uLoBal: Enabling In-Network Load Balancing for Arbitrary Internet Services on SDN

Alex F R Trajano*, Marcial P Fernandez†
Universidade Estadual do Ceará
Fortaleza, Ceará, Brazil
Email: *alex.ferreira@uece.br, †marcial.fernandez@uece.br

***Abstract*—Today's networks should support the increasing demand required by large workloads generated by users who consume several types of service over the Internet. However, the users demand increases at a much higher rate than the networks can evolve due to a number of constraints, including economic reasons. Thus, it is a common practice to adopt load balancing in order to use network resources more efficiently. Although load balancers are an effective way of improving current networks, most of the existing implementations are based on hardware products and dedicated to specific types of services, which is not a good alternative due to the highly dynamic nature of the Internet. In order to address the nature of today's networks, the Software-Defined Networking (SDN) provides an architecture that allows the network to be fully programmable, opening the possibility of implementing such load balancing mechanisms on top of a network controller that provides optimal management of resources. This paper presents uLoBal, an SDN-based load balancer that is able of performing flexible and generic load balancing of arbitrary services through the use of manageable forwarding algorithms that address most of the service types and natures. uLoBal is efficient to load balance services on unstructured networks by considering both network and servers' load metrics. The proposal was evaluated in the Mininet emulation environment and shows an improvement on load balancing, providing better use of network resources and user experience.**

Keywords–Load Balancing; Software-Defined Networking; Internet Services.

I. Introduction

The growing complexity and workload of current computer networks often require large infrastructure investments in order to support new demands. However, it is not feasible to increase the network capacity at the same rate as the demand grows, requiring a set of techniques that aim at a more efficient use of network resources. One of the most-used techniques is to perform load balancing, either by application layer algorithms or network orchestration, in order to optimize network traffic.

In fact, over the last years, it has been common to find specialized hardware appliances or applications capable of performing traffic load balancing of specific types of service. However, the load balancing task should not be coupled with specialized infrastructure items, since it should be an embedded feature of the network itself. This approach, called in-network load balancing, is a way of providing more flexibility and near optimal performance, since it would not be needed to communicate with external devices for choosing to which network path a given packet should be forwarded. Besides, there is a wide set of Internet services that can be boosted using an in-network load balancing technique, which opens the opportunity for developing generic solutions focused on the adherence to current and future services at no cost.

The programmable network concept is coming back thanks to Software-Defined Network (SDN) architecture. SDN is an emerging approach that aims to provide a dynamic, manageable and adaptable platform that is ideal for the nature of current networks and applications. To do so, it decouples the network control from the data plane, enabling the network control to be fully programmable while the underlying infrastructure becomes specialized on data forwarding. The SDN architecture provides a networking environment that is centrally managed by a SDN controller, which is directly programmable, bringing more agility to business process due to the existence of a single point of control. Furthermore, the SDN architecture aims to be vendor-neutral and based on open standards, like the OpenFlow protocol, which is the main SDN technology at the moment.

This paper presents uLoBal, a SDN-based load balancer that is capable of performing flexible and generic load balancing of arbitrary services through the use of different forwarding approaches that address most types of service. uLoBal allows network administrators to manage which services are going to be load balanced and which algorithms will define how such process should occur. uLoBal supports the use of static and dynamic load balancing schemes, working on top of the SDN controller, helping to reduce the load on both the network and the server.

In the last few years, there has been some interest in implementing load balancing functions over SDN, but there has not been much interest in addressing multiple arbitrary services at a single point of management. While some works have focused on specific applications [1], others focused on particular network topologies [2] and others have scalability problems that may lead production networks to collapse [3]. uLoBal tries to gather the best characteristics of each previously proposed solution while providing a novel and efficient load balancing method.

uLoBal has been implemented and tested on the Mininet virtualized environment that emulates a real-world network [4]. The testing scenario was based on a Content Delivery Network (CDN), which is one of the most common types of service that is present on the current Internet.

