# On Bringing Software Engineering to Computer Networks with Software Defined Networking

Alexander Shalimov
Applied Research Center for Computer Networks,
Moscow State University
Email: ashalimov@arccn.ru

Ruslan Smeliansky
Applied Research Center for Computer Networks,
Moscow State University
Email: smel@arccn.ru

***Abstract*—The software defined networking paradigm becomes more and more important and frequently used in area of computer networks. It allows to run software that manages the whole network. This software becomes more complicated in order to provide new functionality that was impossible to imagine before. It requires better performance, better reliability and security, better resource utilization that will be possible only by using advanced software engineering techniques (distributed and high availability systems, synchronization, optimized Linux kernel, validation techniques, and etc).**

## I. Introduction

Software Defined Networking (SDN) is the "hottest" networking technology of recent years [1]. It brings a lot of new capabilities and allows to solve many hard problems of legacy networks. The approach proposed by the SDN paradigm is to move network's intelligence out from the packet switching devices and to put it into the logically centralized controller. The forwarding decisions are done first in the controller, and then moves down to the overseen switches which simply execute these decisions. This gives us a lot of benefits like global controlling and viewing whole network at a time that helpful for automating network operations, better server/network utilization, and etc.

A controller (also known as network operating system) is a dedicated host which runs special control software, framework, which interacts with switching devices and provides an interface for the user-written management applications to observe and control the entire network. In other words, the controller is the heart of SDN networks, and its characteristics determine the performance of the network itself.

**We describe the basic architecture of contemporary controllers**. For each part of a controller we show software engineering techniques are already used and might be used in the future in order to improve the performance characteristics.

**We show the result of our latest experimental evaluation of SDN/Openflow controllers**. Based on this we explain that the performance of single controller is not yet enough to manage data centers and large-scale networks.

**Finally, we present the approach of high performance and reliable next generation distributed controller**. We discuss possible ways to organized it and mention highly demands software engineering techniques.

## II. Background

### A. History

Since early 2000th many researchers in Stanford University and Berkeley University have started rethinking the design and architecture of networking and Internet. The modern Internet and enterprise networks have a very complex architecture and are build using an old design paradigm. This paradigm includes the request for decentralized and autonomous control mechanisms which means that each network device implements both the forwarding functionality and the control plane (routing algorithms, congestion control, etc). Furthermore, any additional functionality in modern networking (for example, load balancing, traffic engineering, access control etc) is provided by the set of complex protocols and special gateway-like devices.

The enterprise and backbone networks, data center infrastructures, networks for educational and research organizations, home and public networks both wired and wireless are build upon a variety of proprietary hardware and software which are cost expensive and difficult to maintain and manage. This leads to inefficient physical infrastructure utilization, high oncost for management tasks, security risks and other problems.

Enterprise networks are often large, run a wide variety of applications and protocols, and typically operate under strict reliability and security constraints; thus, they represent a challenging environment for network management. The stakes are high, as business productivity can be severely hampered by network misconfigurations or break-ins. Yet the current solutions are weak, making enterprise network management both expensive and error-prone. Indeed, most networks today require substantial manual configuration by trained operators to achieve even moderate security [1], [3].

The Internet architecture is closed for innovations [4]. The reduction in real-world impact of any given network innovation is because the enormous installed base of equipment and protocols, and the reluctance to experiment with production traffic, which have created an exceedingly high barrier to entry for new ideas. Today, there is almost no practical way to experiment with new network protocols (e.g., new routing protocols, or alternatives to IP) in sufficiently realistic settings (e.g., at scale carrying real traffic) to gain the confidence needed for their widespread deployment. The result is that most new ideas from the networking research community go untried and untested.

Modern system design often employs virtualization to decouple the system service model from its physical realization. Two common examples are the virtualization of computing resources through the use of virtual machines and the virtualization of disks by presenting logical volumes as the storage interface. The insertion of these abstraction layers allows operators great flexibility to achieve operational goals divorced from the underlying physical infrastructure. Today, workloads can be instantiated dynamically, expanded at runtime, migrated between physical servers (or geographic locations), and suspended if needed. Both computation and data can be replicated in real time across multiple physical hosts for purposes of high-availability within a single site, or disaster recovery across multiple sites. Unfortunately, while computing and storage have fruitfully leveraged the virtualization paradigm, networking remains largely stuck in the physical world [6], [7], [8]. As is clearly articulated in [5], networking has become a significant operational bottleneck.

