From Software-Defined to Human-Defined Networking: Challenges and Opportunities

Elisa Rojas

***Abstract*—The SDN paradigm is still in an early stage of development. Considering full automatization and effortless management as the main objective of these networks, we believe diverse challenges need to be tackled. For this purpose, this article reviews the SDN architecture from top to bottom, paying attention to components yet under standardization or that demand enhancement from a network operator's perspective. The main conclusion is that the SDN area requires a significant amount of research to reach its full potential, which we consider a huge opportunity to innovate toward a truly human-defined networking.**

***Index Terms*—Software-Defined Networking, Network Architecture, Network Management, Future Internet.**

I. Toward Human-Defined Networking

THE Software-Defined Networking (SDN) paradigm emerged in recent years, decoupling the control and data planes and breaking the vendors stranglehold. SDN brought the desired opening of black boxes of functionality to service providers, together with a new hope to decrease the costs involved in provisioning and managing large networks [1].

As one pillar of SDN is logically centralizing the network intelligence, scalability [2] and security [3] are two of the main topics discussed. Nevertheless, diverse challenges related with the SDN architecture need to be tackled, such as the management plane [4] or the evolution of the Northbound Interface (NBI) and Southbound Interface (SBI) [1]. Furthermore, from our experience, many telcos still perceive the SDN world as a jungle to go deep into and initially feel reluctant to completely change their approach to networks, while it should be considered totally the opposite, as a technology that eases the landing of new network applications and services.

To convert the SDN ecosystem into a reality in everyday networks, we need to step toward the concept of Human-Defined Networking (HDN), where information and interactions emanate from the people managing the network instead of being strictly software or even hardware dependent (traditional networks). Basing concepts on human-like behaviors will ease the evolution of networking. HDN is one branch of the SDN concept focused on the human part of the network (network managers) and it should be capable of:

1) **Providing deployment flexibility:** Users might vary their necessities over time, and thus managers require networks to be flexible enough to evolve accordingly. A direct consequence is that network functionality needs to be modular, so that these modules can be activated/deactivated (or even *dragged and dropped* in a graphical interface) following the requirements at each moment and without losing the tight control network managers are used to have.
2) **Constructing complex behaviors automatically:** Composition of network modules should not derive in monolithic solutions, but rather in flexible combinations in which potential conflicts are even resolved automatically based on *hints* provided by the network manager. Ideally, this should be dynamically done at run-time and independently of the NBI. If not, this implies evolving the semantics for writing functionality, related with the next point.
3) **Speaking and understanding a human-like language:** An SDN requires engineers to develop software written in a specific language, even if they lack networking knowledge. While an HDN exhibits a higher abstraction in the NBI to network managers, avoiding the need for specific software programming.
4) **Disengaging from the physical network:** Network managers should be able to control networks independently of what is physically deployed. Thus, SBI protocols should be able to abstract network elements and not depend on them (as it currently does). HDN focuses on serving the users, not the network, and thus its motto: *network as a means, not as an end*.

These four requirements, interrelated as a puzzle, become fundamental pieces that facilitate network management. The first three ease deployment, development and run-time phases in an SDN, particularly the ones operated manually (that is, by humans). While the last one aims to release the –still existing– dependency between software and hardware in SDN, which indirectly affects the three aforementioned ones.

In this paper we present the main challenges, and thus opportunities, to move toward HDN. Combining state of the art and future directions for research, we hope it helps the evolution of the SDN paradigm. For this purpose, we traverse the SDN architecture top to bottom. As depicted in Fig. 1, we start with the concept of an ***SDN app store***, followed by the act of ***Writing an SDN app***, then the review of what could be considered the next steps for OpenFlow in ***Approaching OpenFlow+***, and finally we try ***Unboxing the network boxes***. Hence, according to the SDN architecture defined by the Open Networking Foundation (ONF), we analyze the Application and Data planes, leaving the intermediate Control plane as an independent box, which contains the specificities of the SDN platform deployed.

