rule engine of the AC translates these rules to OpenFlow and forwards it to the OpenFlow controller and then to the switches.

Figure 1 describes a sequence of actions corresponding to the lifetime of a connection. The SP is responsible for monitoring connection open and close events in real time to keep the AC database up-to-date. The SP first intercepts a new connection creation event, retrieves the corresponding user and application names (application-level information), gets the associated IP addresses, protocol and port numbers (flow-level information) and sends it to the AC. Although our approach is mainly focused on managing flows at an IP level, however, but it could be generalized to layer 2 as well.

Upon reception, the AC stores the information in its local database. When a user defines a policy, the AC maps it to the entries in the database to request the OpenFlow controller to create a new rule. The OpenFlow controller then modifies the flow tables of the switch using *flowmod* message. It is important to note that the AC will also store current active rules in its database, such that when the termination of a

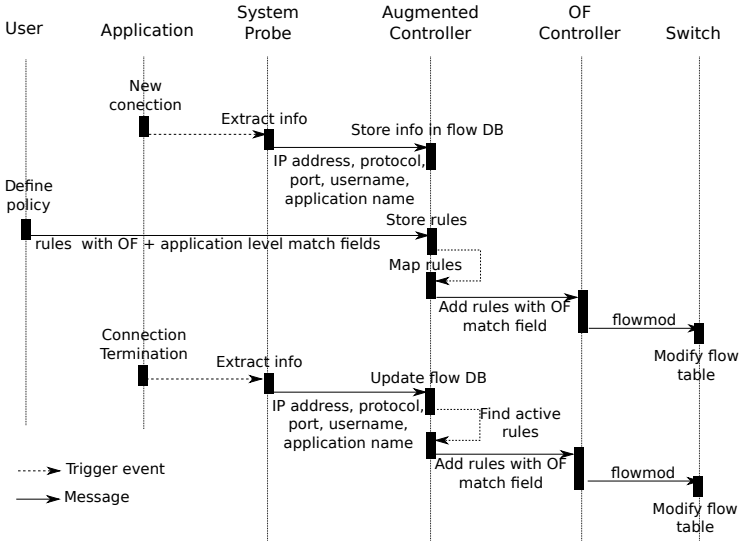


Fig. 1: Installing and updating flow tables

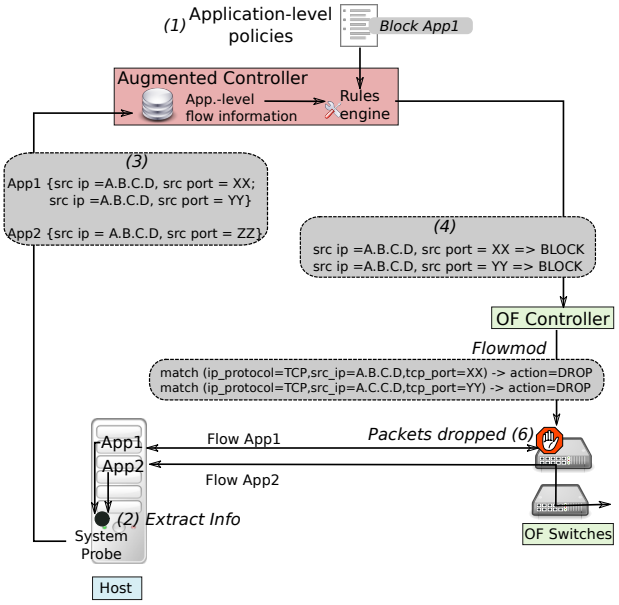


Fig. 2: Example

connection is observed by the SP, it can delete such rules with a similar process. It is an additional benefit of our approach which helps in limiting the size of flow tables while usual timeout-based approach leads to keeping rules longer than the real duration of flows.

B. Controllers

As discussed earlier in Section IV-A, there are two controllers in our system: an existing OpenFlow controller and our proposed AC. As claimed in the introduction, fully opening network configuration to user-level applications can be unrealistic. However, the operator may provide some limited interfaces through OpenFlow, as for example, to let an application express its expected QoS. In our design, the OpenFlow controller acts as a proxy between the AC and the OpenFlow switches. Using our framework, all services offered by the OpenFlow Controller are thus extended to be specified as application-level information. This eases the definition of requests by end-users, who do not have full knowledge.