This work is organized as follows. In Section II, we present some related works, while the basis of SDN and OpenFlow are shown in Section III. In Section IV, we present the uLoBal, the load balancing solution proposal. Sections V and VI show the experimental evaluation and the results. Section VII concludes the paper and presents some intended future works.

II. Related Work

Li et al. [2] have proposed an OpenFlow-based load balancer for Fat-Tree networks that supports multipath forwarding. Their proposal aims to recursively find the current best path from a source to a destination, load balancing the network by enabling the use of alternate paths at runtime, minimizing network congestion. Their algorithm works only on networks that operate on the Fat-Tree topology and use network metrics for choosing the best path.

Wang et al. [5] have proposed an interesting load balancing approach that aims to proactively load balance traffic from clients to servers by slicing the IP address space into trees that isolate a set of clients to a set of servers. The work uses the concept of server weighting, which assigns a fixed number of clients to a server on the network. To do so, the extensive use of wildcards is proposed, which may reduce forwarding performance and create management issues, as can be seen in [6]. Furthermore, the proposed solution requires that, under certain conditions (network topology changes or server weight updates), a part of the network traffic passes through the controller, what may lead to the collapse of the network controller. Network metrics are not considered.

Koerner et al. [3] proposed an architecture that enables in-network load balancing of multiple services using OpenFlow. Their proposal relies on a set of SDN controllers on top of a FlowVisor instance [7], where each controller is responsible for load balancing the traffic of a specific service. The authors have focused on the architecture, so there is no information about particular service implementation, while the performed experiment does not fit real-world scenarios. The idea of using a set of controllers to handle exact services might be interesting in some specific cases, but has the drawback of not permitting multiple services to be handled by a single controller, which is the most common case of SDN deployment.

Handigol et al. [1] show Plug-n-Serve, a module that resides within an OpenFlow controller that is capable of performing load balancing over unstructured networks, aiming to minimize average response time of HTTP servers. Plug-n-Serve load balances HTTP requests by gathering metrics about CPU consumption and network congestion on the network links, which enables its load balancing algorithm to select the appropriate server to direct requests to, while controlling the path taken by packets on the network.

III. SDN and OpenFlow overview

Software-Defined Network (SDN) is an approach to network control and management that allows administrators to manage network services by an abstraction of lower network protocol functions. This is done by decoupling the control plane, that takes decisions about forwarding traffic through the network infrastructure; from the data plane, the network traffic itself. The objective is to simplify the network control in high speed traffic. SDN requires a protocol for the control plane that is able to communicate with devices. The most known protocol is the OpenFlow, often misunderstood to be equivalent to SDN, but other protocols could also be used, such as Forwarding and Control Element Separation (ForCES) [8] and Network Configuration Protocol (NETCONF) [9]. In our work, we will consider only the OpenFlow architecture.

OpenFlow has two main components: the controller, an unique programmable remote control, and the network devices. These two components work together through the OpenFlow Protocol. The main idea is to keep network devices as simple as possible in order to reach better forwarding performance, having no complex decision-making process within the devices, delegating such a task to the network controller.

The OpenFlow Controller is the centralized controller of an OpenFlow network. It sets up all OpenFlow devices, maintains topology information, and monitors the overall status of the entire network. The OpenFlow Device is any OpenFlow-enabled device in a network, such as a switch, router or access point. Each device maintains a Flow Table that indicates the processing applied to any packet of a certain flow. The OpenFlow Protocol works as an interface between the controller and the switches setting up the Flow Table. The controller updates the Flow Table by adding and removing Flow Entries using the OpenFlow Protocol. The Flow Table is a database that contains Flow Entries associated with actions to command the switch to apply some actions on a certain flow. Some possible actions are: forward, drop and encapsulate. Figure 1 shows the structure of a Flow Entry.

