1

Toward a Decentralized SDN Control Architecture: Overview and Taxonomy

1.1. Introduction

In contrast to the decentralized control logic that underpins the construction of the Internet as a complex bundle of box-centric protocols and vertically integrated solutions, the software-defined networking (SDN) paradigm advocates the separation of the control logic from hardware and its centralization in software-based controllers. These key tenets offer new opportunities to introduce innovative applications and incorporate automatic and adaptive control aspects, thereby easing network management and guaranteeing the user's QoE.

However, despite the interest surrounding SDN, adoption raises many challenges, including the scalability and reliability issues of centralized designs that can be addressed with the physical decentralization of the control plane. However, such physically distributed but logically centralized systems bring an additional set of challenges.

This chapter presents a survey on SDN with a special focus on distributed SDN control. In section 1.2, we start by describing the promises and solutions offered by SDN compared to conventional networking. We also elaborate on the fundamental elements of the SDN architecture.

Then, we expand our knowledge of the different approaches to SDN by exploring the wide variety of existing SDN controller platforms. In doing so, we intend to place a special emphasis on distributed SDN solutions and classify them in two different ways. In section 1.3, we propose a physical classification of state-of-the-art SDN control plane architectures into centralized and distributed (flat or hierarchical) in order to highlight the SDN performance, scalability and reliability challenges. In

section 1.4, we put forward a logical classification of distributed SDN control plane architectures, logically centralized and logically distributed, while tackling the associated state consistency and knowledge dissemination issues.

1.2. Software-defined networking: a centralized control architecture

1.2.1. *Conventional networking and the SDN paradigm*

Over the past few years, the need for a new approach to networking has been expressed to overcome the many issues associated with current networks. In particular, the main vision of the SDN approach is to simplify networking operations, optimize network management and introduce innovation and flexibility compared to legacy networking architectures.

In this context, and in line with the vision of Kim and Feamster (2013), four key reasons for the problems encountered in the management of existing networks can be identified:

1) *Complex and low-level network configuration*: network configuration is a complex distributed task in which each device is typically configured in a low-level vendor-specific manner. Additionally, the rapid growth of the network, together with the changing networking conditions, have resulted in network operators constantly performing manual changes to network configurations, thereby compounding the complexity of the configuration process and introducing additional configuration errors.

2) *Dynamic network state*: networks are growing dramatically in size, complexity and consequently in dynamicity. Furthermore, with the rise of mobile computing trends as well as the advent of network virtualization (Bari et al. 2013; Alam et al. 2020) and cloud computing (Zhang et al. 2010; Sharkh et al. 2013; Shamshirband et al. 2020), the networking environment becomes even more dynamic as hosts are continually moving, arriving and departing due to the flexibility offered by VM migration, thus making traffic patterns and network conditions change in a more rapid and significant way.

3) *Exposed complexity*: in today's large-scale networks, network management tasks are challenged by the significant complexity presented by distributed low-level network configuration interfaces. This complexity is mainly generated by the tight coupling between the management, control and data planes, where many control and management features are implemented in hardware.

4) *Heterogeneous network devices*: current networks consist of a large number of heterogeneous network devices including routers, switches and a wide variety of specialized middle-boxes. Each of these appliances has its own proprietary configuration tools and operates according to specific protocols, therefore increasing both complexity and inefficiency in network management.

As a result, network management is becoming more difficult and challenging given that the static and inflexible architecture of legacy networks is ill-suited to cope with today's increasingly dynamic networking trends, and to meet the QoE requirements of modern users. This has fueled the need for the enforcement of complex, high-level policies to adapt to current networking environments, and for the automation of network operations to reduce the tedious workload of low-level device configuration tasks.

In this sense, and to deliver the goals of easing network management in real networks, operators considered running dynamic scripts as a way to automate network configuration settings before realizing the limitations of such approaches, which led to misconfiguration issues. It is worth noting, however, that recent approaches to scripting configurations and network automation are becoming relevant (e.g. Ansible).

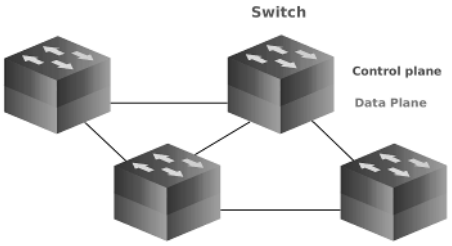
The SDN initiative led by the Open Networking Foundation (ONF), on the other hand, proposes a new open architecture to address current networking challenges with the potential to facilitate the automation of network configurations and, better yet, fully program the network. Unlike the conventional distributed network architecture (Figure 1.1(a)) in which network devices are closed and vertically integrated, bundling software with hardware, the SDN architecture (Figure 1.1(b)) raises the level of abstraction by separating the network data and control planes. In this way, network devices become simple forwarding switches; all the control logic is centralized in software controllers, providing a flexible programming framework for the development of specialized applications and deployment of new services.

Such aspects of SDN are believed to simplify and improve network management by offering the possibility to innovate, customize behaviors and control the network according to high-level policies expressed as centralized programs. This therefore bypasses the complexity of low-level network details and overcomes the fundamental architectural problems raised in points 1) and 3). Added to these features is the ability of SDN to easily cope with the heterogeneity of the underlying infrastructure (outlined in point 4)) thanks to the SDN southbound interface abstraction.

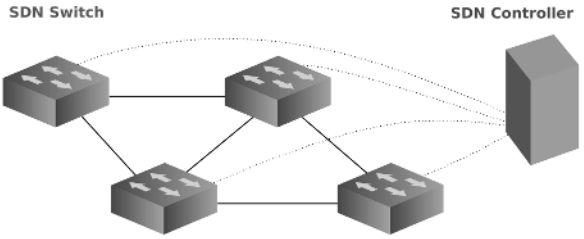
More detailed information on the SDN-based architecture, which is split vertically into three layers (see Figure 1.2), is provided in the next section.

1.2.2. *The SDN architecture*

The SDN-based architecture is split vertically into three layers (see Figure 1.2). Detailed information about the SDN architecture is provided in the sections that follow.



a) Traditional architecture



b) SDN architecture

Figure 1.1. *Conventional networking versus software-defined networking. For a color version of this figure, see www.iste.co.uk/bannour/software.zip*

1.2.2.1. *SDN data plane*

The data plane, also known as the forwarding plane, consists of a distributed set of forwarding network elements (mainly *switches*) in charge of forwarding packets. In the context of SDN, the control-to-data plane separation feature requires the data plane to be remotely accessible for software-based control via an open vendor-agnostic southbound interface.

Both OpenFlow (McKeown et al. 2008; Costa et al. 2021) and ForCES (Forwarding and Control Element Separation) (Doria et al. 2010; Anerousis et al. 2021) are well-known candidate protocols for the southbound interface. They both follow the basic principle of splitting the control plane and the forwarding plane in network elements, and they both standardize the communication between the two planes. However, these solutions are different in many aspects, especially in terms of network architecture design.