Moreover, the user-level applications can specify even finer grained policies through the interface of the AC. For example, a Map-Reduce job can tell the AC about different parameters of its shuffle phase or a FTP transfer can tell the AC about the size of the file being transferred. The AC will forward such additional information to the OpenFlow controller and its applications assuming they can handle it. Designing application specific optimizations are out of the scope of the paper. We discussed related literature in Section II.

C. Example

As an example, in Figure 2, two applications, *App1* and *App2*, are hosted on a single host. First, a policy is issued requesting to block all traffic from *App1* (1). When *App1* and *App2* are executed and opens connections, the SP is able to associate the network flow signatures to processes of *Host* (2). This information is then sent to the AC (3), which reveals that *App1* has two connections on different ports. It is then stored into the local database of the AC. Based on that and the given policies, the rules engine of the AC creates OpenFlow specific rules. Since no policy is related to *App2*, only flows of *App1* are blocked (4). The controller uses a *flowmod* message to define a *DROP* instruction over those flows (5). As a result, flows from *App1* are blocked but not those from *App2* (7).

V. Implementation

In our current implementation of th as a Floodlight module [16], actions are limited to the set of actions that Floodlight can perform on the network configuration but we have extended the definition of matching fields to include application-level information. Our Implementation is released under an open-source license and is available at https://github.com/ldolberg/ALFMAN.

The Augmented Controller is implemented as a service running on Linux/Unix OS, exposing a REST API supporting the following operations:

- *Flow update*: this function allows the SP to specify the associations between flows, applications and users. The AC maintains a database of these associations. We have implemented the database using Redis[1], which provides high performance in-memory storage and allows fast reactivity of the AC.
- *Set Policy*: this function allows to enable or disable a rule in the AC depending on the available features of the underlying OpenFlow controller, *e.g.*: blocking or setting a higher priority or pre-allocating bandwidth for a given application running in a participating host.

[1]http://redis.io/

- *Push Message*: this function allows the SP to forward a custom message upon request of a running application to the AC in the active mode.

The SP is implemented using *Python 2.7* and the standard POSIX tool *Netstat*. It periodically polls the OS tables to extract the *pid* for identifying open sockets and running processes.

For the communication between the SDNS and AC, we have developed an in house protocol. This protocol is based on HTTP and allows SDNS and AC to talk to each other using in a RESTful manner.

VI. Evaluation

A. Experimental environment

A functional verification of our framework have been performed by applying application-level QoS policies. However, we can expect performance degradation implied by the overhead of the SP. In particular, we measure the network overhead induced by the SP, which is also equivalent to the AC network load per monitored system since these two components are communicating together. Since the AC is intended to be deployed in a central component with high computational power or even being distributed, assessing its system overhead (CPU and memory) is not justified.

For the evaluation, a virtual network using Mininet [17] has been deployed with the following elements: Open vSwitch [18], Floodlight controller, the AC and end-host nodes with the SP enabled. The host nodes and the AC were running *Ubuntu 14.04 LTS (GNU/Linux 3.13.0-32-generic x86_64)* (as virtual machines). All are connected to a single LAN. The physical host is a dedicated server running Debian 7.6.0 64-bits, equipped with 16 GB of RAM and 4 2.7 GHz cores. Each virtual machine was provided with 2 GB of RAM and 2 GHz of CPU.

Depending on the underlying applications, we varied the number of end hosts but the number of nodes is not an essential parameter as the core of the evaluation is to assess the performances of the SP, which is executed on every individual node. The results are so independent of the number of nodes, however, considering multiple applications running on a single node is much appropriate.

For evaluation purpose, traffic is generated using Iperf [19], Apache2 HTTP Server[2], Pidgin[3] and Links[4]. Being diverse applications, they are helpful to assess the framework with various traffic patterns. However, the vast majority of the results presented in this work correspond to those obtained using Iperf similarly to many related works mentioned in Section II. Iperf is used with default parameters, except we increase the TCP window size to 64KB to consume as much bandwidth as possible. For Web, we used JMeter[5] to generate HTTP traffic from clients.