Elisa Rojas is with Telcaria Ideas S.L., 28911 Leganes (Madrid), Spain, e-mail: elisa.rojas@telcaria.com

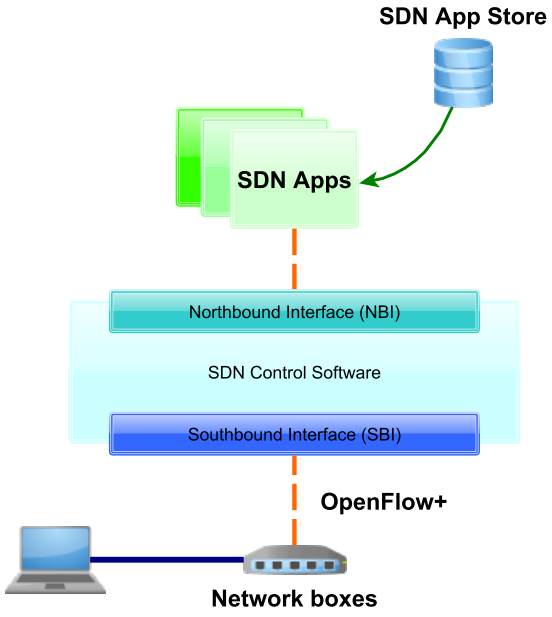

Fig. 1. SDN architecture and the challenges covered in this article

II. SDN APP STORE

Imagine a telco operator with a clear plan to leverage SDN in the incoming years: they know the services and have defined a project outline, so now what is the next step? That is, provided *they have listed all the SDN ingredients, where is the SDN grocery?* Currently, no clear marketplace for SDN applications (or apps) exists. If an operator requires some app, they should check the SDN platforms available and their apps catalog. For this reason, most times SDN frameworks are selected based on the apps or use cases they implement. Moreover, network managers usually end up modifying existing apps because these catalogs are scarce and rarely accomplish all the requirements of the operator. This restricts deployment flexibility, which is the first key component of HDN.

Fig. 2 shows a summary of the use cases published by the two most popular SDN open source projects at the moment: OpenDaylight (ODL) and ONOS. The left side lists the use cases and the right indicates some of the Projects (ODL) or Apps (ONOS) related with them.

There are a few commercial approaches for an SDN app repository. One is HP's *SDN App Store*, which offers a selection of SDN applications to be deployed with the enterprise-ready HPE VAN SDN controller. But altogether represent just a few dozens of apps and entirely designed for their SDN controller.

Another repository is NEC's *SDN Partner Space* or *Open Global SDN Ecosystem Program*, with a slightly different approach, as it presents SDN applications inside well-defined use cases, offering not only the apps, but also hardware, installation and maintenance support.

To accomplish the **deployment flexibility** required by HDN, two concepts need to be further developed related with the SDN app store: SDN app packages, plus support and updates.