Standardized by IETF, ForCES (Doria et al. 2010; Anerousis et al. 2021) introduced separation between the control plane and the forwarding plane. In doing so, ForCES defines two logic entities that are logically kept in the same physical device: the control element (CE) and the forwarding element (FE). However, despite

being a mature standard solution, the ForCES alternative did not gain widespread adoption by major router vendors.

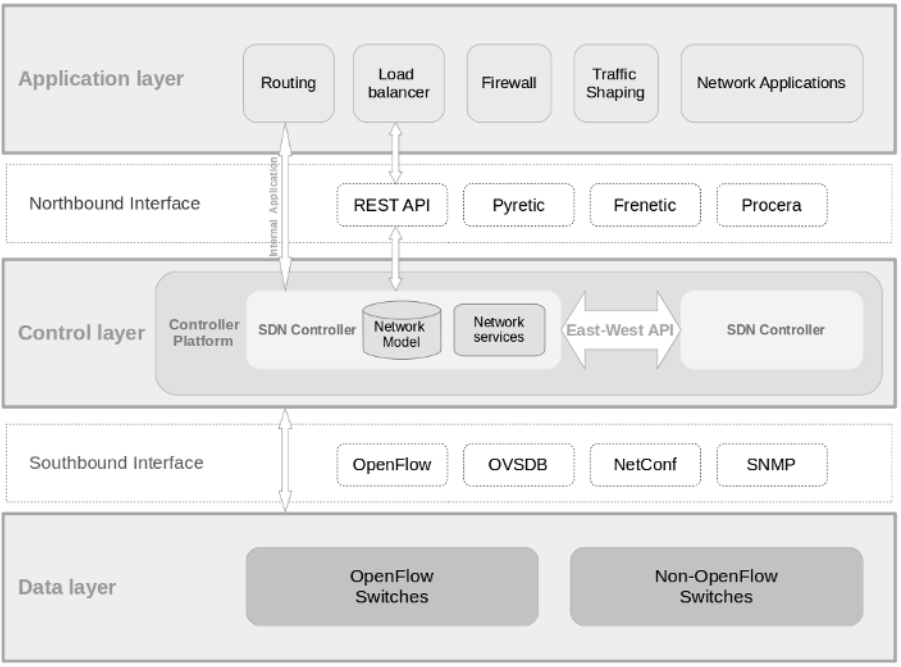


Figure 1.2. *A three-layer distributed SDN architecture. For a color version of this figure, see www.iste.co.uk/bannour/software.zip*

On the other hand, OpenFlow (McKeown et al. 2008; Costa et al. 2021) received major attention in both the research community and industry. Standardized by the ONF, it is considered the first widely accepted communication protocol for the SDN southbound interface.

OpenFlow enables the control plane to specify, in a centralized way, the desired forwarding behavior of the data plane. Such traffic forwarding decisions reflect the specified network control policies and are translated by *controllers* into actual packet forwarding rules, populated in the flow tables of OpenFlow *switches*.

In more specific terms, and according to the original version 1.0.0 of the standard defined by Open Networking Foundation (2009), an OpenFlow-enabled Switch consists of a *flow table* and an OpenFlow *secure channel* to an external OpenFlow controller. Typically, the forwarding table maintains a list of flow entries. Each flow

entry comprises *match fields* containing header values to match packets against, *counters* to update when packets match for flow statistics collection purposes and a set of *actions* to apply to matching packets.

Accordingly, all incoming packets processed by the switch are compared against the flow table, where flow entries match packets based on a priority order specified by the controller. In case a matching entry is found, the flow counter is incremented, and the actions associated with the specific flow entry are performed on the incoming packet belonging to that flow. According to the OpenFlow specification (Open Networking Foundation 2009), these actions may include forwarding a packet out on a specific port, dropping the packet, removing or updating packet headers, etc. If no match is found in the flow table, then the unmatched packet is encapsulated and sent over the secure channel to the controller, which decides how it should be processed. Among other possible actions, the controller may define a new flow for that packet by inserting new flow table entries.

1.2.2.2. *SDN control plane*

Regarded as the most fundamental building entity in the SDN architecture, the control plane consists of a centralized software controller that is responsible for handling communications between network applications and devices through open interfaces. More specifically, SDN controllers translate the requirements of the application layer down to the underlying data plane elements and give relevant information up to SDN applications.

The SDN control layer is commonly referred to as the network operating system (NOS) as it supports the network control logic and provides the application layer with an abstracted view of the global network, which contains enough information to specify policies while hiding all implementation details.

Typically, the control plane is logically centralized and yet implemented as a physically distributed system for scalability and reliability reasons, as discussed in sections 1.3 and 1.4. In a distributed SDN control configuration, east-westbound application programming interfaces (APIs) (Lin et al. 2015; Almadani et al. 2021) are required to enable multiple SDN controllers to communicate with each other and to exchange network information.

Despite the many attempts to standardize SDN protocols, there has been to date no standard for the east-west API, which remains proprietary for each controller vendor. Although a number of east-westbound communications occur only at the data store level and do not require additional protocol specifics, it is becoming increasingly advisable to standardize that communication interface in order to provide wider interoperability between different controller technologies in different autonomous SDN networks.

On the other hand, an east-westbound API standard requires advanced data distribution mechanisms and involves other special considerations. This brings about additional SDN challenges, some of which have been raised by the state-of-the art distributed controller platforms discussed in sections 1.3 and 1.4 but have yet to be fully addressed.

1.2.2.3. *SDN application plane*

The SDN application plane comprises SDN applications, which are control programs designed to implement the network control logic and strategies. This higher level plane interacts with the control plane via an open northbound API. In doing so, SDN applications communicate their network requirements to the SDN controller, which translates them into southbound-specific commands and forwarding rules dictating the behavior of the individual data plane devices. Routing, traffic engineering (TE), firewalls and load balancing are typical examples of common SDN applications running on top of existing controller platforms.

In the context of SDN, applications leverage the decoupling of the application logic from the network hardware along with the logical centralization of the network control to directly express the desired goals and policies in a centralized, high-level manner without being tied to the implementation and state-distribution details of the underlying networking infrastructure. Concurrently, SDN applications make use of the abstracted network view exposed through the northbound interface to consume the network services and functions provided by the control plane, according to their specific purposes.

That being said, the northbound API implemented by SDN controllers can be regarded as a network abstraction interface to applications, easing network programmability, simplifying control and management tasks and allowing for innovation. In contrast to the southbound API, the northbound API is not supported by an accepted standard.

Despite the broad variety of northbound APIs adopted by the SDN community (see Figure 1.2), we can classify them into two main categories.

– The first set involves simple and primitive APIs that are directly linked to the internal services of the controller platform. These implementations include:

- low-level ad hoc APIs that are proprietary and tightly dependent on the controller platform. Such APIs are not considered high-level abstractions as they allow developers to directly implement applications within the controller in a low-level manner. Deployed internally, these applications are tightly coupled with the controller and written in its native general-purpose language. NOX in C++ and POX in Python are typical examples of controllers that use their own basic sets of APIs;

- APIs based on Web services, such as the widely used REST API. This group of programming interfaces enables independent external applications (*clients*)

to access the functions and services of the SDN controller (*server*). These applications can be written in any programming language and are not run inside the bundle hosting the controller software. Floodlight[1] is an example of an SDN controller that adopts an embedded northbound API based on REST.

