Proceedings of the Federated Conference on Computer Science and Information Systems pp. 969–977
DOI: 10.15439/2016F398
ACSIS, Vol. 8. ISSN 2300-5963

Highly customizable framework for performance evaluation of *LOOM*-based SDN controllers

Szymon Mentel
Erlang Solutions,
ul. Batorego 25, Kraków, Poland
Email: szymon.mentel@erlang-solutions.com

Marek Konieczny, Sławomir Zieliński
Department of Computer Science
AGH University of Science and Technology,
al. Mickiewicza 30, Kraków, Poland
Email: {marekko, slawek}@agh.edu.pl

***Abstract*—The article presents an innovative method for assessing performance of modular SDN controllers, focusing on test customization. The method was validated by construction of a testing framework that gives its users the opportunity to emulate various network traffic patterns by using arbitrarily chosen applications, rather than simulating workloads. The presented solution is more comprehensive than others available in contemporary SDN environments also because it that takes into account specific features of modular SDN controllers.**

I. Introduction

Software Defined Networking (SDN) is a promising concept for computer networking. The main idea behind it is the separation of control and data planes. The control plane is moved to a logically centralized, software-based controller. However, as the scale of the application grows, the performance of a controller can become a bottleneck and appropriate techniques for controller scaling need to be employed. For example consider using hierarchical controllers [1], increasing the autonomy of data plane components [2], [3], distributing controllers [4] or elastic scaling [5]. It is also possible to use adaptation mechanisms utilized in SOA systems [6]. Another method - which is of focus for the article - is increasing controller's performance by building upon the modularity offered by modern programming and runtime environments, such as Erlang.

Although there are works related to measuring the performance of monolithic SDN controllers, there is little research regarding modular ones. Therefore, this paper is focused on the development of a method and framework for assessing performance of modular SDN controllers. The framework also provides its users with means required to emulate various network traffic patterns. It was implemented upon open technologies, such as the *LOOM Controller Framework* [7], *Mininet Cluster Edition*[1], and *Open vSwitch (OVS)*[2].

The paper organization is as follows. Sections II and III overview the domain of the presented research, including research in the area of measuring performance of SDN controllers. The survey forms a base for choosing a testing approach and designing the framework architecture - both are presented in Section IV. Section V presents a proof-of-concept implementation of a testing environment and in this way demonstrates the options for customizing the framework to a practical use case. Moreover, it presents results gathered from a series of experiments and the conclusions that were drawn from them. The article ends with concluding remarks and acknowledgments, which form Sections VI and VII, respectively.

[1]https://github.com/mininet/mininet/wiki/Cluster-Edition-Prototype/
[2]http://openvswitch.org/

II. Background

In *SDN*, forwarding logic no longer has to run on specific hardware built into the switches, routers or other networking devices. The *control plane* functions, implemented by a *SDN controller*, can be achieved in general purpose programming languages and run on regular servers. Thanks to that, *control plane* development is much more flexible, cheaper and faster.

To make user traffic reach its destinations, the control plane needs to communicate with data plane devices. Currently, the most popular protocol for control to data plane communication is *OpenFlow*, created and maintained by *Open Networking Foundation*[3]. The *OpenFlow* protocol allows to define simple actions (e.g. drop packets or send packets specific ports), but it is also possible to build more complex policies [8]. *SDN* and *OpenFlow* concepts can also be applied to PON networks [9] , vehicular networks [10] or multimedia transmission over *HTTP* [11].

The research in the area of SDN results in many new concepts and technologies being developed. Traffic patterns for new applications can be highly distributed and can require extremely short response times [12], [13]. As the result, various solutions regarding network topologies have been proposed [14], [2]. However, there is no consistent framework that would facilitate early stage performance evaluation of SDN environments designed to host new applications. The framework presented in this article aims to fill the gap.

Because new control plane developments are typically implemented using open controllers and new data plane developments use virtual switches, this section overviews the respective categories.

Controllers. As *SDN* and *OpenFlow* gain popularity, many *controllers* appear on the market. In most cases, they are distributed as open source projects. Table I lists some of the

[3]https://www.opennetworking.org/

controllers along with their core technologies, the number of contributors and popularity indicators. The latter are expressed by the number of cloned repositories (table column *Forks*) and the number of people which found the project interesting (table column *Stars*). Based on this information (and on the last activity times) one can estimate the potential and size of a particular developing community.

Name	Language	Devs	Stars	Forks	Last activity
Ryu	Python	60	457	339	recently
Floodligth	Java	58	313	323	recently
POX	Python	16	282	285	2 years ago
ONOS	Java	74	147	132	recently
Trema	Ruby	18	238	77	recently
OpenDaylight	Java	66	82	114	recently
NOX	C++	6	75	66	one year ago
OpenMul	C	2	18	8	recently
Beacon	Java	3	8	3	4 years ago
LOOM	Erlang	22	26	8	7 months ago

TABLE I: Popularity of OpenFlow controllers (based on *GitHub* data).

