Evaluation of an SDN-based Microservice Architecture

Anton Hölscher, Mikael Asplund, and Felipe Boeira
Department of Computer and Information Science
Linköping University, Sweden
Email: anton@holscher.se, mikael.asplund@liu.se, felipe.boeira@liu.se

***Abstract*—Microservice architectures decompose applications into individual components for enhanced maintainability and horizontal scaling, but also comes with an increased cost for orchestrating the services. Software-Defined Networks (SDNs) enables the dynamic configuration of network switches using controllers. In this paper we propose a microservice architecture that leverages SDN to orchestrate the microservices with the goal of reducing the orchestration latency cost. We perform a set of experiments using Mininet in which we implement a tailor-made microservice application that uses SDN for orchestration in combination with a set of different controllers and load balancers. Our results show that our proposed architecture performs in the same order of magnitude as a corresponding monolithic system.**

***Index Terms*—microservices, software-defined networking, latency, OpenFlow, load balancing**

I. Introduction

Developing and maintaining large-scale software projects are error-prone and demanding tasks. There is reason to believe that dividing the system into smaller sub-programs, where each sub-program provides a single functionality of the system, could potentially result in reduced overall system complexity. Such a system architecture is typically referred to as a *Microservice Architecture* [3].

While microservices are developed as individual functions that communicate in order to provide the required functionality, they are usually presented to the users as a single system. This can be achieved by adding a *microservice orchestrator* which is a separate microservice responsible for delegating all incoming requests to the intended microservice. However, adding an orchestrator entails increased latency overhead of the system since all incoming packets need to be received and forwarded by the orchestrator.

In this paper we explore the possibility of lowering the latency impacts of the orchestrator by incorporating the orchestrating behaviour within a switch. The main motivation for incorporating the microservice orchestrator into the switch with the help of an SDN-based approach is to reduce the overall latency of service requests from clients. We have performed a set of experiments to assess how well an SDN-based mircoservice orchestrator performs in terms of latency, which will be affected in two ways. First, there is a certain increased latency for every request that goes through the switch due to time taken by the switching logic. Second, if the switch needs to ask the controller for information on how to route the requests then the latency will increase by several orders of magnitude. Such controller intervention is relatively rare, meaning that the average latency will not be so affected, but has a significant impact on the worst-case or 99-percentile latencies.

In addition to assessing the latency of the SDN-based microservice orchestration, we have performed a set of experiments to compare two different SDN controllers and three load-balancing algorithms when applied in this context. The task of the controller is to alter the flow table in the switch and the task of the load-balancing algorithm is to determine the node that will serve each request. These actions are necessary for the microservice orchestration to work, and since there are different designs and implementations for these components, we evaluate how different choices of controllers and load-balancers affect the overall system latency.

Our contributions are threefold, we

- summarise existing SDN controller performance studies,
- propose a novel SDN-based microservice architecture, and
- implement a test environment to evaluate the performance of the proposed architecture.

The remainder of the paper is organised as follows: Section II presents the background and related work, Section III describes our implementation of an SDN-based microservice orchestrator, Section IV provides the methodology used in the experiments, Section V presents the results from the experiments, and finally, Section VI concludes the paper.

II. Related Work

We organise this section in two subsections, (i) comparisons on OpenFlow controllers and (ii) latency measurements of microservice architectures.

A. OpenFlow Controller Comparison

The controller is an external process connected to the OpenFlow switch and is responsible for altering the switches' flow table and may add/remove flow entries in the switch at any time to adapt to the ever-changing network environment. When the switch receives a packet unmatched by the current flow table, the packet is automatically sent to the controller, which then deals with the packet instead. Thus, the controller gets notified of any flow table misses, and may alter the

flow table accordingly. We briefly summarise the most widely known SDN controllers below.

The **NOX** controller was the first publicly available open source controller software. Tootoonchian et al. [13] presented their slightly modified version of NOX, called NOX-MT, to show that some small alterations to the NOX controller could improve its performance significantly.

POX is a Python implementation of NOX with some design alterations to improve performance [6]. However, the POX controller is outperformed by most other available SDN controllers in terms of latency and throughput [9].

Ryu is an SDN controller implemented in Python focusing on simplicity and agile development. Ryu is publicly available and actively developed by NTT and aims to be a framework for building SDN applications rather than a complete controller with all possible features built into the system.