– The second category contains higher level APIs that rely on domain-specific programming languages such as Frenetic (Foster et al. 2011; Kulkarni et al. 2021), Procera (Voellmy et al. 2012) and Pyretic (Monsanto et al. 2013; Kulkarni et al. 2021) as an indirect way for applications to interact with the controller. These APIs are designed to raise the level of abstraction in order to allow for the flexible development of applications and the specification of high-level network policies.

1.3. Physical classification of existing SDN control plane architectures

Despite the undeniable strengths of SDN, there have always been serious concerns about the ability to extend SDN to large-scale networks.

Some argue that these scalability limits are in effect linked to the protocol standards being used for the implementation of SDN. OpenFlow (McKeown et al. 2008; Costa et al. 2021) in particular, although recognized as a leading and widely deployed SDN southbound technology, is currently being rethought for potentially causing excessive overheads on switches (*switch bottleneck*). Scalable alternatives to the OpenFlow standard, which propose to revisit the delegation of control between the controller and the switches with the aim of reducing the reliance on SDN control plane, have been discussed in section 1.2.2.1.

Another entirely different approach to addressing the SDN scalability and reliability challenges – advocated here – is to physically distribute the SDN control plane. This has led to a first categorization of existing controller platforms into centralized and distributed architectures (see Figure 1.3). Please note that in Figures 1.3 and 1.4 controllers that present similar characteristics for the discussed comparison criteria are depicted in the same color.

1.3.1. *Physically centralized SDN control*

A physically centralized control plane consisting of a single controller for the entire network is a theoretically perfect design choice in terms of simplicity. However, a single controller system may not keep up with the growth of the network. It is likely to become overwhelmed (*controller bottleneck*) when dealing with an increasing number of requests while simultaneously struggling to achieve the same performance guarantees.

1 https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/overview.

A centralized SDN controller evidently does not meet the different requirements of large-scale, real-world network deployments. Data centers and service provider networks are typical examples of such large-scale networks, presenting different requirements in terms of scalability and reliability.

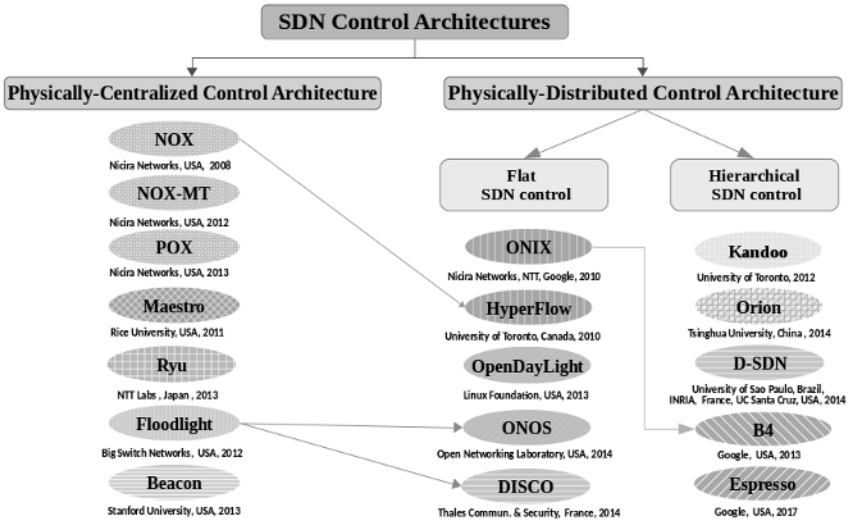


Figure 1.3. *Physical classification of SDN control plane architectures. For a color version of this figure, see www.iste.co.uk/bannour/software.zip*

More specifically, a *data center network* involves tens of thousands of switching elements. Such a large number of forwarding elements, which can grow at a fast pace, is expected to generate a huge number of control events that are enough to overload a single centralized SDN controller (Yeganeh et al. 2013; Azodolmolky et al. 2013). Studies conducted in Benson et al. (2010) show important scalability implications (in terms of throughput) for centralized controller approaches. They demonstrate that multiple controllers should be used to scale the throughput of a centralized controller and meet the traffic characteristics within realistic data centers.

Unlike data centers, *service provider networks* are characterized by a modest number of network nodes. However, these nodes are usually geographically distributed, making the diameter of these networks very large (Yeganeh et al. 2013). This entails different types of controller scalability issues for centralized controller approaches, more specifically, high latencies. In addition to latency requirements, service provider networks have large numbers of flows that may generate overhead and bandwidth issues.

In general, wide area network (WAN) deployments typically impose strict resiliency requirements. In addition, they present higher propagation delays compared to data center networks. Obviously, a centralized controller design in an SD-WAN cannot achieve the desired failure resiliency and scale-out behaviors (Michel and Keller 2017; Sarmiento et al. 2021). Several studies have emphasized the need for a distributed control plane in an SD-WAN architecture. Indeed, they are focused on placing multiple controllers on real WAN topologies to benefit both control plane latency and fault tolerance (Heller et al. 2012; Ul Huque et al. 2017; Sminesh et al. 2019).

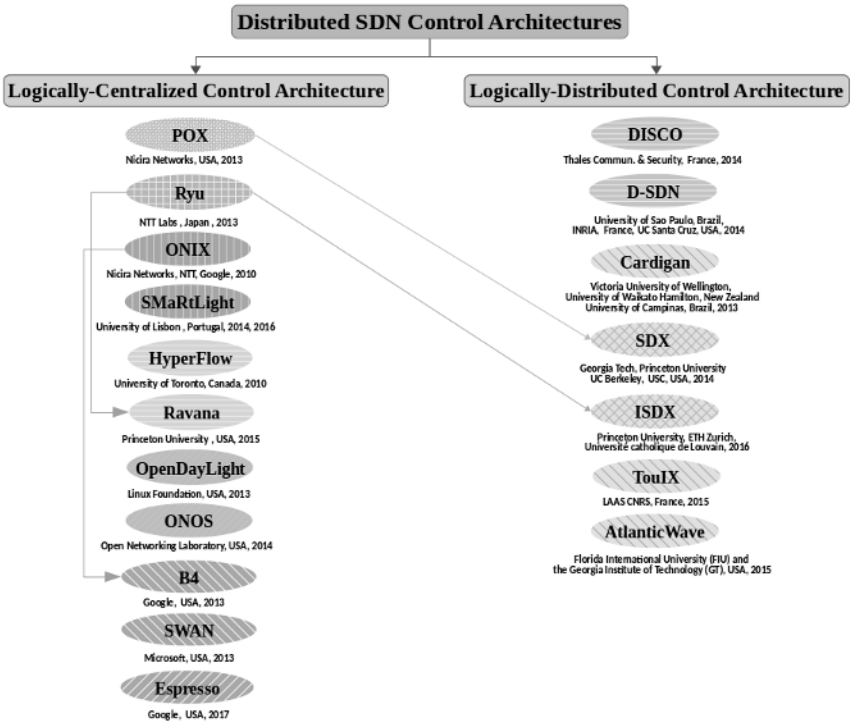


Figure 1.4. *Logical classification of distributed SDN control plane architectures. For a color version of this figure, see www.iste.co.uk/bannour/software.zip*

That said, the potential scalability, reliability and vulnerability concerns associated with centralized controller approaches have been further confirmed through studies (Shalimov et al. 2013; Karakus and Durresi 2017) on the behavior of state-of-the-art centralized SDN controllers such as NOX (Gude et al. 2008), Beacon (Erickson 2013) and Floodlight, within different networking environments.