In many cases *controllers* are in fact entire *SDN platforms* (they are often referred to as *monolithic controllers*). They consist of many components which deliver rich functionalities to users, such as topology discovery or security analysis tools. *OpenDaylight*[4], the leading production controller, is a representative of this group. On top of the controller developers write single, monolithic network applications. They use supported languages, APIs, and libraries provided by the framework, compile the entire platform and run the created application as a single process. Note however that an error in any part of the application can have adverse effects on the entire system.

Erlang community promises to deliver extensible, robust *OpenFlow* controllers based on the *LOOM* [7] framework. The main goal is to make the controller scalable and distributed (that is the main rationale for using *Erlang* virtual machine). *LOOM* has a modular and layered design. In control plane it consists of *Network Execution* and *Application* layers. LOOM-based controllers share core libraries, such as low-level library for encoding and decoding *OpenFlow* messages, an abstract interface to *OpenFlow* switches and a driver with a common base for *OpenFlow* controllers. LOOM developer creates a controller (specific to his needs) that can be later orchestrated by the network execution layer. Such approach clearly differs from monolithic controllers.

Virtual switches. When it comes to switches supporting *OpenFlow*, there are many hardware and software products on the market. However, from the perspective of early stage testing, software ones are the most important, because it is much easier to build test environment based on software switches (and larger environments can be prepared more easily). Moreover, because software switches often implement the latest version of the *OpenFlow* standard, new features can be used as soon as a reference implementation is available. Software switches are also widely used in production environment (e.g., *VMware NSX* [3]), so their evaluation is also valuable.

Erlang community (together with *Infoblox* and *Erlang Solutions*) is currently working on *LINC-Switch*[5]. It is run in operating system user space to facilitate usage flexibility and quick development. *LINC-Switch* is used as a simulator of an optical network, in one of *Open Network Operating System (ONOS)*[6] controller use cases. *LINCX*[7], which alse comes from Erlang community, is a new, faster version of *LINC-Switch*.

In the presented framework we use *Open vSwitch* because of its popularity and implementation quality. The switch supports multiple protocols and network standards. Moreover, because *OVS* is integrated with *Mininet*[8], it is easier to build test environment based on the switches distributed among multiple physical servers.

III. Related work

There are a few noteworthy research efforts in the area of measuring SDN controllers performance that can be leveraged by new ones, especially to identify the metrics to be measured and key test environment parameters. Shalimov et al. [15] measured latency, and throughput of controllers based on (1) the number of cores the controller uses, (2) the number of switches connected to the controller, and (3) the number of hosts in the network. For their tests, they created their own *Haskell* emulator of *OpenFlow* switches - *hcprobe*[9].

Similar work [16], evaluates performance of two Java-based controllers: *OpenDaylight* and *Floodlight*. The authors used a cluster of hosts running *Cbench*[10]. The tool directly stresses a controller by sending OpenFlow Packet-In messages. Another study [17], describes metrics of *ONOS*, including *Flow installation throughput*. The metric shows a number of flow mods, which can be sent by the SDN control plane in response to requests coming from applications or network events.

While all the mentioned characteristics were studied using network emulators, authors of the [18] proposed a methodology that utilizes network virtualization. To interconnect virtualized hosts, *Mininet* was used. They evaluated performance, as the number of Packet-In messages (requesting installation of new flows), a controller can handle per second.

As an example of more structured and holistic approaches, consider *IETF* draft describing the methodology for reporting SDN controller performance [19]. The document touches three aspects, namely: the number of switch sessions a controller can handle, the network size (number of nodes, links, and hosts) a controller can discover, and forwarding table capacity.

[4]https://www.opendaylight.org/

[5]http://flowforwarding.github.io/LINC-Switch/
[6]http://onosproject.org/
[7]https://github.com/FlowForwarding/lincx
[8]http://mininet.org/
[9]https://github.com/ARCCN/hcprobe
[10]https://goo.gl/CEgQ68

No performance tests are covering modular controllers (especially written in Erlang), although the authors of [15] mentioned that they examined *FlowER*, and it was not ready for evaluation under heavy workloads. Nonetheless, Erlang is designed to be used in the telecommunication industry and seems to be well suited for SDN use cases. Therefore, we decided to focus on controllers developed upon *LOOM* framework. To our best knowledge, this article is the first to present any results on Erlang SDN/OpenFlow controller performance testing.

Additionally, the presented framework aims to facilitate profiling of network topologies in context of specific applications (which generate specific traffic) by enabling the developer to define the virtual network topology to be constructed by the framework, and to deploy the applications in the network.

IV. Architecture of solution

Before starting to test performance of a *SDN* controller, it should be decided what to measure, and how to perform the measurements, i.e., what kind of environment to use to generate load against the controller, how to collect the data, and how to interpret it. This article proposes a method that follows a generic methodology consisting of the following steps:

1) Deciding upon testing objectives.
2) Building the test environment.
3) Configuring the test system.
4) Running tests and collecting data.
5) Interpreting the test results.