Beacon is a Java-based controller developed by Erickson [4]. Beacon has shown to be one of the top performing controllers with respect to network throughput [4], [9], [11]. It also performs well in terms of latency [4], [11].

Floodlight is an extension of the Beacon controller, and has an active community and commercial backing[1]. Floodlight is widely used and has been used by companies such as Canonical, CERN, SRI International, and others.

OpenDaylight is a decentralised controller, meaning that more than one controller node may be utilised to deal with unresolved packets and alter the flow tables of the switches in the system [7]. The OpenDaylight controller is widely used by many large corporations[2].

ONOS [1] is a decentralised controller implemented in Java. It was founded by the Open Networking Foundation and serves as a fault-tolerant platform capable of automatic global network discovery. ONOS supports SDN-Application hot-plugging and automatically adapts to changes in the network environment. It is backed by companies such as Google, Intel, AT&T and Samsung.

Multiple studies have been performed in order to compare the different controllers and convey the most fitting controller in various scenarios. Most studies involve measuring the throughput and latency of packets in complex/large network structures while using different controller implementations with varying number of threads. Table I shows how the different controllers performed in each study. Note that the table only considers the listed controllers.

As shown in the Table I, Beacon and ONOS each have the highest throughput in three of the studies and NOX in one. Ryu has the lowest latency in three studies, Beacon in two, and NOX and OpenDaylight in one each. In the evaluation by Tootoonchian et al. [13], they used an older version of Beacon compared to the other studies evaluating Beacon.

TABLE I
SDN-CONTROLLER PERFORMANCE EVALUATIONS

Controller	Included in Study	Lowest Latency	Highest Throughput
NOX	[4], [9], [11], [13]	[13]	[13]
POX	[4], [9], [11], [12], [15]		
Ryu	[4], [8], [9], [11], [12], [15]	[8], [12], [15]	
Beacon	[4], [9], [11], [13], [15]	[4], [11]	[4], [9], [11]
Floodlight	[4], [8], [9], [11], [15]		
OpenDaylight	[8], [9], [12], [15]	[9]	
ONOS	[8], [9], [12], [15]		[8], [12], [15]

B. Latency of Microservice Architectures

Shadija et al. [10] study how the chosen granularity of a microservice affects its total latency. While they note only a minor increase of the total latency, they argue that an increase in the microservice granularity might still lead to a significant increase in the response time of a system.

Ueda et al. [14] examine the latency effects of utilising a microservice architecture based on how the implementation differs from a monolithic server system. They find that the performance of the microservice approach is up to 79% worse than its monolith counterpart, potentitially caused by spending considerable amount of time processing requests, instead of executing the business logic and algorithms of the application.

Gan and Delimitrou [5] implemented two separate movie review and streaming services, one using a monolithic approach and the other using a microservice architecture. Comparing the performance of both systems, the microservice system outperformed the monolithic system at high server loads.

Based on the available evidence there is no obvious conclusion to be drawn on the relative performance of microservice architectures compared to their monolithic counterparts. It is reasonable to assume that this will be very system-dependent and that the level of inter-dependence between services is an important factor. Our analysis focuses on the orchestration of services through an SDN controller and is not intended to answer whether the microservice architecture approach is better or worse than monolithic systems.

III. SDN-BASED MICROSERVICE ORCHESTRATOR

We propose to use an SDN controller to perform the microservice orchestration. To this end we have designed a prototype implementation to showcase this approach and to use as a test object for performance measurements. When a client makes a request to the virtual IP address (VIP) of the microservice, the packet is received at the SDN switch. The switch communicates with the currently attached controller, which utilises its currently attached load balancing algorithm to decide which of the hosts that will receive the packet. Once a receiver is decided, the controller installs a flow entry into the switch and forwards the packet to the recipient.

The prototype implementation consists of a microservice implementation, a load listener and the controller software. The load listener serves as a communication bridge between the microservice implementation and the controller in the cases where a server-aware load balancer is used.

Implementing the microservice requires two separate components. The *request handler* responsible for handling incoming requests and the *load sender* responsible for sending

[1]Big Switch Networks: https://www.bigswitch.com/

[2]https://www.opendaylight.org/use-cases-and-users

the current server load to the *load listener*. We implemented the microservice using C++ and the communication was implemented using the Linux socket interface.

