A practical experience in designing an OpenFlow controller

Roberto Bifulco, Roberto Canonico
Universita' degli studi di Napoli "Federico II"
{roberto.bifulco2, roberto.canonico}@unina.it

Marcus Brunner, Peer Hasselmeyer, Faisal Mir
NEC Laboratories Europe
{brunner, faisal.mir, peer.hasselmeyer}@neclab.eu

***Abstract*—Software Defined Networking (SDN in short) is reshaping the future of computer networks. By decoupling control and data planes, SDN technologies allow a more flexible management of network infrastructures, whose resources may be operated by means of a well defined programming interface. Several approaches have been recently proposed to implement the SDN concept. OpenFlow is maybe the most prominent SDN component, having been supported by several device vendors. This paper discusses a practical experience in designing an OpenFlow controller for a Mobile Cloud Management system. We present the programming model and the designed abstraction and discuss the lesson learned.**

***Index Terms*—Software-Defined Networking, OpenFlow controller, scalability, programming model**

I. Introduction

Software Defined Networking (SDN) suggests the separation of control and data planes, providing well defined interfaces among them, in order to enable flexible configurability and programmability of the network. Programmability is one of the characterizing properties of Software Defined Networks: the control-plane behavior can be defined writing "network programs" that manage a set of switches, providing rich network applications and features. In some way, SDN is bringing into computer networks the same shift that in past decades electronic devices had from special purpose machines, to general purpose ones. With SDN, the current special purpose network can become a general purpose, hence programmable, network. Programmability, in a new context like computer networks, calls for new programming models, tools and languages. In this sense, to continue with the general purpose computer metaphor, SDN are still in the "machine language" phase, where the programming languages are rudimentary, strictly connected to the hardware, and the main part of network programming is still performed manually, without the aid of any tool. The potentially increased complexity, paid to gain in flexibility, can be tamed by using appropriate abstractions and methodologies, as it happens in the software engineering field. Methodologies, abstractions and tools have to address the complexity taking into account, at the same time, classical networks issues, such as scalability.

OpenFlow is one of the most popular SDN-enabling technologies. OpenFlow was born as a means to enable network experiments on campus networks [1], and its first deployments were actually universities' networks. Over time, the advantages of an SDN approach to networks have been explored, leading to applications of OpenFlow to other environments, such as enterprise networks, as in the OpenFlow implementation of the Ethane architecture for network security [2]. More recently, OpenFlow has been also applied to challenging scenarios like datacenter networks [3] and wide-area networks [4]. The Open Networking Foudation [5] (ONF), that is responsible for the OpenFlow specification, currently involves a number of academic and industrial partners. An increasing number of device manufacturers have implemented OpenFlow in their products and Google recently declared the adoption of OpenFlow in its networks [6].

This work presents practical experience in developing a distributed OpenFlow controller for supporting a Mobile Cloud Computing technology, i.e., Follow-Me Cloud (FMC) [7], developed at NEC Laboratories Europe. We introduce the problems we faced in developing the FMC controller and the solutions we adopted in terms of programming methodology and abstractions. In particular, we highlight the scalability issues to be taken into account while developing a controller, how our design describes the network through an object model and handles operations to provide scalability and extendability. The paper is organized as follows: in section II we introduce, the Follow-Me Cloud technology. In section III we present our FMC Controller design and in section IV we discuss the lesson learned. Related works are presented in section V. Finally, we conclude in section VI.

II. Follow-Me Cloud

Follow-Me Cloud (FMC) provides mobility features in a TCP/IP network for both users and services, maintaining all the ongoing network connections active. FMC is applied to a TCP/IP network (see fig. 1.a) in which L2 access networks are connected to an L3 "core" network, that provides connectivity among them, through OpenFlow switches (OFS). Hence, the network core is unchanged and totally unaware of FMC. To provide mobility to a mobile node (MN) that is changing its access network from an "home" to a "foreign" network, FMC requires that a new IP address, belonging to the foreign network, is assigned to MN to work as "locator". The original IP address of MN, that belongs to the home network, is still used by MN itself and by any node that is communicating with MN, since it works as "identifier". When a network node (we

