A Survey on Fault Management in Software-Defined Networks

Article *in* IEEE Communications Surveys & Tutorials · June 2017

DOI: 10.1109/COMST.2017.2719862

2 authors:

Paulo Fonseca
Federal University of Amazonas
6 PUBLICATIONS 99 CITATIONS
SEE PROFILE

Edjard Mota
Federal University of Amazonas
30 PUBLICATIONS 339 CITATIONS
SEE PROFILE

A Survey on Fault Management in Software-Defined Networks

Paulo Fonseca* and Edjard Mota

Institute of Computing, Federal University of Amazonas
{paulo.cesar, edjard}@icomp.ufam.edu.br

***Abstract*—Software-defined networking (SDN) is an emerging paradigm that has become increasingly popular in the recent years. The core idea is to separate the control and data planes, allowing the construction of network applications using high-level abstractions which are translated to network devices through a southbound interface. SDN architecture is composed of three layers: *infrastructure layer*, responsible exclusively for data forwarding; *control layer*, which maintains the network view and provides core network abstractions; and *application layer*, which uses abstractions provided by the control layer to implement network applications. SDN provides features, such as flexibility and programmability, that are key enablers to meet current network requirements (e.g., multi-tenant cloud networks, elastic optical networks). However, along with its benefits, SDN also brings new issues. In this survey we focus on issues related to fault management. Different fault management threat vectors are introduced by each layer, as well as by the interface between layers. Nevertheless, besides addressing fault management issues of its architecture, SDN also must handle the same problems faced by legacy networks. However, programmability and centralized management might be used to provide flexibility to deal with those issues.**

This paper presents an overview of fault management in SDN. The major contributions of this work are as follows: 1) Identification of the main fault management issues in SDN and classification according to the affected layers; 2) Survey of efforts that address those issues and classification according to the affected planes, issues concerned, general approaches and features; 3) Discussion about trade-offs of different approaches and their suitability for different scenarios.

***Index Terms*—Fault management, Software Defined Networks, Fault Tolerance , Resiliency, Failure Recovery, Failure, Survivability, Network Verification, SDN, OpenFlow**

I. Introduction

Among proposals that aimed to simplify network management and facilitate innovation, software-defined networking (SDN) has emerged as one of most important networking paradigms in the recent years [1]. SDN aims to simplify management and provide flexibility to the network through a clear separation of control plane from data plane. In SDN, control functionality is moved to a logically centralized controller, whereas retaining only the forwarding capabilities in the network devices. The network controller provides a northbound interface, which allows the construction of high-level network management applications, and a southbound interface, which manages network devices through a vendor-independent protocol.

*Corresponding author.

In the last years, numerous efforts exploited SDN flexibility and programmability to cope with increasingly stringent network requirements of many technology trends. Cloud computing networks rely heavily on virtualization, which heightens network complexity by adding a virtualization layer on the network stack and increasing network volatility [2]. Big data demands that data transmission, storage, and computation are optimized. Thus, networking takes an important role in efficient big data processing [3]. At last, the management of a 5G network involves the orchestration of multiple different access technologies and multiple hierarchies [4]. In order to meet performance requirements, network management must be efficient and quickly respond to network updates and requests, in spite of the inherent complexity of 5G infrastructure.

Besides, current networks also must meet other requirements. Fault tolerance, reliability, and resiliency to failures are amongst the most important ones. Usually, cloud providers, big data services, and cellular network carriers must follow strict service level agreements (SLAs), which frequently specify desired values for metrics such as response time, processing time, the rate of failure, maintenance time and data losses. Failures have a major financial impact on service providers. For example, from 2007 to 2013, cloud networks from 28 cloud providers amass losses estimated at US$273 million and 1,600 hours of disruptions, due to application and infrastructure failures [5].

A *failure* occurs when the network is not able to correctly deliver a service, whereas a *fault* is the root cause of a failure. In traditional networks, failures are mostly related to link/node failures and faults are mainly software bugs or hardware malfunction [6]. The fault management process consists of fault detection, fault localization, and fault recovery. Fault management mechanisms usually rely on distributed protocols (e.g. OSPF) to handle failures, which lead to a number of issues. Since a failure may be detected by different devices, a single failure may generate multiple alarms. Transient failures may lead to inconsistency among network devices and global network state may take a long time to converge after a failure [7]. Network partitioning during a failure may produce incomplete or incorrect information about the network state. Moreover, other types of failures, such as operator errors and misconfiguration failures are hard to troubleshoot because network administrators are constrained to ad hoc tools, such as `ping` and `traceroute`. In legacy networks, most network devices are closed black boxes that run proprietary software with no standard open interface, making it difficult to extract meaningful information and to reason about network behavior.

Even though the distributed nature of traditional networks benefits scalability and resiliency to failures, it hampers the management of complex networked environments both because of the necessity to configure a large number of devices and because of the inherent complexity of policies themselves [8]. Besides, due to multiple devices with different vendor-specific characteristics, novel solutions that address new requirements must use standardized protocols, such as SNMP and MPLS. Otherwise, these solutions will have limited impact if they are not compatible with the majority of network devices in production networks. Nonetheless, simplified management and potential to innovation are key enablers to realize current network applications' requirements.

SDN's logically centralized control and programmable interfaces are being used as a new way to look out for the resolution to many fault management issues which exist today, for example, a global network view adds many possibilities to the process of fault localization [9]–[11].

SDN architecture is composed of three layers: *infrastructure layer*, which is exclusively responsible for data forwarding and statistics storage, *control layer*, which maintains the network view and provides core network functions, and *application layer*, which uses abstractions provided by control layer to implement network business applications(e.g. firewall, load balancer), enforcing networking policies and requirements.

SDN's layered architecture presents new points of failure, and even augments some issues already present in legacy networks. SDN layers have independent and interdependent fault tolerance threat vectors. For instance, a controller failure may halt network services affecting data plane. In summary, as SDN brings new ways to tackle classic network fault tolerance issues it also brings new threat vectors that need to be addressed.

In this survey, we aim to identify what fault management issues are present in SDN, how they relate to their layers, how current efforts address those issues, what are the major contributions of those efforts and what are the major gaps in the ongoing researches. We surveyed works that address any step of the fault handling process, from fault detection and prevention to failure recovery. It is worth to note that even though security is an aspect of overall network resilience and malicious attacks may lead to faults, we did not include security efforts in this survey because: (1) security attributes require a separate classification, and analysis and (2) recent surveys focused on SDN and security already present a robust literature review [12], [13].

In application plane, we considered faults that are introduced in the network due to software bugs, which may lead to misconfiguration and network correctness violations, and network application failure along with its effect on control and data plane. In control layer, we considered issues that affect network overall fault tolerance. This includes effects of controller failures on the network, maintenance of control plane state during failures, controller placement and control traffic failure recovery. In the infrastructure layer, we focused on fault management of link/node failures, how they relate with other network requirements (e.g., congestion, quality of service) and the different network environments that are addressed in the literature.

This paper is structured as follows: Section II discusses papers that provide discussion about fault management in SDN; section III give a basic overview of SDN architecture, its features and how SDN relates to fault management; section IV presents fault tolerance issues discussed in SDN and classifies them according to the affected layers; section V describes and classifies data plane approaches according to the issue they address, which fault tolerance attribute they provide and which layers are involved in their operation. We also present a comparison between approaches and their key points. Sections VI and VII present a similar analysis to control plane and application plane respectively; section VIII discusses open issues and future directions for research on fault tolerance in SDN; section IX concludes the paper.

II. Related Work

A number of works have discussed issues and/or efforts related to fault management in SDN [14]–[17]. These works differ on the scope of their discussion and on the classification of fault management efforts. Kreutz et al. [14], [15] aim to provide insights about fault tolerance and dependability issues raised by SDN, whereas Silva et al. [16] and Chen et al. [17] provide a classification of the literature. The former categorizes surveyed works in resilience disciplines as defined by Sterbenz et al. [18], and the latter classifies efforts according to the fault management attribute provided (*e.g.*, fault detection, fault recovery).

A. Discussion on SDN fault management challenges

Kreutz et al. [14] provide a comprehensive view of software defined networking architecture, describing the elements that compose SDN, current challenges, and ongoing research efforts. In a subsection about fault tolerance, Kreutz et al. discuss the challenges that are raised by a split architecture, and how they can achieve similar levels of resilience as that of the legacy networks. They list some efforts that address those issues. They also compared different aspects of available network controllers, including how they address availability and fault tolerance challenges, such as redundancy and consistency. In [15], Kreutz et al. present a more in-depth exploration of SDN threat vectors, specifically those related to dependability and security. Kreutz et al. discuss how these issues can be addressed, sketching possible solutions, based on approaches already used in other fields to provide dependability and security (e.g., trust enforcement between network elements, component diversity). In both works, Kreutz et al. focused on the exploration of dependability issues related to SDN, rather than surveying efforts that addressed to those issues.

B. Classification of SDN fault management efforts

Silva et al. [16] present a survey on resilience research, categorizing efforts according to six disciplines [18]: security, survivability, performability, traffic tolerance, disruption tolerance, and dependability. For each discipline, they discuss which resilience attributes are encompassed by the discipline (e.g. dependability encompasses reliability and availability) and briefly describe (or just list) the methods in each category. They also characterized methods with respect to which SDN plane each effort belongs to. In the end, Silva et al. summarize identified problems and challenges, according to the different areas and resilience disciplines. No comparison among surveyed methods is presented.

Chen et al. [17] analyze the intersection of fault tolerance and SDN. They present a discussion about fault tolerance in traditional networks and which features of OpenFlow protocol might be used to provide fault tolerance to the network. Then, methods are categorized according to the planes concerned and the fault tolerance attribute provided. In the data plane, link/node failure detection and failure recovery methods are compared. In the control plane, Chen et al. discuss methods that handle failures of the controller and of the control channel. In both cases, only a small subset of available literature is discussed. Chen et al. do not present a comparison between control plane methods. There is no discussion about application plane issues or other aspects of data plane fault tolerance, such as forwarding loops and forwarding black holes.

C. *Scope of the survey*

The current work aims to provide a comprehensive view of fault management in SDN. Fault management issues are presented in a bottom-up approach, from data plane to application plane, describing the issues that may lead to faults in each plane and/or in the relationship between planes. Then, a broad literature survey is presented and efforts are categorized according to the aspects of network fault tolerance to which they are related. Data plane efforts refer to methods dedicated to providing fault tolerance to network traffic. Failure detection, location, and recovery methods are discussed and compared regarding their characteristics, scope, and goal. Control plane efforts are subdivided into (1) control plane architectures, (2) controller placement strategies and (3) control traffic fault tolerance. Each subdivision encompasses multiple issues that may cause control plane faults, such as controller failure and control channel disruption. Application plane efforts are classified according to the following subdivisions: Application design, application fault tolerance, network troubleshooting, and network correctness.

III. Background

In this section, we provide a more detailed view on SDN architecture and how this architecture relates to fault management.

A. *Software defined networks*

In traditional networks, control plane and data plane are tightly coupled in network devices. Conversely, SDN keeps only data forwarding functionality in network devices, whereas it delegates control plane functionality to a physically separate layer, composed of one or more network entities called *controllers*, which provide the interface between the high-level network applications and the network devices. As depicted in figure 1 [19], SDN architecture comprises three layers (in the course of this work, we use the terms *layer* and *plane* interchangeably):

- *Infrastructure layer*: Also known as the *data plane*, it is responsible for data forwarding and statistics storage; it may include physical and virtual switches. Network devices must comply with a standard interface (e.g. OpenFlow protocol [20]) that is used by the control plane to manage the devices.
- *Control layer*: The control layer consists of one or more SDN controllers. The main function of the control layer is to maintain a *logically centralized network view* that allows network applications reason about network properties and behavior. The control layer observes network state through the open interface with the devices, and provides an *application programming interface* (API) to construct network applications using high-level terms (e.g. host names instead of IP addresses).
- *Application layer*: Uses the API provided by the control layer to implement network business applications (e.g. firewall, load balancer), enforcing networking policies and requirements.

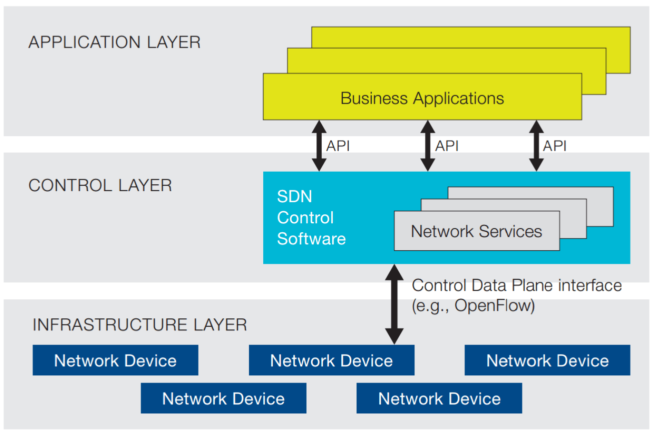


Fig. 1: Software defined networking architecture [19].

