Computer Science • 15 (3) 2014 http://dx.doi.org/10.7494/csci.2014.15.3.253

Paweł Wilk
Piotr Nawrocki

NETWORK MANAGEMENT SERVICES BASED ON THE OPENFLOW ENVIRONMENT

Abstract *The subject of this article is network management through web service calls, which allows software applications to exert an influence on network traffic. In this manner, software can make independent decisions concerning the direction of requests so that they can be served as soon as possible. This is important because only proper cooperation including all architecture layers can ensure the best performance, especially when software that largely depends on computer networks and utilizes them heavily is involved. To demonstrate that the approach described above is feasible and can be useful at the same time, this article presents a switch-level load balancer developed using OpenFlow. Client software communicates with the balancer through REST web service calls, which are used to provide information on current machine load and its ability to serve incoming requests. The result is a cheap, highly customizable and extremely fast load balancer with considerable potential for further development.*

Keywords OpenFlow, software-defined networking (SDN), management services, load balancing

1. Introduction

The aim of this paper is to demonstrate how a connection between software and network data flows can be created. The advantages of this approach will be shown on the basis of a switch-level load balancer service developed using the OpenFlow (OF) protocol. The main motivation to focus on this area have been the developments observed in the data processing model. Cloud computing[1] makes high-performance computers available to everyday users. Multiple applications are being migrated from traditional desktop solutions to web-based ones. Mobile devices that are always connected to the Internet are becoming increasingly popular, thereby generating an enormous amount of traffic. In March 2013, for instance, Facebook had more than a billion active users [6]. Overall, this forces software developers to find new, faster solutions. How can this be achieved? The answer is: by bringing software and hardware closer together. The area with the highest potential for this is the interface between software and network management services. A recently developed paradigm called *software-defined networking (SDN)* opens many new opportunities in this area. Since the day OpenFlow was released, everyone has been able to write their own network protocols and run them in real-life environments. This article demonstrates a practical use for such solutions by showcasing an implementation of a load balancer. As additional functionalities, it provides a health checking mechanism and the ability to securely shut down a server. The communication between the server and the balancer takes place via web service calls and the entire balancing concept is completely transparent from both the client and server sides.

2. Related work

This section is split into two main parts. The first one presents a comparison of available load balancers together with their advantages and disadvantages, while the second one describes the new approach to network management techniques, i.e. software-defined networking. Only a look from both perspectives makes it possible to fully evaluate the solution presented.

2.1. Load balancers

The concept of load balancing has been used for a long time to optimize traffic for web servers. One of the solutions frequently used is LARD (Locality-Aware Request Distribution) strategy [16, 11] which enables load balancing based on the content of client request. Lately, there has been an increase in the importance of optimizing traffic in computer networks thus new implementations of load balancers are being

[1] Cloud computing – a computation model based on making available services whose implementation is hidden from the user. It provides high performance in a virtualized and aggregated manner, where user is charged in accordance with utilization. As a result, there is no need to purchase either physical server hardware or software licenses [3, 13].

developed [5]. Load balancers can be implemented in various completely different ways; the most common are described below.

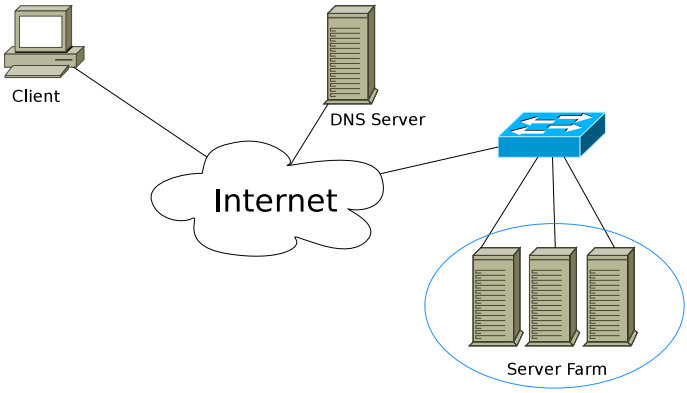


Figure 1. DNS load balancing.

Starting at the network level, there is the round-robin Domain Name System (DNS) [1] presented in Figure 1. It involves linking a single domain name with multiple IP addresses. The domain name server responds to requests with a shuffled IP list and then the client tries to establish a connection, starting with the first IP number. If an attempt fails, it proceeds to the next one. This solution is easy to implement but difficult to manage because DNS responses may be cached on many levels (on intermediate domain name servers and client machines). Furthermore, this solution does not scale well, since every server needs to have a public IP address.

Load balancing can also be implemented directly on routers [4] as a stateless/stateful round robin or according to the number of open sessions (Fig. 2). This is simple to configure but load may be unfairly distributed, in particular where the standard deviation of request processing time is high. The other disadvantage is the fact that such solutions are only available on high-performance routers, which can be expensive.