The following subsections refer to the steps in more detail.

A. Testing objectives

For measuring controllers performance two characteristics seem to be most important: *latency* and *throughput*. Latency describes an average amount of time that is needed to process a request. Although *OpenFlow* controllers have to be able to process various requests, those related to topology changes and new traffic flows are crucial. Regarding networking devices configuration, when a new traffic flow occurs, a controller has to handle *Packet-In* requests. Handling them, from a controller perspective, usually boils down to installing flow entries in the data plane device that originated the Packet-In, in order to instruct it what to do with packets belonging to the new flow. Note that latency in processing a Packet-In depends on the kind and context of a particular request, so the values measured can be different for particular application, network topology, etc.

Throughput is a metric describing how many requests a controller can handle, on average, in a given period. The requests related to receiving an unrecognized traffic flow may be sent simultaneously from multiple data plane devices to a single controller. Thus, this metric is sensitive to the characteristics of the network, i.e., its topology, number of switches and hosts, etc.

Regarding the controller itself, the following parameters affect its performance (i.e., both latency and throughput): number of cores used by the controller, the amount of memory used by the controller, the number of instances of the controller (if it can operate as a distributed system), and workload distribution strategy.

The framework presented in the article provides means for building test environments that enable the user to manipulate all the mentioned variables and measure metrics including, but not limited to, latency and throughput.

B. Building the Test Environment

There are three generic approaches to building an environment that mimics a real network generating Packet-In messages:

- *Building an SDN network from hardware.* Testing a controller with a real network can provide most accurate results. On the other hand, the approach is expensive, not flexible, and not scaling well. Additionally, it is not easy to gather statistics from all the devices on the network.
- *Emulating an SDN network by using specialized software.* There are switch emulators designed for testing SDN networks, e.g., *Cbench* and *Hcprobe*, which offer easy configuration and automation of test scenarios. It is relatively easy to scale the environments based on emulators – it usually involves adding new servers/virtual machines with emulator instances. This approach has been used in evaluation of *Network Management Systems* [20]. Note however, that emulators are hard to synchronize across instances, making it more difficult to coordinate test scenarios, and to gather and analyze the results.
- *Using network virtualization to simulate an SDN network.* In this approach, a tool such as *Mininet* can be used to virtualize a complex network on hardware hosts by using Linux containers interconnected with software switches. Such a tool allows for easy automation, controlling traffic generation and gathering statistics.

The presented list is not exhaustive. It is easy to imagine an approach that combines all of the above. It is also possible to test scalability using discrete-event network simulator such as *NS-3*[11].

For the implementation of our framework, We chose virtualization-based approach because of the ease of test automation and low cost.we chose virtualization-based approach. Figure 1 presents essential components of a test environment that could be developed with the proposed test framework. It outlines three main modules, as well as the data channels between them.

The core elements of the *Network Module* are those related to network simulation: *OpenFlow switches*, *hosts*, and *topology*. To enable the user to reproduce arbitrarily designed topologies, the framework uses *Mininet Cluster Edition*, and *Open vSwitch*. The emulated network characteristics can be very close to what would be observed in production networks also because the *Mininet* hosts come with a full implementa-

[11] https://www.nsnam.org/

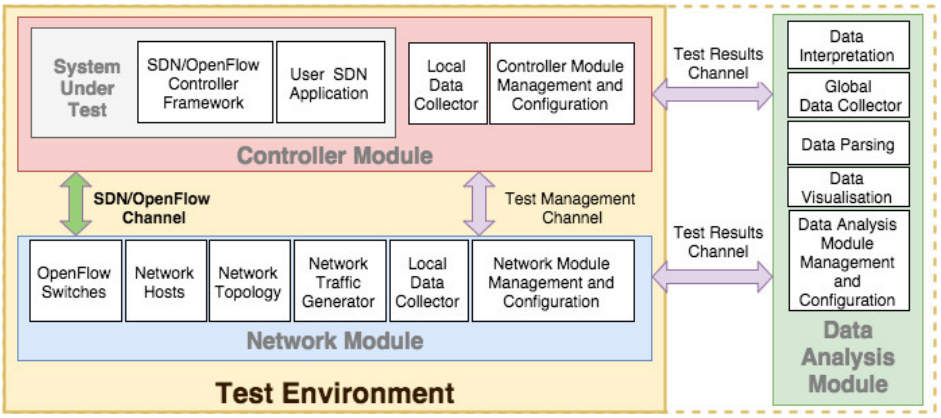


Fig. 1: Test Environment Components

tion of OSI protocol stack, so arbitrarily chosen applications can be used for testing, and act as *traffic generators*.

Local Data Collector is a component responsible for collecting and persisting tests-related data, like traffic metrics or counters of OpenFlow messages sent by the switches (i.e., the load metrics). The *Test Results Channel* indicates the likely transfer of collected data to the *Data Analysis Module*. The *Network Module Management and Configuration* component's task to facilitate automatization of *Network Module* set up and its coordination with the *Controller Module* via the *Test Management Channel*.