While the basic task of routing can be implemented on arbitrary topologies, the implementation of almost all other network services (e.g., policy routes, ACLs, QoS, isolation domains) relies on topology-dependent configuration state. Management of this configuration state is cumbersome and error prone  adding or replacing equipment, changing the topology, moving physical locations, or handling hardware failures often requires significant manual reconfiguration.

Virtualization is not foreign to networks, as networking has long supported virtualized primitives such as virtual links (tunnels) and broadcast domains (VLANs). However, these primitives have not significantly changed the operational model of networking, and operators continue to configure multiple physical devices in order to achieve a limited degree of automation and virtualization. Thus, while computing and storage have both been greatly enhanced by the virtualization paradigm, networking has yet to break free from the physical infrastructure. Furthermore, the network virtualization functionality implemented via additional protocols under L2-L4 layers increase the complexity and cost of network hardware and the difficulty of configuring such hardware.

### B. SDN

Further, to solve all above mentioned problems with network management and configuration, reduce the complexity of network hardware and software and make networks more open to innovations the broad community of academical and industrial researchers Open Networking Foundation [9] propose a new paradigm for networking  the Software Defined Networking (SDN).

The approach proposed by the SDN paradigm is to separate the control plane (i.e. the policy for management network traffic) from the datapath plane (i.e. the mechanisms for real packet forwarding) (see Figure 1).

Traditionally, hardware implementations have embodied the logic required for packet forwarding. That is, the hardware had to capture all the complexity inherent in a packet forwarding decision. According to new paradigm [1], [2], [4] all forwarding decisions are done first in software (remote controller), and then the hardware merely mimics these decisions for subsequent packets to which that decision applies (e.g., all

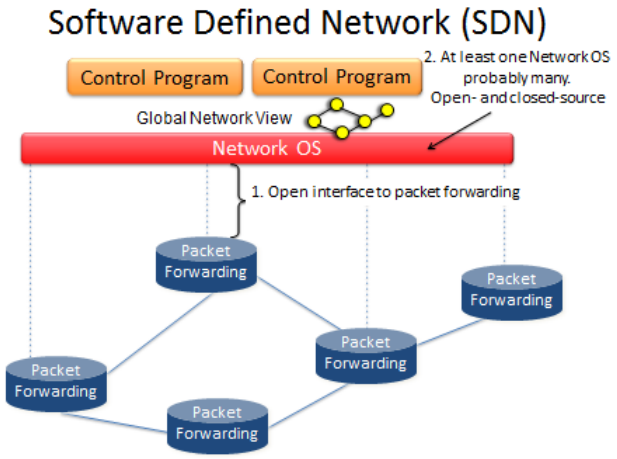


Fig. 1. Software Defined Network organization.

packets of given network flow). Thus, the hardware does not need to understand the logic of packet forwarding, it merely caches the results of previous forwarding decisions (taken by software) and applies them to packets with the same headers.

The key task is to match incoming packets to previous decisions. Packet forwarding is treated as a matching process, with all packets matching a previous decision handled by the hardware, and all non-matching packets handled by the software of remote controller. It is important to mention, that only packet headers are used in matching process.

A network switching hardware now must implement only a simple set of primitives to manipulate packet headers (match them against matching rules and modify if needed) and forward packets [1]. The core feature of such SDN-base switching software is a flow table which stores the matching rules (in form of packet header patterns to match against the incoming packet headers) and set of actions which must be applied to successfully matched packet.

Switching hardware also must provide common and vendor-agnostic interface for remote controller. To unify the interface between the switching hardware and remote controller the special OpenFlow protocol [10] was introduced. This protocol provides the controller a way to discover the OpenFlow-compatible switches, define the matching rules for the switching hardware and collect statistics from switching devices.

Figure 2 shows an interaction between OpenFlow-based controller and OpenFlow-based switching hardware, there controller provides the switch with a set of forwarding rules.