The Request Handler. In order to keep the study focused on the performance of the controllers and load balancers, as opposed to the performance of the servers, the actual workload in the experiments is synthethically generated. Each server node listens to a UDP socket. All packets received consist of a single positive integer denoting the amount of simulated load this packet would require, which is a constant in the static request experiment and chosen randomly in the random request experiment. The load is parsed by the server which increases its simulated load accordingly.

The Load Sender. The *load sender* is a component responsible for continuously updating the *load listener* with the load of the server node. In a set time interval, the *load sender* sends the current load of the system along with a server node identifier to a specific UDP-port of the *load listener*.

The Load Listener. The *load listener* is responsible for listening for the current load of each server-node and forwarding it to the SDN controller. It forwarded the load to the controller using POSIX shared memory.

IV. Experiment Method

A. Experiment Design

Since the performance of using an SDN switch as an orchestrator might be tied to the performance of the controller itself, we performed a set of latency measurements using different SDN controllers.

Performance Experiments. Both the *Switch Performance Experiment (SPE)* and the *Controller Performance Experiment (CPE)* consist of an emulated network containing two clients and an SDN switch connected to an SDN controller. Once the environment was set up and the controller initialised, one of the clients issued 100 000 ping requests to the other client, measuring the RTT of each of those requests.

In the *SPE* the flow entries for the request is pre-installed in the switch, resulting in no controller interaction. In the *CPE*, flow entry installation is disabled, resulting in all requests having to travel through the controller. The purpose of these experiments is to gauge the latency impact of the controller communication overhead.

Microservice Architecture Experiments. In the *Microservice Architecture Experiments (MAE)*, several hosts resides in an emulated network. Some hosts in the network, referred to as microservice-hosts, start hosting the server program. Once the microservice-hosts are configured, a *Network Flooding Swarm (NFS)* is started, consisting of a multitude of clients that all start to flood the network with requests. Each request sent towards the microservice IP is caught by the SDN switch and forwarded to one of the microservice-hosts. The SDN switch chooses the microservice-host using its flow-table or the connected SDN controller. The controller decides the recipient using its associated load balancer. Each of these requests increases the total workload of the recipient microservice-host, and the workload of each microservice-host is observed to evaluate the effectiveness of the utilised load balancer. Once the entire NFS is started, another client, referred to as the *Main Client*, begins issuing ping-requests to the microservice IP and record their round-trip time (RTT). An overview of the experiment is depicted in Figure 1.

For each controller and load balancer combination, the experiment was performed ten times. Five times where each NFS client made a request with a static workload, referred to as the *Static Microservice Architecture Experiment (MAE-Static)*. The other five times, each NFS client got assigned a random request workload, and each request made by that client increased the server workload by the assigned amount. This experiment is referred to as the *Random Microservice Architecture Experiment (MAE-Random)*.

Metrics. We perform measurements for five different latency percentiles, 50% (median), 90%, 99%, 99.99%, 99.999%, as well as min and max values.

The other metric that we focus on in this work is the *server load imbalance* which can be seen as an inefficiency in the orchestration. DeRose et al. [2] designed the *Load Imbalance Percentage* metric that quantifies how unevenly the workload was between several processes. The metric is defined as follows, where I is the resulting load imbalance percentage, m is the maximum workload, a is the average workload and n is the number of processes in the system:

$$I = \frac{m-a}{m} \times \frac{n}{n-1} \times 100$$

In the cases where $m = 0$ or $n = 1$ we assign $I = 0$, since either there is no workload or there is only one process in the system. Both cases imply that there is no server imbalance.

Latency Comparison. To assess the latency effects of SDN-based microservice orchestration it is important to find an approach of fairly comparing the different systems. Therefore, we assume that the different systems could be implemented with similar latency w.r.t. code performance.

We measure latency by issuing ping requests to the microservice host for all experiments. In the analysis of the *SPE*, we establish that the controller had no impact on the latency. Additionally, the network layout of the *SPE* and would be identical to a monolithic architecture if the monolith replaced the switch, and the two clients in the experiments instead was one single client sending requests to itself through the switch. However, one of the ping requests in the *SPE* experiment would then represent two ping requests to the monolith. Thus, we can estimate the ping percentiles of the monolith by simply halving the response times from the *SPE* results. We conclude in the analyis of the *SPE* that the controller doesn't matter in that experiment. Therefore, we can use the mean of both those experiments to get a more reliable result.