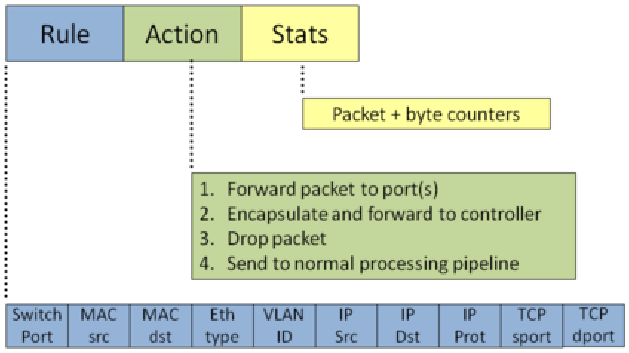

Figure 1. The OpenFlow Flow Entry [10].

Each OpenFlow device has a Flow Table with flow entries. A Flow Entry has three fields: Rule, Action and Stats. The Rule field is used to define the set of conditions to characterize the packets that will match that specific flow. The Action field defines the set of actions that must be applied to a packet if it matches the conditions defined in the Rule field. The Stats maintains a set of counters that are used to monitor flow's statistics, which can be used for management purposes. Each incoming packet is matched against the entries of the Flow Table. If a set of Flow Entries matches that packet, the device will select the entry with the highest priority, and the actions will be executed, having its statistics updated at the end of the process. If there is no matching entry for an incoming packet, the device sends a *PacketIn* message to the controller, wrapping the unmatched packet.

Once the controller receives a *PacketIn* message, it can take some actions on that packet, such as send *FlowMod* messages to the network devices in order to install new flow entries that match the incoming packet, or send a *PacketOut* message to "manually" forward the packet, or even ignore that packet. Besides, the controller can send messages to query statistics on every OpenFlow-enabled device within its network. An example is the *StatsRequest* message, which aims to query flow, port or queue statistics on the devices, which can be useful for determining the current state of the network. Each *StatsRequest* message is sent asynchronously, so the controller must listen for a *StatsResponse* message that will gather the requested statistics' data.

IV. uLoBal: Enabling Load Balancing on SDN

The uLoBal architecture was designed to provide high flexibility and to fit multiple and diverse scenarios and needs. As described previously, it is essential that an in-network load balancer can address different types of service, not being focused on specific scenarios. The uLoBal follows such design directive and aims to identify a set of servers of a service using a unique identifier that will set the load balancing mode for that service. Furthermore, the uLoBal needs to be aware of network statistics in order to enable a load balanced forwarding, allowing a more efficient use of network resources.

To this extent, uLoBal has three main modules: (1) the network monitoring module; (2) the load balancing module; (3) the management module, a Representational State Transfer (REST) Application Programming Interface (API) that allows management by the network administrator and integration with other monitoring tools. All these components are embedded on the SDN controller in order to avoid unnecessary communication with external services. Figure 2 shows how these components are connected.

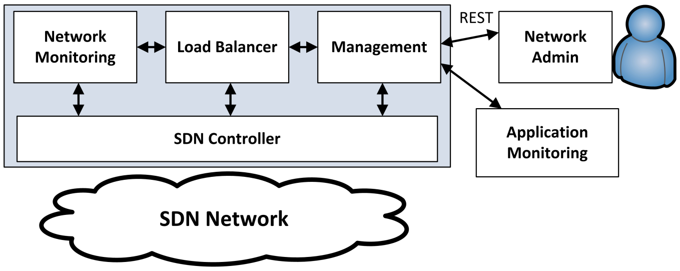

Figure 2. uLoBal architectural modules integration.

In an environment where there is a set of services that can be accessed through multiple endpoints, it is possible to perform in-network load balance in order to allow an even distribution of requests. The main objective behind uLoBal is to enable such service load balancing yet performing network load balancing. In networks where there are multiple paths between clients and endpoints, it is possible to use alternate paths as the network workload grows, mitigating problems related to networking congestion and reducing end-to-end latency. uLoBal can load balance the servers of the services by algorithms that use static and dynamic approaches that account for recent load information on both servers and network. Further subsections will give detailed information about each module.