The control functionality in SDN paradigm is implemented by the remote controller  a dedicated host which runs special control software. At the present time there exist a number of controllers. The most well known are NOX [12], POX [13], Beacon [14], Floodlight [15], MUL [16], Ryu [19], and Maestro [18]. Again, a controller is a framework which interacts with OpenFlow-compatible switching devices and provides an interface for the user-written management applications to observe and control the entire network. A controller does not

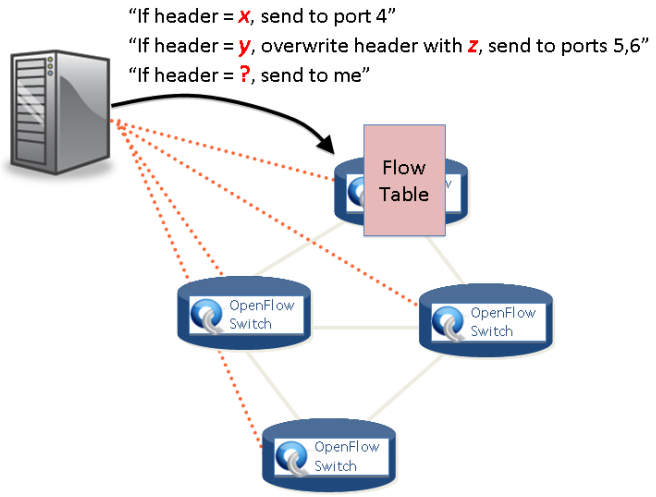


Fig. 2. Software Defined Network paradigm. Remote controller provides the forwarding hardware with rules describing how to forward packets according to their headers.

manage the network itself; it merely provides a programmatic interface. Applications implemented on top of the Network Operating System perform the actual management tasks.

A controller represents two major conceptual departures from the status quo. First, the Network Operating System presents programs with a centralized programming model; programs are written as if the entire network were present on a single machine (i.e., routing algorithms would use Dijkstra to compute shortest paths, not Bellman-Ford). Second, programs are written in terms of high-level abstractions (e.g., user and host names), not low-level configuration parameters (e.g., IP and MAC addresses). This allows management directives to be enforced independent of the underlying network topology, but it requires that the Network Operating System carefully maintain the bindings (i.e., mappings) between these abstractions and the low-level configurations.

### C. OpenFlow

The OpenFlow protocol is used to manage the switching devices: adding new flow, deleting the flow, get statistics, and etc. It supports three message types: controller-to-switch, asynchronous, and symmetric.

Controller-to-switch messages are initiated by the controller and used to directly manage or inspect the state of the switch. Asynchronous messages are initiated by the switch and used to update the controller of network events and changes to the switch state. Symmetric messages are initiated by either the switch or the controller and sent without solicitation.

The full set of messages and the detailed specification of OpenFlow protocol could be found in [11].

## III. Controller

Based on analyzing available materials about almost twenty four SDN/OpenFlow controllers, we proposed the reference architecture of SDN/OpenFlow controller shown on Figure 3.

The main components are:

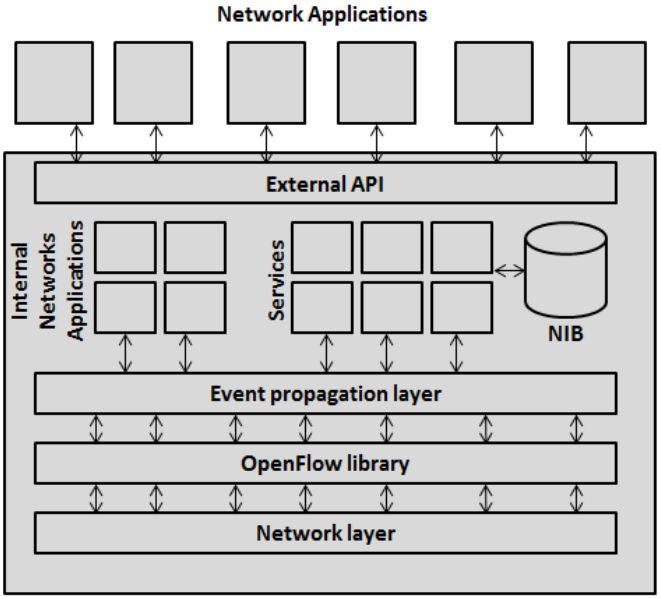


Fig. 3. The basic architecture of an OpenFlow/SDN controller.