In order to quantify the *MAE* latencies in relation to the monolith latencies we can calculate the ratio using the the following formula, where R is the relative latency, E is the estimated latency of the monolith and M is the latency measured in the microservice experiment: $R = \frac{M}{E}$

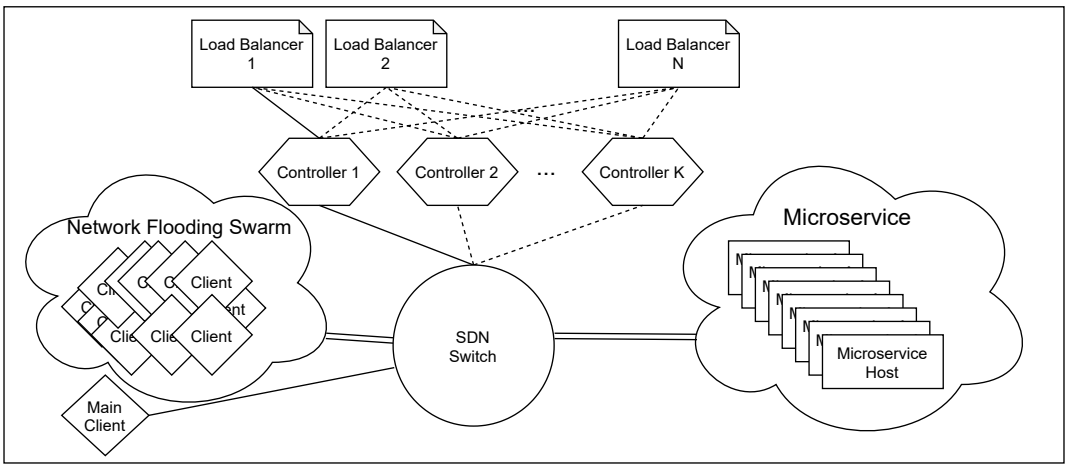

Fig. 1. Experiment Network Layout - The SDN switch forwards all packets from the client to the microservice, consulting the attached controller which uses the current load balancer to decide recipients for requests with no flow entries yet installed.

B. Experimental Setup

Simulation Environment. We use Mininet to emulate a network environment. In the setup, the Mininet network contains an emulated Open vSwitch[3] which is connected to an OpenFlow controller running outside of the Mininet network.

Choice of Controllers. As shown in Table I, OpenDaylight, Ryu, Beacon and NOX have been regarded as the controllers with lowest latency in at least one of the surveyed controller evaluation studies. Of these NOX and OpenDaylight were omitted due to configuration issues.

Load Balancers. Due to their simplicity and wide-spread use, we include the Round Robin and Random Assign algorithms in our comparison. In order to examine the effects of a server-aware load balancing algorithm, the Least Loaded algorithm has also been implemented.

Gathering Data. The current load and timestamp of each microservice-host is captured once every 100 ms. After the experiment has finished, these loads and timestamps are written to a file and analyzed. Similarly, the latency of each ping-request made by the main client is recorded to a file.

V. Results

This section presents the results of our experiments. First, the results of *SPE* and *CPE* are presented and explained, followed by the *MAE-Static* and *MAE-Random*, respectively.

A. Latency Experiments

The results for the *SPE* and the *CPE*, shown in Table II, consist of the latency of the RTT between the two hosts in the system. A higher value in the table indicates a higher latency for that controller.

The table shows that when installing flow entries, the latency of a request is constantly below a millisecond. Regardless of controller, the lowest latency is 3 microseconds and for 90% of all requests, the latency is below 5 microseconds. For the remaining requests, the latency increases to around 8 microseconds at the 99th percentile and around 20 microseconds for the 99.9th percentile. For the slowest requests, the latency increases to 422 microseconds when the Beacon controller was measured and 161 microseconds when measuring the Ruy controller. When not installing flow entries, the latency increases significantly, reaching as high as 20 milliseconds for the slowest requests. Comparing the controllers, for the fastest 90% of all requests with each controller, Beacon yields at least 5 times faster responses. For the remaining 10%, the latency of the requests measured when using Beacon approaches the latency measured when using Ruy with both measurements being approximately 20 milliseconds for their slowest request.

TABLE II
Latency percentiles in *SPE* and *CPE* (milliseconds).