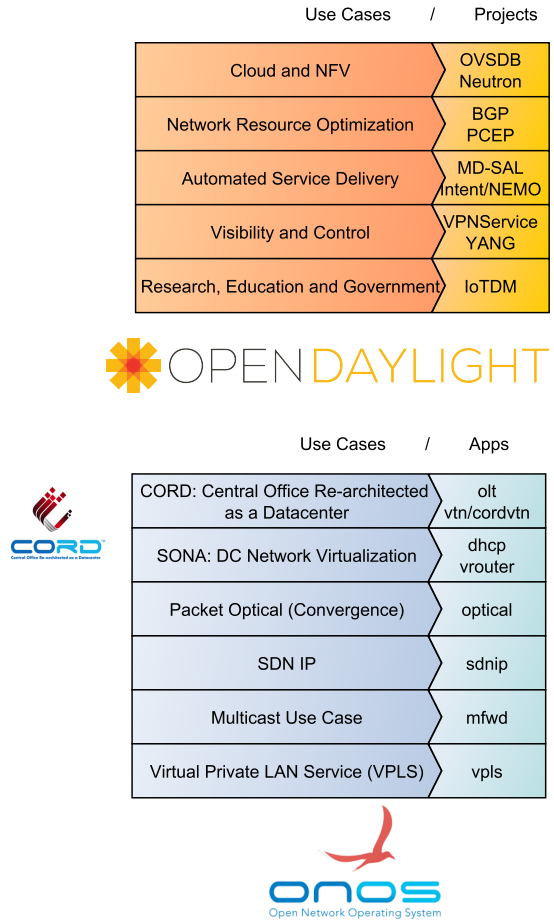

Fig. 2. Well-known use cases for the most popular open source SDN platforms: ODL and ONOS

A. SDN app packages

A first step toward the concept of an app store is the definition of an SDN app package. For example, an Android package needs to accomplish a set of conditions before being published in the Google Play store, such as: accepting terms and policies, establishing hardware and software requirements or determining the installation price. In the particular case of the SDN world, an initial idea would be reusing two common concepts of general software applications: *system requirements* and *configuration parameters*. The question remains open though: how should we express them?

1) System requirements: System requirements are usually defined during the development phase of the SDN app. Apart from the SDN platform, the system requirements include hardware and software requirements (CPU, memory or specific software packages) and also network requirements (such as SBI protocols supported, type of SDN switch or type of network). For example, a layer-two switch application could require any free-loop topology and this should be verified prior to deployment.

2) Configuration parameters: Configuration parameters are usually defined during the deployment phase of the SDN

app, but they previously require some hooks in the app that allow them. Examples of configuration parameters are specific default rules for a firewall or the table cache duration for a routing application. Additionally, an SDN package could define how to combine itself with another package, that is, specific parameters for SDN app composition. For instance, if we want to deploy the firewall and the routing apps, they could have one parameter to indicate their execution priority. However, composition of SDN apps is still under research and not fully covered by current SDN platforms.

B. Support and updates

Finally, SDN apps and their supporting platforms should allow updates to enrich them with new functionalities or solving reported bugs. Although some SDN platforms, such as ONOS, are already considering the possibility of allowing gradual upgrades of the platform and its apps (including rolling them back), no clear consensus exists in the community about support and updates of software. Thus, a telco willing to install an SDN scenario should hire its own experts and support for the deployment, which increases the costs considerably.

Updates are specifically challenging in the case of big networks, which are constantly changing in response to increasing traffic demand and deploying new services (thus, upgrading the network on a daily basis) [5]. The requirements for these updates include:

1) *Updating network elements consistently:* Control and data plane need to be synchronized. This provides consistency at the cost of a convergence time to reach a common status.
2) *Verifying large control plane updates:* A test procedure should be defined for automatic verification before and after onboarding new functionality.
3) *Being incremental:* Smaller and frequent updates are usually safer and easier to verify using automated tools, resulting in more robust systems [5].
4) *Preserving availability vs. Security:* Decisions need to be made if some inconsistency is found in the network. Should we turn off network elements as soon as an inconsistency/error is found, or should we degrade them gracefully? Should other adjoining components follow the same instructions?

Classical software methods such as rollback operations or revision control could be applied in the context of SDN. However, SDN not only requires a repository to track and maintain the copy of the network, but it would probably need certification authorities to avoid security risks, particularly if the network is medium or large size.

Finally, other aspects to consider involve: detecting malfunctioning parts of the network (*software aging* also affects SDN), or determining when/where an update should be applied (e.g. for non-critical updates, users might determine whether they want to apply changes now or later).

III. Writing an SDN app

If none of the existing SDN apps accomplish what a network administrator requires, they will be developed from scratch. For this task, several aspects have to be taken into account, namely: the SDN platform as well as its NBI, the SBI protocol deployed, the shape and size of the network, the network devices to be controlled by the SDN app and other apps already running in the network.