The *Controller Module* module encapsulates the *SDN/OpenFlow Framework* and *User SDN Application* components, that together form a fully capable SDN controller, which is the *System Under Test*(SUT). Because the framework is intended to be used mainly with modular, Erlang-based SDN controllers, a natural choice was to leverage the LOOM framework as the basis for SUT implementations, so that custom controllers can be easily built into the framework. The *Local Data Collector* and *Management and Configuration* components have similar responsibilities as their counterparts from the *Network Module*. Similarly to the *Network Module* case, the *Test Results Channel* indicates the likely transfer of collected data to the *Data Analysis Module*.

The *Data Analysis Module* is related to data produced during the tests: gathering, parsing, presenting and interpreting information. Depending on a particular test scenario, this component could be placed either inside the environment (in the case of runtime analysis) or outside (in the case of offline analysis). *Global Data Collector* is a component that gathers all the metrics from other modules so that they can be processed further in a consistent way. *Data Parsing* and *Data Visualization* components are designed for presenting the data in readable formats. *Data Interpretation* denotes additional tools that facilitate interpretation processes. *Data Analysis Module Management and Configuration* component is intended for management and configuration of the data analysis process.

C. Configuring the Test System

The configuration of a test environment can result in changes of traffic patterns. The key network configuration parameters include number of switches, number of hosts per switch, topology, and deployed applications.

Regarding the controller, there are usually less parameters to change. However, changing the hardware specification (e.g., memory, CPU), setting some options at the controller level (e.g., the number of cores it can use) or choosing the controller system structure option (centralized, distributed) have to be considered.

D. Data Collection and Interpretation of Results

Metric probes (i.e., values read from a given metric) can be taken at different intervals, cover different time spans, and have various aggregation rules. For example, when measuring load as the number of Packet-In requests, one needs to decide on how often the value will be measured, what time it will cover (e.g., last 10 minutes), and how it will be aggregated (e.g., maximum, average). Those decisions have to be taken keeping in mind the planned test scenarios.

In the presented framework, the core of the component responsible for collecting test data is implemented by using the *exometer*[12] package - it is run as a user application dependency in the same Erlang Virtual Machine. The package allows for instrumentation of Erlang code, so that data reflecting system performance can be exported to a variety of monitoring systems.

The following *exometer* metrics are implemented:

Application Packet-In Handle Time: histogram metric that captures elapsed time it takes for the SDN application (part of the SUT) to handle a single Packet-In request.

LOOM Packet-In Handle Time: similar as above, but time for the whole SUT (i.e., *LOOM* framework and SDN application) is captured.

Packet-In Count: counts Packet-In messages that are processed by the SDN application.

[12]https://github.com/Feuerlabs/exometer

Packet-Out Count: counts Packet-Out messages that are sent out by the SDN application.

Flow-Mod Count: counts Flow-Mod messages sent out of the SDN application.

In order to make updates to the **LOOM Packet-In Handle Time** metric, the *LOOM* framework code had to be instrumented. In the proof of concept implementation it is assumed that **each Packet-In message has a corresponding Packet-Out message**. Based on that, each Packet-In message that arrives to the controller framework is marked with a timestamp just after being decoded. Then, the mark is copied into the corresponding Packet-Out message. When the message is about to be encoded before sending, the mark is retrieved and the metric value is computed. Note that such instrumentation would not be possible with closed controller code.

The value of **Application Packet-In Handle Time** is based on the same assumption. The first timestamp is made when a Packet-In message enters the application process, and the second when Packet-Out and possibly Flow-Mod messages are sent.

The presented set of metrics is not closed and can be extended by metrics specific for a particular controller. To make use of the metrics collected during a test run, appropriate subscriptions and reporters need to be configured. Each reporter reads values from a metric at given interval and writes them to a file. The values are then processed by the *Data Analysis Module*. Depending on test scenarios' specifics, data can be analyzed (and, e.g., visualized) at run time or saved for later reference.

V. Framework Evaluation

In order to prove the framework usability and test whether it fulfills the requirements, a learning switch application was developed. The switch (called later *LOOM Switch*), was built upon the *LOOM Framework*. Together with the framework it formed a simple, but fully functional, SDN controller. The basic design decisions regarding the *LOOM Switch* were as follows:

1) There is no topology detection service: *LOOM Switch* fills in the forwarding tables based on the standard MAC addresses learning algorithm.
2) There is exactly one forwarding table (FT) for each data plane switch connected to the controller.
3) A packet that does not match any of the flow table entries is buffered in the data plane switch and its portion is sent to the controller using a Packet-In message.
4) At most one flow table entry is installed in the data plane switch that sent a Packet-In message.
5) Flow tables of other data plane switches are not changed.

Figure 2 depicts an example test setup. It demonstrates how the most important parts of the test environment are related to each other. Thick solid lines represent network links. Arrows show OpenFlow channels, connecting each of the switches to the *LOOM Switch* controller. Test management and data collection components were omitted for clarity.