Controller	Experiment	Min	$P_{50.0}$	$P_{90.0}$	$P_{99.0}$	$P_{99.9}$	$P_{99.99}$	Max
Beacon	SPE	0.003	0.004	0.004	0.008	0.026	0.078	0.422
Ryu	SPE	0.003	0.004	0.004	0.007	0.021	0.058	0.161
Beacon	CPE	0.162	0.268	0.381	0.957	3.870	9.620	20.000
Ryu	CPE	1.010	1.430	1.650	2.070	5.030	7.850	18.700

When comparing the controllers in the *SPE*, the latency is almost identical. This was expected, since when the flow entries are pre-installed, the controller will never be involved in the experiment, which effectively turns the tests identical to each other.

Comparing the latencies from *CPE* shows that Ryu is around five times slower for the 50th percentile, with Ruy never resulting in latencies lower than a millisecond and Beacon being able to produce latencies less than 200 microseconds. For the slowest requests, however, the results show that the controller has little effect on the resulting latency. This is likely due to the communication between the switch and controller being the slower factor rather than the performance of the controller itself in some cases.

Based on these results we can conclude that the latency caused by the controller communication plays a major part in the total latency of the slowest requests. When striving for

[3]Open vSwitch https://www.openvswitch.org/

minimal latency, it would be beneficial to pre-install as many flow entries as possible into the switch in order to minimise the number of requests communicated to the controller.

B. Microservice Architecture Experiment

For the *MAE-Static* and *MAE-Random*, the results consist of the the ping requests latencies in the Main Client as well as the load imbalance chart of the system. The balance chart depicts, for each controller/load balancer combination, how the load imbalance percentage of the system changes throughout the session. A higher value in the y-axis indicates a more unfair balance of the workload between the server-nodes.

The ping table consists of the percentiles in the RTT achieved for each controller and load balancer. Thus, a higher value in the table indicates a higher latency for that controller/load balancer combination.

Static Workload. The upper chart in Figure 2 shows that the load imbalance percentage is almost identical when comparing the controllers for each load balancing algorithm. The chart shows that the imbalance is rather high and volatile in the beginning, especially for the random load balancers, then turning stable around halfway through the experiment. The chart shows a rapid decline in load imbalance for both the Server- and Round Robin load balancers, settling at less than 10%. For the Random load balancer, the imbalance declines to about 70% shortly after its volatile start, only to increase to 80% and then starting its slow decline towards 50% after 20 seconds.

Observing Table III, the latency increases in an almost equal pace for all load balancer and controller combinations, starting at 3 microseconds and slowly increasing to about 50 microseconds at 99.9th percentile. In the remaining 0.1 percent of the requests, the latency of all requests increases drastically, having around 1 millisecond of latency for the 99.99th percentile almost 10 milliseconds for the 99.999th percentile and at least 25 milliseconds for the slowest request. For the Beacon controller the slowest requests took around 30 milliseconds, whereas for Ruy the slowest requests averaged around 40 milliseconds.

Random Workload. Similarly to the static workload experiment, Figure 2 also shows that the controller have no noticeable effect on the load imbalance. In this experiment, all three load balancers yield different results. Random load balancing results in an initial decline to 70%, followed by an incline to 80% and then a slow decline towards 50% load imbalance. Using the Round Robin approach resulted in 60-70% load imbalance initially, and then slowly declining to 30% after a slight increase to 70-80%. For the server aware load balancer, the first 25 seconds show a steady decrease to 70% in load balance, followed by a steep decrease from 70% to 20% in around 5 seconds. Thereafter, the load imbalance stabilises at around 10%.

Table III shows that the latencies are quite similar for the lowest 99.9 percent of the requests. The last 0.1 percent greatly varies as well showing a major increase in latency.