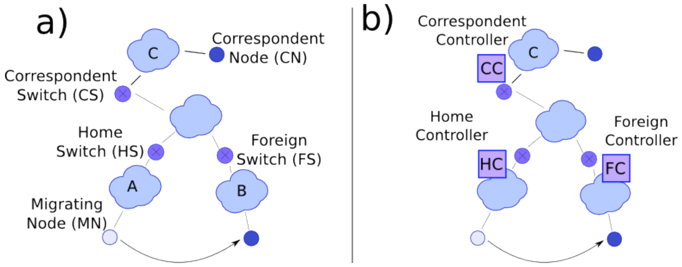

Fig. 1. a. FMC example scenario; b. FMC distributed architecture

call such a node *correspondent node* or CN in short) sends a packet to MN, it uses the *identifier* address as destination address. The OFS connecting the CN's network (CNet) to the core network performs an address translation, to substitute the *identifier* with the *locator* address. When the packet reaches the foreign network, the OFS at the edge of such network performs a new translation, substituting the *locator* with the *identifier*, in order to deliver the original packet to the MN.

There are several issues to be solved in order to make FMC usable. In particular, (i) FMC must scale with the number of users and migrations[1], and (ii) must be easily deployable in traditional networks.

Scalability is provided by a distributed architecture. The design of the distributed architecture follows the principle of distributing knowledge to where it is actually needed. The needed information at a particular network is the *identifier/locator* mapping for a given network entity, while the networks that need to know about this mapping are the ones that are connected to the core network through the Home Switch (HS), the Foreign Switch (FS) or Correspondent Switches (CS). To manage this information and to distribute it among networks, FMC uses the architecture depicted in Figure 1.b. With respect to the migrating node, the FMC architecture comprises three different roles: *Home Controller (HC)* that controls the network to which the *identifier* address belongs to; *Foreign Controller (FC)* that controls the network to which the *locator* address belongs to; *Correspondent Controller (CC)* that controls one or more CSes. The architecture is flexible enough to enable a single controller to play one, two or all the roles for the same migrating node, e.g., because the same controller is in charge of managing multiple networks. This approach also offers the possibility to adapt the number of controllers used in the network, in order to tackle the actual network load. Upon MN migration, the HC informs the FC that MN is moving to the foreign network, so that a new *identifier/locator* mapping can be established. During this process, the HS and the FS are configured accordingly. When a CN that resides on a correspondent network, that is unaware of the MN migration, sends a packet to MN, the packet reaches the home network where it is intercepted by the HC and redirected to the correct location. Moreover, the HC sends to the CC the updated *identifier/locator* mapping information for MN, so that subsequent packets are early redirected to the correct location.

To make deployment of FMC in a TCP/IP network as easy as the placement of OFSes at the edge of L2 access networks, FMC managed OFSes have to provide traditional switching functions in addition to FMC ones. E.g., OFSes work as Learning Switches (LS), i.e., they learn MAC addresses and associate them with switch ports. When a node migration has to be handled, the switch extends the packet handling with FMC functions.

III. Controller design

The presented logical architecture has been implemented as a distributed OpenFlow controller on top of the NOX [8] OpenFlow Controller Framework (OCF), even if we believe that all the concepts can be ported to any other OCF. In this paper, we are are referring to NOX as an OCF, even if its authors define it as a Network Operating System (NOS) in [8]. NOX provides a set of helper methods and APIs to interact with OpenFlow switches, while we assume that a NOS should also provide advanced hardware and programming abstractions. Moreover, we use the term "OpenFlow controller" to identify the combination of an OCF with the OF applications running on top of it.