In particular, NOX classic (Gude et al. 2008), the world's first-generation OpenFlow controller with an event-based programming model, is believed to be limited in terms of throughput. Indeed, it cannot handle a large number of flows, namely, a rate of 30k flow initiation events per second (Tavakoli et al. 2009; Karakus and Durresi 2017). Such a flow setup throughput may seem sufficient for an enterprise network, but it could be arguable for data-center deployments with high flow initiation rates (Benson et al. 2010). Improved versions of NOX have subsequently been developed by the same community (Nicira Networks), such as NOX-MT (Tootoonchian et al. 2012) for better performance and POX for a more developer-friendly environment.

However, while none of these centralized designs is believed to meet the above scalability and reliability requirements of large-scale networks, they have gained greater prominence as they were widely used for research and educational purposes.

Additionally, Floodlight, which is a very popular Java-based OpenFlow controller from Big Switch Networks, suffers from serious security and resiliency issues. For instance, Dhawan et al. (2015) reported that the centralized SDN controller is inherently susceptible to denial-of-service (DoS) attacks. Another subsequent version of Floodlight, called SE-Floodlight, has therefore been released to overcome these problems by integrating security applications. However, despite the introduced security enhancements aimed at shielding the centralized controller, the latter remains a potential weakness that compromises the whole network. In fact, the controller still maintains a single point of failure and bottlenecks even if its latest version is less vulnerable to malicious attacks.

On the other hand, given its obvious performance and functionality advantages, the open-source Floodlight has been extensively used to build other SDN controller platforms that support distributed architectures such as ONOS (Berde et al. 2014; Ruslan et al. 2020) and DISCO (Phemius et al. 2013).

1.3.2. *Physically distributed SDN control*

Alternatively, physically distributed control plane architectures have received increased research attention in recent years since they appeared as a potential solution to mitigate the issues brought about by centralized SDN architectures (poor scalability, single point of failure (SPOF), performance bottlenecks, etc.). As a result, various SDN control plane designs have been proposed in the recent literature. However, we discern two main categories of distributed SDN control architectures based on the physical organization of SDN controllers: a flat SDN control architecture and a hierarchical SDN control architecture (see Figure 1.3).

1.3.2.1. *Flat SDN control*

The flat structure implies the horizontal partitioning of the network into multiple areas, each of which is handled by a single controller in charge of managing a subset of SDN switches. There are several advantages to organizing controllers in a flat design, including reduced control latency and improved resiliency.

Onix (Koponen et al. 2010), HyperFlow (Tootoonchian and Ganjali 2010) and ONOS (Berde et al. 2014; Ruslan et al. 2020) are typical examples of flat, physically distributed controller platforms, initially designed to improve control plane *scalability* through the use of multiple interconnected controllers sharing a global network-wide view and allowing for the development of centralized control applications. However, each of these contributions takes a different approach to distribute controller states and provide control plane scalability.

Onix, for example, provides good scalability through additional partitioning and aggregation mechanisms. To be more specific, Onix partitions the NIB (Network Information Base), giving each controller instance responsibility for a subset of the NIB; it aggregates by making applications reduce the fidelity of information before sharing it between other Onix instances within the cluster. Similar to Onix, each ONOS instance (composing the cluster) that is responsible for a subset of network devices holds a portion of the network view that is also represented in a graph. In contrast to Onix and ONOS, every controller in HyperFlow has the global network view; thus, obtaining the illusion of control over the whole network. However, HyperFlow can be considered a scalable option for specific policies in which a small number of network events affect the global network state. In that case, scalability is ensured by propagating these (less frequent) selected events through the event propagation system.

Furthermore, different mechanisms are put in place by these distributed controller platforms to meet *fault tolerance* and *reliability* requirements in the event of failures or attacks.

Onix (Koponen et al. 2010) uses different recovery mechanisms depending on the detected failures. An Onix instance failure is predominantly handled by distributed coordination mechanisms among replicas, whereas network element/link failures are under the full responsibility of applications developed atop Onix. In addition, Onix is assumed to be reliable in terms of connectivity infrastructure failures because it can dedicate the failure recovery task to a separate management backbone that uses a multi-pathing protocol.

Similarly, HyperFlow (Tootoonchian and Ganjali 2010) focuses on ensuring resiliency and fault tolerance as a means of achieving availability. When a controller failure is discovered by the failure detection mechanisms deployed by its publish/subscribe WheelFS system (Stribling et al. 2009), HyperFlow reconfigures

the affected switches and redirects them to another nearby controller instance (from a neighbor's site). Alongside this ability to tackle component failures, HyperFlow is resilient to network partitioning thanks to the partition tolerance property of WheelFS. In fact, in the presence of network partitioning, WheelFS partitions continue to operate independently, thus favoring availability.

Similarly, ONOS (Berde et al. 2014; Ruslan et al. 2020) considers fault tolerance to be a prerequisite for adopting SDN in service provider networks. ONOS's distributed control plane guards against controller instance failures by connecting, from the onset, each SDN switch to more than one SDN controller; its master controller and other backup controllers (from other domains) which may take over in the wake of master controller failures. Load balancing mechanisms are also provided to balance the mastership of switches among the controllers of the cluster for scalability purposes. In addition, ONOS incorporates additional recovery protocols, such as the Anti-Entropy protocol (Bianco et al. 2016), for healing from lost updates due to such controller crashes.

More recent SDN controller platform solutions (Chandrasekaran and Benson 2014; Shin et al. 2014; Katta et al. 2015; Yeganeh and Ganjali 2016; Chandrasekaran et al. 2016; Spalla et al. 2016; Mantas and Ramos 2019) focused specifically on improving fault tolerance in the distributed SDN control plane. Some of these works assumed a simplified flat design in which the SDN control was centralized. However, since the main focus was placed on the fault tolerance aspect, we believe that their ideas and their fault tolerance approaches can be leveraged in the context of medium- to large-scale SDNs in which the network control is physically distributed among multiple controllers.

In particular, Botelho et al. (2014) developed a hybrid SDN controller architecture that combines both passive and active replication approaches for achieving control plane fault tolerance. SMaRtLight adopts a simple Floodlight-based multi-controller design following OpenFlow 1.3, where one main controller (the primary) manages all network switches, and other controller replicas monitor the primary controller and serve as backups in case it fails.

This variant of a traditional passive replication system relies on an external data store that is implemented using a modern active replicated state machine (RSM) built with a Paxos-like protocol (BFT-SMaRt (Bessani et al. 2014)) to ensure fault tolerance and strong consistency. This shared data store is used for storing the network and application state (the common global NIB) and also for coordinating fault detection and leader election operations between controller replicas that run a lease management algorithm.