[2]https://httpd.apache.org/, accessed on 12/03/15
[3]https://pidgin.im/, accessed on 12/03/15
[4]https://links.twibright.com/, accessed on 12/03/15
[5]http://jmeter.apache.org/

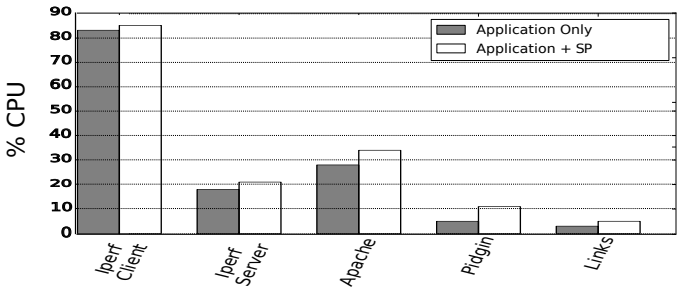


Fig. 3: CPU overhead of the SP

B. Metrics

As highlighted previously, the SP is the most sensitive service of our approach because it runs on the same machine as the user applications while the AC may run on a dedicated server. For evaluating the impact caused by the SP, the consumption of system resources, CPU and memory, during the total length of the experiments are recorded and expressed as a percentage of the total amount of CPU and memory respectively.

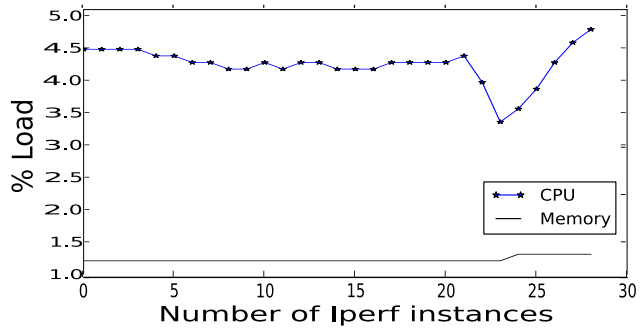
C. Single Application Overhead

In this experiment, the host system performance is measured in terms of CPU usage and memory consumption. All applications described in Section VI-A have different communication profiles. They are first run without the SP instantiated and then they are run again while the SP is running. The difference of resource consumption and throughput in these two scenarios will give the overhead of the SP. We report the result in Figure 3. The impact of the SP remains very low for all applications. More precisely, a maximal difference of 6 percentage points and an average increase of 2% in the total CPU consumption i observed. Therefore, the SP has a negligible impact on a host running a single application which would so not suffer from performance degradation.

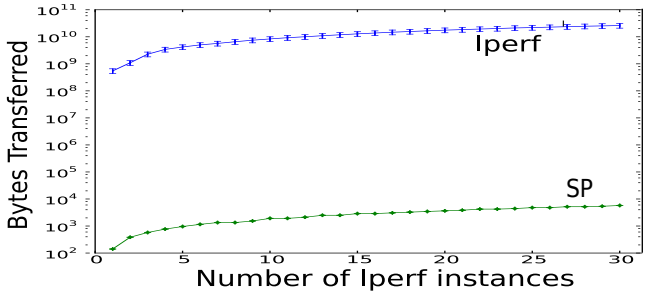
D. Performance with Multiple Applications

To emulate multiple co-existing applications, several instances of the Iperf client with different users are executed on a single virtual machine and they are individually by the SP. This setting allows us a fine control over the number of flows to monitor by just varying the Iperf instances (one flow per instance).

Figure 4 presents the average results based on 100 executions (Figure 4(b) includes also the Y axis error-bar). In these experiments, the overhead of the SP was measured individually in order to assess the SP as a single entity. In Figure 4(a), the consumption of system resources (CPU and memory) remains steady with increasing number of monitored *Iperf* instances. The variation among experiments is bounded within the interval 0.2%-0.5%. The CPU usage is more variable than memory but cannot be even considered as significant considering the scale. Moreover, similar variations are exhibited by all applications. After investigation, we found that the larger variation observed with 23 Iperf instances is an



(a) CPU and Memory consumption



(b) Network usage

Fig. 4: Overhead of the SP

artifact due to operating system operations, such as memory pagination, which shortly pauses Iperf, making it consuming less CPU. Based on this experiment and the previous one, the SP service impact remains low regardless the type and number of monitored applications.