Even though first proposals of control and data plane separation are not recent [21], [22], discussions about its implications and real deployments have not gained traction until about the past few years [23]–[26] with the proposal and the subsequent large adoption of the OpenFlow protocol [27], a southbound protocol that provides a communication interface which can be used to configure switches' forwarding policies. The OpenFlow matching process is depicted in Figure 2: When a switch receives a packet, (1) it checks if the packet's header fields matches with one or more rules; (2) If no match occurs, the packet is forwarded to the network controller, (3) which will handle the packet and generate a new flow rule; (4) otherwise, if there is a matching rule, (5) a list of actions is applied to the packet. Figure 3 lists which header fields are available for matching in OpenFlow 1.1.0. Actions that can be applied to flows include basic forwarding, packet modification, and QoS support. The set of possible actions and fields that can be used for rule matching increases with newer specifications of OpenFlow protocol. Although there are alternatives to OpenFlow, such as ForCES [28] and OpFlex [29], most of SDN solutions are implemented using OpenFlow, including those related to fault management.

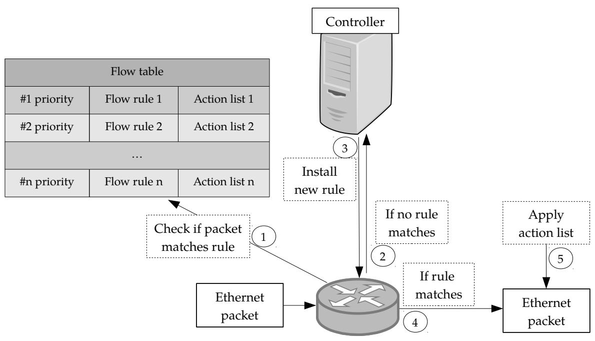


Fig. 2: Overview of OpenFlow rule-matching process

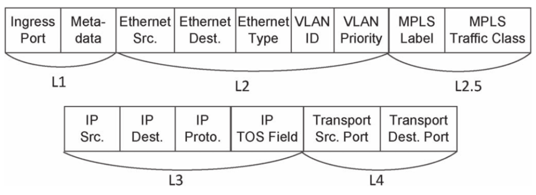


Fig. 3: Match fields of a flow table entry in OpenFlow 1.1.0 [30]

SDN offers three main features to network administrators: *centralized management*, *flexibility*, and *programmability*. Centralized control simplifies the management because it allows the construction of control logic from a complete view of the network. A designated controller to which all network devices must report their states decreases convergence time and coordination complexity. The southbound API gives flexibility to network admins by allowing them to separately handle each type of traffic, according to network needs. Since network devices from different vendors must agree on a standardized specification, network admins can build network applications on a well-defined common ground. At last, programmability is achieved through a northbound API that provides a set of abstractions. Network services and business applications are programmed using these high-level abstractions, which may implement entities from network resources to business policies and more. Also, programmability allows network applications to change its behavior at run-time, reacting to network events.

These features open possibilities for addressing network requirements and challenges in novel ways. A complete view of the network facilitates routing decisions, traffic engineering, and fault localization because it delegates the handling of partial information and communication asynchrony to the network controller. Centralized management also allows network applications to detect complex network behavior, such as periodic traffic patterns and distributed denial-of-service attacks [31]. The ability to handle traffic using fine-grained criteria helps traffic slicing and management of multi-tenant environments [32]. Taking advantage of programmability, it is much more simple to manage virtual networks using high-level terms and abstracting the details of the underlying infrastructure. Also, applications that require frequent reconfigurations, such as big data processing [33], can take advantage of SDN's ability to program the network at runtime.

To further discussion, a number of works provide detailed investigation on SDN architecture [30], [34], [35], on how SDN may help the realization of current network requirements [36]–[40].

B. Fault management in computer networks

The fault management process refers to the handling of the whole lifecycle of faults, which includes *faults*, *errors*, *failures* and *symptoms*:

- A *fault* is the root cause that may lead the system to an error state. A fault can remain dormant for a long period of time before causing an error.
- An *error* is the manifestation of a fault, it occurs when the system enters in an incorrect state. An error may or may not be detected by the system and it does not necessarily cause a disruption of the service.
- A *failure* occurs when a network service deviates from its expected correct behavior due to one or more errors.
- A *symptom* is a side-effect caused by one or more failures and that can be observed in the network behavior.

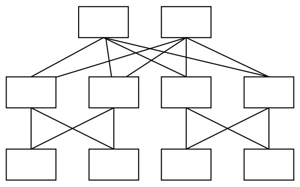
For example, a buggy implementation of router software that, under specific conditions, generates a network black hole represents a *fault*. When these conditions are met, the network enters in an incorrect state (i.e. some packets will be silently discarded), thus, an *error* occurs. When a packet enters the black hole, a *failure* takes place. The increase of packet loss and absence of acknowledgment messages are *symptoms* of the black hole.

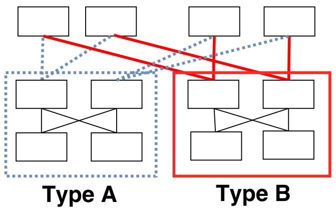
The result of a survey conducted among network administrators shows that the most common causes of network failures are switch/router software bugs, hardware failures, protocol misconfiguration and external factors, whereas the most common symptoms of failures are reachability problems, degraded throughput/latency, and congestion [6].

Legacy networks and SDN are different in regard to which types of faults may occur and which fault management mechanisms are supported by each architecture.

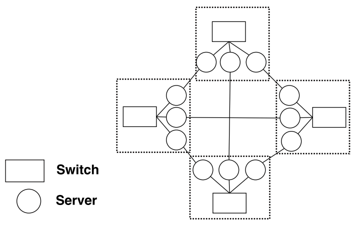
1) Legacy networks: In traditional networks, the lack of centralization and heterogeneity of network devices imposes constraints to fault management solutions. For example, most network devices run closed proprietary software, thus, there are few options to handle switch/router software bugs besides updating the device firmware version.

Many fault management solutions are based on distributed protocols, such as OSPF, since the control plane is encapsulated in the network devices. Distributed communication lead to numerous issues: a single failure may generate multiple alarms; transient failures may lead to state inconsistency, global network state may take a long time to converge [7].


(a) Aspen tree



(b) AB fat-tree



(c) 2-level DCell topology

Fig. 4: Examples of resilient topologies

Network administrators can alleviate some of these issues by integrating control plane mechanisms with infrastructure layer characteristics. A number of efforts propose novel network topologies which enable efficient fault management mechanisms and enhance network resiliency. Walraed-Sullivan et al. [41] propose the Aspen tree topology, a modification of the fat-tree topology [42] that decreases the convergence time and reduces overhead by increasing the number of redundant links between switches. Since there are more alternate paths between switches, the fault management process is limited to a small subset of devices located near to the failure. Liu et al. [43] introduce the *AB fat-tree* topology, which increases the fault tolerance of the fat-tree topology by breaking the tree symmetry. The topology is composed of two types of subtrees (type A and B) that are wired to their parents in two different ways, as depicted in Figure 4b. This change increases the number of alternate paths because a parent with a failed child can always reroute through the parents of a child in a type B subtree (and vice-versa). Guo et al. [44] propose DCell, a recursive topology aimed for data centers, in which a high-level DCell is composed by multiple low-level DCells and DCells at the same level are fully connected with one another. This organization does not have a single point of failure and enables near shortest-path routing even in the presence of multiple failures. In Figure III-B1, simple examples of these topologies are depicted in order to display their properties.

There are complex trade-offs involved in such design choices, since scalability, latency, construction cost, and other factors frequently are network requirements as critical as network resilience. AB fat-trees allow for a temporary increase in latency for paths affected by faults in exchange for no increase in hardware cost. The Aspen tree topology increases the number of links between the switches in exchange for supporting fewer hosts. DCell is robust and fault-tolerant but it does require more hardware. In AB fat-trees and Aspen trees switches are the key component in forwarding, whereas, in DCell the servers perform most of forwarding. A number of efforts evaluate and compare different resilient topologies with respect to multiple metrics [45], [46].

2) Software defined networks: In SDN, due to its decoupling between control and data plane, faults may appear in different layers and network entities.

SDN layers have independent and interdependent fault tolerance threat vectors. In application layer, the design of network applications must take into consideration the hazards, such as race conditions and deadlocks, that may lead to failures. Application failures may also affect the control layer if the application execution space is not isolated from the controller space, and the infrastructure layer if the failed application was related to data forwarding and network devices have no alternative strategy. In control layer, a centralized controller may incur a single-point of failure, whereas distributed controllers must keep their state consistent in order to avoid anomalies due to non-deterministic applications. In infrastructure layer, network devices must quickly detect node/link failures and notify network controllers. The unreliability of communication between the control and infrastructure layers may cause loss of network events, leading to inconsistencies between logical network view and infrastructure physical state.

Furthermore, SDN even augments some issues already present in legacy networks. For example, in legacy networks, most of the network devices have closed proprietary software that varies depending on the vendor. Whereas, in SDN, network devices must comply with a standardized and open protocol. If a protocol characteristic can be exploited on an attack or may lead to a fault under specific circumstances, all protocol-compliant devices may be compromised [15].

An intuitive way to address these new issues is to borrow concepts from fault tolerance solutions of other areas, such as distributed systems and operating systems. Consensus protocols are widely used in distributed systems in order to guarantee consistency between nodes. Operating system design has developed fault isolation techniques to prevent OS halting due to application failures. Those approaches can be adapted to satisfy SDN needs, considering computer networks properties and requirements [47], [48]. Besides, SDN features may use such concepts to support a wider range of failures. Fault tolerance primitives and abstractions can be defined using control and application layer to define high-level fault tolerance policies. A standard southbound protocol can be leveraged to support failure protection and other fault management capabilities that may be exploited by network applications. In summary, as SDN brings new ways to tackle classic network fault tolerance issues it also brings new threat vectors that need to be addressed.

IV. Fault Tolerance Issues in SDN

In this section, we discuss which fault tolerance issues are present in each layer and which might arise from the interaction between layers. In each layer/interface between layers, it was investigated: (1) What are possible sources of *faults* in this layer/interface? and (2) what *failures* necessarily impact this layer/interface? Therefore, layers that *may or may not* be affected by a type of failure or that can be used to address the issue are not considered in the classification. For instance, the failure of a link that it is not part of the control network (*i.e.*, connects network devices to the controllers) may or may not affect control and application layer. Thus, link failures and control network failures are classified separately, the former in the infrastructure layer and the latter in the interface between infrastructure and control. We use a bottom-up approach, from infrastructure plane to application plane.

A. Infrastructure layer

Fault tolerance issues of the infrastructure layer are mostly related to issues already present in traditional networks, namely link and node failure. However, centralized management and programmability shed a different light on those issues, enabling novel strategies and bringing new challenges.

a) Network failure detection and location: In traditional networks, fault detection and location must be performed in a distributed manner. A number of challenges arise: a single failure may trigger multiple alarms [49], end-to-end failure detection, network device state may take a long time to stabilize [7], network partitioning may be difficult to detect, and so on. SDN can ease this task by keeping a centralized network view that all network devices must update upon link failure. For example, a controller might identify which switches are affected by a specific failure and notify only those switches [49]. However, overhead caused by this communication will impact recovery time, thus it must be carefully designed.

b) Network failure recovery: Link and node failure is a problem almost as old as communications. In computer networks, there are two general failure recovery approaches: *protection* and *restoration*. In protection, backup routes are configured before a failure occurs. Whereas in restoration, backup paths are computed on-demand, upon network failures. If the protection paths are not fairly distributed among the links, network protection may lead to congestion scenarios, and switch memory may exhaust if the number of forwarding rules is huge. On the other hand, failure restoration may increase recovery time, overload network controllers, among other issues. SDN supports hybrid approaches with multiple protocols and mechanisms coexisting and coordinating among themselves to improve failure recovery.

Centralized control enables novel strategies for the network recovery from link and node failures since the network view may be used to support traffic engineering applications, QoS capabilities, and congestion-aware routing. Features of different network environments must be investigated in order to provide fault tolerance to a wide variety of scenarios, such as data center networks, smart grid and optical networks.