In the case of a failure of the primary controller, the elected backup controller starts reading the current state from the shared consistent data store in order to

mitigate the cold-start (empty state) issue associated with traditional passive replication approaches, and thereby ensure a smoother transition to the new primary controller role.

The limited feasibility of the deployed controller fault tolerance strategy is warranted by the limited scope of the SMaRtLight solution, which is only intended for small- to medium-sized SDN networks. On the other hand, in large-scale deployments, adopting a simplified master–slave approach, and, more importantly, assuming a single main controller scheme in which one controller replica must retrieve all the network state from the shared data store in failure scenarios, have major disadvantages in terms of increased latency and failover time.

Similarly, the Ravana controller platform proposal (Katta et al. 2015) addresses the issue of recovering from complete fail-stop controller crashes. It offers the abstraction of a fault-free centralized SDN controller to unmodified control applications that are relieved of the burden of handling controller failures. Accordingly, network programmers write application programs for a single main controller, and the transparent master–slave Ravana protocol takes care of replicating the control logic – seamlessly and consistently – to other backup controllers for fault tolerance.

The Ravana approach deploys enhanced RSM techniques that are extended with switch-side mechanisms to ensure control messages are processed transactionally with ordered and exactly-once semantics, even in the presence of failures. The three Ravana prototype components, namely the Ryu-based controller (Ryu SDN Framework) runtime, the switch runtime, and the control channel interface, work cooperatively to guarantee the desired correctness and robustness properties of a fault-tolerant logically centralized SDN controller.

More specifically, when the master controller crashes, the Ravana protocol detects the failure within a short failover time and elects the standby slave controller to take over using Zookeeper-like (Hunt et al. 2010) failure detection and leader election mechanisms. The new leader finishes processing any logged events in order to catch up with the failed master controller state. Then, it registers with the affected switches in the role of the new master before proceeding with normal controller operations.

1.3.2.2. *Hierarchical SDN control*

The hierarchical SDN control architecture assumes that the network control plane is vertically partitioned into multiple levels (layers) depending on the required services. According to Liu et al. (2015) and Amiri et al. (2019), a hierarchical organization of the control plane can improve SDN scalability and performance.

To improve *scalability*, Hassas Yeganeh and Ganjali (2012) assumes a hierarchical two-layer control structure that partitions control applications into local

and global. Contrary to DevoFlow (Curtis et al. 2011; Abuarqoub 2020) and DIFANE (Yu et al. 2010; Abuarqoub 2020), Kandoo proposes to reduce the overall stress on the control plane without the need to modify OpenFlow switches. Instead, it establishes a two-level hierarchical control plane, where frequent events occurring near the data path are handled by the bottom layer (local controllers with no interconnection running local applications) and non-local events requiring a network-wide view are handled by the top layer (a logically centralized root controller running non-local applications and managing local controllers).

Despite the obvious scalability advantages of such a control plane configuration, in which local controllers can scale linearly as they do not share information, Kandoo did not envisage *fault tolerance* and resiliency strategies to protect itself from potential failures and attacks in the data and control planes. In addition, from a developer perspective, Kandoo imposes some Kandoo-specific conditions on the control applications developed on top of it in such a way that makes them aware of its existence.

On the other hand, Google's B4 (Jain et al. 2013; Kumar et al. 2015), a private intra-domain software-defined WAN connecting their data centers across the planet, proposes a two-level hierarchical control framework for improving *scalability*. At the lower layer, each data center site is handled by an Onix-based (Koponen et al. 2010) SDN controller hosting local site-level control applications. These site controllers are managed by a global *SDN Gateway* that collects network information from multiple sites through site-level TE services and sends them to a logically centralized *TE server,* which also operates at the upper layer of the control hierarchy. Based on an abstract topology, the latter enforces high-level TE policies that are mainly aimed at optimizing bandwidth allocation between competing applications across the different data center sites. That being said, the TE server programs these forwarding rules at the different sites through the same gateway API. These TE entries will be installed in higher priority switch forwarding tables alongside the standard shortest-path forwarding tables. In this context, it is worth mentioning that the *topology abstraction,* which consists of abstracting each site into a *super-node* with an aggregated *super-trunk* to a remote site, is key to improving the scalability of the B4 network. Indeed, this abstraction hides the details and complexity from the logically centralized TE controller, thereby allowing it to run protocols at a coarse granularity based on a global controller view and, more importantly, preventing it from becoming a serious performance bottleneck.

Unlike Kandoo (Hassas Yeganeh and Ganjali 2012), B4 (Jain et al. 2013) deploys robust *reliability* and *fault tolerance* mechanisms at both levels of the control hierarchy in order to enhance the B4 system availability. These mechanisms have been specially enhanced after experiencing a large-scale B4 outage. In particular, Paxos (Chandra et al. 2007) is used for detecting and handling the primary controller failure within each data center site by electing a new leader controller among a set of

reachable standby instances. On the other hand, network failures at the upper layer are addressed by the logically centralized TE controller, which adapts to failed or unresponsive site controllers in the bandwidth allocation process. Additionally, B4 is resilient against other failure scenarios in which the upper level TE controller encounters major problems in reaching the lower level site controllers (e.g. TE operation/session failures). Moreover, B4 guards against the failure of the logically centralized TE controller by geographically replicating TE servers across multiple WAN sites (one master TE server and four secondary hot standbys). Finally, another fault recovery mechanism is used in case the TE controller service itself faces serious problems. This mechanism stops the TE service and enables the standard shortest-path routing mechanism to be an independent service.

Similarly, Espresso (Yap et al. 2017) is another interesting SDN contribution which represents the latest and more challenging pillar of Google's SDN strategy. Building on the previous three layers of this strategy (the *B4* WAN (Jain et al. 2013), the *Andromeda* NFV stack and the *Jupiter* data center interconnect), *Espresso* extends the SDN approach to the peering edge of Google's network, where it connects to other networks worldwide. Considered a large-scale SDN deployment for the public Internet, Espresso, which has been in production for more than 2 years, routes over 22% of Google's total traffic to the Internet. More specifically, the Espresso technology allows Google to dynamically choose from where to serve content for individual users based on real-time measurements of end-to-end network connections.

To deliver unprecedented *scale-out* and efficiency, Espresso assumes a hierarchical control plane design split between *global controllers* and *local controllers*, which perform different functions. In addition, Espresso's software programmability design principle externalizes features into software, thereby exploiting commodity servers for scale.

Moreover, Espresso achieves higher availability (*reliability*) when compared to existing router-centric Internet protocols. It supports a fail static system in which the local data plane maintains the last known good state to allow for control plane unavailability, without impacting data plane and BGP peering operations. Finally, another important feature of Espresso is that it provides full *interoperability* with the rest of the Internet and the traditional heterogeneous peers.

1.4. Logical classification of existing SDN control plane architectures

Aside from the physical classification, we can categorize distributed SDN control architectures according to the way knowledge is disseminated among controller instances (the *consistency* challenge) into logically centralized and logically distributed architectures (see Figure 1.4). This classification has been recently adopted by Benamrane et al. (2015).