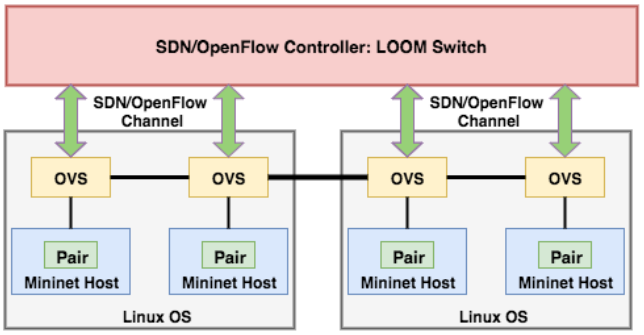


Fig. 2: An example test setup

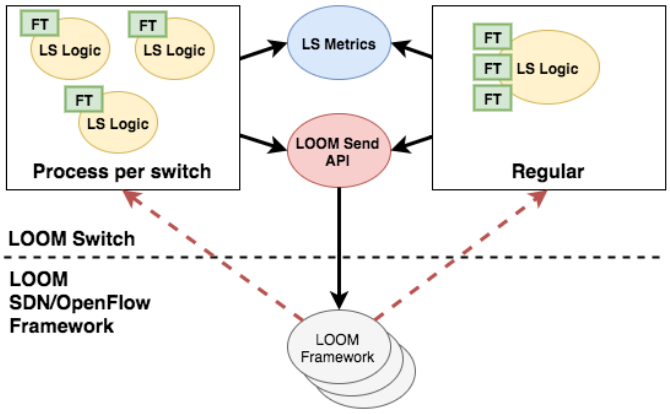


Fig. 3: Options of LOOM Switch Deployment

The network used for proof-of-concept testing was built from a number of switches that formed a linear topology, each of which had an even number of hosts attached. Network traffic was generated by a simple application called *Pair*, that was deployed on *Mininet hosts*. The application was sending out a UDP datagram to a random counterpart, receiving a reply, changing its configuration (including MAC address) and flushing its *ARP* table. Such an application provided an evenly distributed, constant load during the test runs.

A. Testing scenarios

Because Erlang is a highly concurrent functional language, and the presented framework supports the specifics of Erlang LOOM, it was a natural choice to evaluate two options of *LOOM Switch* deployment. In the first, called later *regular*, a single Erlang process handled requests coming from all the switches. In the second, called later *process per switch*, each switch had its dedicated process. The intention was to compare performance offered by the deployment options.

Figure 3 shows the *LOOM Switch* deployment architectures from the perspective of Erlang processes, for *LOOM Switch* application instance that serves 3 switches. The dashed arrows in the figure represent invocations of *LOOM Switch* callbacks from the LOOM framework.

In the case of *process per switch* deployment, each forwarding table (implemented as a hash map) is associated with

a separate Erlang process. In the *regular* deployment case, all the tables are accessed from one process. Regardless of the deployment option, *LOOM Switch* uses a single process for asynchronously handling metrics (**LS Metrics**) and for sending messages to the switches via *LOOM Send API*.

B. Experimental results

We conducted framework proof-of-concept tests for configurations in which the controller (i.e., *LOOM Switch*) used 2, 4 or 8 cores. As a representative test case, we chose the results gathered for 4 cores serving the *LOOM Switch*, run in the *process per switch* deployment option.

Figures 4a, 4b, 4c and 4d contain plots with metrics obtained during respective test runs. Each graph is labeled with *(X, Y)*, where X is a number of switches in the (linear) topology and Y is a number of hosts per switch. The overall number of hosts (240) was constant across all the test runs, and the same number of traffic generator (i.e., *Pair*) instances was used, so the performance was expected to be dependent only on the deployment option and the number of switches used. Note that the same number of hosts and conversations resulted in different loads in terms of Packet-In requests due to the assumptions regarding LOOM Switch and the varying number of data plane switches in the topology.

The following metrics are presented:

- **Application Packet-In Handle Time**, i.e., SDN application (part of the controller) latency, called later **LS Time**,
- **LOOM Packet-In Handle Time**, i.e., SDN controller latency, called later **LOOM Time**.

The collected values correspond to a 20 minute period of a stable load. To get a time frame in which the load is stable, the measurement was started after 15 minutes of test execution. The values were sampled every minute.

The difference in values of **LS Time** and **LOOM Time** is quite impressive. Average **LS Time** values, indicating the time it takes *LOOM Switch* to handle a request are becoming lower and less significant in relation to **LOOM Time**, which represents the overall time needed to serve a request by the controller (which consists of the *LOOM Switch* and the LOOM framework). In percentages, they take $38.7\,\%$, $9.8\,\%$, $1.7\,\%$, and $0.6\,\%$ of **LOOM Time**, respectively.