The most sophisticated and fastest solution is a hardware balancer called F5 Network Manager [2] (Fig. 3). The balancing process can be configured in several different ways: random distribution, round robin, weighted round robin (making it possible to spread the load unevenly), dynamic round robin (costs can be computed based on the number of active connections or average response time), least connection, fastest (according to response time). The only drawback of this device is its price, which is in the thousands of dollars. The final method described here is a purely software-based implementation (Fig. 4).

This would be the most flexible and cheapest approach, but it has two major drawbacks. Firstly, every request has to pass through the balancer and this means that

each packet needs to be processed and subsequently its headers must be completely rebuilt. This introduces unnecessary latency. Secondly, if there is a huge number of requests, the balancer may turn out to be a bottleneck.

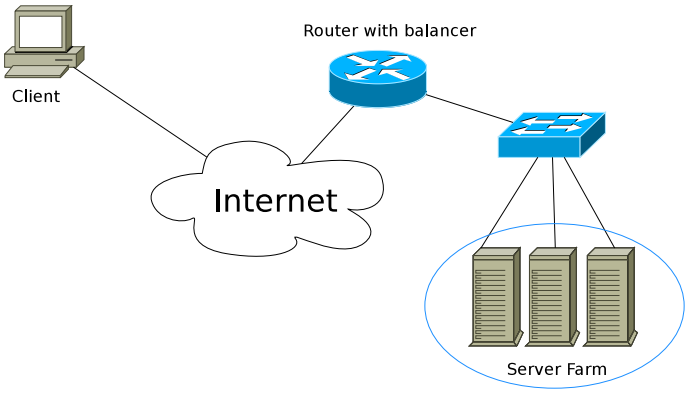


Figure 2. Router load balancing.

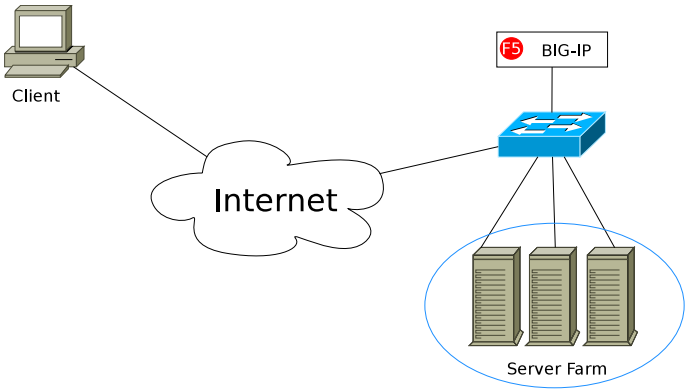


Figure 3. F5 load balancing.

To sum up, each of the above propositions has its advantages and disadvantages, which makes it difficult to choose the optimum one. However, software-programmable switches may introduce completely new solutions, and one of those will be presented in this article.

2.2. Software-Defined Networking

In solutions that are currently popular, the decision on where a packet is to be sent is made on network devices such as switches or routers.

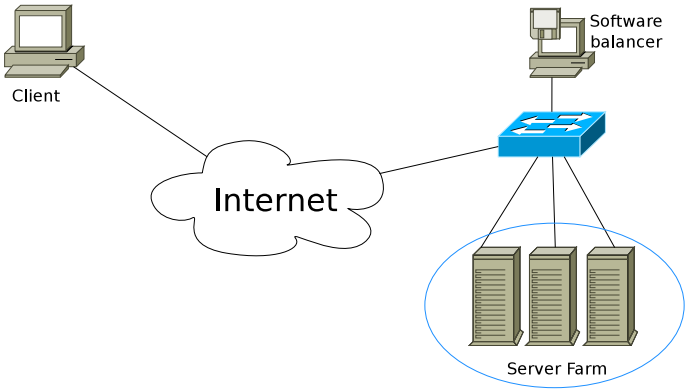


Figure 4. Software load balancing.

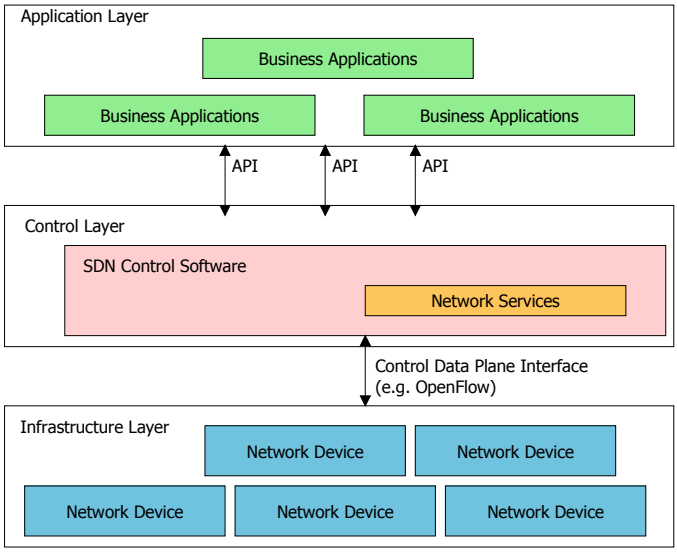


Figure 5. Software-defined networking – Architecture [14].