The **SDN platform** defines the NBI and thus the language to develop the app (Java in the case of ODL or ONOS, Python for Ryu, etc.)[1]. This is a first constraint and it also determines the interfaces to access the network devices, which takes us to the next point: the **SBI protocol** (OpenFlow, NETCONF, etc.). The SDN platform creates an abstraction so that the developer writing code for the NBI avoids handling SBI specificities, but in the end they are still tightly related. For example, writing an application in Ryu not only depends on the SBI protocol, but also on its version; other platforms such as ODL or ONOS bypass that version dependency, but still designate a driver and interface per SBI protocol.

The **shape and size of the network** also affect how an app will be written; the shape determines how the app handles events (for example, if the network has a loop, broadcasting messages should be avoided) and the size decides whether the app should be partitioned in modules (to guarantee scalability, for instance). Finally, the developer needs to know **which devices will be controlled by the app and what other apps are running in the network** to avoid conflicting rules and misbehaviors.

The direct consequence of these factors is that network managers now need to be both expert software developers and networking engineers [6], which causes SDN expertise to be in vogue. In practice, this represents an obstacle for network operators, who demand the advantages of SDN without the complexity of looking for such concrete profiles. This constraint could be solved by following the second and third requirements of HDN: **constructing complex behaviors automatically** (i.e. no need for expert networking engineering) and **speaking and understanding a human-like language** (i.e. no need for software developers). The next sections describe the current state of the art and missing items to accomplish each requirement, respectively.

A. Composing applications

Most of SDN platforms define some criterion for composing application modules when more than one is running. For example, the Floodlight controller executes apps in a specific order (determined previously to deployment) and each application decides whether to pass the event to the next module or drop it, thus stopping the global execution associated to the event. Other platforms such as ODL or ONOS also employ priority levels. Therefore, modules with higher priority are always executed first and there is no chance of modules competing at the same level or changing priority levels depending on the event type. Consequently, some behaviors cannot be characterized due to lack of flexibility at the SDN controller, while users

[1]Language-independent APIs, such as REST or AMQP, are currently supported by diverse SDN platforms. However they are supplementary to the core APIs of the platform and usually leveraged for asynchronous behavior, as strictly using them alone would involve additional TCP connections, thus deteriorating performance

demand some common semantics for composing applications at the NBI.

One of the first semantics to tackle this problem was defined in CoVisor [7], as parallel and sequential composition sentences. For instance a monitor and a forwarding app might run in parallel, while a firewall and a router are executed sequentially, as the firewall filters the traffic first (see Fig. 3). Other approaches based on parallel and sequential semantics are NetIDE (which extends CoVisor to support more SBI protocols) or FlowBricks (which requires modifying OpenFlow). Finally, specific architectures for composition are: Redactor (which uses declarative languages), Statesman or Corybantic (which require a custom interface for network modules). Later on, proposals based on graphs such as PGA [8] consider that the parallel and sequential operators are unable to express some composite behaviors and propose graphs to enrich the composition capabilities.

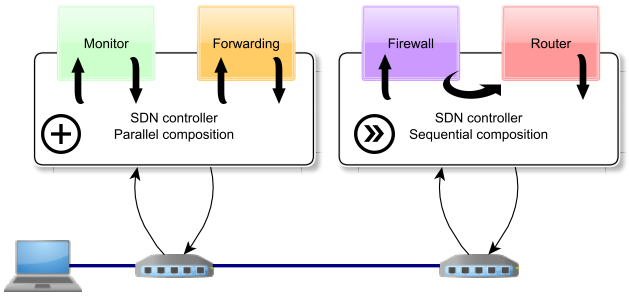

Fig. 3. Examples of composition: parallel (left) and sequential (right)

A particularly challenging part of composing applications are *vetoing* actions. Composing apps in parallel, in sequence or in graphs comes quite naturally chaining the outputs, but how to response when two or more apps are in conflict and specifically when one app does not want to allow some action whatsoever? How can we define this hierarchy and how can we avoid loops in actions (for example, one app installs a flow and another vetoes it and deletes it, and then the first app installs it again)? These questions still remain unanswered.