A. Management Module

The uLoBal Management API can be accessed through a REST service, in order to allow administration and integration with external systems that can give updated load information of the services' servers. As the SDN controller cannot perform complex load monitoring at the servers, it is necessary to expose such service in order to allow an external monitoring tool to provide such information to the load balancer. Table I shows the uLoBal API methods.

The uLoBal uses a tuple of three values to identify an endpoint (or server) of a service: the *ServiceId*, *IP* address and transport *port* values. The *ServiceId* value is any string that uniquely identifies the provided service on a set of servers, being each server identified by the *IP* address and transport *port* values. Any kind of service that uses the TCP or UDP protocols can be addressed using these values if all of its endpoints are accessed on the same transport port, which is the most common case. Methods 1 and 2 of the API use exactly this tuple in order to add or remove endpoints to/from the load balancing. Method 3 is used by an external monitoring system that updates the load information of each server that provides access to the service, being the *Load* value the last load measure of a server normalized within the interval $[0, 10]$. The *Load* is a generic value that can be calculated using any application's specific metric. In order to allow the network administrator to change the load balancing mode that must be used for a given *ServiceId*, the *Mode* value must be provided to method 4 using one of three possible values: *ServerRoundRobin*, *ServerIpHash* or *NetServerLoad*, making the load balancer to change the balancing algorithm.

TABLE I. uLoBal API methods and parameters

#	method	parameters	used by
1	insertServiceEndpoint	(ServiceId, IP, Port)	Network Admin
2	deleteServiceEndpoint	(ServiceId, IP, Port)	Network Admin
3	updateServerLoad	(ServiceId, IP, Port, Load)	Monitoring System
4	changeLBMode	(ServiceId, Mode)	Network Admin

B. Network Monitoring Module

Since uLoBal uses network load information in order to load balance the service traffic on multiple forwarding paths, it is necessary to account for such load data to perform the load balancing. The network monitoring consists of two steps: (1) collect statistics on every port of every switch of the SDN at predefined time intervals; (2) calculate the minimum spanning tree of the network graph using the collected statistics as the cost metric.

The collecting process is made through the use of *StatsRequest* messages sent from the SDN controller to all switches on the network. Adrichem et al. describe a similar process in [11]. When the load statistics are collected, the Dijkstra algorithm is used to compute the minimum spanning tree that will be internally cached to be queried by the load balancing component. The cost metric used to compute the tree is given by $LinkCost = b + e$, where b is the percentage of the used link's bandwidth and e is the percentage of packets that have suffered of either drops or transmission errors. The $LinkCost$ must be normalized within the interval $[0, 10]$ before the spanning tree is calculated.

C. Load Balancer Module

The uLoBal load balancer module is responsible for load balancing requests based on three operational modes: Round-Robin (RR), IP Hashing (IPH), and Network and Server Load (NSL). Each one is identified in the REST API by *ServerRoundRobin*, *ServerIpHash* and *NetServerLoad*, respectively.

The load balancing mechanism is based on the principle of SDN, where the controller can push flows on the switches when there is no matching flows for an arriving packet. At this moment the controller receives a *PacketIn* message, that will be handled by the load balancer module if the destination IP and port match some previously inserted endpoint. The handling algorithm will depend on the configured load balancing mode for the matching ServiceId, with the NSL mode as the default mode. Algorithm 1 shows how the *PacketIn* message is handled by the controller.

Algorithm 1: PacketIn handling algorithm

```
Data: A PacketIn message
if Packet's destination IP and port belongs to any ServiceId then
    $sId$ := get the matching ServiceId; $lbMode$ := get the configured load balancing mode for $sId$;
    switch $lbMode$ do
        case ServerRoundRobin
            call RR(PacketIn, $sId$);
        end
        case ServerIpHash
            call IPH(PacketIn, $sId$);
        end
        case NetServerLoad
            call NSL(PacketIn, $sId$);
        end
    endsw
    Send the packet on a PacketOut message;
end
else
    Ignore the packet, not interfering the normal processing;
end
```