1) Network layer is responsible for communication with switching devices. It is the core layer of every controller that determines its performance. There are two main tasks here:
   - Reading incoming OpenFlow messages from the channel. Usually this layer relies on the runtime of chosen programming language. For faster communication with NIC we can also use fast packets processing framework like netmap [20] and Intel DPDK [21].
   - Processing incoming OpenFlow messages. The common approach is to use multi-threading. One thread listens the socket for new switch connection requests and distributes the new connections over other working threads. A working thread communicates with the appropriate switches, receives flow setup requests from them and sends back the flow setup rule. There are a couple of advanced techniques. For instance, Maestro distributes incoming packets using round-robin algorithm, so this approach is expected to show better results with unbalanced load.
2) OpenFlow library. The main functionality is parsing OpenFlow messages, checking the correctness, and according to a packet type producing new event like "packetin", "portstatus", and etc. The most interesting part here, that is not in modern controllers yet, is resilience to incorrectly formed messages.
3) Event layer. The layer is responsible for event propagation between the controller's core, services, and network internal network applications. The network application subscribes on events from the core, produces other events to which other applications may subscribe. This is usually done by publishing/subscribing mechanism, either by writing your-own implementation or using the standard one like

libevent for C/C++, RabbitMQ for Erlang.

4) Sevices. This is the most frequently used network functionality like switches discovery, topology creating, routing, firewall.
5) Internal network applications. This is your-own application like L2 learning switch. "Internal" means that it's compiled together with the controller in order to get better performance.
6) External API. The main idea behind the layer is to provide language independent way to communicate with controller. This common example is the web-based RESTful API.
7) External network applications. Applications in any language leveraging services via External API exposed by controller services and internal applications. These applications are not needed in good performance and low latency communication with the controller. The common example is monitoring applications.
8) Web UI layer. It provides WEB-based user interface to manage the controller by setting up different parameters.

Also, the most important general question before choosing the controller or creating new one is what programming language to use. There is a trade off between the performance and the usability. For instance, POX controller written on Python is good for fast prototyping but it is too slow for production.

## IV. Experimental controllers evaluation

We performed an experimental evaluation of the controllers.

Our test bed consisted of two servers connected via 10Gb link. The first server was used to launch the controllers. The second server was used for traffic generation according to a certain test scenario.

We chose the following seven SDN/OpenFlow controllers:

- NOX [12] is a multi-threaded C++-based controller written on top of Boost library.
- POX [13] is a single-threaded Python-based controller. It's widely used for fast prototyping of network application in research.
- Beacon [14] is a multi-threaded Java-based controller that relies on OSGi and Spring frameworks.
- Floodlight [15] is a multi-threaded Java-based controller that uses Netty framework.
- MUL [16] is a multi-threaded C-based controller written on top of libevent and glib.
- Maestro [18] is a multi-threaded Java-based controller that uses JAVA.NIO library.
- Ryu [19] is Python-based controller that uses gevent wrapper of libevent.

Each controller runs the L2 learning switching application provided by the controller. There are several reasons for that. It's quite simple and at the same time representative. It fully uses controller's internal mechanisms, and it also shows how effective the chosen programming language is by implementing single hash lookup.

We used the latest available sources of all controllers dated March, 2013. We run all controllers with the recommended settings for performance and latency testing, if available.

As a traffic generators we used freely available cbench [17] and our-own framework hcprobe for controllers testing. Cbench and hcrpobe emulates any number of OpenFlow switches and hosts. Cbench is intended for measuring different performance aspects of the controller including the minimum and maximum controller response time, maximum throughput. Hcprobe allows to investigate various characteristics of controllers in a more flexible manner by specifying patterns for generating OpenFlow messages (including malformed ones), varying the number of reconnection attempts in case the controller accidentally closes the connection, choosing traffic profile, and etc. It is written in Haskell that is high-level programming language and allows users to easily create their own scenarios for controllers testing.