To achieve this, the first step would be analyzing what network functionality will be the foundations of any SDN deployment. Future research directions should investigate the set of actions (*add*, *modify*, *delete*, *avoid*, etc.) and operators (*linear merging*, *graphs*, *diagrams*, etc.), to express it.

B. Intent-Based Networking

As already mentioned, one of the big deals to write an app is that the NBI is still maturing. Consequently, each SDN platform ends up designing its own interface according to its own requirements, complicating the idea of a generalized method for app development.

One of the first attempts to create a higher-level abstraction for the NBI was Pyretic (currently deprecated and absorbed into the Frenetic SDN controller), by defining the concept of *network policies* and a set of operators to combine them that later inspired CoVisor. However, it was restricted to OpenFlow 1.0 and the POX controller, it required an additional TCP connection and its semantic was still quite dependent on the SBI protocol.

Later on, the concept of Intent-Based Networking (IBN) appeared to standardize the NBI with a unified language common to all SDN platforms. Thanks to IBN, developers write their apps based on *what* they want to do in their network instead of *how*. IBN provides a higher abstraction of the network, so that developers do not necessarily need to know concrete details of the deployment. To accomplish the HDN criteria, IBN should procure the capabilities of inserting a *shapeless* app in the network. A *shapeless* app does not require to match specific network devices or network layers and it only defines the logic independently from the hardware, and thus, independently from the SBI protocol. These shapeless apps could be combined by the network administrator as functional blocks. For example, they would be defined as *I want these two hosts to communicate, just for web services*, instead of *I want to install these TCP flow rules in these switches, to filter web traffic between hosts*.

Nevertheless, current proposals for IBN languages, such as NeMo or the ONOS intent framework, are unable to accomplish this abstraction since they still rely on commands related with the TCP/IP stack, while we require a human-readable language independent from the network characteristics. Other approaches provide abstractions for the NBI without reaching the level of a human-like language, such as: KnowNet (which implements the NBI as a knowledge graph), Gavel (which represents networks as graph databases) or Ravel (which is considered a Database-Defined Network).

This ideal or *pure* IBN is still under research, being it critical for the development of future apps, especially now that new scenarios come into the SDN spectrum, such as the Internet of Things, SmartHome environments or 5G networks. Some open questions include: how to specify *intents*, how to deploy the network based on them, or how to reconcile *intents* and the network elements [5]. Intent-Driven Networking (IDN) [9] proposes a roadmap toward IBN without rewriting the entire network stack and providing backward compatibility (non-IBN behaviour).

IV. Approaching OpenFlow+

When the SDN term appeared, many people could not distinguish between OpenFlow and SDN, mainly because OpenFlow was –and still is– the core SBI protocol, contributing to the arrival of the SDN paradigm. OpenFlow cracked the network into data and control planes, demarcating one before and one after, but still lacks many desirable properties for a completely functional SDN concept. To accomplish the fourth requirement of HDN, the SBI protocol should provide the capability of **disengaging the controller from the physical network**. Thus, research should focus on how to communicate hardware and software without creating a dependency; for instance, the protocol should not depend on how the hardware implements flow tables. Particularly, this requirement smooths network management, as the person operating the network will only have to care about the software and not the underlying hardware anymore.

A. Why OpenFlow 1.X is not enough

OpenFlow versions are mainly extensions of OpenFlow 1.0, adding extra fields for messages and flow matching, plus fixes for edge cases. In fact, the most popular SDN platforms just support OpenFlow 1.0 and 1.3, and do not plan to integrate more versions in the near future. The reason behind is that OpenFlow 1.0 was the first release and 1.3 was the version most supported by vendors, primarily due to the important features provided for data center and campus networking: IPv6, Quality of Service (QoS), and Q-in-Q tunneling. The main problem with OpenFlow is that new specifications were developed at high rate, while the hardware development cycle rate was much slower. Thus vendors constrained the evolution of the protocol, keeping at the same time the stranglehold of switching hardware.