In software-defined networking (SDN) [15], this is left to software controllers, thus allowing network devices to focus exclusively on transmitting packets as shown in Figure 5.

This solution has many advantages such as:

- faster development of new features – users can run their own network solutions without waiting for manufacturers to implement them in their hardware;
- SDN's ability to coexist with traditional networks whereby only part of the traffic is processed by SDN;

- improved hardware utilization by introducing completely new configurations such as implementing network policing in a reactive manner using functional programming languages [9].

All these advantages have contributed to the widespread use of this paradigm in hardware devices and business applications.

SDN is just a general paradigm whose most popular implementation is OpenFlow [17] created by the Open Networking Foundation (ONF), which is backed by major network hardware manufacturers such as Cisco, IBM, HP and NEC.

The SDN architecture consists of three elements: a controller, an OpenFlow-compatible switch and a secure connection between them. On the switch, there is a flow table, which matches packets to its entries based on their source, destination Ethernet address, IP address or port, IP type of service, ICMP code/type, VLAN ID or incoming port. If there is no successful match, the packet is sent to the controller, otherwise the actions listed in the relevant flow table entry are executed. The actions available are forward or drop. A packet can be forwarded to all ports except the incoming port, to the controller, to the local switch stack or to a specific port. The most important optional action is the modification of packet headers. The controller is a software process that listens for OpenFlow messages. OpenFlow only specifies the message format so that the controller can be written in any language. The general use case for OpenFlow is as follows:

- a packet arrives at the switch;
- the flow table is empty so the packet is sent to the controller;
- the controller pushes appropriate flow table entries;
- the controller forwards the packet received;
- from this point forward, every similar packet goes directly to its destination, bypassing the controller.

Authors of many papers [7, 12, 18, 10] consider the problem of load balancing in the OpenFlow paradigm. Most solutions are attempts to find a static routing path during the initialization step without considering the dynamics of changes in network configuration [12]. However, some solutions take network changes into account. For instance, the LABERIO path-switching algorithm [12] makes it possible to balance the traffic dynamically during data transmission.

Compared to the works listed above, the proposed solution (described in Section 3) provides a load balancer service with additional functionality such as the health checking mechanism and secure server shutdown.

2.3. OpenFlow framework comparison

OpenFlow only defines the desired pattern of communication between the switch and the controller by stipulating the order of bits in OpenFlow messages. Writing a controller from scratch would be extremely time-consuming and error-prone but multiple frameworks have been developed in different languages such as Java, C, C++, Python, Ruby and others. This section includes a comparison of the most

popular frameworks on the market. The basis of comparison was the implementation in each framework of a standard switch, which learns and stores MAC addresses for the purpose of providing traditional switching capabilities.

2.3.1. NOX

When it was first created, this controller used two programming languages: C++ and Python. The assumption was that specific modules would be developed in C++ and connected using Python. Unfortunately, the users misunderstood this concept and felt compelled to develop entire controllers using one of these languages, which led to unnatural structures. Because of that, NOX creators decided to fork the project, creating two new ones. Since that time, NOX has been used to create controllers purely in C++ and POX has been developed for those who prefer Python. Our opinion on this framework is as follows:

- complicated deployment process;
- contains many external dependencies that need to be resolved manually;
- only supports Linux.

Its main advantage is performance – creators quote delays of around 0.01 milliseconds and a throughput of ca. 50 000 flows per second.

2.3.2. POX

This framework has been developed entirely using Python and thus it supports Linux, Mac OS and Windows. Its main purpose is to accelerate the development process without having to worry about performance issues. Compared to NOX, it offers delays of around 0.06 milliseconds and a throughput of around 31,000 flows per second. POX comes with a few ready-made modules, of which the most interesting are:

- messenger – allows communication via JSON messages;
- discovery – informs the controller of network connection updates (link establishment or link failure);
- proxy ARP mechanism.

Furthermore, a special external library has been created for POX called POXDesk. It is a web-based GUI that enables browsing of the flow table and its modifications as well as the presentation of network topology.

2.3.3. Trema

Trema has been created using C and Ruby, but users can write controller code using Ruby only. It only supports the Linux operating system with Ruby 1.8.7 installed. Compared to the other frameworks, Trema provides a complete development environment with a network emulator and debug tools. Its main features are:

- simple installation – users receive a complete solution for creating controllers;
- TremaShark – the environment is automatically delivered with an integrated WireShark;
- integration with RSpec – the ability to implement controller unit tests;

- modular design with predefined modules such as:
  - network topology recognition;
  - Routing Switch functionality;
  - slice-able routing switch – makes it possible to divide the network into separate segments and only service some of them.

2.3.4. Beacon

Beacon is written fully in Java, with an emphasis on modular nature (related to the use of the OSGi framework) and on performance gains from multi-threading (as at 17 May 2011, it was twice as fast as NOX). Producers claim that after some tweaking, it can be run on Android devices. Its main features are:

- multi-threading;
- JUnit integration;
- REST API for flow management;
- web-based GUI.