B. Control/infrastructure interface

The physical separation of control logic from network devices inherently creates an interdependency between network devices and controllers for proper operation. Centralized management also increases network requirements, since links responsible for control traffic are more important to the overall operation than links that only carry data traffic. We first discuss issues already present in legacy networks but are aggravated in SDN: control network failure and controller-switch reliable communication, which are analogous to link/node failure and data delivery, respectively. Then, two issues introduced by SDN are described: controller placement and controller failover.

a) Control network failure: A control network is composed of all controller links that are used for network management. A link failure in the control network may cause a disruption in the management of a switch and all of its downstream switches. Also, if backup paths are computed or activated on demand, control channel failure might prevent data traffic restoration. Control traffic recovery goal is to restore switch connectivity with any available controller, differing from data traffic recovery where a specific destination path must be restored. Experiments showed that efficient control channel failure detection is also an important factor to reduce recovery time [11].

b) Controller-switch reliable communication: The core concept of software-defined networking is the separation of control and data plane, where network devices communicate about the relevant network events to the control plane and control plane provides network services by sending commands to devices. If an event or a command is not delivered to its destination, network management may be compromised. For example, if a link failure event is not delivered to control plane, routing services can not know that this link is down and generate invalid paths. On the data plane, if a new device connects to the network, and authentication service fails to deliver a command granting access to the network, the device will stay disconnected. Reliable message delivery is an old network problem with an extensive literature and a number of protocols addressing this issue, however, in SDN this problem is aggravated, since the communication between a centralized control and network devices is critical, and presents itself in new forms, *e.g.* network events can be lost during controller failover.

c) Controller placement: Since controllers must be assigned to the set of network devices, which it will control, multiple controllers gives rise to two questions during network design: 1) How many controllers are needed to meet network requirements? 2) Where in the topology should they be placed?

Underprovisioning the number of controllers can lead to overloaded controllers, possibly causing downtime due to resource exhaustion, while overprovisioning may result in underutilization, wasting money and resources. Additionally, research has demonstrated that the location of a controller in the topology can affect network performance [50]. Since connectivity between controllers and devices is essential to network management, a non-optimal controller placement may allow that even if only a minority of links fails, the majority of devices still lose control channel connectivity. Other aspects, such as recovery time and inter-controller latency are also be affected by controller placement.

d) Controller failover: Currently, network controllers can connect to SDN devices and manage them in two modes: *Hierarchical*, where the switch sends all its request to a primary controller and, upon a primary controller failure, the switch must connect to a pre-configured backup controller;

Non-hierarchical, where the switch connects simultaneously to multiple controllers and sends each request to all of them, which are responsible for coordinating in order to handle redundant requests and responses.

OpenFlow support this feature through *role request messages* since version 1.2 [51]. Each switch can assign roles to the controllers connected to it: *master*, the controller can access and control the switch, only one controller can be the master at any given time; *slave*, the controller can access the switch reading its state but it can not control the device, writing in switch flow table; *equal*, the controller can manage the switch without restrictions and any number of controllers can be configured as equal.

Controllers may differ in performance, network localization, and latency to network devices, thereby affecting the overall performance of the network. This gives rise to different questions related to fault tolerance. Which controller must be chosen as the initial primary? How should the list of backup controllers be composed and ordered? How frequently must it be updated? How to detect primary and backup failures? The failover process of the non-hierarchical mode is simpler, as all controllers are already connected, however, requests must be ordered in such way that any of the controllers is able to respond to any request during the failover.

C. Control layer

Since control layer is the translation layer between applications and infrastructure, most of its issues directly affect or are caused by other layers. Only the communication among controllers exclusively refers to its own operation.

a) Inter-controller consistency: The use of multiple controllers is an intuitive approach to avoid the single point of failure problem [52], [53], as it can leverage solutions ranging from a centralized controller replicated in different controllers to distributed controllers coordinating to provide network services while increasing resiliency. Despite the approach used, in order to keep a logical centralization of control, all controllers must share a consistent global network view. Levin et al. [54] discuss state distribution trade-offs and perform experiments with different consistency levels. Controller inconsistency can raise a wide range of problems, for example, in passive replication all backup controllers and primary controller must be in the same state, otherwise, if primary fails and a replica with inconsistent state becomes new primary, incorrect assumptions can be made by the controller and the network devices will not be properly managed. In active replication, event delivery ordering must be enforced in order to guarantee controller consistency.

D. Application/control interface

Analogously to operating systems and programs, network controllers and applications share the same execution space. Thus, countermeasures must be implemented to seamlessly handle failures in both layers. Two types of issues were identified: *application failure isolation*, related to runtime, and *control plane state fault tolerance*, related to storage.

a) Application failure isolation: In most of the available SDN controllers [52], [55], [56], applications and network OS are tightly coupled in the same execution space. This kind of architecture leads to a fate-sharing relationship between the controller and applications. If an application crashes, the controller also fails. Additionally, these controllers do not monitor application resource consumption, thus, a faulty application may exhaust shared resources, leading to a controller failure.

The network operating system design must take into account fault tolerance and reliability. Similar to the traditional operating system kernel design [57], network OS kernel architecture must be constructed considering fault tolerance and performance trade-offs. In the *monolithic kernel architecture*, all controller services are in a single protection zone, which means that a failure of one service may cause the failure of the network controller itself. Whereas, in *microkernel architecture*, independent services run on the top of a kernel and communicate among themselves using an interprocess communication (IPC) interface. The performance of microkernel architectures usually falls short of monolithic kernel performance due to its overhead caused by the IPC communication. Hybrid architectures are also proposed, addressing the fault tolerance and performance trade-off.

b) Control plane state fault tolerance: One of the SDN principles is the logical centralization of control, allowing a reduction of management complexity and network heterogeneity. Two possible approaches to achieve this are *physically distributed control* and *physically centralized control*. The former approach provides more flexibility, enabling clustering techniques and increasing resilience, however it requires coordination capabilities in the control plane. The latter option is less complex, but faces scalability and resiliency issues, since it represents a single point of failure. The usual approach is to maintain backup replicas that may take control of the network in case of failure. In both approaches, a controller failure causes loss of state. In order to guarantee service availability, control plane state must have some level of redundancy. Both *controller failure* and *application failure* may lead to state loss, thus, they must be handled separately if they do not share the same storage.

E. Application layer

One of the main goals of SDN is to provide programmability to networks, moving network management closer to software development. Thus, in order to achieve desired levels of fault tolerance, network application development must consider fault management. Issues in the application plane are related to the development of reliable network applications, during application design and guaranteeing application correctness through testing.

a) Application design: Network requirements can be enforced since the design phase of network applications. If fault tolerance requirements are not taken into account, applications may generate faults due to bugs. Considering that SDN applications belong to a specific knowledge domain, domain-specific features can be used to provide fault tolerant constructs and abstractions. Such abstractions can be used to

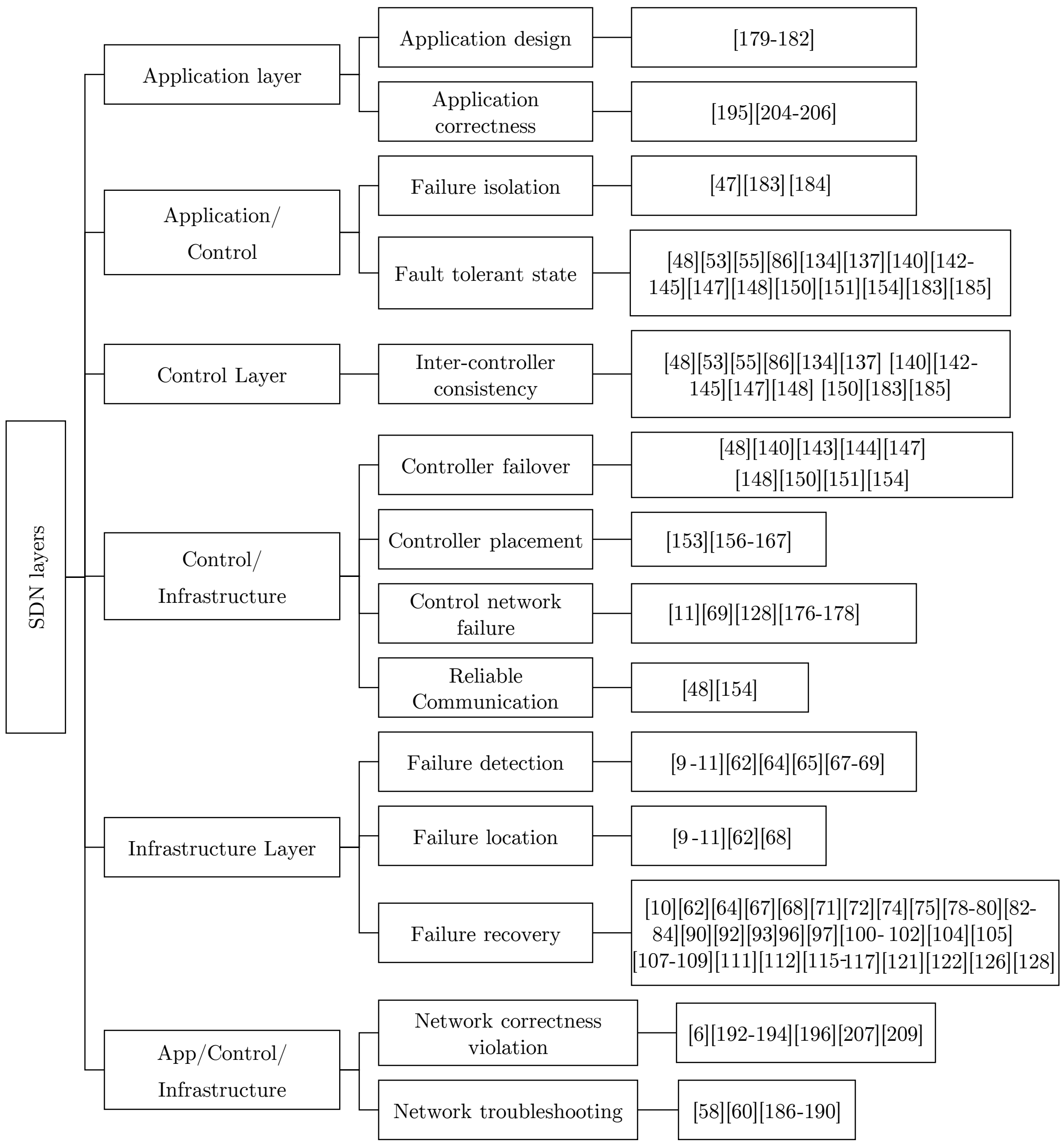


Fig. 5: Fault management issues in SDN

handle applications' points of failure, such as deadlocks, and race conditions, and to provide expressiveness to developers write high-level fault management policies.

b) Application correctness: In order to ensure that an application has the intended behavior, they must pass through a comprehensive testing routine. Application correctness can be ensured using one of the many approaches used for software verification and validation: formal verification, programming language instrumentation, black-box and white-box testing. As in application design, domain-specific knowledge can be used to provide specialized tools. Since testing must be performed before deployment and the application function does not necessarily involve the infrastructure layer (*e.g.*, application state manager, application logging), this issue is categorized in the application plane.

F. Application/control/infrastructure interface

At last, there are some pervasive issues that may affect or may be generated at any layer. Two main issues were identified: *network correctness violations* may arise from faults at any layer or interface, and *network troubleshooting* may involve all layers in order to effectively perform fault diagnosis.

TABLE I: Fault management issues in SDN

Issue	**Attribute**	**Section**	**Specific to SDN?**	**App plane**	**Ctrl plane**	**Data plane**
Network failure detection	Failure detection	V-A	No			✓
Network failure location	Failure location	V-A	No			✓
Network failure recovery	Failure recovery	V-B	No			✓
Control network failure detection	Failure detection	VI-C1	Aggravated by SDN		✓	✓
Control network failure location	Failure location	VI-C1	Aggravated by SDN		✓	✓
Control network failure recovery	Failure recovery	VI-C2	Aggravated by SDN		✓	✓
Controller-switch reliable communication	Fault masking	VI-A3	Aggravated by SDN		✓	✓
Controller placement	Failure isolation	VI-B	Yes		✓	✓
Controller failover	Failure recovery	VI-A2	Yes		✓	✓
Inter-controller consistency	Fault tolerance	VI-A1	Yes		✓	
Application failure isolation	Failure isolation	VII-B1	Yes	✓	✓	
Control plane state fault tolerance	Fault tolerance	VI-A1, VII-B2	Yes	✓	✓	
Fault tolerant application design	Fault tolerance	VII-A	Yes	✓		
Application correctness	Fault detection	VII-D1	Yes	✓		
Network correctness violation	Fault detection	VII-D2	Aggravated by SDN	✓	✓	✓
Network troubleshooting	Fault diagnosis	VII-C	No	✓	✓	✓

a) Network correctness violation: Network-wide policies and invariants are defined in high-level terms that are translated to device configuration. Eventually, the actual behavior of the network may violate policies (*e.g.* routing policies, security policies) and invariants (*e.g.* connectivity) due to some untested application bug, a network controller mistranslation, or a faulty protocol implementation in network devices. Therefore, network correctness must be constantly monitored in order to detect possible violations.

b) Network troubleshooting: When a failure occurs, network administrators must identify which applications are involved and which sequence of actions led to the failure. This is not a trivial problem, considering that in large networks a massive number of dependent and interdependent events may take place during a failure. Strategies to assist network troubleshooting include recording and replaying network events [58], determining minimal causal sequence [59] and analyzing network behavior by observing packets backtraces [60].