1.4.1. *Logically centralized SDN control*

1.4.1.1. *Onix and SMaRtLight*

Both Onix (Koponen et al. 2010) and SMaRtLight (Botelho et al. 2014) are logically centralized controller platforms that achieve controller state redundancy through state replication. However, the main difference is that Onix uses a distributed data store, while SMartLight uses a centralized data store to replicate the shared network state. They also deploy different techniques for sharing knowledge and maintaining a consistent network state.

Onix is a distributed control platform for large-scale production networks which stands out from previous proposals by providing a simple general-purpose API, a central NIB abstraction and standard state distribution primitives for easing the implementation of network applications.

In more specific terms, Onix uses the NIB data structure to store the global network state (in the form of a network graph) that is distributed across running Onix instances and synchronized through Onix's built-in state distribution tools, according to different levels of consistency as dictated by application requirements. In fact, in addition to interacting with the NIB at runtime, network applications on top of Onix initially configure their own data storage and dissemination mechanisms by choosing between two data store options already implemented by Onix in the NIB. A replicated transactional database that guarantees strong consistency at the cost of good performance for persistent but slowly changing data (state), and a high-performance memory-only distributed hash table (DHT) for volatile data that does not require strict consistency.

While the main advantage of Onix is its programmatic framework – created for the flexible development of control applications with desired trade-offs between performance and state consistency (strong/eventual) – it carries the limitations of eventually consistent systems that rely on application-specific logic to detect network state inconsistencies for the eventually consistent data and provide conflict resolution methods for handling them.

As mentioned in section 1.3.2.1, SMaRtLight is a fault-tolerant logically centralized master–slave SDN controller platform in which a single controller is in charge of all network decisions. This main controller is supported by backup controller replicas that should have a synchronized network view in order to take over the network control in case of the primary failure. All controller replicas are coordinated through a shared data store that is kept fault-tolerant and strongly consistent using an implementation of RSM.

Consistency between the master and backup controllers is guaranteed by replicating each change in the network image (NIB) of the master in the shared data

store before modifying the state of the network. However, such synchronization updates generate additional time overheads and have a drastic impact on the controller's performance. To address this issue, the controllers keep a local cache (maintained by one active primary controller at any time) to avoid accessing the shared data store for read operations. By keeping the local cache and the data store consistent even in the presence of controller failures, the authors claim that their simple master–slave structure, in the context of small to medium-sized networks, achieves a balance between consistency and fault tolerance while keeping performance at an acceptable level.

1.4.1.2. *HyperFlow and Ravana*

Both HyperFlow (Tootoonchian and Ganjali 2010) and Ravana (Katta et al. 2015) are logically centralized controller platforms that achieve state redundancy through event replication. Despite their similarities in building the application state, one difference is that the Ravana protocol is completely transparent to control applications, while HyperFlow requires minor modifications to applications. In addition, while HyperFlow is eventually consistent favoring availability, Ravana ensures strong consistency guarantees.

More specifically, HyperFlow (Tootoonchian and Ganjali 2010) is an extension of NOX into a distributed event-based control plane where each NOX-based controller manages a subset of OpenFlow network switches. It uses an event-propagation publish/subscribe mechanism based on the distributed WheelFS (Stribling et al. 2009) file system for propagating selected network events and maintaining the global network-wide view across controllers. Accordingly, the HyperFlow controller application instance running on top of an individual NOX controller selectively publishes relevant events that affect the network state and receives events on subscribed channels to other controllers. Then, other controllers locally replay all the published events in order to reconstruct the state and achieve the synchronization of the global view.

By this means, all controller instances make decisions locally and individually (without contacting remote controller instances). Indeed, they operate based on their synchronized eventually consistent network-wide view as if they are in control of the entire network. Through this synchronization scheme, HyperFlow achieves the goal of minimizing flow setup times and also congestion; in other words, cross-site traffic required to synchronize the state among controllers. However, the potential downside of HyperFlow is related to the performance of the publish/subscribe system, which can only deal with non-frequent events. In addition, HyperFlow does not guarantee a strict ordering of events and does not handle consistency problems. This makes the scope of HyperFlow restricted to applications that do not require a strict event ordering with strict consistency guarantees.

To correctly ensure the abstraction of a "logically centralized SDN controller", an elaborate fault-tolerant controller platform called Ravana (Katta et al. 2015) extended beyond the requirements for controller state consistency to include that for switch state consistency under controller failures.

Maintaining such strong levels of consistency in both controllers and switches in the presence of failures requires the entire event-processing cycle to be handled as a transaction in accordance with the following properties: (i) events are processed in the same total order at all controller replicas so that controller application instances reach the same internal state; (ii) events are processed exactly once across all the controller replicas; and (iii) commands are executed exactly once on the switches.

To achieve such design goals, Ravana follows a Replicated State Machine (RSM) approach but extends its scope to deal with switch state consistency under failures. Indeed, while Ravana permits unmodified applications to run in a transparent fault-tolerant environment, it requires modifications to the OpenFlow protocol, and it makes changes to current switches instead of involving them in a complex consensus protocol.

To be more specific, Ravana uses a two-stage replication protocol that separates the reliable logging of the master's event delivery information (stage 1) from the logging of the master's event-processing transaction completion information (stage 2) in the shared in-memory log (using Viewstamped Replication (Oki and Liskov 1988)) in order to guarantee consistency under joint switch and controller failures. In addition, it adds explicit acknowledgment messages to the OpenFlow 1.3 protocol and implements buffers on existing switches for event retransmission and command filtering. The main objective of the addition of these extensions and mechanisms is to guarantee the exactly-once execution of any event transaction on the switches during controller failures.

Such strong correctness guarantees for a logically centralized controller under Ravana come at the cost of generating additional throughput and latency overheads, which can be reduced to a reasonable extent with specific performance optimizations. Since the Ravana runtime is completely transparent and oblivious to control applications, achieving relaxed consistency requirements for the sake of improved availability, as required by some specific applications, involves consideration of new mechanisms that relax some of the correctness constraints on Ravana's design goals.

A similar approach to Ravana (Katta et al. 2015) was adopted by Mantas and Ramos (2016) to achieve a consistent and fault-tolerant SDN controller. In their work, the authors claim to retain the same requirements expressed by Ravana, namely, the transparency, reliability, consistency and performance guarantees but without requiring changes to the OpenFlow protocol or to existing switches.

Similarly, Kandoo (Hassas Yeganeh and Ganjali 2012) falls into the category of logically centralized controllers that distribute the control state by propagating network events. Indeed, at the top layer of its hierarchical design, Kandoo assumes a logically centralized root controller for handling global and rare network events. Since its aim was to preserve scalability without changing the OpenFlow devices, Kandoo did not focus on knowledge distribution mechanisms for achieving network state consistency.

1.4.1.3. *ONOS and OpenDayLight*

ONOS[2] and OpenDayLight (ODL)[3] (Bondkovskii et al. 2016; Badotra and Panda 2020) represent another category of logically centralized SDN solutions, which set themselves apart from state-of-the-art distributed SDN controller platforms by offering community-driven open-source frameworks and providing the full functionalities of network operating systems. Despite their obvious similarities, these prominent Java-based projects present major differences in terms of structure, target customers, focus areas and inspirations.