2.3.5. Floodlight

Floodlight was developed as a variant of Beacon – some programmers split from the group, claiming that OSGi was creating too much overhead, and developed their own solution, which has mostly been funded by the BigSwitch company. Beacon is written in Java but may also be extended using Jython. Instead of OSGi, a customized modularity concept is used. Its main features are:

- multi-threading;
- can be a back-end for OpenStack (the standard for private and public clouds);
- actively developed by BigSwitch;
- the only controller with declared intentions to support subsequent OpenFlow versions (above 1.0).

This framework has been selected for the implementation of the project described in this article because it is the most mature and well-thought controller of all those listed above. Furthermore, BigSwitch involvement in its development guarantees regular updates and high support quality.

2.3.6. Summary

A detailed comparison of frameworks is presented in Table 1. The broad choice available allows every user to pick one no matter what programming language they prefer. The only thing that has to be borne in mind is the fact that even after more than two years since the new version of OpenFlow was introduced together with OF-Config, there are only a few controllers available that support it.

Table 1
OpenFlow framework comparison.

	NOX	**POX**	**Trema**	**Beacon**	**Floodlight**
Programming language	C++	Python 2.7	Ruby $\geqslant$ 1.8.7	Java	Java
Operating system	Ubuntu 12.04	Linux, Mac OS, Windows	Ubuntu 10-13, Debian, Fedora 16–19	Any with JVM	Any with JVM
OpenFlow version	1.0	1.0	1.0	1.0	1.0 (1.2, 1.3 planned)
Multi-threading support	–	+	+	+	+

3. Case study

The main purpose of this article is to demonstrate that OpenFlow can be used to create a switch-level load balancer with an interface that enables communication between the balancer and the software that needs balancing. In the solution presented, a server farm is accessible through a single IP address completely transparent for both the end user and the server administrator. Detailed architecture is presented below.

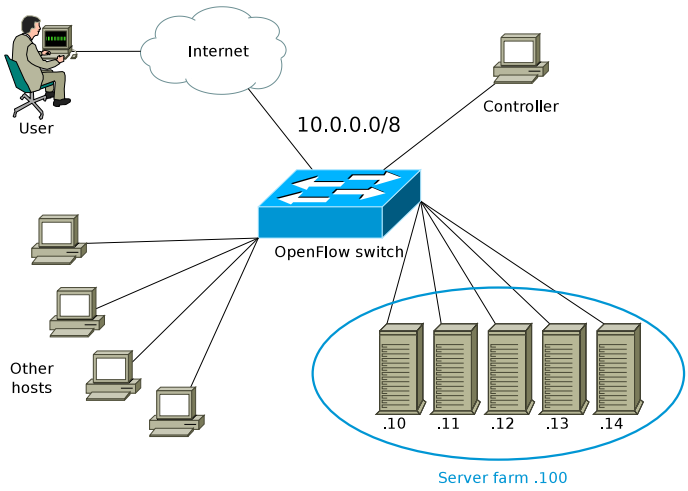


Figure 6. System architecture.

In Figure 6, a computer network is shown that features an OpenFlow-compatible switch. There are servers and normal hosts in this network; each machine has its own IP address and can ping other computers. Furthermore, servers are grouped into a single farm accessible via the 10.0.0.100 IP address. Each server periodically registers

its load in the controller. Based on that information, the controller directs new users to the server with the smallest load.

3.1. Implementation

In Figure 7, a general use case is presented that can help readers understand how the system works.

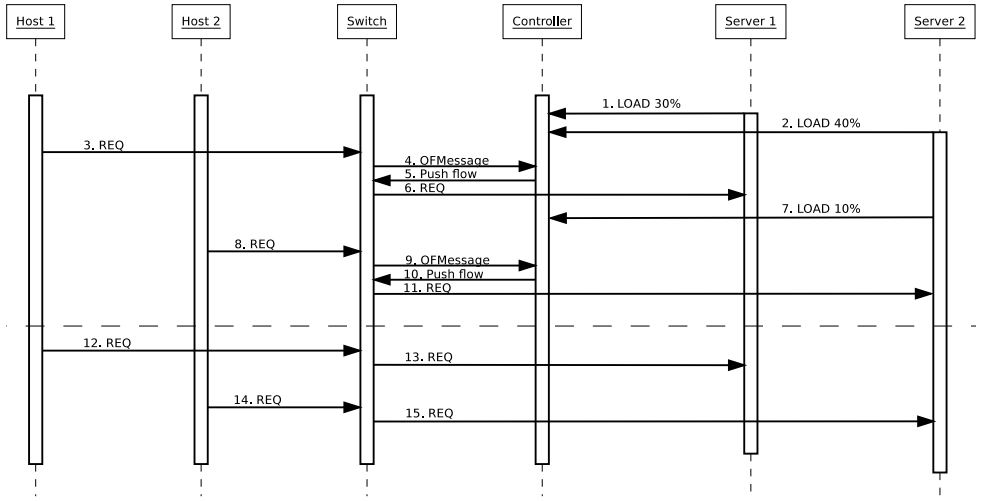


Figure 7. General communication in the system.