The uLoBal provides two approaches for load balancing service requests. The first is static, an approach that does not consider either network or server metrics in order to make decisions to where forward incoming traffic, while the second approach is dynamic and uses this metrics in order to make traffic orchestration. The static approach has the advantage of less overhead since there is no need to be aware of such metrics, which may be useful for networks where there is little congestion and to services that need some predictability about which server will handle a given request. On the other hand, the dynamic approach enables better network traffic orchestration, using resources according to their most up-to-date metric information, which can help to reduce network congestion and, consequently, improve the users Quality of Experience (QoE). Algorithms 2 and 3 use the static approach, while the Algorithm 4 uses the dynamic approach.

Algorithm 2: RR algorithm

```
Data: A PacketIn message and the $sId$
Increment the RR packet counter for the given $sId$;
$pktC$ := the RR packet counter for the given $sId$;
$sLen$ := get the amount of servers that belongs to $sId$;
$sIndex$ := $pktC \mod sLen$;
From the list of servers of $sId$, get the server $dstSrv$ stored at the $sIndex$ position;
Get the less costly network path from the packet's source to $dstSrv$ and send FlowMod messages to the switches on the path;
```

Algorithm 2 is a basic function that simply forwards requests by choosing the destination server through the Round-Robin algorithm. Once it chooses the server, it gets the cheapest current network path from the source to the destination in order to load balance the network. Its main characteristic is that the servers constantly receive a similar amount of requests, which can be useful for services that the costs of the requests are always the same.

Algorithm 3: IPH algorithm

```
Data: A PacketIn message and the $sId$
Create a circular list $srvCirLst$;
foreach $server\ srv\ that\ belongs\ to\ sId$ do
    Calculate the hash $h$ of the $srv$ IP;
    Insert $h$ into $srvCirLst$;
end
Get the packet's source IP and calculate its hash $srcH$;
Insert $srcH$ into $srvCirLst$ and get its index $srcIdx$;
Get the server $dstSrv$ whose hash is stored at the $srcIdx + 1$ position on $srvCirLst$;
Get the less costly network path from the packet's source to $dstSrv$ and send FlowMod messages to the switches on the path;
```

Algorithm 3 aims to map a set of clients to the same endpoint following a Consistent Hashing approach [12]. The main goal is to always forward requests made by a user to the same endpoint, which may be useful for services that need to fetch context information before serving the request, since this context information can be locally cached.

Algorithm 4: NSL algorithm

```
Data: A PacketIn message and the $sId$
Create a Hash Map $cstMap$ capable of storing multiple values mapped by a single key;
foreach $server\ srv\ that\ belongs\ to\ sId$ do
    Get the less costly network path $nPth$ from the packet's source to $srv$;
    $netCst$ := the cost of $nPth$;
    $srvCst$ := the current $srv$ cost value;
    $cost$ := $\sqrt[2]{netCst \times srvCst}$;
    Insert the $nPth$ on $cstMap$ mapped by $cost$;
end
Get the minimum key $k$ from $cstMap$;
Get a random entry $nPth$ mapped by $k$;
Send FlowMod messages to the switches on the path $nPth$;
```

Algorithm 4 was designed for considering the current network and server loads in order to choose the destination server. It works by selecting the endpoint that can be accessed with the minimum cost, being the cost calculated through the geometric mean of both server and network costs. As a dynamic approach, it is not easy to predict which requests are going to reach a determined server, since it will depend exclusively on the current load from both servers and network.

Note that Algorithms 2, 3 and 4 send *FlowMod* messages to the switches within the network path from the source to the selected endpoint. Each *FlowMod* message consists of a header that matches the packet and two actions, (1) rewrite the packet's destination/source address; and (2) send the packet to the next hop. Furthermore, note that even though Algorithms 2 and 3 perform a static server load balancing, the chosen network path remains dynamic, since the selection process is based on the network metrics collected by the monitoring module. The flows generated by uLoBal use a *soft timeout* approach in order to set the duration of flows on switches, configuring the inactivity timeout to 5 seconds.