From a programming perspective, to support FMC operations, the controller has to provide these features:

1) it should easily become a distributed application if needed, i.e., different parts of the controller should be able to be moved to different computing nodes (in a different network location);
2) it must be extensible, providing the ability to combine different network functions, even not FMC related;

The raw outcomes of an OpenFlow controller are Flow Table Entries (FTE) to be installed at switches, and network packets to be forwarded by switches, both generated in response to network events and/or in response to high level control operations. To accomplish such tasks providing the aforementioned features, efficient models and abstractions must be provided. In particular, controller design has to provide both a data model to describe the network and its state, and a control logic programming model to interact with such data model. The design phase has to address also the so called non-functional requirements, e.g., performance and scalability of the implemented system. Using again the general purpose computer metaphor, in a general purpose computer the system behavior depends both on the hardware architecture and on the software running on top of such architecture, likewise in OpenFlow, it depends both on the network architecture and on the control logic implemented by the controller. Understanding the effects brought by different ways of interacting with the OpenFlow network is an important step to drive the design decisions. Hence, during the development of the FMC OpenFlow controller, we performed a preliminary analysis of the different dimensions that characterize an OpenFlow controller. We refer to this dimensions as "*Control Logic Dimensions*".

[1] Scalability of FMC is discussed in [7]

A. Control logic dimensions

In OpenFlow the actual control plane behavior is defined by the Controller. Given the flexibility of the OpenFlow approach, we can try to perform an high level classification of the characteristics of different control logics, identifying coarse-grained dimensions to classify such applications, the same way computers' programs are classified as CPU-bound, I/O bound and so on.

- **Flows Granularity**: defines the granularity of the network flows managed by the control logic. The granularity is defined after the header fields of current data packets. For example, a flow identified by the solely destination MAC address is coarse grained, while a flow identified by the combination of IP addresses and port numbers is much more fine grained;
- **Network Visibility**: an OF application may need detailed network traffic information or links statuses, for, e.g., load balancing or route reconfiguration. Depending on the control logic, the quantity and frequency of switches status update may vary;
- **Network state**: network state is related to the information the control logic has to manage, in order to provide its functions. Typical examples of network state are routing tables, end-points identity information, etc.;
- **Reactivity**: OpenFlow provides a mean to reactively program switches, through the forwarding of network packets to the controller. A control plane can range from being fully reactive, when each OF table entry is installed in response to a packet coming to the controller, to proactive, when all entries are installed before network traffic arrives to the switch.

In designing the FMC controller, some dimensions where dictated by the mobility technique adopted in FMC, e.g., the flows granularity. The other dimensions may vary according to the taken implementation decisions. In particular, in the FMC implementation, we decided that we can tolerate an increased network state to maintain at controllers, instead of increasing the number of OpenFlow messages exchanged to retrieve the switches' status. At the same time, we adopted a control logic as much proactive as possible, i.e., pre-installing FTEs when it is possible.

B. Hierarchical control

FMC architecture suggests a hierarchical controller organization. In FMC the hierarchy is mainly related to the geographic locations, and, in particular, there are two hierarchical levels. A (i) local level, related to the handling of the Learning Switch functions and low level FMC mobility technique operations, and a (ii) global level, that provides a global view of the network and that coordinates the local levels to provide FMC functions on a geographic scale. The two levels differ for both the performed operations and the processed data. The local level handles OFSes directly, providing FTEs and handling network events. The global level task is to coordinate the local levels to provide the required network functions, i.e., in the FMC case the network addresses mobility. Hence, the global level is also the place in which the controller north-bound interface is implemented. The north-bound interface, in the FMC example, provides methods to, e.g., define identifier/locator mappings or to request a node migration[2].

C. Data model

The controller is built around an easily accessible view of the switches, so that advanced functions can be defined using a common network model. We defined the network model using an object-oriented (OO in short) approach. The decision is motivated by the suitability of the OO paradigm for the description of network devices like OF-switches, and by the deep understanding of the OO model by programmers, that are highly involved in the design and management of a Software Defined Network.

The object-oriented network model is composed of the following base classes:

- *Network*: contains a globally unique identifier and a set of OFSwitch objects. It works mainly as a container for OFSwitch objects and for network state that is related to the whole network, i.e., it is part of the global level of the hierarchy;
- *OFSwitch*: is the base class used to represent and manage an OpenFlow switch. It includes both the state of the switch, that can be used programmatically by controller functions, and a set of methods used to handle network events.