Our testing methodology includes performance and scalability measurements as well as advanced functional analysis such as reliability and security. The goal of performance/scalability measurements is to obtain maximum throughput (number of outstanding packets, flows/sec) and minimum latency (response time, ms) for each controller. For reliability we measured the number of failures during long term testing under a given workload profile. And as for security we study how controllers work with malformed OpenFlow messages.

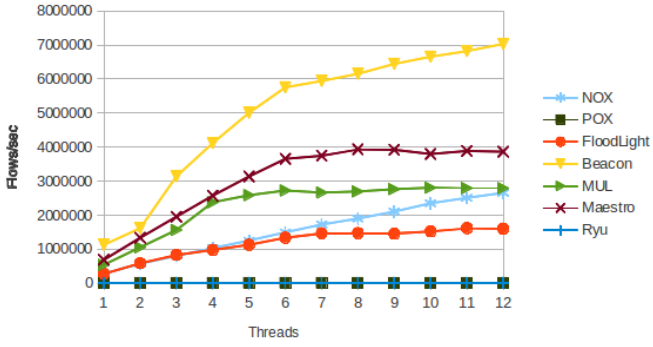


Fig. 4. The average throughput achieved with different number of threads.

The figure 4 shows the maximum throughput for different number of available cores per one controller. The single threaded controllers (Pox and Ryu) show no scalability across CPU cores. The performance of multithreaded controllers increases steady in line for 1 to 6 cores, and much slower for 7-12 cores because of using hyper threading technology (the maximum performance benefit of the technology is 40%). Beacon shows the best availability, achieving the throughput near 7 billion flows per second. This is because of using shared queues for incoming messages and batching for outgoing messages.

The average response times of all controllers are between 80-100ms. The long-term tests show that most controllers

when running for quite a long time start to drop connections with the switches and loose PacketIn messages. The average number is 100 errors for 24 hours. And almost all controllers crashes or loosing the connection with a switch when they received malformed messages.

Let us come back to throughput numbers and understand if the current performance enough. In the data centers new flow request arrives every 10us in maximum and 300us to 2ms in average [22]. Assuming small data center with 100K hosts and 32 hosts/rack, the maximum flow arrival rate can be up to 300M with the median rate between 1.5M and 10M. Assuming 2M flows/sec throughput for one controller, it requires only 1-5 controllers to process the median load, but 150 for peak load! In large-scale networks the situation can be tremendously worse.

The solving of the problem should go two ways. The first way is improving single controller itself by doing more advanced multi-threaded optimizations. The second way is using multiple controller instances which collaboratively manage the network. This approach is called a distributed controller.

## V. Moving to distributed controller

As we see in previous section single controller is not enough for managing the whole network. There are two problems here:

1) *Scalability*. Because networks are growing rapidly, the controller's resources are not enough to maintain state of all network devices. Moreover, the flow setup latency in a bigger networks is also increasing.
2) *Reliability*. The controller is a single point of failure. If the controller crashes, the network stops.

To solve the above problems, we need physically distributed control plane with centralized view of the entire network.

The scheme of the solution is presented in Figure 5.

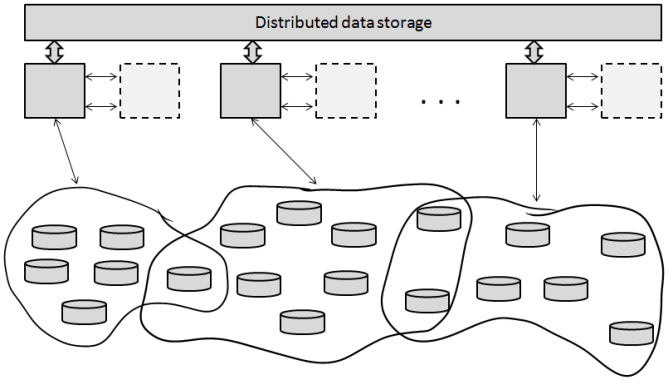


Fig. 5. The organization scheme of distributed controller.

The networks divides into segments, which controlled by dedicated instance of the controller. Network segments may overlap to ensure network resiliency in case of failure of any controller. In this case the switches will be redistributed over appropriate instances of the controller.