In addition to system overhead, Figure 4(b) highlights the network overhead by showing the number of transmitted bytes. To fairly evaluate network overhead, we compare the number of transmitted bytes by the SP to the AC and by the applications themselves, *i.e.* the Iperf instances. In both cases, the number is growing but in highly different proportions since the scale is logarithmic. Actually, for each new Iperf instance, the SP interacts with the AC but most of the transfered bytes are related to the connection management between the SP and the AC, for example initialization and keep-alive messages to avoid connection reinitialization. This explains the steady value for the number of transmitted bytes by the controller around 192 Bytes per flow (excluding L2 to L4 headers). Thus, our approach does not introduce significant overhead in terms of CPU, memory and network resource consumption.

VII. Conclusion

In this paper, we have proposed an application-level flow management framework. It is composed of distributed probes gathering necessary system-level information and a central controller acting as an intermediary with the SDN controller. The experiments show its viability and practicability since the induced computational and network overhead remain low with increasing number of flows to monitor. Our framework especially fits with a cloud infrastructure where port- and address- based application classification is not relevant due to multi-tenancy. Some limitations are related to security issues, which we plan to address in our future work.

References

[1] N. Feamster, J. Rexford, and E. Zegura, "The road to sdn: An intellectual history of programmable networks," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, pp. 87–98, Apr. 2014.

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[3] Z. Li, Y. Shen, B. Yao, and M. Guo, "Ofscheduler: a dynamic network optimizer for mapreduce in heterogeneous cluster," *International Journal of Parallel Programming*, pp. 1–17, 2013.

[4] K. Finley, "Encrypted web traffic more than doubles after NSA revelations." [Online]. Available: http://www.wired.com/2014/05/sandvine-report/

[5] S. Das, Y. Yiakoumis, G. Parulkar, N. McKeown, P. Singh, D. Getachew, and P. Desai, "Application-aware aggregation and traffic engineering in a converged packet-circuit network," in *Optical Fiber Communication Conference and Exposition (OFC/NFOEC)*, 2011, pp. 1–3.

[6] M. Wijnants, B. Cornelissen, W. Lamotte, and B. De Vleeschauwer, "An overlay network providing application-aware multimedia services," in *Workshop on Advanced architectures and algorithms for internet delivery and applications*. ACM, 2006.

[7] M. Jarschel, F. Wamser, T. Hohn, T. Zinner, and P. Tran-Gia, "SDN-based application-aware networking on the example of youtube video streaming," in *European Workshop on Software Defined Networks (EWSDN)*. IEEE, 2013.

[8] J. Bolliger and T. Gross, "A framework based approach to the development of network aware applications," *Software Engineering, IEEE Transactions on*, vol. 24, no. 5, pp. 376–390, 1998.

[9] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with varys," in *Proceedings of the 2014 ACM conference on SIGCOMM*. ACM, 2014, pp. 443–454.

[10] A. D. Ferguson, A. Guha, Liang *et al.*, "Participatory networking: An api for application control of sdns," in *ACM SIGCOMM 2013*. ACM, 2013, pp. 327–338.

[11] A. Das, C. Lumezanu, Y. Zhang, V. Singh, G. Jiang, and C. Yu, "Transparent and flexible network management for big data processing in the cloud," in *Presented as part of the 5th USENIX Workshop on Hot Topics in Cloud Computing*. USENIX.

[12] G. Wang, T. Ng, and A. Shaikh, "Programming your network at run-time for big data applications," in *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 2012, pp. 103–108.

[13] J. Liu, L. Zhu, W. Sun, and W. Hu, "Scalable application-aware resource management in software defined networking," in *Transparent Optical Networks (ICTON)*, July 2015.

[14] P. Xiong, H. Hacigumus, and J. F. Naughton, "A software-defined networking based approach for performance management of analytical queries on distributed data stores," in *SIGMOD International Conference on Management of Data*. ACM, 2014.

[15] 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 *Network Operations and Management Symposium (NOMS)*. IEEE, 2014.

[16] "Floodlight, SDN Controller," http://Floodlight.openflowhub.org/, accessed: 2014-08-14.

[17] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *SIGCOMM Workshop on Hot Topics in Networks (HotNets)*. ACM, 2010.

[18] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker, "Extending networking into the virtualization layer," in *Workshop on Hot Topics in Networks (HotNets)*. ACM, 2009.

[19] M. Gates, A. Tirumala, J. Dugan, and K. Gibbs, *Iperf version 2.0.0*, Part of Iperf's source code distribution, NLANR applications support, University of Illinois at Urbana-Champaign, Urbana, IL, USA, May 2004. [Online]. Available: http://iperf.sf.net