The network model is dynamic, OFSwitch objects are added or removed in response to the connections initiated by corresponding switch devices. The model is dynamic also in the sense that an OFSwitch object contains information about the switch current internal state, such as the installed FTEs. The implementation of the described classes can be tuned according to the desired impact on the different Control Logic Dimensions. For example, the OFSwitch can be designed to cache FTEs installed at the represented switch, but can also dynamically retrieve the FTEs from the switch, i.e., sending OpenFlow messages.

D. Scalability

To provide scalability, the controller uses several *Controller nodes* to execute the network control logic. At this aim, the network model is extended to include in each OFSwitch class, in addition to switch related network state, also the related control logic. The control logic implemented in an OFSwitch object can operate only on the switch represented by the object itself. The network control logic, hence, belongs to the OFSwitch instances and not to the controller as a whole. Since each OFSwitch instance contains also all the data related to the represented switch (as explained in sec. III-C), each instance can be moved among controller nodes when needed.

[2] In the current implementation the north-bound interface is implemented as a REST interface that exchanges JSON serialized data.

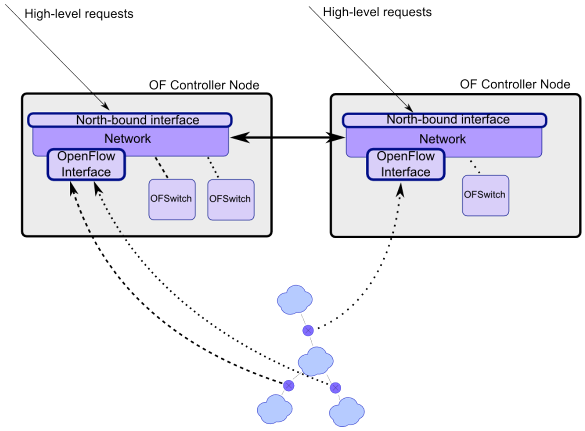

Fig. 2. FMC controller architecture and deployment example: network OFSes are distributed among two OF Controller Nodes. OFSes are managed through OFSwitch objects. The Network object dispatches events coming from switches to the correspondent OFSwitch objects, and implements transparent communication among OFSwitch objects hosted at different controller nodes.

Each controller node contains at least a Network object, that is in charge of dispatching network events it receives from controlled switches: each event is forwarded to the corresponding OFSwitch object, that executes proper algorithms to handle it (See Figure 2). Using this programming model, distributing the controller application becomes a problem of partitioning the OFSwitch objects among different controller nodes. The controller nodes are assumed to be placed in locations that are near the switches they are controlling (from the network perspective), in order to reduce the delay of controller-switch communication. The current FMC controller implementation takes care also of the distribution of messages destined to OFSwitch instances, that are used to develop control algorithms that involve more than one switch, i.e., algorithms belonging to the global level of the hierarchy. Such communications are handled in a way that is transparent to the programmer, using a "proxy" object in case the OFSwitch object is located on a different OFC node. The model provides the programmer with a clear separation of the control decisions that could be taken at the single switch level (local level), from the ones that need a broader view of the network (global level).

E. Extensibility

Extensibility is provided using OO paradigm characteristics, like inheritance (See Figure 3.a). The OFSwitch class can be extended to provide new or enhanced functions, e.g, by overriding and extending methods. For example, it is possible to provide a LearningSwitch implementation, that resembles classical L2 switches functions, and then, extending this class, other functions can be added (e.g., the FMCSwitch class implements the FMC functions). The main issue in providing extendability through an OO model is the paradigm mismatch between OO programming and OF switches programming, since the switches programming is performed by means of FTEs. Because of this, the OO paradigm is not used to program the network itself, but to interact with the network devices in a (hopefully) simplified way.