V. Evaluation

In order to evaluate how the load balancer would behave in production networks, two scenarios were elaborated to evaluate the effectiveness of the proposal and give some insights about further deployment.

The first scenario aims to compare how the network load balancing approach affects the clients perceived delay when requesting some content on a CDN server that operates through HTTP. This scenario aims to evaluate only the network load balancing by avoiding the server load balancing, allowing better analysis of the proposal behavior. To this extent, the network topology has only a single server that is accessed by several clients spread over the network. Since there is a single server, all the requests are forwarded to a single point of the network, making the load balancer change the forwarding paths at runtime, balancing the traffic load. For comparability, the experiment has addressed the use of the proposed load balancer operating on the NSL mode and the use of a traditional Shortest Path First (SPF) approach.

The second scenario aims to compare how the different load balancing modes affect (1) the client perceived latency and (2) the server load. Again, the clients are going to request contents on CDN servers that operate through HTTP. In this scenario, we used three CDN servers, each one providing an endpoint for content delivery, making the load balancer forward requests to one of these servers, following the configured load balancing mode.

The evaluation environment was based on a virtualized network using Mininet [4]. The Mininet system permits the specification of a network interconnecting virtualized devices. Each network device, hosts, switch and controller are virtualized and communicate via Mininet. A Python script is used to create the topology in Mininet, and the traffic flow control is made by the OpenFlow controller. Therefore, the test environment implements and performs the actual protocol stacks that communicate with each other virtually. The Mininet environment allows the execution of real protocols in a virtual network. The possibility to set link bandwidth and delay in Mininet allowed us to perform an experiment similar to an actual real scenario. The chosen OpenFlow controller was the Floodlight [13], due to its simplicity and development flexibility.

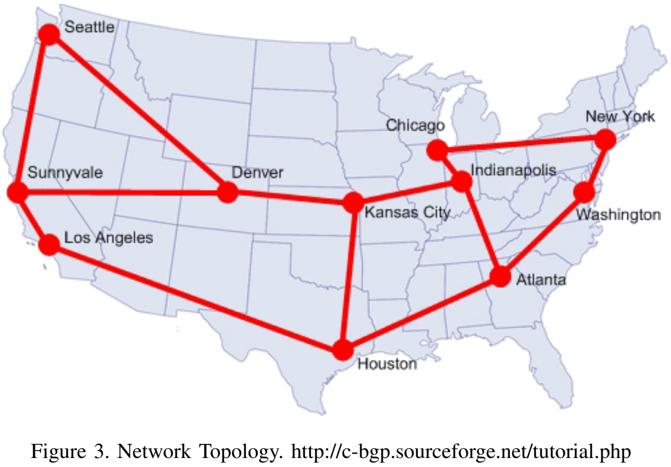

Figure 3. Network Topology. http://c-bgp.sourceforge.net/tutorial.php

Figure 3 shows the Abilene network topology, which has been used to perform the experiments on top of Mininet. The topology's sources were obtained from the TopologyZoo [14] and parsed according to the method described by [15]. In the first scenario, the server was positioned at the network node represented by the Indianapolis city. In the second scenario, the servers were positioned at the New York, Washington and Sunnyvale cities. Neither link latency nor bandwidth has been modified in the experiments.

In each city that did not contain any CDN endpoint, a set of 10 clients has been placed in order to make content requests. Each client was configured to perform sequential requests to a randomly chosen server from the available servers of the experiment. Of course, it is the job of the load balancer to redirect the request to the proper server, following the load balancing mode.

VI. Results

After each client finished 10^6 requests, the experiments results were collected and the graphs in Figures 4, 5 and 6 were built.