Summary: Figure 5 associates the fault management issues with the surveyed efforts that address those issues in the following sections. Table I lists issues according to the following features: 1) fault tolerance attributes related to each issue; 2) section in which related efforts are discussed; 3) whether legacy networks are affected by the same problem and whether the problem is aggravated by SDN and 4) planes associated with the issue. Besides fault tolerance attributes already discussed, we also use *fault masking* to classify issues. Fault masking is the ability to a network seamlessly ignore faults [61]. We deem as *fault tolerance* those issues related to overall network fault tolerance (*e.g.*, application design). Control plane state fault tolerance is discussed both in control plane and application plane sections, Sections VI and VII, respectively. Section VI describes how control plane architectures manage their state and Section VII describes network applications that provide fault tolerance to the control plane state.

V. Data Plane Fault Management

Unlike application plane and control plane fault tolerance issues, which were mostly related to SDN architecture and its characteristics, fault tolerance in data plane is related to issues already present in traditional networks. However, SDN sheds a different light on those issues, enabling novel strategies and bringing new challenges. For example, upon link failure, centralized network view allows to reason about network behavior and workload in order to calculate an optimal path or locate where the failure occurred, but, it also bounds recovery time to control channel latency. Most methods focus on providing failure detection and failure recovery to a wide range of network scenarios.

A. Failure detection

Most network devices have native link failure detection, sending a *failure* notification to the controller in case of failure and performing failover to a backup path, if any is configured. However, research has demonstrated that if failure recovery relies only upon built-in mechanisms of commercial switches, recovery time may require hundreds of milliseconds [11], [62]. Also, more sophisticated methods are required to detect path failures and quality degradation without connectivity loss.

Sharma et al. [62] leveraged SDN to support standardized failure detection, such as Bidirectional Forwarding Detection (BFD) and Loss-of-Signal (LoS). Loss of Signal detects link failures by checking whether a specific port of a device is down, while BFD [63] detects failures along paths, by establishing a session between two end-points, and then they both agree on their sending and receiving intervals. If an endpoint stops receiving BFD packets for long enough, the path is declared failed.

In order to increase the speed of failure detection, Adrichem et al. [64] defined a BFD session per link, instead of per path, and modified *Open vSwitch* implementation to monitor BFD status in real-time.

Kempf et al. [65] provide failure detection in end-to-end paths for transport networks by moving part of the control plane to forwarding devices and using MPLS OAM, which uses MPLS BFD [66]. To detect failures along a path, packet generators are implemented at switches, sending probe packets interleaved with data packets. The egress node of a path detects a failure if consecutive probe packets are missing or if it indicates a failure along its path. This method enables scalability by allowing different network entities use the same packet generators, later separating the packets using MPLS.

Gyllstrom et al. [67] propose *PCount*, a failure detection method. *PCount* monitors link connectivity installing rules to tag all outgoing packets and periodically checking how many packets were received at destination switch. If the error rate is higher than a given threshold, the algorithm assumes the link has failed.

Liu et al. [68] present an SDN elastic optical network (SDN-EON) architecture and propose a failure recovery and detection method. An OpenFlow agent periodically checks the bit-error rate (BER). If BER is higher than a specified threshold, the link is assumed to be failed. If the failed node or link belongs to a segment, all subsequent agents will send alarm messages to the controller. To isolate the failed node, the controller checks if the segment belongs to a lightpath connection and localizes the origin of failure in the first link of the failed segment. Failure detection time is bounded by a δt timeout, that guarantees that the controller will receive all alarm messages.

Lee et al. [10], [11] propose a failure detection method that seeks to reduce recovery time. Monitoring circle paths are responsible for detecting and locating link failures in networks with out-of-band management. In order to minimize the number of monitoring cycles and hops, monitoring cycle assignment is modeled as an instance of the *min-max k-Chinese postman problem*, which finds k closed tours of a graph where each tour starts and ends at the same vertex and all edges are covered. Binary search is used to find the smallest k, and then a heuristic [70] is implemented to perform the assignment.

Kozat et al. [9] also use monitoring cycles to detect and locate failures. The authors explore the possibility of asymmetric failure (e.g. if only one direction of link communication fails) installing flows for both directions. In order to minimize the overhead of control messages and detection time, binary search is used to locate the failure. At each iteration, a packet is sent with a different returning point such that half of the location candidates can be pruned.

Summary: Logically centralized control makes it easier to reason about link failure notification, locating failures and restoring connectivity; however, even an efficient recovery mechanism time is bounded by failure detection time. Also, as the network complexity scales up, failure detection and location become more difficult, as the monitoring traffic overhead may impair network performance. These issues are addressed moving failure detection closer to the data plane [67], minimizing the number of monitoring packets [9] or limiting path length [11]. Ideally, failure recovery mechanisms must coordinate with failure detection and failure location methods, for example, a control channel failure may prevent the controller from receiving failure notification until the control connectivity is restored. Another important factor is the different network features (e.g. in-band control, multiple controllers) and scenarios (e.g. multi-link failures, control channel failures) that must be covered by detection mechanisms. Table II compares failure detection efforts according to their scope and whether they provide failure location, along with a brief description.

B. *Failure recovery*

After a network failure is detected a failure recovery method must be employed in order to restore network connectivity. There are two main link failure recovery strategies: *protection* and *restoration*. In failure protection, backup paths are configured proactively through the installation of flow rules in the switches. In failure restoration, the controller acts upon the link failure notification, calculating an alternative path and installing flow switches. Both methods can be used to achieve different requirements, such as reduced recovery time, meeting bandwidth guarantees and minimizing TCAM consumption. They can be also applied to provide fault tolerance to a number of different network environments: Optical networks, data center networks, virtual networks, hybrid networks with legacy and SDN devices and/or coexisting protocols. In this subsection, various methods were organized according to which network requirement or environment they address. Due to space constraints, the comparison between failure recovery methods was separated into two tables. Tables III and IV summarize and compare failure restoration and protection methods, respectively.

1) Recovery time: One of the main goals of network fault management is to minimize the amount of network downtime. There are different approaches in SDN to reduce network downtime, which vary with respect to the part of recovering process that they optimize.

a) Failure protection: The most simple failure protection method is to compute a primary path and a backup path for each flow and install respective flow rules in switches. Since OpenFlow 1.0 does not provide failover capabilities, the early efforts extend or leverage OpenFlow to enable path protection.

Sgambelluri et al. [71], [72] propose a mechanism where the controller precomputes backup paths, installs flow rules with low priority along primary path to redirect traffic. Since the flow table has a flow expiration timeout, as long the backup path is active, the switch generates a flow renewal packet to avoid expiry backup flow entries. Once the failed link is restored, the switch notifies the controller, which must reinstall former working path. When the switch detects a link failure on the working path, affected switches automatically deletes rules that use the failed link. OpenFlow protocol was extended

TABLE II: Failure detection methods comparison

Method	Scope	Failure location	Description
LoS [62]	Link failure	Out-of-band	Checks whether a device port has failed
BFD [62]	Path failure	N/A	Establishes a session between two end-points and sends monitoring packets between them
Adrichem et al. [64]	Link failure	N/A	Establishes a BFD session for each link along the path
Gyllstrom et al. [67]	Link failure	N/A	Checks error rate between two devices
Kempf et al. [65]	Path failure	N/A	Packet generators send packets along MPLS BFD sessions
Liu et al. [68]	Path failure	Out-of-band	Assumes link failed if BER is above a threshold
Kotani et al. [69]	Control path failure	N/A	Sends probe messages between controllers and switches at selected events
Lee et al. [10], [11]	Control path failure	In-band	Sends packets through monitoring cycles
Kozat et al. [9]	Path failure	In-band	Sends packets through monitoring cycles and performs binary search to locate failures

with a `OFPT_FLOW_RESTORE` message, sent by the switch to notify the controller upon primary path restoration.

Sharma et al. [73], [74] discuss how SDN can meet carrier grade reliability requirements (*i.e.* recovery time <50 *ms*). They move fault management to control plane aiming to provide fast connectivity recovery after a link failure is detected. When the controller receives a link failure event from a switch, it checks whether it will impact installed paths; if necessary, it recalculates affected paths and sends respective flow modifications to switches. BFD and LoS are used to detect failures. Experiments demonstrated that SDN is able to achieve carrier-grade recovery time.

In a later work [62] they propose a protection method that makes use of OpenFlow 1.1 *fast failover* group type to handle link failures. The controller computes disjoint-paths for each flow request and installs protection rules in switches.

b) Fast detection: Adrichem et al. [64] aim to reduce recovery time with a fast failure detection, which will be explained in section V-A, and a failure protection mechanism that handles single-link failure with precomputed backup paths installed using OpenFlow *fast failover* group type. If there is no backup configured, switches use reverse forwarding until achieving a feasible path.

c) Backup path header encoding: Gonçalves et al. [75] use *fast-failover* group type and OpenvSwitch *learn* feature to provide protection and reduce recovery time in data plane. *Learn* feature allows switches to apply a different forwarding action upon receipt of a previously sent packet, optimizing paths without the need to contact network controller. The authors discuss the trade-off between two link recovery approaches, a routing algorithm that prioritizes disjoint paths and the other one that allows overlapping between primary and backup routes. The latter achieved lower response time since overlapping paths enable packets to traverse fewer nodes.

Ramos et al. [76] propose SlickFlow, a link protection approach that leverages OpenFlow to support encoding of alternative paths in packet header. The controller calculates a primary path minimizing latency and a disjoint backup path, and then, encodes them in the packet header along with an *alternative* bit, which signifies the path being used. If a switch detects a failure in the primary path, it forwards the packet through the backup path, changing the *alternative* bit to indicate to the subsequent switches, which one must be used. Since existing headers are used to encode backup paths, they have a limited number of hops (*i.e.*, 16 hops).

d) Local failover: Loop-free alternates (LFAs) [77] are a standardized link protection method used for fast IP reroute in which, a backup route for a link failure scenario preserves loop-freeness to avoid loop scenarios. Braun and Menth [78] leverage OpenFlow to support LFAs enabling fast recovery from multiple link and node failures. Backup paths are ordered according to its level of loop protection (*i.e.* number of failure scenarios which may result in loops), and then they are configured in switches using *fast failover* group type. To avoid loops, visited nodes are encoded in packets' header. If a node is visited twice, a loop is detected and packets are dropped.

e) Record and replay: Kuźniar et al. [79] propose a restoration mechanism called AFRO, a failure recovery mechanism that aims to generate a valid controller and switch state after a failure occurs. The AFRO operation has two phases: *record* and *recovery*. During the record phase, AFRO records all packets directed to controllers and flow modifications are performed on switches. When a link or device failure occurs, AFRO enters in a recovery mode. A copy of the network, without the failed elements, is emulated and the network events are replayed, achieving a valid state that can be used by the actual network. Resulting switch configuration is replicated using flow modifications and the controller state is transferred or the resulting controller itself takes control of the network.

f) Graph search: In most restoration methods, the switch requests a backup path from the controller, bounding the recovery time with control channel latency and path computation. In order to avoid such issues, some restoration methods move failure recovery to data plane. Liu et al. [80] delegate the basic connectivity to data plane mechanisms, while retaining the path optimality computation in the control plane, ensuring fast failover while guaranteeing eventual path optimality. In this method, all possible paths to a destination are modeled as a directed acyclic graph (DAG), where the destination is a *sink* node (*i.e.* does not have *outgoing* links). Upon failure communication, the disconnected node reverses all links, which were not yet reversed, based on a partial reversal version of Gafnis-Bertsekas algorithm [81], and further repeating this operation in subsequent links while guaranteeing that a path will be eventually found. In parallel, the control plane may calculate optimal paths, since link reversal path only guarantees connectivity.

Borokhovich et al. [82] model the link failover (*i.e.* forwarding a packet to a destination) as a graph search problem. Three algorithms are applied: 1) *Modulo* algorithm, that forwards the packets to switch ports in a round-robin fashion, until a packet reaches its destination; 2) *Depth-first search*, that tries to forward the packets to the next hop until it reaches a failed link, and then it sends the packet back to its parent; 3) *Breadth-first search*, that sends the packet immediately back to its parent, forwarding it only when all neighbors of its parent have been visited. In all these methods packet header is used to indicate the nodes traversed. Based on classic graph search algorithms, the implemented approaches guarantee that the packet will reach its destination.