The final outcomes of the execution of methods from OFSwitch class and derivatives are a set of FTEs to be installed on the switch, and a set of packets to be sent by the switch. The extended classes and methods, hence, are in charge of providing a set of FTEs and network packets modified accordingly to the desired result. This process is tricky: the addition of a FTE can have unexpected effects on the behavior of the switch, that is defined by the combination of all the FTEs installed on that switch. Moreover, some packets coming from a super-class's methods may not be sent anymore in the extended function, and so on. We are actually handling a two levels programming problem:

1) high-level programming is performed by using API provided by OFSwitch classes and derivates. Programmers are in charge of defining convenient APIs to allow the extensions of the functions they are providing in a given class;
2) low-level programming is performed by means of FTEs and packets sent through the switch. All the high-level functions are finally translated in FTEs and packets to be sent.

Our purpose is to provide extendability in any case, so, also when the developer of an OFSwitch sub-class is not providing methods to easily modify its application behavior before the FTEs and network packets are generated.

To this end, the OFSwitch class provides convenient methods to perform network events handling. For example, when a packet is forwarded from a switch to the controller, a *packet_in* network event is generated. The event is handled by a specific method in the OFSwitch class, that implements the control logic to handle the packet, and provides (i) an ordered list of FTEs to install at switches, and (ii) an ordered list of network packets that must be sent by the selected switches. This approach allows for the extension of the OFSwitch class: a subclass that inherits from the OFSwitch can still use the methods from the superclass, to get the lists of FTEs and packets to send, and adjust them according to the extended control logic.

In addition to this feature, the OFSwitch class provides a dedicated method to install rules on the corresponding OFswitch, in order to intercept all the FTE installation requests (and network packets sending requests) before actually issuing them at the corresponding switch. Using this approach, it is possible to introduce some extended logic that, before the actual switch programming happens.

IV. Discussion

In this section we discuss how the design we made helped us in developing the FMC distributed controller. As stated in section III-B, FMC is well suited to use a hierarchical organization of the control logic, with a good separation between local and global operations. The Object Oriented model used to describe switches and networks is particularly useful at this aim. The Network class, working as a container for a set of OFSwitch objects, is a good place to implement the

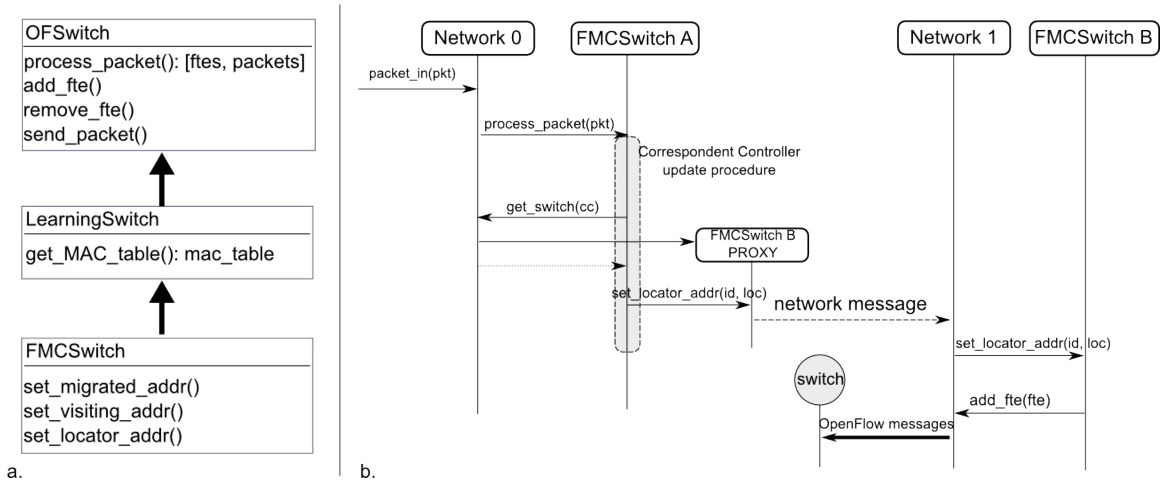

Fig. 3. a. FMC controller object model: the basic OFSwitch class is extended to introduce LearningSwitch functions and, then, it is further extended to introduce FMC functions. b. FMC controller operations sequence diagram, when an Home Switch receives a packet from a not updated correspondent network: the Network object contains information about the location of the correspondent node, hance, the related CC can be updated with the *identifier/locator* mapping information.

global control logic, while the OFSwitch objects, being closely related to the network device, are in charge of handling the local control logic. If necessary it could be possible also to provide more intermediate levels in the hierarchy, but for our purposes it was not the case.