The phenomenon of decreasing values of **LS Time** metric can be explained as follows. In the *process per switch* deployment option, computations related to switching decisions are spread across the available cores, because each switch is served by a separate process. Along with the increase of Packet-In messages rate the *LOOM* framework gets overloaded and slower in serving requests, in comparison to the *LOOM Switch* application. As a result, as the framework needs more time to process the messages, and they are delivered at a lower pace to the *LOOM Switch*. Consequently, a message spends less time being processed by *LOOM Switch*, because it is able to process it instantly, since a major part of the messages is buffered (and stuck) in the framework.

C. Comparison of deployment options

In the presented test case of *process per switch*, four cores deployment scenario, the LOOM framework was observed to be a bottleneck. As presented in section V-B, the **LS Time** constituted only a few precent of the whole **LOOM Time**, which clearly denotes that the Packet-In messages were stuck in the framework. The highest load measured was about $228\,000$ Packet-Ins per minute with average times of $3\ \mathrm{ms}$ and $572\ \mathrm{ms}$ for **LS Time** and **LOOM Time** metrics, respectively.

Similar values of **LOOM Time** ($580\ \mathrm{ms}$) were observed in case of the *regular* deployment for the load of 73000 Packet-Ins per minute. In that case, the values of **LS Time** and **LOOM Time** metrics were almost equal, indicating that most of the processing time was consumed by the *LOOM Switch* application. Its pace of serving requests was significantly slower than the pace of request delivery performed by the *LOOM* framework.

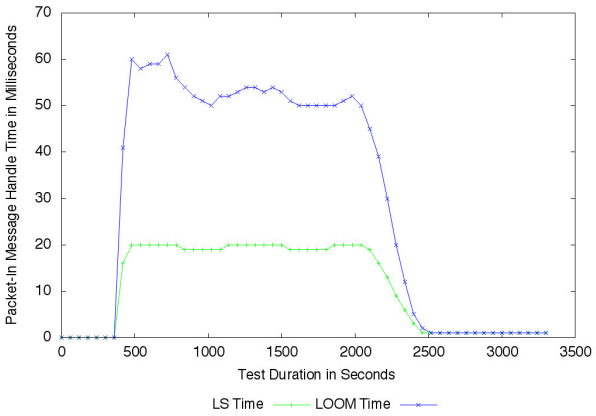
From the results gathered from two test sets for the two *LOOM Switch* deployment options, it is clear that the way of deploying a modular SDN controller has a tremendous impact on its performance. The same hardware configuration, 4 cores per controller, yielded about three times better processing capability in the *process per switch* case than in the *regular* one while offering the same processing time. Note that the processing time values cannot be directly compared with results presented in other articles, because it was measured for the maximum number of Packet-In messages that the controller was able to process. As shown in [5], the processing time grows rapidly after excessing a certain threshold, mainly due to request queuing.

In the following section, we present an experiment that was conducted to check whether adding more cores to the LOOM controller improved the situation, i.e., lowered the processing time for the maximum load identified for 4 cores.

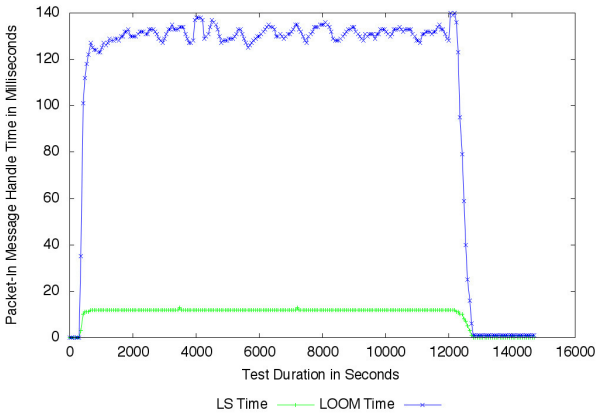
D. Scalability Test for Process Per Switch Deployment

Figure 5 presents a plot for the tested controller running in the *process per switch* mode, that depicts the relation between controller performance and the number of cores available for the schedulers. The plot was created based on experiments conducted on the same topologies that were used for comparing deployment options. Each test scenario (denoted (X,Y) on the X axis) has three values for three corresponding controller configurations that vary only in the number of used cores. Each point represents an average value of the **LOOM Time** metric measured in a given test scenario and controller configuration. Note that the experiment was conducted under heavy load (228000 Packet-In messages per minute processed), which proved to be the limit of processing capability of the tested controller running on 4 cores.