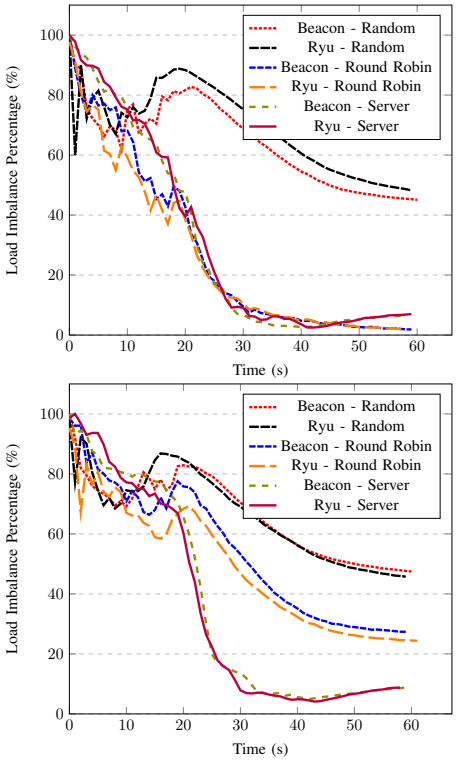

Fig. 2. Server Load Imbalance in the Microservice Archtecture Experiments - These charts show how the load imbalance of the server-node workloads changes throughout the experiments. The upper chart depicts the results from MAE-Static, while the bottom chart shows the results of the MAE-Random.

Analysing the Microservice Architecture Experiments. The results of both MAE experiments show that, in terms of load balancing, the chosen controller had no significant impact. This is expected, since the controller does not alter the logic of the load balancing implementation. The charts also show that the the Random load balancer results in a 50% load imbalance, regardless if the requests have a randomised imposed load or all requests have the same load. This high imbalance is probably due to the balancer being client-aware, due to installing flow entries for each host, which, in turn, causes the amount of requests being randomly chosen to drastically decrease. This results in the randomness of the system being too low for the random load balancer to distribute requests evenly enough.

The results also show that the Round Robin is really well suited for servers where each request results in similar workload for the server, but is somewhat lacking when the request workload is randomised. The adaptiveness of the server-aware load balancer makes it able to handle both request types without an issue, and for the experiments in this study, the extra network load imposed by constantly communicating the current workload to the switch, did not seem to affect the network to any noticeable effect.

The latency measurements made in these experiments show that for the vast majority of all requests, the response time

TABLE III
LATENCY PERCENTILES IN ALL THE REQUEST EXPERIMENTS. THE EXPERIMENT COLUMN INDICATES THE TYPE OF MAE BEING MEASURED.

Experiment	Controller	Balancer	min	$P_{50.0}$	$P_{90.0}$	$P_{99.0}$	$P_{99.9}$	$P_{99.99}$	$P_{99.999}$	max
Static	Beacon	Random	0.003	0.004	0.006	0.016	0.045	0.940	9.504	39.420
Static	Beacon	Round Robin	0.003	0.004	0.006	0.020	0.089	1.798	8.920	27.360
Static	Beacon	Server	0.003	0.004	0.006	0.018	0.054	1.304	7.906	25.940
Static	Ryu	Random	0.003	0.004	0.006	0.016	0.045	1.039	9.344	53.860
Static	Ryu	Round Robin	0.003	0.004	0.006	0.015	0.042	0.903	8.206	47.620
Static	Ryu	Server	0.003	0.004	0.006	0.016	0.044	0.944	8.408	27.740
Random	Beacon	Random	0.003	0.004	0.006	0.020	0.065	1.392	10.800	39.280
Random	Beacon	Round Robin	0.003	0.004	0.006	0.017	0.047	1.072	11.008	49.120
Random	Beacon	Server	0.003	0.004	0.006	0.017	0.054	1.442	9.956	28.020
Random	Ryu	Random	0.003	0.004	0.006	0.015	0.043	0.990	11.628	54.200
Random	Ryu	Round Robin	0.003	0.004	0.006	0.018	0.050	1.058	10.482	51.700
Random	Ryu	Server	0.003	0.004	0.006	0.016	0.045	0.985	9.098	44.300

will be less than a millisecond in this architecture, even when the network is flooded with other requests. However, looking at the requests past the 99.999th percentile, we see that the slowest requests measured are even slower than those made in the *CPE* experiment. This would indicate that when the network is flooding with requests, the communication with the switch is also affected to some extent, with the response time being more than doubled in some scenarios compared to the worst case of the *CPE* results.

When comparing the latency of using each controller in the *MAE* experiments, the difference is not nearly as significant as the differences shown in the *CPE* experiment. This could be explained by there being such few requests being measured where the request is handled by the controller, since the first request between a client and the microservice results in a flow entry installation.