2) Bandwidth guarantees: Some network services demand that a minimum bandwidth is guaranteed in order to comply with SLAs or support real-time critical applications. In the presence of network failures, local failovers frequently lead to high-load scenarios [83]. Some SDN failure recovery services support recovery according to traffic priority and strategies to mitigate network congestion.

a) Quality of Service: Sharma et al. [84], [85] propose a recovery mechanism that supports different levels of quality of service. Failure recovery is performed considering the traffic priorities (*i.e.* control, business or best-effort traffic) and assigning flows to switches and guaranteeing connectivity accordingly.

Phemius and Bouet [86] investigate resiliency between WANs and propose a resilient traffic engineering application that provides fault tolerance while meeting QoS requirements. Upon link failure, a traffic engineering service calculate a new path, considering two priority levels of flow: *critical* or *non-critical*. If a link failure occurs in the path used by a critical flow, a new path can be assigned for any path, including those used by lower priority flows, interrupting their streams.

b) Congestion: During failure restoration, flow assignment must take link congestion into account, otherwise, the same links can be used by multiple flows, leading to network congestion. Harry et al. [87] propose a traffic engineering method that guarantees congestion-free traffic in networks with up to *k* arbitrary faults. In this method, a traffic engineering controller calculates how much bandwidth must be reserved to guarantee congestion-free traffic, given *k* faults, and then, assign traffic among switches reserving required bandwidth.

Paris et al. [88] also aim to provide fast failure recovery while optimizing network usage through network reconfiguration. The proposed architecture has two main components: Fast Recovery Setup (FRS), responsible for quick computation of backup paths, only considering connectivity, potentially non-optimal; and Network Garbage Collector (GC), that performs periodic or event-based network reconfiguration. They formulate network reconfiguration as an extension of the NP-complete *min-cost Multi-Commodity Flow* problem. GC solves it by applying an extension of the simplex method iteratively for each source-destination demand, in order to find a flow that satisfies active constraints at minimum cost.

Network events are modeled such that failure events increase the optimality gap (*i.e.* distance to optimal solution), since it requires fast connection setup, without optimization, thus, requiring network reconfiguration in order to reduce optimality gap. However, network reconfiguration takes time and cause disturbances in performance. Paris et al. analyze this trade-off and propose an optimization model to limit network changes without compromising optimality.

In path protection, it is difficult to guarantee congestion-free traffic during failure scenarios, since there is no prior information. Lee et al. [10] address this issue by assigning backup paths according to ingress and egress ports. Based on *Interface Specific Forwarding*(IFS) [89], flows with different interface sources and node destinations will have different backup paths. Then, each single-link failure scenario is considered and paths are assigned while maximizing load balance.

Borokhovich and Schmid [83] prove that fast local failover techniques can tolerate at most $n-1$ failures before disconnecting source-destination pairs, where n is the number of nodes and the network is a full-mesh. They also prove that failures lead to high load scenarios even though is available bandwidth. They propose two methods that achieve an almost optimal tradeoff: random failover scheme (RFS), that precomputes loop-free random paths for source-destination pairs, and deterministic failover scheme (DFS), that proceeds as RFS and checks whether the candidate solution leads to a high load scenario, if yes, another solution is generated.

Cascone et al. [90] rely on OpenState [91] to leverage OpenFlow to address efficient failure detection and recovery, and backup path planning [92]. OpenState provides stateful data plane abstraction extending OpenFlow pipeline with an extra *state table*. Packet matching is performed in two steps. First, packets are matched against the state table, and then, according to its associated state, matched against *flow table*.

Based on OpenState, failure detection and recovery [90] are realized according following steps: 1) Backup path pre-planning module computes disjoint paths for all F_i, one-link *failure states* for each network element i. Those states are stored along the backup path nodes. 2) Heartbeat messages are constantly exchanged between devices to quickly detect failures. 3) When node j detects that node i is unreachable, incoming packets are labeled F_i. If node j is responsible for F_i it changes its forwarding state and sends all packets to the backup path. Otherwise, it sends packets back to their incoming port, until they find the node that keeps the F_i state.

Later, Cascone et al. [92] further investigate backup path calculation, meeting different requirements. Two optimization models, along with their associated constraints, are proposed: 1) Backup path length and path reuse optimization model and 2) Link-congestion avoidance model.

Sahri and Okamura [93] also propose a failure recovery mechanism that handles connectivity and optimality separately. Initially, controller pre-installs backup paths with low priority at switches. Switch behavior was modified to automatically timeout flows using a failed link in case of link failure. Pre-installed rules are used to avoid loss of connectivity, while network controller computes an optimal path.

3) Switch memory consumption: In path protection, since rules need to be preinstalled, the number of rules installed may escalate quickly, leading to switch memory exhaustion. Intensive operations of data center networks demand to process of a high number of flows; studies observed up to 10,000 network flows per second per server rack [94]. Additionally, flow entries are longer than other descriptors, since they encode match fields and actions. For example, an OpenFlow 1.0 flow entry with a 10-field header have 288 $bits$, while an Ethernet forwarding descriptor has 60 $bits$ [95].

a) Rule generation minimization: Kitsuwan et al. [96] aim to reduce the number of backup flows required to provide connectivity. Initially, the controller computes a routing tree with the destination at the root and installs flows with low priority in switches. When a switch detects a failure, it modifies flows to use a backup link, generating a new routing tree with minimal change. Evaluation showed a reduction of number of flows needed for protection, compared to Sgambelluri et al. [71].

Kim et al. [97] aim to simplify route computing using VLAN paths instead of physical paths. Applications running in control plane slice the network in different VLANs, where switch ports are mapped to different VLAN IDs, leading to a reduction in number of flows since flow rules will specify a logical path implemented by VLAN instead of the physical path. Upon failure notification control plane assigns a random VLAN path in order to balance workload.

b) Rule compression: Stephens et al. [98] propose a forwarding table compression algorithm along with a compression-aware routing mechanism, that generates more easily compressible forwarding tables. The primary requirement to compress flow rules is that they should have the same output and packet modifications, which depends on the forwarding model used. Table compression is leveraged using Plinko forwarding model [99], where all switches traversed by a packet are retained in its header. Packets that traversed the same switches have the same output path. Initially, table compression algorithm sorts rules based on the size of the set of rules with same output and action. Then, entries are greedily merged masking off bits in which they differ. During backup path selection, routing mechanism first searches the set of active backup paths for a path that can be reused. Path selection maximizes path reuse by choosing paths using breadth-first search (*i.e.* all possible t-resilient paths are constructed before (t+1)-resilient paths), which generates more reusable paths.

Mohan et al. [100] propose a routing mechanism that minimizes TCAM consumption. Two routing strategies are implemented: *Backward Local Rerouting*(BLR) and *Forward Local Rerouting*(FLR). In BLR, for each installed flow, a node-disjoint path is calculated and upon failure along the primary path, packets are sent back to their origin and rerouted to the backup path. In FLR a 2-phase algorithm is applied. First, for each link in the primary path, FLR determines the backup path that traverses minimum number of additional nodes. Then, FLR resolves conflicting rules (*i.e.* different match fields with same output port) by choosing one rule that complies with both paths and eliminating others. In order to reduce TCAM usage, instead of using *fast failover* group type, OpenFlow protocol was extended to add a smaller entry in action set called `BACKUP_OUTPUT`, which specifies an output port in case of failure.

4) Network environments: Leveraging SDN to support fault tolerance mechanisms in different network environments has been proposed by various works:

a) Hybrid: Yu et al. [101] address OSPF slow link failure recovery by coordinating control plane with legacy networks control capabilities in OpenFlow-enabled switches (*i.e.* support OpenFlow protocol but still have routing and control capabilities). During normal operation packets are handled by RIB. When a link failure is detected by the switch, controller recalculates paths using Floyd algorithm and install flow rules. As soon as links are operational again, RIB takes control of packet forwarding.

Zeng et al. [102] evaluates resiliency aspect of a Software Defined Routing Platform (SDRP), namely RouteFlow [103], in comparison with legacy networks. It was demonstrated that RIPv2 and OSPF performance in SDRP are comparable to legacy networks, with the latter achieving shorter failover time than its legacy counterpart.

Gil et al. [104] propose a fault tolerant architecture for hybrid networks, which is composed of an SDN controller, responsible for providing an interface to configure network infrastructure and a *Software Defined Network Operations Center* (SD-NOC) which manages the network exchange information along with the SDN controller, while configuring SDN and non-SDN devices. The hybrid network is a combination of circuit-oriented network and packet-switching, with SDN and non-SDN devices. Upon link failure, the SD-NOC calculates and installs a new path. In case of SDN controller failure, SD-NOC might take control of the network.

Chu et al. [105] propose a hybrid approach where OpenFlow management coordinates with distributed protocols to perform failure recovery while considering network load-balance. Chu et al. argue that efficient failure recovery can be achieved even if only a group of SDN switches is placed in the network. They propose a greedy heuristic that, given a network topology, defines the minimum number of SDN switches, their locations and to which legacy network devices they must be connected to guarantee single-link failure recovery. Each legacy device is connected to a designated SDN switch through an IP tunnel. When the device detects a link failure, it will encapsulate and forward all packets to designated SDN device. Backup paths are calculated periodically and selected to minimize maximal link utilization.

Markovitch et al. [106] also argue that in order to increase network flexibility, only a small part of the network devices must be SDN-compliant. In their approach, the network is decomposed into multiple loop-free components and SDN devices are placed at monitoring locations capable of accurately observe network changes. Each SDN device is responsible for a network segment and receives network updates through *multiple spanning tree protocol* (MSTP) updates. If the controller receives a link failure, it uses information contained in the MSTP message to quickly locate the failure and re-route all affected traffic around the failure. Simulations demonstrated that only 2% to 10% of network devices must be SDN-compliant to implement their approach.

Proactive strategy may not scale multiple failures without high demand of memory while reactive approaches are bounded to control plane detection and mitigation. As an alternative to both methods, Tilmans et al. [107] propose a hybrid

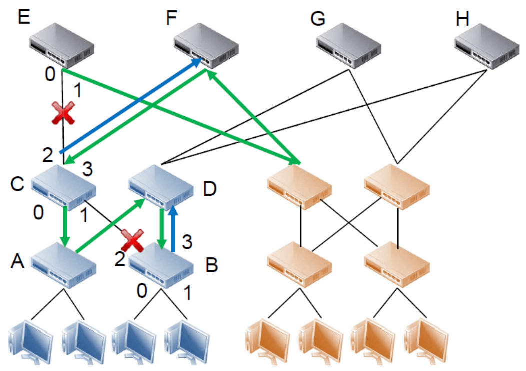


Fig. 6: Link failover for the uplink routing path is denoted using a blue arrow line; for the downlink routing path, it is denoted using the green arrow line. [108]

SDN architecture that uses a centralized network controller, which provides primary forwarding and enforce policies, and in case of failure falls back to Interior Gateway Protocol to guarantee basic connectivity. IGP-as-a-Backup SDN (IBSDN) architecture is composed by an IBSDN controller and local routing agents run on top of OpenFlow-enabled switches. During normal operation, IBSDN applies high level policies in the network installing their correspondent rules, while local agents run IGP (e.g. OSPF, RIP, IS-IS) keeping forwarding paths in local agent memory. In case of failure, IGP forwarding paths are applied to the switches and tagged to indicate that they are IGP packets, which is used by local routing agents to identify an occurrence of failure and replace primary rules.

b) Data center: Li et al. [108] propose a local optimal path recalculation strategy focused on fat-tree networks. In case of failure, connectivity of uplink is restored assigning flows to the higher level link with lowest load. Downlink restores communication by redirecting its flows to a lower level switch that has access to a higher level switch that might send to a backup link connected to the original destination. An example of traffic flow configuration after link failures is depicted in Figure 6. Since fat-tree topologies include link redundancy between levels, there must be another higher level switch connected to the original downlink destination. This approach aims to provide fast path recalculation while minimizing the number of switches affected, reducing the number of update messages generated and also reducing configuration time.

Araújo et al. [109] propose a resilient architecture called INFLEX that integrates transport and network layers to leverage end-to-end fault tolerance. INFLEX architecture is composed of three main elements: *end-hosts*, *edge switches* and an *inflector controller*. When end-hosts detect a failure (e.g. retransmission timeout), the transport layer requests a new path by setting a *inflection request* flag in the IP headers of all outbound packets. The edge switch process packets, matching packets against flow rules and policies. If there is an inflection request and no matching rule to an alternative path, packet is sent to the inflector. The inflector is a controller that processes inflection requests and assigns forwarding planes to flows. INFLEX allows the configuration of multiple routing planes, each with its own routing paths. Since INFLEX demands a total control over the network stack, from host to switches and controllers, its application is suitable for data-center networks.

c) Multicasting protection: In multicast communication, a node must send information concurrently to multiple nodes. To avoid unnecessary network resource consumption, the number of links must be minimized and packets only cloned during path diversion.