The steps are as follows: the first server registers with a load of 30% (1) and the second server registers with a load of 40% (2). The first host generates a request to the server farm (3). Since there are no flow entries on the switch, the request received is transferred to the controller, encapsulated in an OF message (4). The controller selects the less-loaded server to process the request. For this to happen, appropriate flows need to be pushed to the switch (5). Then the request finally arrives at the server (6). In the meantime, the second server updates its load to 10% (7). Subsequently, a new host generates a request to the server farm (8). Because this is the first request from this unique user, an OF message containing the request is sent to the controller (9). At this point, the second server is the least loaded, so a flow to it is pushed to the switch (10). The request is sent to the second server (11). All further requests from the same host will be automatically (without reaching the controller) transferred to the appropriate server in accordance with the sessions started (12–15).

Figure 8 presents communication in more detail, focusing on exact packet routes and details and leaving out the balancing aspect and the server registration process altogether.

The host wants to send a request to the server, so it (the host or else a router where there is traffic incoming from other networks) generates an ARP request to learn

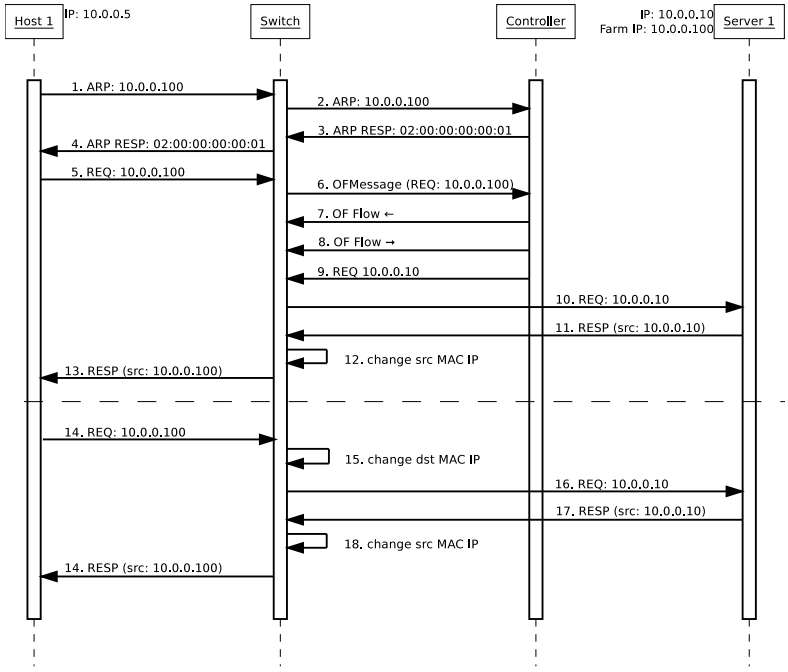


Figure 8. Detailed request path through the network.

the MAC address for the server farm IP (1). Because the flow table is empty, the ARP request encapsulated in the OF message is sent to the controller (2). The controller is programmed so that it acts as a Proxy ARP mechanism in the case of ARP requests for the server farm address. If the ARP request is for any other IP address, the switch will act in a traditional way. Proxy ARP responds with a configurable MAC address – 02:00:00:00:00:01 (3). This interaction is required where traffic is coming from multiple users in other networks. In this case, all packets pass through the router so there has to be a single MAC address representing the server farm instead of load-balancing ARP requests because the router could cache the ARP response and always send traffic to a single server. When the host/router gets the ARP response (4), it sends a request to the farm address (5). There are still no flows on the switch, so packets get transferred to the controller inside an OF message (6). The controller pushes flows from the server with the smallest load to the client (7). The next step is to create flows from the client to the server (8). Because everything has been set up, the controller can now send the packet to the server (9). At this stage, it is important to change the destination IP and MAC addresses for the specific server. Packet checksum has to be recalculated as well. Then the switch passes the request to the server (10), which sends a response (11). All flows have already been initialized, so the switch only changes packet source MAC and IP addresses to appropriate farm addresses (12). All checksums are recalculated automatically. As a final step, the response reaches the

client (13). Further communication goes directly to the server through the switch, bypassing the controller (14–19). All MAC and IP addresses are replaced directly on the switch.

The final aspect of the solution described is communication between the servers and the controller. This is based on REST web service calls. Two separate services have been developed – one to register a server and update its load value and the second to provide a way for excluding a server from the farm list. The project developed features two additional functionalities that have not been presented on the diagrams above.

The first feature is a health checking mechanism that operates cyclically at configurable intervals. Where the controller notices that there have not been any updates from a server for three update intervals, it assumes that the server in question has crashed. The controller then removes all flows leading to the server in question. In this case, user sessions are lost but users are distributed among the remaining servers.