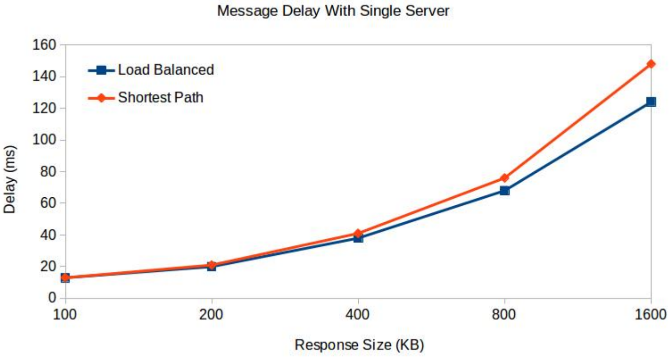

Figure 4. Results of the first experiment.

Figure 4 shows the results for the first experiment, where the network load balancing was compared to a SPF approach. It is possible that at low workloads, these approaches did not show significant differences, suggesting that the proposed network load balancing scheme is not relevant. However, when the workload starts to grow, there is an improvement in order of tens of milliseconds on the clients perceived delay, suggesting that this network load balancing approach can be useful as the workload grows. Besides, it is possible to conclude that at high workloads, the network would benefit form such load balancing mechanism, since congested paths would be avoided.

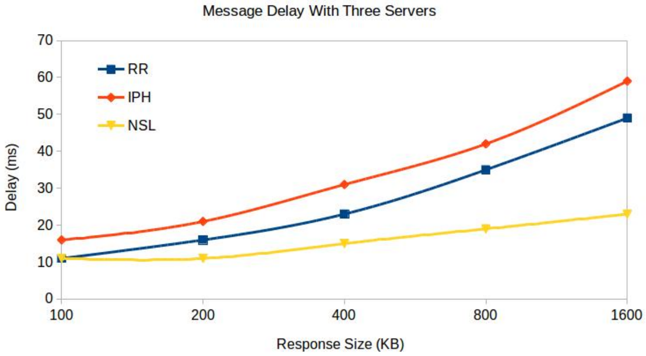

Figure 5. Results of the second experiment, from the point of view of the clients perceived delay.

Figure 5 shows the results for the second experiment, where the load balancing modes can be compared with each other. Both RR and IPH have similar behaviors at different workloads, although the RR shows better latency results. Such results can be explained by the fact that the network nodes are spread over an area of a whole country, with the servers being positioned at the edges of the network, which makes the use of these techniques a bad idea due to the high distance, increasing the end-to-end delay. It is possible to notice that the NSL mode outperforms both RR and IPH modes, which can be explained by the use of both server and network metrics when deciding to which server the requests will be forwarded, considering always the less costly network path at the moment. When the responses' sizes were 1600 KB, the improvement was about 53% and 62%, compared to RR and IPH modes, respectively.

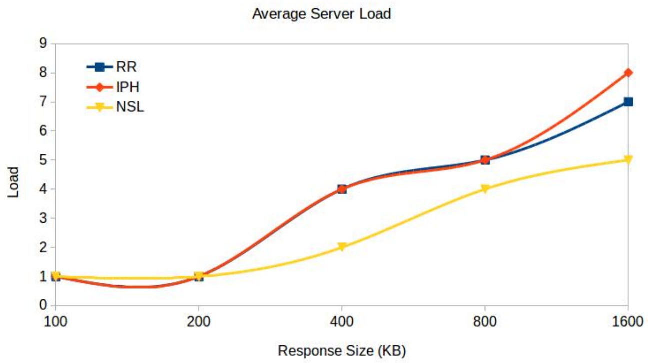

Figure 6. Results of the second experiment, from the point of view of the servers average load.

Figure 6 shows the results for the second experiment, where the average servers' load can be compared according to the load balancing mode. Again, both RR and IPH modes are similar to different workloads, unless the responses' sizes were 1600 KB. This result was expected, since these load balancing schemas aim to distribute the requests evenly across the servers, not looking for any external factor on the decision-making process. For this reason, it is also possible to notice that NSL can alleviate the average server load in most of the workloads, suggesting that the proposed load balancer can be useful in different production networks with distinct workloads when configured to operate in this mode. Furthermore, it was not observed any significant change in the consumption of resources in the network controller, even when using the NSL mode, suggesting that uLoBal follows the same scalability levels of the network controller.