Unlike ODL (Badotra and Panda 2020), which is applicable to different domains, ONOS (Berde et al. 2014; Ruslan et al. 2020) from ON.LAB is specifically targeted toward service providers and is thus architected to meet their carrier-grade requirements in terms of scalability, high-availability and performance. In addition to the high-level northbound abstraction (a global network view and an application intent framework) and the pluggable southbound abstraction (supporting multiple protocols), ONOS, in the same way as Onix and HyperFlow, offers state dissemination mechanisms (Muqaddas et al. 2016) to achieve a consistent network state across the distributed cluster of ONOS controllers, a required or highly desirable condition for network applications to run correctly.

More specifically, ONOS's distributed core eases the state management and cluster coordination tasks for application developers by providing them with an available set of core building blocks for dealing with different types of distributed control plane state, including a *ConsistentMap* primitive for state requiring strong consistency and an *EventuallyConsistentMap* for state tolerating weak consistency.

In particular, applications that favor performance over consistency store their state in the shared eventually consistent data structure that uses optimistic replication assisted by the gossip-based anti-entropy protocol (Bianco et al. 2016). For example, the global network topology state, which should be accessible to applications with minimal delays, is managed by the *network topology store* according to this eventual consistency model. Recent releases of ONOS treat the network topology view as an in-memory state machine graph. The latter is built and updated in each SDN

2 https://opennetworking.org/onos/.

3 https://opendaylight.org/.

controller by applying local topology events and replicating them to other controller instances in the cluster in an order-aware fashion based on the events' logical timestamps. Potential conflicts and loss of updates due to failure scenarios are resolved by the anti-entropy approach (Bianco et al. 2016), in which each controller periodically compares its topology view with that of another randomly selected controller in order to reconcile possible differences and recover from stale information.

On the other hand, state imposing strong consistency guarantees is managed by the second data structure primitive built using RAFT (Ongaro and Ousterhout 2014), a protocol that achieves consensus via an elected leader controller in charge of replicating the received log updates to follower controllers and then committing these updates upon receipt of confirmation from the majority. The mapping between controllers and switches, which is handled by ONOS's *mastership store,* is an example of a network state that is maintained in a strongly consistent manner.

Administered by the Linux Foundation and backed by the industry, ODL is a generic and general-purpose controller framework that, unlike ONOS, was conceived to accommodate a wide variety of applications and use cases concerning different domains (e.g. Data Center, Service Provider and Enterprise). One important architectural feature of ODL is its YANG-based Model-Driven Service Abstraction Layer (MD-SAL) which allows for easy and flexible incorporation of network services requested by the higher layers via the northbound interface (OSGi framework and the bidirectional RESTful Interfaces) irrespective of the multiple southbound protocols used between the controller and the heterogeneous network devices.

The main focus of ODL was to accelerate the integration of SDN in legacy network environments by automating the configuration of traditional network devices and enabling their communication with OpenFlow devices. As a result, the project was perceived as adopting vendor-driven solutions that mainly aim to preserve the brands of legacy hardware. This represents a broad divergence from ONOS, which envisages a carrier-grade SDN platform with enhanced performance capabilities to explore the full potential of SDN and demonstrate its real value.

The latest releases of ODL provided a distributed SDN controller architecture referred to as ODL clustering. In contrast to ONOS, ODL did not offer various consistency models for different types of network data. All the data shared across the distributed cluster of ODL controllers for maintaining the logically centralized network view are handled in a strongly consistent manner using the RAFT consensus algorithm (Ongaro and Ousterhout 2014) and the Akka framework.

1.4.1.4. *B4 and SWAN*

Google's B4 (Jain et al. 2013) network leverages the logical centralization enabled by the SDN paradigm to deploy centralized TE in coexistence with the

standard shortest-path routing for the purpose of increasing the utilization of the inter-data center links (near 100%) compared to conventional networks and thereby enhancing network efficiency and performance. As previously explained in section 1.3.2.2, the logically centralized *TE server* uses the network information collected by the centralized *SDN gateway* to control and coordinate the behavior of site-level SDN controllers based on an abstracted topology view. The main task of the TE server is indeed to optimize the allocation of bandwidth among competing applications (based on their priority) across the geographically distributed data center sites.

That being said, we implicitly assume the presence of a specific consistency model used by the centralized SDN gateway for handling the distributed network state across the data center site controllers and ensuring the centralized TE application runs correctly based on a consistent network-wide view. However, very little information has been provided on the level of consistency adopted by the B4 system. In fact, one potential downside of the SDN approach followed by Google could be that it is too customized and tailored to their specific network requirements, given no general control model has been proposed for future use by other SDN projects.

Similarly, Microsoft has presented SWAN (Hong et al. 2013) as an intra-domain software-driven WAN deployment that takes advantage of the logically centralized SDN control using a global TE solution to significantly improve the efficiency, reliability and fairness of their inter-DC WAN. Like Google, Microsoft did not provide much information about the control plane state consistency updates.

1.4.2. *Logically distributed SDN control*

The potential of the SDN paradigm has been fully explored within single administrative domains like data centers, enterprise networks, campus networks and even WANs, as discussed in section 1.4.1. Indeed, the main pillars of SDN – the decoupling between the control and data planes together with the consequent ability to program the network in a logically centralized manner – have unleashed productive innovation and novel capabilities in the management of such intra-domain networks. These benefits include the effective deployment of new domain-specific services as well as the improvement of standard control functions, following the SDN principles such as intra-domain routing and TE. RCP (Caesar et al. 2005) and RouteFlow (Rothenberg et al. 2012) are practical examples of successful intra-AS platforms that use OpenFlow to provide conventional IP routing services in a centralized manner.

However, this main feature of logically centralized control, which has been leveraged by most SDN solutions to improve network management at the intra-domain level, cannot be fully exploited for controlling heterogeneous networks

involving multiple autonomous systems (ASes) under different administrative authorities (e.g. the Internet). In this context, recent works have considered extending the SDN scheme to such inter-domain networks while remaining compatible with their distributed architecture. In this section, we shed light on these SDN solutions that adopted a logically distributed architecture in accordance with legacy networks. For this reason, we place them in the category of logically distributed SDN platforms as opposed to the logically centralized ones mainly used for intra-domain scenarios.

1.4.2.1. *DISCO and D-SDN*

The DISCO project (Phemius et al. 2013), for example, suggests a logically distributed control plane architecture that operates in such multi-domain heterogeneous environments, more precisely WANs and overlay networks. Built on top of Floodlight, each DISCO controller administers its own SDN network domain and interacts with other controllers to provide end-to-end network services. This inter-AS communication is ensured by a unique lightweight control channel to share summary network-wide information.

The most obvious contribution of DISCO lies in the separation between intra-domain and inter-domain features of the control plane, while each type of feature is performed by a separate part of the DISCO architecture. The intra-domain modules are responsible for ensuring the main functions of the controller such as monitoring the network and reacting to network issues, and the inter-domain modules (messenger, agents) are designed to enable message-oriented communication between neighbor domain controllers. Indeed, the AMQP-based messenger offers a distributed publish/subscribe communication channel used by agents, which operates at the inter-domain level by exchanging aggregated information with intra-domain modules. DISCO was assessed on an emulated environment according to three use cases: inter-domain topology disruption, end-to-end service priority request and virtual machine migration.