Kotani et al. [110], [111] tackle IP multicasting fault tolerance. Two disjoint multicast trees are calculated: 1) a primary tree, applying Dijkstra's algorithm on the multicast group graph, and 2) a disjoint backup tree. These rules are preinstalled in switches for path protection. Switches rewrite packet headers to identify the routing tree being used by a given flow.

Pfeiffenberger et al. [112] model multicasting communication as an instance of NP-hard *Steiner tree problem*, where groups of nodes in a graph must be connected using the minimum number of links (unlike *minimum spanning tree*, virtual nodes can be added), and implements an approximation algorithm [113]. The algorithm constructs a complete graph including all nodes of a multicast group, and calculates its minimum spanning tree.

Fault tolerance is achieved by repeating the approximation algorithm considering the failure of each link. Rules with backup paths are installed using *fast failover* group type to protect links. Each multicast route assigns an ID in the VLAN field to guarantee end-to-end communication.

Gyllstrom et al. [67] use a similar approach to address efficient multicast fault-tolerance. They aim to maximize path reuse to minimize control overhead and number of flows in switches. Path reuse problem is modeled as an instance of Steiner arborescence problem [114], a modified version of Steiner tree problem, which must find the minimum weight directed tree, rooted at r and spanning to S nodes. Primary paths are computed using an approximation algorithm [114]. *Bunchy* algorithm computes backup paths considering one-link failure scenarios and decreasing weights of links used in the primary path, thus, optimizing path reuse. After rules are computed, a *Merger* algorithm unifies rules with the same outgoing port and installs them in switches.

Raja et al. [115] propose failure detection and protection for multicasting trees. A multicasting tree represents communication flow with the root being the *sender node*. Failure detection and restoration are performed in four steps: 1) Subtrees of multicasting trees are computed, where a node with more than one child node is a root of a subtree. 2) LLDP protocol detects failures and the controller is notified. 3) The controller identifies in which node and subtree the failure has occurred 4) Restore connectivity installing flows that connect affected nodes to the closest functional subtree root placed at higher level.

Nagano and Shinomiya [116] uses fundamental tie-sets, minimal cycles in a graph, to guarantee network connectivity. Initially, primary path is defined as a network spanning tree and the controller computes all network tie-sets(*i.e.* every minimal cycle including links not used in the primary path). Upon failure notification, the controller locates the link failure, chooses the minimum tie-set that includes a link that will restore connectivity, and install corresponding rules in the switch.

d) Inter domain: Most of distributed controllers assume that network communication latency is low enough to be negligible. For most of the practical scenarios, this limits clusters to include only controllers that are geographically close. ICONA [117] is an ONOS application that enables geographically distributed control by providing inter-cluster communication and fault tolerance.

ICONA sees each geographical region as one domain controlled by an ONOS cluster. ICONA runs on top of each ONOS, applications that manage inter-domain operations: 1) Topology manager responsible for discovery, storage of state and propagation of network events and its own state; 2) Service manager in charge of inter-domain services, namely MPLS VPN and L2 pseudo-wire tunnels 3) Backup manager that handles failures of inter-cluster links. Backup inter-cluster links are pre-computed and activated by the backup manager in case of failure.

e) Optical networks: SDN focused mainly on enterprise networks right from its inception and has been gradually deployed in a wide range of networked environments, such as optical networks.

SDN leveraged flexible management in optical networks while integrating features as elasticity [118] and QoS-awareness [119]. A number of efforts were aimed to provide fault tolerance while benefiting from SDN capabilities.

Liu et al. [68] present an SDN elastic optical network (SDN-EON) architecture resilient to link degradation and failure. Upon link degradation, the controller reassigns flows to a new modulation to improve optical signal-to-noise ratio. In case of link failure, a backup path is calculated and, according to its characteristics, a new modulation format may be assigned. OpenFlow protocol was extended to cope with EON features and an OpenFlow *alarm message*, used to notify controllers. This work was later extended [120], optimizing backup path selection and reducing recovery times by precomputing shortest backup paths for each node pair and storing them in a lookup table. Then, upon failure notification, the controller checks for a path in the lookup table, with preference on paths having data rates similar to the rates used previously.

Giorgetti et al. [121] compares different OpenFlow-based restoration strategies and GMPLS/PCE standard restoration. Two OpenFlow-based restoration methods are proposed: 1)*SDN-ind*, a *flow modification message* is sent to switches after each backup path is computed; 2)*SDN-bund*, if there are multiple link failures, the controller sends a *flow modification bundle message* consisting of all backup paths to the affected switches, only requiring a single hardware reconfiguration. OpenFlow protocol is extended to support `BUNDLE_FLOW_MOD` type message. Both methods are compared with GMPLS/PCE method, which uses RSVP-TE messages to release and reserve paths, and Path Computation Element(PCE) to calculate paths.

Yang et al. [122] provide path protection to software defined elastic optical network architecture. The architecture proposed, based on [118], is composed of OpenFlow-enabled bandwidth-variable optical switches with OpenFlow agent daemon running in them and a controller with a link protection mechanism. OpenFlow protocol was extended to represent optical networks features (*i.e.* central frequency, spectrum size) and an additional *type* field that distinguishes working and backup paths, along with their priorities. A path calculation module installs primary and backup disjoint paths, assigning more backup paths to more important services. When a failure is detected by an optical performance monitor, controller is notified, and then primary path flow rules are removed only from source and end nodes, since the paths are disjoint. Backup paths are allocated using bandwidth squeezed protection [123], where spectral efficiency is achieved by allocating minimum required bandwidth to backup paths.

f) Wireless networks: One of main challenges of software defined wireless networks is to maintain connectivity between controller and devices, since in wireless networks a number of external factors may affect connectivity, being more dynamic than wired environments. Detti et al. [124] propose a wireless mesh SDN (wmSDN) that relies on Optimized Link State Routing (OLSR) protocol to preserve network functionality upon a control plane connectivity disruption. Control and data traffic are separated into two different subnets which are managed by two independent levels of controllers, a centralized controller for data traffic and local controllers responsible for control traffic. Each wireless mesh router (WMR) runs an OLSR daemon that monitors network topology and maintains a dummy IP routing table. If the OLSR daemon detects a controller disconnection, all flow rules inserted by the controller are removed and replaced by the rules of OLSR IP routing table. Then, the routing becomes controlled by OLSR protocol.

In [125] , they extend wmSDN to support multiple controllers in order to increase network resiliency. Salsano et al. argue that a master election is not suited for wireless mesh networks due to link unreliability and topological changes. Instead, each WMR is responsible for its own master selection. During normal operation, WMR periodically checks which controllers are reachable. Following controller disconnection, WMR connects to the next reachable controller, according a globally defined list. This method does not take link load into consideration, which may lead to controller overload.

g) Virtual networks: Network virtualization is one of the main SDN applications, however, there are few efforts addressing virtual network resilience in SDN context. Wang et al. [126] address virtual backup path provisioning efficiency presenting two methods to calculate backup paths by minimizing bandwidth consumption. When the hypervisor receives a virtual network (VN) request, it must compute a primary and backup path. Two methods are proposed for backup path calculation, *OBT-I* and *OBT-II*. In OBT-I, the problem is modeled using ILP where the objective function minimizes the total amount of bandwidth constraints and it also supports path splitting (*i.e.* a virtual link can be mapped to multiple paths [127]). In OBT-II, backup path is computed generating a spanning tree that satisfies bandwidth requirements.

Summary: Link failure recovery methods differ in a number of aspects: backup path computation and dissemination strategy; switch memory consumption; network congestion avoidance strategy; in which plane they take place. Each aspect leads to different trade-offs and the selection of an approach affects the scenarios and requirements that can be met, e.g. preinstalled backup paths require more switch memory usage and thus, path protection is not suited for data center networks, since they might have thousands of possible paths [108].

Some methods focus on specific scenarios to optimize failure recovery for them, such as INFLEX [109], which leverages the total control of the data center over the whole network stack, ranging from hosts to switches and controllers, enabling a failure detection and backup path encoding at host level.

Cascone et al. [90] argue that time critical features, as failure recovery, must be kept in the data plane, eliminating the need to contact controller, consequently bounding recovery time to control channel latency and path computation. This is inherent to failure protection methods, whereas in failure restoration this can be achieved by moving the control plane path computation capabilities to switches. Among other issues, it is difficult to effectively implement load-balancing strategies in path protection because there is no information about the network load at the time of flow installation. Some approaches assign flows to different backup paths according to source interface [10] or expected link load [83] in order to distribute load among links upon network failures. However, unforeseen load scenarios may still lead to link congestion.

Most of path protection methods only guarantee single-link failure protection [83], [105]. In link protection methods multi-link failure support is more frequent [78], [115], however, Borokhovich et al. [83] demonstrated that local failover might lead to potential network congestion and loops, despite backup path alternatives that may be still available.

A usual approach to *connectivity-optimality* trade-off is to place connectivity functionality in the data plane and path optimality at the control plane [80], [88], [96], since at data plane level it is difficult to quickly find an optimal solution without a centralized network view.

SDN proved itself suitable for a wide variety of network environments, being able to provide efficient failure recovery mechanisms, adding flexibility to current solutions [102], [121] and leveraging new approaches [90], [92], [110], [122].

C. Southbound protocol support

In order to optimize performance and standardize basic features, communication protocol between controller and network devices as well as switch processing must support fault handling features.

1) Protocol behavior: Earlier versions of OpenFlow did not specify any actions to be taken upon network or controller failures, leading some of the efforts to extend protocol functionality to support link protection [72] and controller failover [53].

OpenFlow 1.1 [20] allows specification of *fast failover group* type, *i.e.*, actions that must be taken upon the failure of a primary forwarding action, such as, forwarding packet to another device port. OpenFlow 1.2 [51] introduces multiple controller management through definition of *roles*. Each switch can assign the roles of *master* and *equal* to the controllers connected to it. These can manage the switch without restrictions, though, only one controller can be the master at any given point of time, whereas any number of controllers can be configured as equal. *Slave* can only read switch state, and does not have permission to write in its flow table. Both features can be combined in order to realize controller and link failover upon network failures.

In order to guarantee controller-switch communication reliability Katta et al. [48] extended OpenFlow protocol to support the exchange of acknowledgment messages between controller and switch and buffer operations. Sgambelluri et al. [71] modify switches to send `OFPT_FLOW_RESTORE` messages to controller in order to indicate that a link is up again, while updating the controller network view.

Liu et al. [68] and Yang et al. [122] extend OpenFlow match fields to meet optical networks requirements. Extra match fields include central frequency, spectrum size and modulation format information. Instead of port monitoring, Liu et al. [68] notify controllers if the *bit-error rate* is greater than predefined threshold, by sending a `OFPT_ALARM` message.

OpFlex [29] is an SDN southbound protocol that gives more intelligence to network devices, through a distributed control system architecture based on a declarative policy information model. In OpFlex architecture, policy definition and network view are separated in different logical components. The *managed objects*(*i.e.*, physical or virtual devices) detect policy faults and raise exceptions to the *Observer*, the logical component responsible for network monitoring.

2) Header rewrite/repurpose: Many fault tolerance efforts use packet header to provide network information to the switches in order to avoid controller communication. In most of these works, existing header fields are repurposed to store forwarding information. For example, Kitsuwan et al. [96] use `priority` and `tag` fields to indicate which forwarding plane is currently being traversed.

Current efforts may rewrite headers because of the following reasons: *path searching signalling*, where packets are tagged to indicate that they are searching for a viable path [82], [109]; *path traversed encoding*, where the path already traversed is encoded in packet headers to avoid black holes [78], [98]; *forwarding plane assignment*, used by switches to check which forwarding rule is correspondent to the path being traversed [67], [96], [109], [111], [112].