VII. Conclusion and Future Work

This paper has presented uLoBal, a SDN-based load balancer that is capable to load balance arbitrary services through the use of different forwarding approaches that address services of several types and nature. The work showed that uLoBal works by aggregating a set of servers that defines the services' endpoints, allowing the network administrator to enable the load balancing of requests through the use of three different load balancing modes. Besides, uLoBal supports the use of static and dynamic load balancing, making it adaptable to different types of service. Experimental results showed that the network load balancing performs better when compared to classic network forwarding, while enabling load balancing at the servers of distributed services.

As a future work we intend to aggregate more load balancing modes on the current implementation of uLoBal, while optimizing the existing algorithms to provide even better performance. Since uLoBal can work over unstructured networks, it is needed to verify how the network topology can affect the performance of each load balancing mode, which can give important insights about the positioning of servers on the network according to the type of service.

References

[1] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari, "Plug-n-serve: Load-balancing web traffic using openflow," ACM SIGCOMM Demo, vol. 4, no. 5, 2009, p. 6.

[2] Y. Li and D. Pan, "Openflow based load balancing for fat-tree networks with multipath support," in Proc. 12th IEEE International Conference on Communications (ICC'13), Budapest, Hungary, 2013, pp. 1–5.

[3] M. Koerner and O. Kao, "Multiple service load-balancing with openflow," in IEEE 13th International Conference on High Performance Switching and Routing (HPSR2012). IEEE, 2012, pp. 210–214.

[4] B. Lantz and B. Heller, "Mininet: rapid prototyping for Software Defined Networks," Last accessed, Aug 2015. [Online]. Available: http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/Mininet

[5] R. Wang, D. Butnariu, J. Rexford et al., "Openflow-based server load balancing gone wild," in Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE 2011, 2011.

[6] B. Lopes Alcantara Batista, G. Lima de Campos, and M. Fernandez, "Flow-based conflict detection in openflow networks using first-order logic," in Computers and Communication (ISCC), 2014 IEEE Symposium on, June 2014, pp. 1–6.

[7] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Flowvisor: A network virtualization layer," OpenFlow Switch Consortium, Tech. Rep, 2009.

[8] A. Doria, J. H. Salim, R. Haas, H. Khosravi, W. Wang, L. Dong, R. Gopal, and J. Halpern, "Forwarding and Control Element Separation (ForCES) Protocol Specification," RFC 5810 (Proposed Standard), Internet Engineering Task Force, Mar. 2010. [Online]. Available: http://www.ietf.org/rfc/rfc5810.txt

[9] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman, "Network Configuration Protocol (NETCONF)," RFC 6241 (Proposed Standard), Internet Engineering Task Force, Jun. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6241.txt

[10] Open Networking Foundation, "Openflow switch specification, version 1.3.5," Last accessed, Sep 2015. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.3.5.pdf

[11] N. L. Van Adrichem, C. Doerr, F. Kuipers et al., "Opennetmon: Network monitoring in openflow software-defined networks," in Network Operations and Management Symposium (NOMS), 2014 IEEE. IEEE, 2014, pp. 1–8.

[12] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in Proceedings of the twenty-ninth annual ACM symposium on Theory of computing. ACM, 1997, pp. 654–663.

[13] D. Erickson, "Floodlight Java based OpenFlow Controller," Last accessed, Aug 2015. [Online]. Available: http://floodlight.openflowhub.org/

[14] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The internet topology zoo," Selected Areas in Communications, IEEE Journal on, vol. 29, no. 9, 2011, pp. 1765–1775.

[15] M. Großmann and S. J. Schuberth, "Auto-Mininet: Assessing the Internet Topology Zoo in a Software-Defined Network Emulator," Otto-Friedrich University, Tech. Rep., 2013.