The main advantage of the DISCO solution is the potential to adapt it to large-scale networks with different ASes such as the Internet (Benamrane et al. 2015). However, we believe that there are also several drawbacks associated with such a solution including the static non-evolving decomposition of the network into several independent entities, which is in contrast to emerging theories such as David D. Clark's theory (Clark et al. 2003) about the network being manageable by an additional high-level entity known as the Knowledge Plane. Moreover, following the DISCO architecture, network performance optimization becomes a local task dedicated to local entities with different policies, each of which acts in its own best interest at the expense of the general interest. This leads to local optima rather than the global optimum which achieves the global network performance. Additionally, from the DISCO perspective, one SDN controller is responsible for one independent domain. However, an AS is usually too large to be handled by a single controller. Finally, DISCO did not provide appropriate reliability strategies suited to its

geographically distributed architecture. In fact, in the event of a controller failure, it could be inferred that a remote controller instance will be in charge of the subset of affected switches, thereby resulting in a significant increase in the control plane latency. In our opinion, a better reliability strategy would involve local per-domain redundancy. Local controller replicas should indeed take over and serve as backups in case the local primary controller fails.

Similarly, INRIA's D-SDN (Santos et al. 2014) enables a logical distribution of the SDN control plane based on a hierarchy of *Main Controllers* and *Secondary Controllers*, matching the organizational and administrative structure of the current and future Internet. In addition to dealing with levels of control hierarchy, another advantage of D-SDN over DISCO is related to its enhanced security and fault tolerance features.

1.4.2.2. *SDX-based controllers*

Unlike DISCO, which proposes per-domain SDN controllers with inter-domain functions to allow autonomous end-to-end flow management across SDN domains, recent trends have considered deploying SDN at Internet eXchange Points (IXPs), thus giving rise to the concept of Software-Defined eXchanges (SDXes). These SDXes are used to interconnect participants of different domains via a shared software-based platform. This platform is usually aimed at bringing innovation to traditional peering, easing the implementation of customized peering policies and enhancing the control over inter-domain traffic management.

Prominent projects adopting this vision of software-defined IXPs and implementing it in real production networks include Google's Cardigan in New Zealand (Stringer et al. 2014), SDX at Princeton (Gupta et al. 2014), CNRS's French TouIX (Lapeyrade et al. 2016) (the European ENDEAVOUR project) and the AtlanticWave-SDX (Heidi Morgan 2015). Here, we chose to focus on the SDX project at Princeton since we believe in its potential for demonstrating the capabilities of SDN to innovate IXPs and for bringing answers to deploying SDX in practice.

The SDX project (Gupta et al. 2014; Mostafaei et al. 2021) takes advantage of SDN-enabled IXPs to fundamentally improve wide-area traffic delivery and enhance conventional inter-domain routing protocols that lack the required flexibility for achieving various TE tasks. Today's BGP is indeed limited to destination-based routing, it has a local forwarding influence restricted to immediate neighbors and it deploys indirect mechanisms for controlling path selection. To overcome these limitations, SDX relies on SDN features to ensure fine-grained, flexible and direct expression of inter-domain control policies, thereby enabling a wider range of valuable end-to-end services such as inbound TE, application-specific peering, server load balancing and traffic redirection through middle-boxes.

The SDX architecture consists of a smart SDX controller handling both SDX policies (*policy compiler*) and BGP routes (*route server*), conventional Edge routers and an OpenFlow-enabled switching fabric. The main idea behind this implementation is to allow participant ASes to compose their own policies in a high-level (using Pyretic) and independent manner (through the virtual switch abstraction), and then send them to the SDX controller. The latter is in charge of compiling these policies to SDN forwarding rules while taking into account BGP information.

In addition to offering this high-level softwarized framework, which is easily integrated into the existing infrastructure while maintaining good interoperability with its routing protocol, SDX also stands out from similar solutions like Cardigan (Stringer et al. 2014) because of the efficient mechanisms used for optimizing control and data plane operations. In particular, the scalability challenges faced by SDX under realistic scenarios have been further investigated by iSDX (Gupta et al. 2016; Abdullahi et al. 2021), an enhanced Ryu-based version of SDX intended to operate at the scale of large industrial IXPs.

However, one major drawback of the SDX contribution is that it is limited to the participant ASes being connected via the software-based IXP, implying that non-peering ASes would not benefit from the routing opportunities offered by SDX. Moreover, while solutions built on SDX use central TE policies for augmenting BGP and promote a logical centralization of the routing control plane at the IXP level, SDX controllers are still logically decentralized at the inter-domain level since no information is shared between them about their respective interconnected ASes. This brings us back to the same problem we pointed out for DISCO (Phemius et al. 2013) regarding end-to-end traffic optimization being a local task for each part of the network.

To remedy this issue, some recent works (Kotronis et al. 2015) have considered centralizing the whole inter-domain routing control plane to improve BGP convergence by outsourcing the control logic to a multi-AS routing controller with a "bird's-eye view" over multiple ASes.

It is also worth mentioning that SDX-based controllers face several limitations in terms of both security and reliability.

Because the SDX controller is the central element in the SDX architecture, security strategies must focus on securing the SDX infrastructure by protecting the SDX controller against cyber attacks and by authenticating any access to it. In particular, Chung et al. (2015) argue that SDX-based controllers are subjected to the potential vulnerabilities introduced by SDN in addition to the common vulnerabilities associated with classical protocols. In this respect, they distinguish three types of current SDX architectures and discuss the security concerns involved.

In their opinion, Layer-3 SDX (Stringer et al. 2014; Gupta et al. 2014) will inherit all BGP vulnerabilities, Layer-2 SDX (Internet2) will obtain the vulnerabilities of a shared Ethernet network and SDN SDX (Lin et al. 2015) will also bring controller vulnerabilities like DDoS attacks, comprised controllers and malicious controller applications. Moreover, the same studies (Chung et al. 2015) point out that SDX-based controllers require security considerations with respect to policy isolation between different SDX participants.

Finally, since the SDX controller becomes a potential single point of failure, fault tolerance and resiliency measures should be taken into account when building an SDX architecture. While the distributed peer-to-peer SDN SDX architecture (Chung et al. 2016) is inherently resilient, centralized SDX approaches should incorporate fault tolerance mechanisms such as those discussed in section 2.3 and should also leverage the existing fault tolerant distributed SDN controller platforms (Berde et al. 2014; Ruslan et al. 2020; Das et al. 2020).

1.5. Conclusion

In this chapter, we have provided a detailed analysis of state-of-the-art distributed SDN controller platforms. We assessed their architecture components and design patterns, and classified them in novel ways (physical and logical classifications) in order to provide useful guidelines for SDN research and deployment initiatives.

Additionally, our extensive analysis of these SDN platform proposals has enabled us to gain a thorough understanding of their advantages and drawbacks and, most importantly, to develop a critical awareness of the challenges facing distributed control in SDNs. These open challenges are further discussed in Chapter 2.