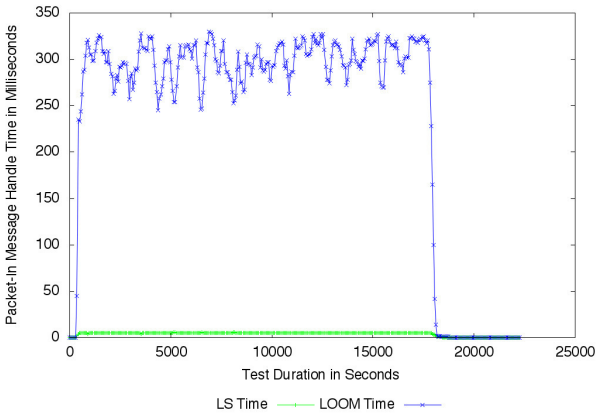
The most important conclusion from comparing the results depicted in figure 5, is that while switching from 2 cores to 4 gives a significant increase in performance in all test scenarios, such situation does not happen when doubling the number of cores again, from 4 to 8. In the case of 5 schedulers the controller performed even a bit worse. As a consequence, it



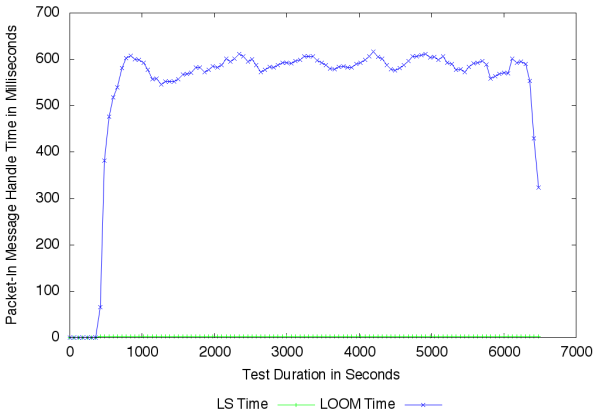
(a) Test Scenario (5, 48);
61000 Packet-Ins/minute



(b) Test Scenario (10, 24);
117000 Packet-Ins/minute



(c) Test Scenario (15, 16);
172000 Packet-Ins/minute



(d) Test Scenario (20, 12);
228000 Packet-Ins/minute

Fig. 4: Handle Request Time in Function of Time for Different Test Scenarios

should be stated that the bottleneck (for 8 cores) does not result from insufficient computing power, but rather from the LOOM framework implementation.

The reason that *LOOM* running *LOOM Switch* in the discussed deployment option scales well only up to the certain point (to 4 cores given the presented results), may be related to *locks*. Some processes in *LOOM* may be acquiring the same *lock*. The more cores available to the system, the more processes can execute simultaneously. As a result, there could be more failed attempts to acquire a particular *lock*. Of course, there could be other potential reasons but checking the locks seems to be a good starting point in searching for an explanation.

VI. Conclusions

In the article, we presented our research on the development of an innovative framework for measuring the performance of SDN controllers. As a part of the presented research, we built a test environment to evaluate the framework practically by testing specific features of modular SDN controllers. The presented results are promising, and form a good base for further development and analysis. We plan to extend our environment with different traffic generators to emulate various application workloads (e.g. multimedia sessions, big data processing). It will allow measuring performance of SDN controllers in real scenarios, with real applications and with different traffic conditions.

The framework presented in this article is designed for assessing various network topologies, and controller deployments, with arbitrarily chosen applications generating network traffic. Such a tool is needed, e.g., for early stage testing of a new network-intensive application that creates network traffic with certain characteristics. Using the framework presented in the article one can define both the network topology and traffic patterns that run on top of it as well and observe the effects of modifying multiple attributes on the performance of the controller.