The second feature is secure server shutdown. If a server needs to be turned off or removed from the server farm, for example to execute a software update, this may be done in a manner transparent to users. In order to do this, the server should call the appropriate REST service. After this service has been called for the first time, no new users are directed to the server in question. The response to that call is a JSON object indicating if all sessions have timed out; this means that the server no longer belongs to the server farm. This functionality has been developed based on the session tracking mechanism. Each flow has its own time-out set to the same value as the session time-out. If a user is inactive for a longer period and a flow disappears, the user's next request is checked against previously saved sessions. If a session exists for the user, the user is assigned to the same server again. The session tracking mechanism can remove session records when the time since the last update has exceeded two session time-outs.

3.2. Performance evaluation

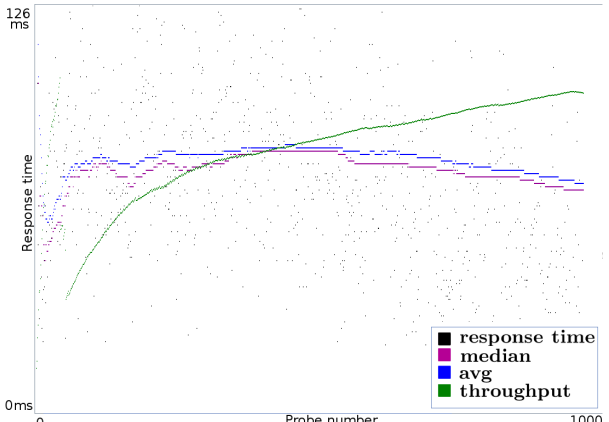
Performance evaluation has only been conducted for web services. This is because the rest of the system is fully virtualized, so running performance tests on a single machine that runs the controller and the OpenVSwitch while simulating hosts at the same time would be unreliable. Thus only functional tests were run for the entire system.

An additional argument for conducting tests in a virtualized environment is the state of implementation of OpenFlow specifications in real-life devices (switches). In most equipment, OpenFlow specification version 1.0 is implemented. By using a virtualized environment, one can take advantage of subsequent OpenFlow specification versions (up to version 1.3), which will significantly expand capabilities, including a multi-table processing pipeline, MPLS, IPv6 and QoS.

In the execution of the performance test, the tools listed below were used:

- Mininet – virtual network;
- Jmeter – monitoring parameters and service performance;

- iostat – monitoring current device load;
- VisualVM – monitoring the Java Virtual Machine.



median	avg	min	max	throughput
69ms	71ms	10ms	198ms	40.6/sec

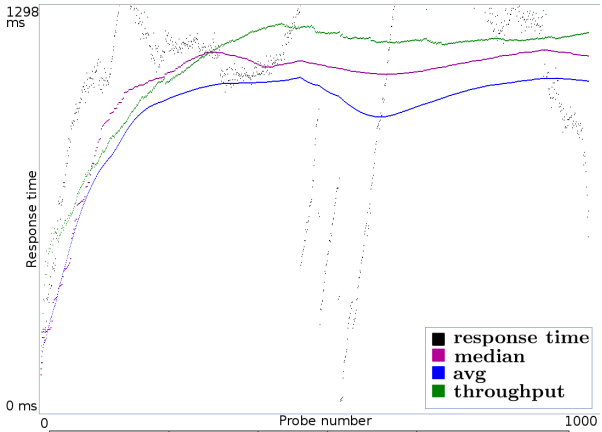
Figure 9. Response time for the service with actual server load (5 servers, 200 calls).

All tests were executed on a machine with the following specifications:

- processor: Intel SU2300, 2x1.2 GHz;
- memory: 4GB, Java heap memory size Xmx=512MB;
- Java 1.7_025;
- operating system: Ubuntu 12.04.

The results presented in Figure 9 illustrate a situation where there are few servers (five in that case) but they are updating their statuses heavily. This may be the case where the controller is configured so that server errors are detected as soon as possible. In this case, as shown, each server could update its status $40.6/5 = 8.12$ times per second. Thus a server failure would be discovered after 369 milliseconds. As the graph shows, throughput was constantly increasing and response times were stable. Based on those two parameters, a conclusion may be drawn that the solution presented is promising and may be deployed in environments that need to be highly available. Even distribution of the response times suggest there is no request queueing and they are being served on the fly. This may be also affirmed by the fact that controller performance limits have not been reached.