For instance, a missing must-have in OpenFlow are transactional operations. Transactions would allow consistency along the network, especially considering a state change triggered by the controller might take a prolonged period of time to be effective or it might fail in particular nodes. Different studies show that the lack of synchronization of rule updates across network devices not only cause temporary inconsistencies, but also lead to packet losses [10]. Transactional operations were in the table since the beginning of the protocol, but were not developed in the end due to their complexity. Some proposals suggest how transactions could be a reality, what characteristics the network should have and why a global solution to the problem is complicated [11], [12].

B. Alternatives to OpenFlow

Programming Protocol-Independent Packet Processors (P4) [13] proposes the forwarding plane processes packets disregarding implementation details. It relies on a compiler that will later transform the program to be mapped in a target switch.

Protocol Oblivious Forwarding (POF) suggests how the primitive instruction set should be. A POF forwarding element does not understand the standard packet format and flow table search keys are defined as $\{offset, length\}$ tuples.

As a consequence, and based on the protocol-independent forwarding nature of P4 and POF, the ONF created the protocol independent layer for OpenFlow (OF-PI), a promising but still immature project.

We believe a first step for a true disassociation from the network elements would be questioning the match/action tuple abstraction used in SBI protocols. Thus, reimagining how network functionality could be expressed in alternative and more sophisticated ways.

V. Unboxing the network boxes

Apart from the SBI protocol, the type of devices that shape the network are also critical to accomplish the fourth requirement of HDN. Adding features in the hardware is directly translated into more control possibilities for the network manager. In this section, we envision the following research directions: how white boxes should be ideally designed, and why network devices should not be totally dumb, applied both to forwarding and end devices.

A. White boxes

In traditional networks, telcos and their network administrators work with *black boxes* of functionality. These boxes are designed and mantained by manufacturer companies, and thus closed to modification by their clients. Usually, these boxes provide the required services plus some extras to increase market competitiveness. Therefore, in practice many telcos end up with duplicated services and they can only manage proprietary interfaces and parameters. For example, it might happen that a telco buys two routers from two different providers and they both offer a firewall service as an extra, thus obtaining a duplicated service even when it was not requested. Furthermore, just a few companies produce the majority of the telecommunication hardware, so these devices usually have high prices.

Thanks to OpenFlow, vendors were forced to open their interfaces allowing fine-grained control over their boxes. However, these *gray boxes* were still limited by the programmability of the existing hardware technology. As we mentioned in the past section, network administrators could now only implement what the device supported. In order to allow full programmability and control of the network, pure *white boxes* are required. These boxes would use generic network programming models, as described in previous sections.

Additionally, although the white boxes concept could be achieved theoretically, further research should be carried out in terms of performance and network features support for commercialization. Currently, OpenFlow switches (and even software switches such as Open vSwitch or CPqD's ofsoftswitch13) differ from one vendor/implementation to other. Thus, this variability should be tackled for an ideal white box concept.

B. Dummy boxes

Another school of thought believes that not all the complexity should go to the controller, as it makes the network too dependent on controller implementations, which is not a balanced approach. Smarter boxes could fit in different SDN scenarios, making the global solution far superior to others that base their intelligence just in the controller. Likewise, allowing these alternative ways of allocating the SDN intelligence would extricate creative networking.

In DevoFlow [14], the authors question the costs of involving the controller frequently and propose switches only target specific flows, reducing the communication between control and data plane. They conclude betting that the dummy box approach of OpenFlow will not meet the requirements of high performance networks.

OpenState [15] outlines the concept of a *stateful* data plane, which would be beneficial to offload controllers from decisions based on local knowledge. They define an extension of the OpenFlow match/action abstraction and prove it with a hardware implementation based on FPGA boards. Some approaches also believe in an updated architecture in which network devices implement parts of the SDN apps directly in the data plane.