VI. CONTROL PLANE FAULT MANAGEMENT

In this section we discuss current efforts to tackle control plane issues discussed in section IV. During our research, we observed that most of control plane architectures deal simultaneously with many issues related to control plane failures. Thus instead of presenting approaches separated by issues, we grouped different control plane architectures and described how they address control plane state redundancy and consistency, controller coordination after failures, and communication with network devices. We also compared controller placement strategies and how they affect network

TABLE III: Failure restoration methods comparison

Authors	Plane	Scenario	Path set up	Scope	Method
Sharma et al. [84], [85]	Control Plane	-	On-demand	Path	Shortest path considering QoS
Yu et al. [101]	Control Plane	-	On-demand	Path	SDN computes path until OSPF converges
Phemius et al. [86]	Control Plane	-	On-demand	Path	QoS
Harry et al. [87]	Control Plane	-	On-demand	Path	Reserve bandwidth to avoid congestion
Kuzniar et al. [79]	Control Plane	-	On-demand	Path	Record & Replay
Paris et al. [88]	Control Plane	-	On-demand	Path	Fast recovery with periodic optimization
Nagano et al. [116]	Control Plane	-	Pre-computed	Link	Use tie-set to perform recovery
Liu et al. [80]	Data Plane	-	On-demand	Link	Reverse forwarding
Borokhovich et al. [82]	Data plane	-	On-demand	Link	Graph search
Kim et al. [104]	Control Plane	-	Pre-computed	Path	Reduce number of flow rules using VLAN paths instead of physical paths
Watanabe et al. [128]	Data plane	Control channel	On-demand	Link	OSPF
Gil et al. [104]	Control Plane	Hybrid	On-demand	Path	SD-NOC and SDN controller coordinate to perform failover operations
Araujo et al. [109]	Data Plane	Data Center	On-demand	Path	Forwarding planes encoded in packet header
Li et al. [108]	Control Plane	Fat Tree	On-demand	Link	Redirects to uplink with controller connectivity
Zeng et al. [102]	Control Plane	Hybrid	On-demand	Path	OSPF
Gerola et al. [117]	Control Plane	Inter-domain	On-demand	Link	Failover to inter-domain backup link
Raja et al. [117]	Control Plane	Multicasting	Pre-computed	Link	Connects affected nodes to the closest functional subtree root placed in a higher level
Giorgetti et al. [121]	Control Plane	Optical	On-demand	Path	Sends flow modifications to switches independently or in a bundle
Liu et al. [68]	Control Plane	Optical	Pre-computed	Path	Computes new modulation or shortest path

resiliency. Additionally, since network devices are managed through control channels, a channel failure may lead to a control plane failure. We discuss methods that aim to quickly detect and recover from control channel failures.

A. Control plane architecture

The network controller is a fundamental element in SDN fault tolerance, because network services and applications run on top of it, depending on its services and logical centralized view to perform network management. Control plane architecture, which comprises of controller internal architecture, inter-controller and controller-switch communication, must provide or leverage coordinated mechanisms that address control plane fault tolerance issues, such as single-point of failure and controller consistency (previously discussed in section IV). In this subsection we discuss the efforts that deal with control plane fault tolerance issues. These control plane architectures are categorized and compared in Table V, columns were filled when relevant information was provided.

1) Control plane state redundancy: Fault-tolerant controller architectures can be categorized with respect to two aspects: *control distribution* and *state redundancy*. Control distribution can be, *Centralized*, where a primary controller is responsible for domain management, while backup controllers keep their state consistent with the primary, so they can take over network control in case of primary failure; or *Distributed*, where multiple controllers take control of a network simultaneously, while coordinating with each other to exchange network information necessary to process their requests.

Control state redundancy can be achieved through three approaches: *State replication* encapsulates network view and/or applications' state directly and sends them to replicas, which must update their states; *Event propagation* sends all, or selected, events in the same order to each replica, so that they can achieve the same state after event processing; *Traffic duplication* replicates all traffic, at switch-level, into replica controllers. Event and traffic replication approaches must consider ordering, because message reordering may lead controllers to inconsistent states.

a) State replication: Control state replication strategies vary in regard to control distribution and replication strategies. In distributed architectures, network state is partitioned and replicated along multiple controllers. In centralized architectures the primary controller might store the network state in an external data store or send directly to the backup controllers to update their state.

Distributed data store: In distributed control, multiple controllers must share the network view in order to complete network-wide operations. A common strategy is to partition the network and assign a master controller and multiple backup controllers to each partition. Then, each master is responsible for updating its partition view in a distributed data store that replicates to other controllers.

Onix [53] is an early example of distributed controller with fault tolerance support. Onix API is data centric, where the network view is represented in a graph and maintained in a Network Information Base (NIB). Network operations are done through modifications in the NIB and its state is distributed among the controllers. Onix provides coordination facilities that allow different storage strategies. Applications can use two types of storage: a *transactional storage* with strong consistency for stable and critical applications, and in-memory *distributed hash table* (DHT) for weak consistency and dynamic applications. Onix also supports Zookeeper, which can leverage different teaming strategies, such as leader election and failover.

ONOS [55] is an open-source distributed controller with focus on scalability and high availability. Similar to Onix,

TABLE IV: Failure protection methods comparison

Authors	Scope	Multi	Scenario	Goal	Method
Sharma et al. [62]	Path		-	Failure protection	Backup paths using *fast failover* group type
Adrichem et al. [64]	Path	✓	-	Decrease recovery time	Fast detection with backup paths and reverse forwarding rules
Ramos et al. [76]	Path		-	Minimize latency	Backup paths encoded in packet header
Goncalves et al. [75]	Path		-	Reduce backup path length	Disjoint and overlapping backup paths minimized using *learn* feature
Braun et al. [78]	Link	✓	-	Multiple link failure protection	Loop-free alternates
Sgambelluri et al. [71], [72]	Path		-	Failure protection	Backup paths using low priority flows
Lee et al. [10]	Path		-	Avoid congestion	Computes different backup paths according to source-destination pairs
Borokhovich et al. [83]	Path		-	Avoid congestion	Computes random permutations according to source-destination pairs
Cascone et al. [90], [92]	Path		-	Minimize flow rules Avoid congestion	Uses OpenState to assign different failure states considering path reuse or link-load
Sahri et al. [93]	Path		-	Fast recovery with eventual optimality	Backup paths using low priority flows while controller computes optimal solution
Mohan et al. [100]	Path		-	Minimize number of flow rules	Backward rerouting sends packets back to origin, where a backup is configured Forward rerouting determines shortest path and merges conflicting rules
Stephens et al. [98]	Path		-	Minimize number of flow rules	Maximizes path reuse, then applies a forwarding table compression algorithm
Kitsuwan et al. [96]	Link		-	Fast recovery with eventual optimality	Local failover to a transient forwarding plane while backup plane is configured
Yang et al. [122]	Path		Optical networks	Path protection optimizing bandwidth usage	Computes disjoint paths and uses bandwidth squeezed protection
Pfeiffenberger et al. [112]	Path	✓	Multicasting	Minimize number of links used	Computes paths using an approximation algorithm to solve Steiner tree problem
Gyllstrom et al. [67]	Path		Multicasting	Minimize number of links used Minimize number of flow rules	Computes paths using an approximation algorithm to solve Steiner arborescence Merge similar rules with same outgoing port
Kotani et al. [111]	Path		Multicasting	Failure protection	Computes primary and disjoint paths
Raja et al. [115]	Segment	✓	Multicasting	Failure protection	Computes multiple subtrees and connects affected subtree to higher functional root node
Chu et al. [105]	Path		Hybrid	Avoid congestion	Legacy devices encapsulate and forward all packets to designated SDN device
Tilmans et al. [107]	Path		Hybrid	Failure protection	Routing daemons running on switches precompute backup paths using IGP (e.g. OSPF, RIP, IS-IS)
Wang et al. [126]	Path		Virtual networks	Avoid congestion	Computes spanning tree considering bandwidth usage and virtual path splitting

network view is also represented in a graph, and each service state has a store. In distributed mode, multiple instances of ONOS compose a cluster, where each instance (or node) is responsible for a subset of network devices. Each node has a portion of the network view. Network services propagate their state using a *publish-subscribe* mechanism that may achieve different consistency levels. Since ONOS is under continuous development, different distribution mechanisms were used since its inception, including Cassandra data store [129] for eventual consistency, and distributed file systems, such as Zookeeper [130] and Hazelcast [131] for strong consistency. Currently, it employs Raft consensus protocol [132] for strong consistency and a lazy replication protocol [133] for eventual consistency.

HP Virtual Application Network (VAN) [134] is an SDN controller with focus on modularity and extensibility. HP VAN was implemented using OSGi specification [135], a standardized modular architecture that aims to provide application independence and reuse. Communication between applications is performed using an implementation of Advanced Message Queue Protocol (AMQP) [136], an application layer protocol to ensure message delivery guarantee. HP VAN offers a clustering mode, where various controllers take control of a domain. Similar to Onix, teaming capabilities are offered, but not detailed, through controller coordination and synchronization using ZooKeeper.

Spalla et al. [137] propose a proof-of-concept fault tolerant controller that provides basic state distribution and clustering capabilities. The network is partitioned among multiple controllers, where each switch assigns *master* role to a controller and *slave* roles to others. The state between controllers is shared using OpenReplica [138], a coordination service that keeps consistency between distributed instances of an application implementing synchronization and Paxos consensus protocol [139].

Centralized Data Store: One approach to provide control plane with state fault tolerance is to decouple state storage from network controllers. Primary controller replicates network state into a shared external data store, from which the backup controllers can retrieve it. SMaRtLight controller [140] has a 3-tier architecture composed by (1) data plane, (2) network controllers replicas and (3) shared data store. Figure 7 illustrates SMaRtLight architecture. One controller is configured as primary, which is responsible for processing all switch requests, and the others are configured as backups. Controllers coordinate their actions using a *lease management algorithm*, which the controllers use to detect whether a primary is active. The data store consists of a set of servers kept consistent using an implementation of the state machine replication [141].

State-update messages: An alternative replication approach for centralized architecture is passive replication. In this approach, the primary controller encapsulates its application state in *state-update messages* and sends it to backup controllers, which update their states. Fonseca et al. [142] implement *CPRecovery*, a proof-of-concept passive replication mechanism.

Passive replication operation can be divided into two phases: *replication phase* and *recovery phase*. The replication

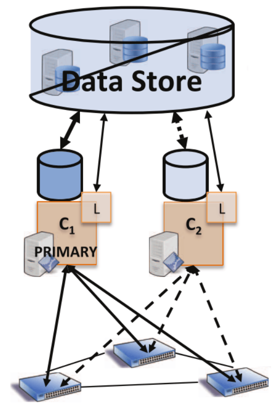


Fig. 7: Centralized data store architecture [140]

phase represents most of its operation, which occurs during failure free scenarios. In replication phase, the primary controller monitors its application state and sends each state change to the backup replicas, while the replicas keep synchronizing their state with the primary. If a primary controller failure is detected (e.g. heartbeat messages, disconnection notification) the recovery phase starts. In recovery phase, the switch connects to the configured backup controller, which updates its role to *primary*, continuing network normal operation.

Pashkov et al. [143] propose a *High Availability Controller* (HAC) architecture, a hybrid approach that alternates between both strategies, according to service priority. Network view changes are sent to all backup controllers as soon as they are available. Information about the state of each network service running on top of the network controller is sent periodically. Whereas, controller state is stored in an external shared data store. Different coordination services are used to orchestrate the replication operation and messages are ordered using serialization.

b) Event replication: The OpenFlow protocol is heavily event-oriented. Each significant network change is sent to the network controller for proper processing. Therefore, it is possible to achieve state replication replaying events in backup replicas. However, in high traffic volume networks, the amount of network events may overwhelm the controller. In distributed architectures, this issue may escalate as the number of controllers increase.

Hyperflow [144] is a distributed controller that provides scalability by propagating selected events. Events that only require local processing are not propagated, e.g. forwarding requests that can be resolved locally. Only those events that change network view are propagated (e.g. *link state change* events). Tootoonchian et al. argue that such events are much less frequent, on the scale of tens per second for networks with thousands of hosts [52]. Events are propagated through a publish-subscribe distributed file system.

If a controller fails to send three consecutive heartbeats, Hyperflow assumes that controller has failed and configures orphan switches to connect with their nearest neighbour. Since HyperFlow does not guarantee delivery ordering, it might not be used with methods or applications that require event ordering.

DISCO [86], [145] uses a similar approach, propagating only network-wide events. Each controller only keeps its network domain state consistent, and propagates network-wide events. Instead of using a data store, inter-domain events are propagated using AMQP protocol, guaranteeing message delivery. DISCO also provides more sophisticated inter-domain information, such as bandwidth use and latency, to leverage QoS and reservation features.

Ravana [48] is a centralized controller platform that uses viewstamped replication [146] to replicate events and provide fault tolerance in both control and data plane. Replication of controller state and switch consistency is achieved through a two-stage protocol that keeps three properties: 1) *Total Event Ordering*, controller replicas must process in the same order, thus, all replicas must reach the same state; 2) *Exactly-Once Event Processing*, the controller always processes all the events issued together. If an event is lost due to some link failure, it must be re-transmitted until its processing; 3) *Exactly-Once Command Execution*, the switches execute any given series of events once and only once.

In Ravana architecture, a master controller receives any network request, and then, the master stores the event in the shared replicated log and processes it. The replicas retrieve the event and process it after the master has acknowledged its completion.

Ravana achieves *total event ordering* implementing the shared log using viewstamped replication, that ensures that the events are replicated into the backups in a consistent linearized order. The replicas keep track of the IDs of the logged events to avoid repeated events, thus conforming to the *exactly-once event processing* property. The switch also keeps track of received commands in a local buffer, filtering and ignoring repeated commands. Thus, *exactly-once command execution* property is also guaranteed.