Local control logic includes also the handling of local network events, generated by switches, that are actually handled at corresponding OFSwitch objects. This approach helped in the distributed controller implementation, since all the local events are kept local, requiring no interactions among different Controller nodes. The OFSwitch class (and its sub-classes), in this context is used both as an aggregation and filtering point for local network events, that then are passed up in the hierarchy shaped as “high-level events”. The implementation of such filtering and aggregation functions , i.e., the shape of “high-level events“, is tightly coupled to the required global control logic. E.g., in figure 3.b the event linked to the reception, at the home network, of a packet destined to a migrated MN is handled locally by the HC and then translated into a high-level event, that is implemented through the call of the *set_locator_addr()* method on the (proxy) of the FMCSwitch object representing the CC. Using OO inheritance, The OFSwitch class can be extended by a sub-class that implements the required filtering and aggregation logic, then, the sub-class is associated with the switches that require the application of such logic. The final outcome is the association of different OFSwitch classes to different switches, according to the control logic we have to implement at such switches, in an elegant and easy way.

The combination of OFSwitch sub-classes that expose an interface and network events related to the global control logic, with the presence of global-level Network objects made the implementation of the north-bound interface functions straight forward. The Network object is a good place to expose the north-bound interface, while the mapping of this interface to the low level FTEs is made easy using OFSwitch sub-classes, that translate a set of high-level methods in corresponding FTEs. Hence, the north-bound interface interacts with OFSwitch sub-classes instead of having to take care of low-level FTEs, separating the development of low level FTEs programming from the development of high-level network functions.

While OO paradigm is really useful in providing separation of concerns and scalability, it does not help in providing extensibility, that still requires a direct handling of FTEs, i.e., the function that is going to be extended must be well known by the programmer, to correctly handle the provided FTEs. Our design helped the extensibility by providing a mean to manipulate FTEs before they are installed at switches. In particular, the abilities to intercept FTEs that are going to be installed and network packets that are going to be sent are crucial to this end. Moreover, the use of FTEs caching at OFSwitch objects made the analysis of the actual switches behavior much easier, simplifying both the network functions extension and controller debugging.

V. Related Work

The design of an OpenFlow controller, like the design of any software application, requires the application of languages, methodologies/abstractions and tools. In this section we provide an overview of available network languages, models and abstractions as implemented by OFCs, and tools to aid OpenFlow controller development.

A. Programming languages

Frenetic [9] is a high-level language based on the functional programming paradigm, that provides the programmer with an omniscient, centralized view of the network. A run-time system, linked to the language, “translates” the high-level instructions to a set of low-level packet processing rules, and manages them interacting with network equipment. NetCore

[10] is an evolution of Frenetic, that extends the high-level language and provides some improvement in the compilation algorithms and run-time system, trying to speed up the network performance.

B. Programming Frameworks

Hyperflow [11] provides a Distributed OpenFlow Controller Framework that separates the network in partitions. Each partition is assigned to a controller instance and all the instances are synchronized by means of a publish/subscribe mechanism. In this approach each controller instance runs the same control logic, while the framework distributes the network events and OpenFlow messages to the appropriate controller instance. Hence, HyperFlow provides transparent scalability of the Controller. To provide its features, Hyperflow distributes consistently the network state updates and OpenFlow messages. Onix [12] defines a Network Information Base (NIB) that is read and written by the control logic. The NIB is actually distributed using different strategies defined by the developer, that chooses the storage system type for the NIB data, according to their needs in terms of speed, consistency and reliability. All the synchronizations and operations on the physical network are performed through the NIB, so, by partitioning the NIB among several controller instances, it is possible to share the overall load and distribute responsibilities. A third approach is based on a service view of the Controller: the control logic is implemented as a collection of services that collaborate among them. Each service is able to run on a separate controller instance, hence, several instances can be used to share the workload. Clearly this approach requires a careful design of the control logic, whose implementation as a collection of services, and the way such services are distributed, dictates the actual shape of the system [13].