C. Dummy hosts

End devices, currently secluded from the SDN ecosystem, could also be part of it. One of the main reason to keep them out of the scope is their disparate nature: laptops, mobile phones, IoT chips, etc. Nevertheless they keep implementing classic protocols and the TCP/IP stack, which hampers SDN deployments: a fact that should not be omitted.

For example, an SDN controller might prepare the routes between a couple of hosts, but this does not avoid Address Resolution Protocol (ARP) or Neighbor Discovery (ND) messages, generated at the host previously to any communication. The solution to this consists in implementing a proxy service for ARP and ND requests, in which the edge nodes (the first neighbors of the end devices controlled via SDN) contact the controller to resolve those petitions. The fact is that the controller might be aware of the hosts and all their information from the start, but as the hosts are isolated from this knowledge, they will still request the information as usual.

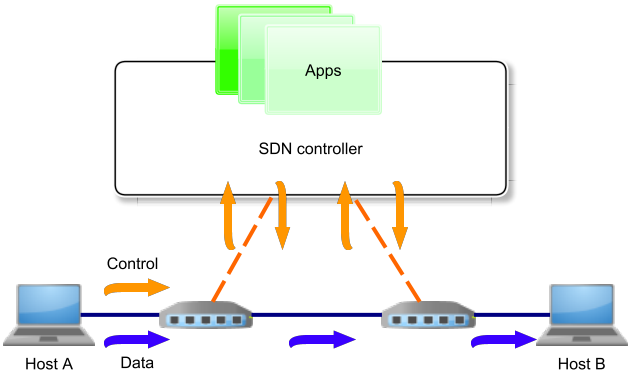

Fig. 4. Host A sends data packets to Host B and, at the same time, control packets to the edge switch (in-band SDN communication)

The main handicap of including intelligence at host nodes is they belong to miscellaneous types of hardware and execute a wide range of software (operating systems). However, ongoing research believes in moving network functions to the edge to offload the core and running a virtual network appliance at end devices (e.g. smartphone) could be seen as feasible in the near future. Modifying the hosts to support SDN might seem unrealistic, but one could think of a device running an OpenFlow agent as it can execute an SSH client. This agent could then exchange knowledge with the controller via in-band communication (not requiring an out-of-band specific network interface for the control plane), as depicted in Fig. 4.
In any case, although many may presume redefining the hosts as part of the SDN paradigm is not necessary, the discussion is still open.

VI. Conclusion

The SDN paradigm is flourishing rapidly, but so far the focus remains on replicating most of the functionalities that traditional networks implemented before. Network operators demand an open network with effortless management and reduced cost, and expect SDN to accomplish all of it. In order to assist SDN in accomplishing these requirements, we define the concept of HDN, followed by four principles.

Along the article, we examine the state of the art and present what we consider challenges (and thus opportunities) for the forthcoming research in the SDN area. Proceeding up to down in the SDN architecture, we explore the concept of SDN app stores, the development of SDN apps independent of the SBI restrictions or specific languages, the evolution of SBI protocols and the implementation of network devices.

The main conclusion draws that SDN is still in an early stage, and hence profuse research and standardization effort needs to be carried out for a smooth transition from traditional networks. Specifically, lessons learned are:

- To allow deployment flexibility in the network, the concept of *app module* or *app package* needs to be defined and generalized for different SDN frameworks. Afterward, an *app store* could be created accordingly to fulfill the needs of operators, who would simply *drag and drop* these services into their SDN deployments.
- To guarantee the growth of the *app store*, development of apps needs to be easy and straightforward. Thus, the NBI needs research efforts toward a human-like language interface, which also should permit effortless composition of different *app packages*. This would drastically decrease costs for network operators.
- To provide independence of the NBI and the SBI, the SBI should evolve to support truly open and generalized protocol(s). Currently, OpenFlow is not enough and some initiatives such as P4 are appearing. Moreover, network devices could refine the paradigm by adding some SDN intelligence in them or possibly including end devices as part of SDN deployments, which needs further study to prove pros and cons.