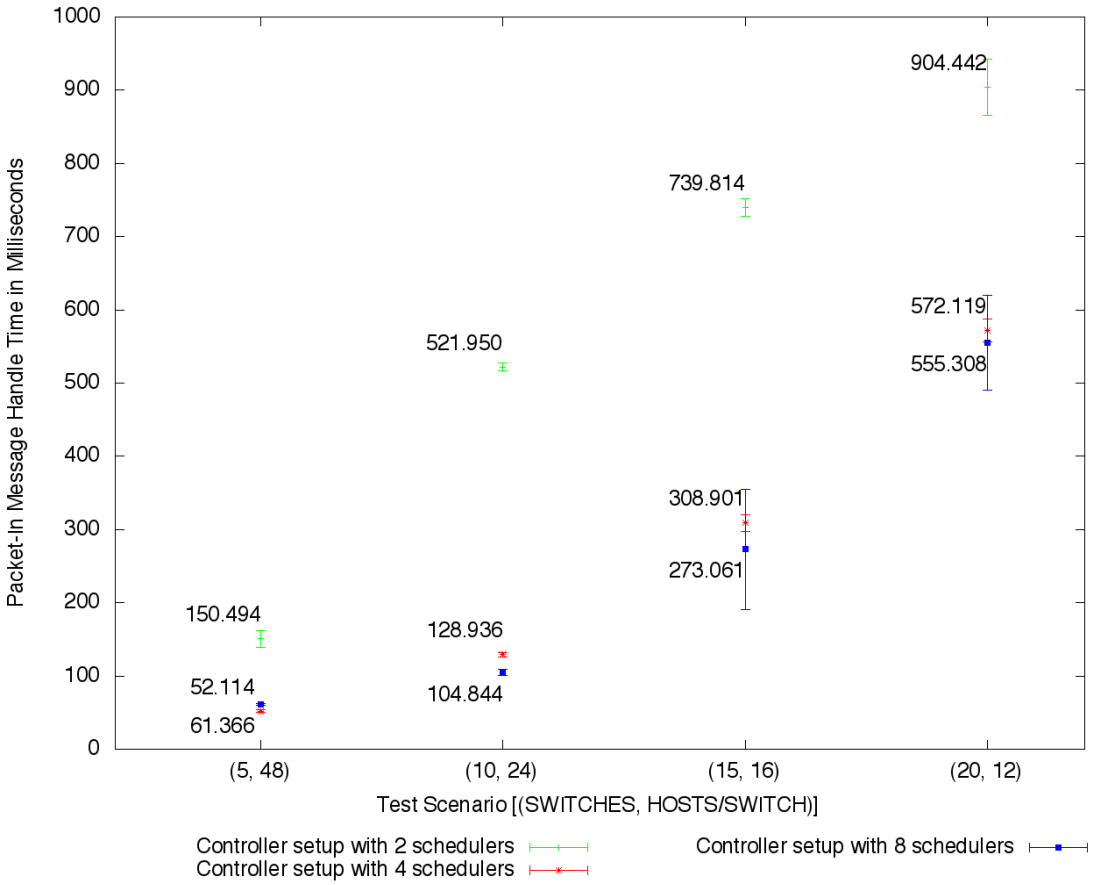


Fig. 5: *LOOM* Handle Request Time in Different Test Scenarios

VII. Acknowledgments

The research presented in this paper was partially supported by the National Centre for Research and Development (NCBiR), Poland, project PBS1/B9/18/2013 and AGH statutory research grant no. 11.11.230.124.

References

[1] S. Hassas Yeganeh and Y. Ganjali, "Kandoo: a framework for efficient and scalable offloading of control applications," in *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 2012, pp. 19–24.

[2] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Vl2: a scalable and flexible data center network," in *ACM SIGCOMM computer communication review*, vol. 39, no. 4. ACM, 2009, pp. 51–62.

[3] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, P. Ingram, E. Jackson *et al.*, "Network virtualization in multi-tenant datacenters," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014, pp. 203–216.

[4] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama *et al.*, "Onix: A distributed control platform for large-scale production networks." in *OSDI*, vol. 10, 2010, pp. 1–6.

[5] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella, "Towards an elastic distributed sdn controller," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 7–12, 2013.

[6] T. Szydlo and K. Zielinski, "Adaptive enterprise service bus," *New Generation Computing*, vol. 30, no. 2-3, pp. 189–214, 2012.

[7] LOOM, http://flowforwarding.github.io/loom/.

[8] D. Jullier, M. Konieczny, and S. Zieliński, "Applying software-defined networking paradigm to tenant-perspective optimization of cloud services utilization," in *Computer Networks*. Springer, 2015, pp. 193–202.

[9] P. Parol and M. Pawlowski, "Towards networks of the future: Sdn paradigm introduction to pon networking for business applications," in *Computer Science and Information Systems (FedCSIS), 2013 Federated Conference on*. IEEE, 2013, pp. 829–836.

[10] I. Stojmenovic and S. Wen, "The fog computing paradigm: Scenarios and security issues," in *Computer Science and Information Systems (FedCSIS), 2014 Federated Conference on*. IEEE, 2014, pp. 1–8.

[11] C. Cetinkaya, Y. Ozveren, and M. Sayit, "An sdn-assisted system design for improving performance of svc-dash," in *Computer Science and Information Systems (FedCSIS), 2015 Federated Conference on*. IEEE, 2015, pp. 819–826.

[12] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan, "Speeding up distributed request-response workflows," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 219–230.

[13] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. ACM, 2015, pp. 123–137.

[14] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4, pp. 63–74, 2008.

[15] A. Shalimov, D. Zuikov, D. Zimarina, V. Pashkov, and R. Smeliansky, "Advanced study of sdn/openflow controllers," in *Proceedings of the 9th Central and Eastern European Software Engineering Conference in Russia*. ACM, 2013.

[16] Z. K. Khattak, M. Awais, and A. Iqbal, "Performance evaluation of

opendaylight sdn controller," in 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS), 2014.

[17] Open Network Operating System, "Raising the bar on sdn control plane performance and scalability," http://goo.gl/AizqcC, 2015.

[18] M. P. Fernandez, "Evaluating openflow controller paradigms," in IEEE 27th International Conference on Advanced Information Networking and Applications, 2013.

[19] B. Vengainathan, A. Basil, M. Tassinari, V. Manral, and S. Banks, "Benchmarking methodology forsdn controller performance," Working Draft, IETF Secretariat, Internet-Draft draft-bhuvan-bmwg-sdn-controller-benchmark-meth-01, July 2015. [Online]. Available:https://tools.ietf.org/html/draft-bhuvan-bmwg-sdn-controller-benchmark-meth-01

[20] K. Grochla and L. Naruszewicz, "Testing and scalability analysis of network management systems using device emulation," in Computer Networks. Springer, 2012, pp. 91–100.