VI. CONCLUSION

In this paper we have investigated the effects of transforming a monolithic server architecture to a microservice architecture orchestrated by an SDN switch. To conduct experiments a microservice architecture has been created using an SDN switch as the microservice orchestrator. The effects of employing said system have been evaluated both with regular and varying request workload using different load balancing algorithms. From this work, we draw three main conclusions.

First, the study shows that for the vast majority of requests, the latency for a SDN-supported microservice orchestrator is about three times slower than an optimistic estimate of a monolithic solution, which led to approximately 40 extra microseconds in response time. However, for a small portion of requests, the latency increases significantly, resulting in more than 20x slower requests in some cases and hundreds of times slower for the slowest requests.

Second, comparing Beacon and Ryu, the Beacon controller resulted in the lowest latency when considering the fastest 99.9% of all requests.

Third, comparing different load balancing algorithms, since the OpenFlow architecture enforces a client-aware load-balancing model, we conclude that for requests of varying imposed workload, a server-aware load-balancing algorithm is needed. However, when all requests result in similar workload, the round robin load balancer is encouraged, due to its simplicity and slightly better performance.

Our results show that using SDN to implement the microservice orchestration directly into the switch is feasible. It does increase the latency compared to a corresponding monolithic version, but if a microservice approach is desired, our solution is a good option to consider.

REFERENCES

[1] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, et al. ONOS: towards an open, distributed SDN OS. In *Proceedings of the third workshop on Hot topics in software defined networking*. ACM, 2014. doi: 10.1145/2620728.2620744.

[2] L. DeRose, B. Homer, and D. Johnson. Detecting application load imbalance on high end massively parallel systems. In *European Conference on Parallel Processing*. Springer, 2007. doi: 10.1007/978-3-540-74466-5_17.

[3] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. Microservices: yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*. Springer, 2017. doi: 10.1007/978-3-319-67425-4_12.

[4] D. Erickson. The beacon openflow controller. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 2013. doi: 10.1145/2491185.2491189.

[5] Y. Gan and C. Delimitrou. The architectural implications of cloud microservices. *IEEE Computer Architecture Letters*, 17(2):155–158, 2018.

[6] Y. Jarraya, T. Madi, and M. Debbabi. A survey and a layered taxonomy of software-defined networking. *IEEE communications surveys & tutorials*, 2014. doi: 10.1109/COMST.2014.2320094.

[7] D. Kreutz, F. M. Ramos, P. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 2015. doi: 10.1109/JPROC.2014.2371999.

[8] L. Mamushiane, A. Lysko, and S. Dlamini. A comparative evaluation of the performance of popular sdn controllers. In *2018 Wireless Days (WD)*. IEEE, 2018. doi: 10.1109/WD.2018.8361694.

[9] M. Paliwal, D. Shrimankar, and O. Tembhurne. Controllers in SDN: A review report. *IEEE Access*, 2018. doi: 10.1109/ACCESS.2018.2846236.

[10] D. Shadija, M. Rezai, and R. Hill. Microservices: granularity vs. performance. In *Companion Proceedings of the10th International Conference on Utility and Cloud Computing*, 2017. doi: 10.1145/3147234.3148093.

[11] A. Shalimov, D. Zuikov, D. Zimarina, V. Pashkov, and R. Smeliansky. Advanced study of SDN/OpenFlow controllers. In *9th Central & Eastern European Software Engineering Conference in Russia*. ACM, 2013. doi: 10.1145/2556610.2556621.

[12] A. L. Stancu, S. Halunga, A. Vulpe, G. Suciu, O. Fratu, and E. C. Popovici. A comparison between several software defined networking controllers. In *12th International Conference on Telecommunication in Modern Satellite, Cable and Broadcasting Services*. IEEE, 2015. doi: 10.1109/TELSKS.2015.7357774.

[13] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood. On controller performance in software-defined networks. In *2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE 12)*, 2012.

[14] T. Ueda, T. Nakaike, and M. Ohara. Workload characterization for microservices. In *2016 IEEE international symposium on workload characterization (IISWC)*. IEEE, 2016. doi: 10.1109/IISWC.2016.7581269.

[15] L. Zhu, M. M. Karim, K. Sharif, C. Xu, F. Li, X. Du, and M. Guizani. Sdn controllers: A comprehensive analysis and performance evaluation study. *ACM Comput. Surv.*, 53(6), 2020. doi: 10.1145/3421764.