It is worth noting that research directions toward HDN might affect other aspects of SDN, such as security. For instance, providing a higher abstraction to the NBI should not imply loosening control over the network. Also, third parties might be required for certification of SDN apps if the app store becomes a reality. Finally, including end hosts in the SDN arena might create security risks to forwarding devices, which could be solved defining a hierarchy or isolating parts of the SDN from others.

Above all, we hope this article serves as a summary and inspirational tool for future research in the SDN area.

Acknowledgment

This work is partially supported by the EC FP7 NetIDE project, G.A. 619543 and by grants from Comunidad de Madrid through Project TIGRE5-CM (S2013/ICE-2919).

References

[1] D. Kreutz, F. M. V. Ramos, P. E. Verssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.
[2] S. H. Yeganeh, A. Tootoonchian, and Y. Ganjali, "On scalability of software-defined networking," *IEEE Communications Magazine*, vol. 51, no. 2, pp. 136–141, February 2013.

[3] M. C. Dacier, H. Knig, R. Cwalinski, F. Kargl, and S. Dietrich, "Security Challenges and Opportunities of Software-Defined Networking," *IEEE Security Privacy*, vol. 15, no. 2, pp. 96–100, 2017.

[4] J. A. Wickboldt, W. P. D. Jesus, P. H. Isolani, C. B. Both, J. Rochol, and L. Z. Granville, "Software-defined networking: management requirements and challenges," *IEEE Communications Magazine*, vol. 53, no. 1, pp. 278–285, January 2015.

[5] R. Govindan, I. Minei, M. Kallahalla, B. Koley, and A. Vahdat, "Evolve or Die: High-Availability Design Principles Drawn from Google's Network Infrastructure," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16, 2016, pp. 58–72.

[6] D. M. Batista, G. Blair, F. Kon, R. Boutaba, D. Hutchison, R. Jain, R. Ramjee, and C. E. Rothenberg, "Perspectives on software-defined networks: interviews with five leading scientists from the networking community," *Journal of Internet Services and Applications*, vol. 6, no. 1, p. 22, 2015.

[7] X. Jin, J. Gossels, J. Rexford, and D. Walker, "CoVisor: A Compositional Hypervisor for Software-defined Networks," in *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'15. Berkeley, CA, USA: USENIX Association, 2015, pp. 87–101. [Online]. Available: http://dl.acm.org/citation.cfm?id=2789770.2789777

[8] C. Prakash, J. Lee, Y. Turner, J.-M. Kang, A. Akella, S. Banerjee, C. Clark, Y. Ma, P. Sharma, and Y. Zhang, "PGA: Using Graphs to Express and Automatically Reconcile Network Policies," *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 29–42, Aug. 2015. [Online]. Available: http://doi.acm.org/10.1145/2829988.2787506

[9] Y. Elkhatib, G. Tyson, and G. Coulson, "Charting an Intent Driven Network," *CoRR*, vol. abs/1604.05925, 2016. [Online]. Available: http://arxiv.org/abs/1604.05925

[10] T. Truong, Q. Fu, and C. Lorier, "FlowMap: Improving network management with SDN," in *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*, April 2016, pp. 821–824.

[11] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid, "A distributed and robust SDN control plane for transactional network updates," in *2015 IEEE Conference on Computer Communications (INFOCOM)*, April 2015, pp. 190–198.

[12] S. Dudycz, A. Ludwig, and S. Schmid, "Can't Touch This: Consistent Network Updates for Multiple Policies," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2016, pp. 133–143.

[13] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming Protocol-independent Packet Processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014. [Online]. Available: http://doi.acm.org/10.1145/2656877.2656890

[14] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling Flow Management for High-performance Networks," *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 254–265, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2043164.2018466

[15] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "OpenState: Programming Platform-independent Stateful Openflow Applications Inside the Switch," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, pp. 44–51, Apr. 2014. [Online]. Available: http://doi.acm.org/10.1145/2602204.2602211