The results shown in Figure 10 illustrate the performance of the same service in different conditions. In this case, there are 100 servers with each potentially making 0.711 service requests per second, which means that a server failure would be diagnosed after 4.2 seconds. This scenario reflects usage in a farm developed for highly scalable solutions. This directly results in slightly lower availability, but the results are still acceptable. Furthermore 20 times increase of the server number caused only 14.88



median	avg	min	max	throughput
1139ms	1057ms	37ms	1508ms	71.1/sec

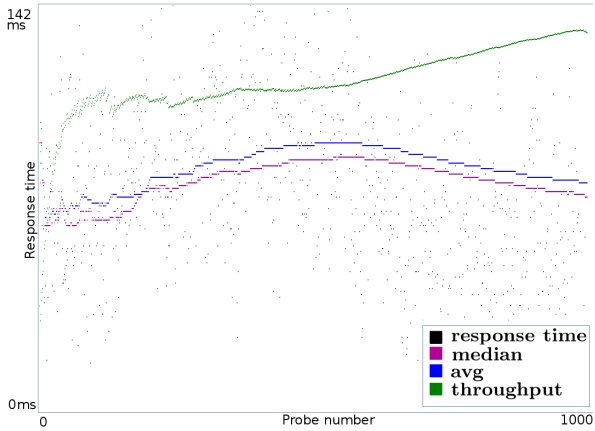
Figure 10. Response time for the service with actual server load (100 servers, 10 calls).

increase of the average response time, this may suggest linear dependency between those two values. Sudden decrease of the response time for the probes between 450 and 550 might have been caused by the garbage collector mechanism on the stress testing software side. Nevertheless constant throughput suggest that the controller side is functioning properly in a predictable way.

This part conclude stress tests for the service responsible for refreshing of the server load. For the both cases – the high availability and scalable environment – presented solution is suitable and may be successfully applied.

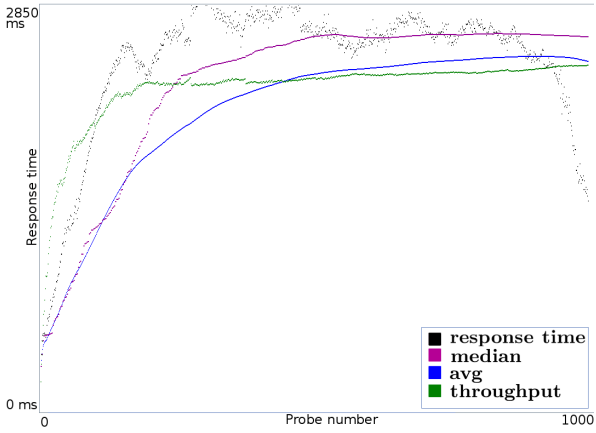
Two further figures present performance test results related to the service that provides the secure shutdown capability. Figure 11 presents results for five servers that are making requests continuously. The throughput achieved allows each server to make requests 7.66 times per second. As can be seen from the graph, results were constantly improving so the final call frequency could be even higher. Response times suggest that this solution is fully sufficient. As previously even distribution of the response times suggests processing them on the fly. As compared to the load update service both the average response times and the throughput are similar. This may suggest that when talking about the small server number most of the latency comes from the RESTful library and processing the requests with its parameters.

The final test, which is presented in Figure 12, stresses the service somewhat more. As can be seen, the secure server shutdown service does not scale as well as server load updating. Given a higher number of servers, overall throughput dropped compared to the case where there were only five. Moreover twenty times increase of the server number caused thirty time increase of the latency. This suggest the complexity is higher than linear. As a result, each server could check session states every 2.7 seconds. This value is acceptable but indicates that the solution may not be fully



median	avg	min	max	throughput
75ms	80ms	12ms	250ms	38.3/sec

Figure 11. Response time for the secure server shutdown service (5 servers, 200 calls).



median	avg	min	max	throughput
2633ms	2459ms	316ms	2936ms	36.7/sec

Figure 12. Response time for the secure server shutdown service (100 servers, 10 calls).

scalable. Therefore, for deployments with a server farm larger than 100 machines, administrators should consider using this service only to prevent further client assignments. The number of active sessions should be verified directly in server software. Also bearing in mind the application of the service, probability that 100 servers would needed to be shutdown at the same time is quite low. To prevent such unnecessary load on the controller and unpredictable behaviour of the rest of servers, it is advised to shutdown servers sequentially or in relatively small batches.

4. Summary

This article addresses the question whether it is possible and useful to create a connection between network management capabilities and the software that runs on network nodes. As the primary example, the functionality of a load balancer developed on the switch level is presented.

As demonstrated, the answer to the question above is affirmative. Closing the gap between software and network management may improve general performance and could also lead to completely new solutions that were previously unimaginable. This article describes the architecture and operating scheme of a load balancer that provides additional functionalities such as a health checking mechanism or secure server shutdown. It has been fully implemented using the OpenFlow standard and Floodlight framework. Both these tools leverage the software-defined networking paradigm and provide completely new ground for developing comprehensive solutions wherein software cooperates with hardware, especially with computer networks.

4.1. Further work

The system presented can provide an excellent starting point for further enhancements. To make the solution available in a production environment, providing security to the web services would have to be considered. Currently, any host within the network can call the web service, thereby causing the controller to enter an invalid state. To prevent this, a simple iptables (firewall) configuration might be sufficient.

Another thing worth analyzing would be the provision of backup servers to be used in contingencies, for example during primary server group updates. Furthermore, the entire system could be integrated with Continuous Delivery[2] software so that servers would be updated automatically without disrupting users.