Each controller is connected to a distributed data storage that provides a consistent view of whole network. It stores all switch- and application- specific information. Application state is kept in the distributed data store to facilitate switch migration and controller failure recovery.

In addition, each controller has failover controller in case of its failure. It might be cold or hot. The cold failover is turned off by default and starts only when the master controller crashes. The hot failover receives the same messages as the master controller, but has read-only access. This provides the smallest recovery time.

There is a lot of open research questions like how to organized controllers consistency in the right way, how to reduce overhead on using distributed data store, how to do switch migration, how to run applications on distributed controllers, what the best controllers placement is, and etc.

## VI. Conclusion

Software Defined Networking (SDN) has been developed rapidly and is now used by early adopters such as data centers. It offers immediate capital cost savings by replacing proprietary routers with commodity switches and controllers; computer science abstractions in network management offer operational cost savings, with performance and functionality improvements too.

However there is a lot researching has to be done especially in SDN software area. Controllers are not yet ready to use in production because of insufficient performance to operate with data centers and large scale networks load.

Distributed controller is the next step in developing SDN/Openflow controllers. It's solving the scalability and reliability problems of modern controllers. For this we must use the techniques already existed in software engineering.

## References


[1] M. Casado, T. Koponen, D. Moon, and S. Shenker, *Rethinking Packet Forwarding Hardware*. In Proc. of HotNets, Nov. 2008.

[2] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Natasha Gude, Nick McKeown, Scott Shenker, *Rethinking enterprise network control*, IEEE/ACM Transactions on Networking (TON), v.17 n.4, p.1270-1283, August 2009

[3] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, Scott Shenker. *Ethane: Taking Control of the Enterprise*, ACM SIGCOMM 07, August 2007, Kyoto, Japan.

[4] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford , Scott Shenker, Jonathan Turner, *OpenFlow: enabling innovation in campus networks*, ACM SIGCOMM Computer Communication Review, v.38 n.2, April 2008

[5] J. Hamilton, *Data center networks are in my way*, Talk at Stanford Clean Slate CTO Summit, 2009.

[6] M. Casado, T. Koponen, R. Ramanthan, S. Shenker S. *Virtualizing the Network Forwarding Plane*. In Proc. PRESTO (November 2010)

[7] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, S. Shenker, *Extending Networking into the Virtualization Layer*, HotNets-VIII, Oct. 22-23, 2009

[8] J. Pettit, J. Gross, B. Pfaff, M. Casado, S. Crosby, *Virtual Switching in an Era of Advanced Edges*, 2nd Workshop on Data Center Converged and Virtual Ethernet Switching (DC-CAVES), ITC 22, Sep. 6, 2010

[9] Open Networking Foundation, *https://www.opennetworking.org*

[10] Openflow, *http://www.openflow.org*

[11] Openflow specification, *http://www.openflow.org/wp/documents*

[12] Gude, N., Koponen, T., Pettit, J., Pfaff, B., Casado, M., McKeown, N., and Shenker, S. *NOX: towards an operating system for networks*. SIGCOMM Computer Communication Review 38, 3 (2008), 105-110.

[13] Pox documentation, *http://www.noxrepo.org/pox/about-pox/*

[14] Beacon documentation, *https://openflow.stanford.edu/display/Beacon/Home*

[15] Floodlight documentation, *http://floodlight.openflowhub.org/*

[16] Mul documentation, *http://sourceforge.net/p/mul/wiki/Home/*

[17] Cbench documentation, *http://www.openflow.org/wk/index.php/Oflops*

[18] Zheng Cai, *Maestro: Achieving Scalability and Coordination in Centralized Network Control Plane*, Ph.D. Thesis, Rice University, 2011

[19] Ryu documentation, *http://osrg.github.com/ryu/*

[20] Luigi Rizzo, *netmap: a novel framework for fast packet I/O*,Usenix ATC'12, June 2012

[21] Packet Processing is Enhanced with Software from Intel DPDK, *http://intel.com/go/dpdk*

[22] Theophilus Benson, Aditya Akella, and David A. Maltz, *Network traffic characteristics of data centers in the wild*, IMC, 2010