C. Tools

To test the correctness of OpenFlow applications, NICE was proposed in [14]. NICE is a tool for automatic OpenFlow applications testing, that combines model checking and concolic execution to explore the state space of OpenFlow programs written for the NOX controller platform. In FlowChecker [15] the aim is to detect OFS misconfigurations. FlowChecker uses manually built binary decision diagrams to encode OpenFlow rules and then applies model checking in order to detect OpenFlow switches misconfigurations.

VI. Conclusions

Software Defined Networking is a promising paradigm for future network management, and OpenFlow is emerging as a successful industry-supported SDN building block. In this paper we presented the design decisions and the experience we made in developing a distributed OpenFlow controller for supporting the Follow-Me Cloud technology. We introduced a hierarchical view of the control operations, as well as a network model based on the Object Oriented paradigm. Furthermore we introduced a first coarse-grained classification of OpenFlow controller behavior in respect to a few parameters, and explained how the OO paradigm can be exploited to support both scalability and extendability of the OpenFlow controller, taking into account the paradigm mismatch among OO model and the OpenFlow programming model.

References

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1355734.1355746

[2] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: taking control of the enterprise," in *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, ser. SIGCOMM '07. New York, NY, USA: ACM, 2007, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/1282380.1282382

[3] A. Tavakoli, M. Casado, T. Koponen, and S. Shenker, "Applying nox to the datacenter," in *Proc. of workshop on Hot Topics in Networks (HotNets-VIII)*, 2009.

[4] Y. Kanaumi, S. Saito, and E. Kawai, "Toward large-scale programmable networks: Lessons learned through the operation and management of a wide-area openflow-based network," in *Network and Service Management (CNSM), 2010 International Conference on*, oct. 2010, pp. 330 –333.

[5] "Open networking foundation," `https://www.opennetworking.org/`. [Online]. Available: https://www.opennetworking.org/

[6] Wired - going with the flow: Googles secret switch to the next wave of networking. [Online]. Available: http://www.wired.com/wiredenterprise/2012/04/going-with-the-flow-google/

[7] R. Bifulco, M. Brunner, R. Canonico, P. Hasselmeyer, and F. Mir, "Scalability of a mobile cloud management system," in *Proc. of SIGCOMM workshop on Mobile Cloud Computing (MCC-2012)*, 2012.

[8] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "Nox: towards an operating system for networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 3, pp. 105–110, 2008.

[9] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: a network programming language," *SIGPLAN Not.*, vol. 46, no. 9, pp. 279–291, Sep. 2011. [Online]. Available: http://doi.acm.org/10.1145/2034574.2034812

[10] C. Monsanto, N. Foster, R. Harrison, and D. Walker, "A compiler and run-time system for network programming languages," *SIGPLAN Not.*, vol. 47, no. 1, pp. 217–230, Jan. 2012. [Online]. Available: http://doi.acm.org/10.1145/2103621.2103685

[11] A. Tootoonchian and Y. Ganjali, "Hyperflow: a distributed control plane for openflow," in *Proceedings of the 2010 internet network management conference on Research on enterprise networking*, ser. INM/WREN'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 3–3. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863133.1863136

[12] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker, "Onix: a distributed control platform for large-scale production networks," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–6. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924968

[13] H. Shimonishi, S. Ishii, L. Sun, and Y. Kanaumi, "Architecture, implementation, and experiments of programmable network using openflow." *IEICE Transactions*, vol. 94-B, no. 10, pp. 2715–2722, 2011.

[14] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford, "A nice way to test openflow applications," *EPFL Technical Report*, Oct. 2011.

[15] E. Al-Shaer and S. Al-Haj, "Flowchecker: configuration analysis and verification of federated openflow infrastructures," in *Proceedings of the 3rd ACM workshop on Assurable and usable security configuration*, ser. SafeConfig '10. New York, NY, USA: ACM, 2010, pp. 37–44. [Online]. Available: http://doi.acm.org/10.1145/1866898.1866905