The most attractive feature for the business environment would be enabling some servers to be turned off. For example at night, when traffic is lower, there is no need for all servers to operate. Turning them off could lead to considerable energy savings, thus reducing server maintenance costs.

Acknowledgements

The research presented in this paper was partially supported by the Polish Ministry of Science and Higher Education under AGH University of Science and Technology Grant 11.11.230.124 (statutory project).

References

[1] Aitchison R.: *Pro DNS and BIND 10.* Apress, 2011.

[2]Continuous Delivery – it is a process whose goal is the full automation of the software delivery process (from code submission through automatic testing to production deployment) [8].

[2] BIG-IP: *BIG-IP Local Traffic Manager: Concepts*, 2013.

[3] Buyya R., Broberg J., Gościński A.: *Cloud Computing - Principles and Paradigms*. WILEY, 2011.

[4] Cisco: *Cisco IOS IP Configuration Guide, Release 12.2*.

[5] Dabrowski J., Feduniak S., Balis B., Bartynski T., Funika W.: Automatic Proxy Generation and Load-Balancing-based Dynamic Choice of Services. In: *Computer Science*, vol. 13(3), 2012. ISSN 2300-7036.
URL `https://journals.agh.edu.pl/csci/article/view/13`.

[6] Facebook: Facebook Reports First Quarter 2013 Results. 2013.

[7] Handigol N., Seetharaman S., Flajslik M., McKeown N., Johari R.: Plug-n-Serve: Load-Balancing Web Traffic using OpenFlow. ACM SIGCOMM Demo, 2009.

[8] Humble J., Farley D.: *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010.

[9] Kim H., Feamster N.: Improving Network Management with Software Defined Networking. In: *IEEE Communications Magazine*, pp. 114–119, 2013.

[10] Koerner M., Kao O.: Multiple service load-balancing with OpenFlow. In: A. Smiljanic, M. Hamdi, H.J. Chao, E. Oki, C. Minkenberg, eds., *HPSR*, pp. 210–214. IEEE, 2012. ISBN 978-1-4577-0831-2.
URL `http://dblp.uni-trier.de/db/conf/hpsr/hpsr2012.html#KoernerK12`.

[11] Lei Y., Gong Y., Zhang S., Li G.: Research on Scheduling Algorithms in Web Cluster Servers. In: *J. Comput. Sci. Technol.*, vol. 18(6), pp. 703–716, 2003.
URL `http://dblp.uni-trier.de/db/journals/jcst/jcst18.html#LeiGZL03`.

[12] Long H., Shen Y., Guo M., Tang F.: LABERIO: Dynamic load-balanced Routing in OpenFlow-enabled Networks. In: *2013 IEEE 27th International Conference on Advanced Information Networking and Applications (AINA)*, vol. 0, pp. 290–297, 2013. ISSN 1550-445X.
URL `http://dx.doi.org/http://doi.ieeecomputersociety.org/10.1109/AINA.2013.7`.

[13] Nawrocki P., Soboń M.: Public cloud computing for Software as a Service platforms. In: *Computer Science*, vol. 15(1), 2014. ISSN 2300-7036.
URL `https://journals.agh.edu.pl/csci/article/view/519`.

[14] Open Networking Foundation: *Software-Defined Networking:The New Norm for Networks*, 2012.

[15] Open Networking Foundation: *Software-Defined Networking: The New Norm for Networks*, 2013.

[16] Pai V.S., Aron M., Banga G., Svendsen M., Druschel P., Zwaenepoel W., Nahum E.M.: Locality-Aware Request Distribution in Cluster-based Network Servers. In: D. Bhandarkar, A. Agarwal, eds., *ASPLOS*, pp. 205–216. ACM Press, 1998. ISBN 1-58113-107-0.
URL `http://dblp.uni-trier.de/db/conf/asplos/asplos98.html#PaiABSDZN98`.

[17] Pfaff B., Heller B., Talayco D., Erickson D., Gibb G., Appenzeller G., Tourrilhes J., Pettit J., Yap K., Casado M., Kobayashi M., McKeown N., Balland P.,

Price R., Sherwood R., Yiakoumis Y.: *OpenFlow Switch Specification*. Stanford University, 2009.

[18] Wang R., Butnariu D., Rexford J.: OpenFlow-based Server Load Balancing Gone Wild. In: *Proceedings of the 11th USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, Hot-ICE'11, pp. 12–12. USENIX Association, Berkeley, CA, USA, 2011.
URL `http://dl.acm.org/citation.cfm?id=1972422.1972438`.

Affiliations

Paweł Wilk
AGH University of Science and Technology, 30-059 Krakow, Poland,
`pawel.wilk.mail@gmail.com`

Piotr Nawrocki
AGH University of Science and Technology, 30-059 Krakow, Poland, `piter@agh.edu.pl`

Received: 26.11.2013
Revised: 10.01.2014
Accepted: 13.01.2014