The Ravana protocol provides transactional semantics for the whole control loop of event delivery, event ordering, event processing and command execution, guaranteeing *safety* (for any given series of events, the resulting commands executed could have been executed in a fault-free system) and *liveness* (every event is eventually processed by the controller and every command is eventually executed by the switch).

c) Traffic replication: A third approach to controller state replication is to duplicate network traffic, at the data-plane level, into all the controllers.

Fonseca et al. [147] implements a proof-of-concept active replication mechanism, where all switches connect to all controllers simultaneously, and send the requests to all controllers, which coordinate among them to avoid response duplication. When a controller receives a request from a network device, it sends the message arrival timestamp to other controllers, if all the controllers reply indicating that none of them has an older timestamp, it sends the reply to the switch. Otherwise, the controller only updates its internal state.

Gramoli et al. [148] uses traffic replication to quickly recover from failures in a disaster scenario. They aim to minimize two key metrics *recovery time objective* (RTO), the time duration of outage during a disaster, and *recovery point objective* (RPO), how much data can be lost in case of disaster. RTO reflects how long it takes to detect and mitigate a failure, and RPO reflects how many updates are lost.

Gramoli et al. propose a method that proactively replicates data and quickly detects failures. At intra-domain level information is stored in a key-value store and replicated among replicas, which ensures fault tolerance for intra-data center failures. At inter-data center level, traffic is safely replicated to guarantee that all data centers are in the same state. When a client sends a *put* or *update* request to a data center, its request is replicated into all data centers through a primitive SDN-multicast, that replicates packets and acknowledges the client after all data centers safely synchronize. TCP header is modified to keep replication process transparent to the client and data centers. Gramoli et al. argue that this process guarantees nil RPO, since data is replicated before being stored and the client being acknowledged. If the primary data center fails to reply a request, the client will repeat the process. If a client exceeds a number of attempts *n*, a backup data center will take control of the orphan client, redirecting all client's flows using an SDN-anycast directive. Consistency is guaranteed using linearization locally and inter-data centers. Experimental results showed that setting $n = 5$ achieved an average 30 $seconds$ RTO. All packets must be forwarded to the controller, this leads to a significant overhead. There is no discussion relative to an ideal n and whether 5 can lead to false positives (e.g. due link congestion).

2) Controller failover: After controller failure, the backup controllers that will take control of its orphan devices must be chosen carefully. For example, if the orphan devices are assigned to a controller that is only accessed through a congested link, the request processing latency may increase. Some fault-tolerant control frameworks focus on efficient controller failover approaches.

Chan et al. [149] propose a fast controller failover for multi-domain SDN (FCF-M). In this distributed approach, each domain is managed by a controller which is replicated to a local backup using strong consistency. Controller failure is detected using a circular heartbeat mechanism, where each controller checks whether its predecessor is still alive sending probe packets. Controller failover is performed locally, if the backup controller is available, or using controllers from other domains if necessary. Devices are assigned to controllers by minimizing controller distance and verifying whether its residual capacity is enough to accommodate orphan switches. Otherwise, orphan switches are distributed among other domain controllers.

Obadia et al. [150] propose two controller failover mechanisms. The first is a greedy algorithm that, during failure, sends LLDP packets searching for controllers. When a switch receives a LLDP packet it forwards it to its controller, then the controllers take control of orphan switches. In the second mechanism each controller computes which controllers will take control of switches from its domain in case of failure, then each one sends its list to neighboring controllers. Both mechanisms use AMQP to guarantee message delivery and are implemented on top of DISCO controller.

Kuroki et al. [151], [152] aim to provide high controller availability through inter-data center controller failover. OpenFlow 1.2 role capabilities are used to assign the master controller and other controllers as backup controllers. At the intra-cluster level, when a master controller fails to respond to a heartbeat message, sent at a fixed 50 *ms* interval in order to increase responsiveness, a backup takes control of the network. For inter-cluster recovery, a Role Management Server(RMS) is devised, which is responsible for sending heartbeats to each cluster controller, every 50 *ms*. If a controller misses it, RMS requests that each cluster controller sends a *keepalive* message to check the suspected controller status. If the majority of controllers confirm the failed status, RMS checks each controller CPU load and assigns the orphan switches to the controller with lower CPU usage, distributing to more controllers if needed. Experiments used a rate of 100 packets/s, very unrealistic considering data-center scenarios, only capable to prove the approach basic functionality. The local recovery is straight-forward only differing from standard detection methods in its high responsiveness demand.

Muller et al. [153] propose a generic heuristics framework, which is used to compose an ordered list of backup controllers that must take control of the network. To assign each index of its list the controller gets all devices connected, selects the optimal value according to a defined metric among the possible candidates, and update the candidate list. The result is a backup controller list that is configured in each device controlled by the evaluated controller.

3) Controller-switch reliable communication: Since controller state is updated through network events, if an event is lost it may cause incorrect behavior. For example, when there is a failure in a primary controller, a backup replica takes over the network control. Until this operation is complete, new requests from switches can not be processed and will be lost, leading the backup controller to an inconsistent state. Thus controller-switch communication must be reliable in order to guarantee network correctness.

Ravana [48] provide a message delivery guarantee mechanism. During controller and switch communication, both exchange acknowledgement messages, while the switch keeps the event in a temporary buffer in case of controller failure in order to guarantee that the events will be eventually delivered. The OpenFlow protocol was also modified in order to support the exchange of acknowledgment messages between controller and switch, and buffer operations through the following messages: `EVENT_ACK`, an event arrival notification; `CMD_ACK`, a command arrival notification; `EBuf_CLEAR` and `CBuf_CLEAR` are used to clear *event buffer* and *command buffer*, respectively.

Li et al. [154] propose Non-stop Network Controller (NSNC), a primary-backup architecture that employs leveraging high availability TCP to guarantee message delivery. A packet filter captures all packets originating from or destined to the controller. When the packet filter receives a TCP connection request from a new controller it means that the former primary has failed and the new primary is trying to connect. Then, the packet filter rewrites the TCP header with a wrong sequence number and sends it back to the switch, which will check the packet sequence number, mark the packet as an error and resend it. Packet filter performs packet header transformations necessary for this operation to be transparent to both sides (e.g. dropping ACK messages).

When the primary controller fails, the controller election algorithm in NSNC uses the following metrics: Load of controller, number of flow entries of a switch, historic frequency of switch requests to its controller and round-trip time from the controller to each device. These metrics are normalized, and then, parameterized weights are applied on each metric and summed. The controller with the minimum value associated to a given switch becomes the master. These metrics are stored in a shared mapping table and are collected periodically, to avoid additional processing during controller failover.

Summary: Control plane state redundancy is one of the central issues in SDN fault tolerance, along with controller failover, since it is related to one of the defining characteristics of SDN: *logical centralization of control*. Therefore, controller state fault tolerance should be considered in control plane design, because different architectures may enable different strategies (e.g. distributed control [144] vs. centralized control [48]). Naturally, control plane redundancy may require some kind of processing/communication overhead, which might affect other network aspects, such as, scalability and performance.

There are a number of distributed applications that can be used to achieve *state replication*, such as distributed file systems, distributed data structures and consensus protocol. Since implementations of these applications are widely available, this is a popular approach. However, consistency might affect network and controller performance [54]. Different applications may be assigned to different consistency levels [55], [143] in order to reduce redundancy overhead.

Event and traffic replication can be used to reduce communication overhead caused by strong consistency, since the same event or packet can be used to update multiple applications' state. Besides, there is no need to identify state changes and which application data structures must be distributed. However, some applications may require ordering in order to guarantee consistency. Also, as the network size increases the number of network events and traffic, which need to be replicated might greatly increase.

Efficient controller failover can be performed if information about other controllers can be accessed and processed in a timely manner. Network controller load, latency, failure rate and other features should be used to choose the master from the available controllers in order to minimize probability of a controller failure [152], [154].

In order to avoid event loss, which can lead to inconsistent controller state, some strategies are proposed to enforce reliable delivery. However, such strategies may require modification of OpenFlow protocol [48] or a high granularity packet processing [154]. Ideally, such features should be supported natively by OpenFlow, at data plane level, in order to increase processing speed, or provided at control plane level, enabling lower granularity.

TABLE V: Control plane architectures comparison

Authors	Control	Replication	Storage	Consistency	Hierarchical	Delivery guarantees	Election
Phemius et al. [86], [145]	Distributed	Events	Local store	Partial, Inter-domain state only	No	AMQP	-
Gramoli et al. [148]	Distributed	Traffic	Local store	Full	No	No	Closest data center
Fonseca et al. (2013) [147]	Distributed	Traffic	Local store	Full	No	No	First to process request
Pashkov et al. [143]	Centralized	Control state	Local store	Full	No	Serialization	Highest ID or IP
HP Controller Architecture [134]	Distributed	Control state	DFS[a] and Local Store	Full	No	AMQP	-
Tootoonchian et al. [144]	Distributed	Events	DFS	Full	No	No	Nearest neighbour
Kuroki et al. [151]	Distributed	Control state	Local store	No	No	-	Residual capacity
Li et al. [154]	Centralized	-	Local store	-	Yes	-	Residual capacity Number of flow entries Frequency of switch requests
Obadia et al. (GF) [150]	Distributed	Control state	Local store	Partial, Inter-domain state only	No	AMQP	First to process request
Obadia et al. (PPF) [150]	Distributed	Control state	Local store	Partial, Inter-domain state only	No	AMQP	Preconfigure routes
Koponen et al. [53]	Distributed	Control state	DFS and Local Store		No	Ordering	-
Berde et al. [55]	Distributed	Control state	DFS and Local Store	Full, eventual consistency	No	Ordering	-
Fonseca et al. (2012) [142]	Centralized	Control state	Local store	Full	Yes	No	-
Katta et al. [48]	Centralized	Events	Local store	Full	Yes	Yes, VR[b]	First to process request
Botelho et al. [140]	Centralized	Control state	Shared data store and local cache	Full	Yes	Ordering, Paxos	First to process request
Spalla et al. [137]	Distributed	Control state	DFS	Partial	No	Ordering, Paxos	-

[a]Distributed file system
[b]Viewstamped replication

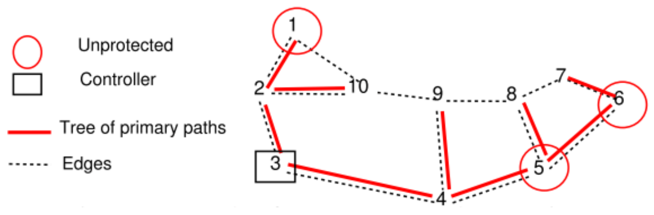


Fig. 8: Example of unprotected switches, according to [162].

B. Controller placement and assignment

A controller placement strategy needs to address two network design choices: How many controllers are needed and where they should be placed. Research has demonstrated that the location of a controller can affect network performance, from latency [50], [155] to control and data network resiliency [156]. A fault tolerant controller placement may prioritize different network aspects, such as failure isolation (e.g. failures of each partition do not affect others) [156], [157], or control path resiliency [158]. Most of the approaches model controller placement using graph representation [159] or *integer linear programming* [153]. In general, resilient controller placement efforts differ with respect to the fault tolerance aspect they prioritize, metrics that they consider and methods they use to optimize these metrics.

Device assignment is a related problem, where the switches are assigned to the controllers but the controller location is already defined [160], [161]. Efforts on both problems are summarized in table VI.

Zhang et al. [156] tackle the controller placement modeling as a graph partitioning problem. The proposed method aims to reduce total failure likelihood minimizing inter-partition connectivity, using a modified version of min-cut, where the network is partitioned with minimum cuts across boundaries, while balancing number of devices and links per partition. For each partition, a controller is assigned to the location which has the shortest paths to all devices in the same partition.

Hu et al. [158] first proposed a new metric to measure network reliability called *expected percentage of control path loss*, where control path is defined as the set of routes that are used for communications between controllers and switches. Then, they also proposed greedy algorithms and simulated annealing [168] implementations where the optimal value is the minimum percentage of control path loss. Evaluation was performed on Internet2 OS3E topology, and it showed that simulated annealing achieved the best result. It was also demonstrated that the ratio between the number of controllers and number of nodes must be in a "sweet spot". Few controllers can increase the probability of control path failure, whereas a large number of controllers make control paths similar to a full mesh. Tradeoffs between reliability and latency are investigated and experiments showed a reduction of 13-17% when one is optimized in detriment of the other.

Beheshti et al. [162] tackle the controller placement problem in topologies with only one controller where each switch has only one path to the controller. The set of all node paths to the controller are called *controller routing tree*. A switch is considered protected if in case of failure of its parent node or corresponding link there is a *neighbour link* that it can use to connect with the control path (in Figure 8, controller is placed at node 3, leaving switches 1,5 and 6 unprotected). The *weight of a node* corresponds to the number of descending nodes. Considering that an unprotected switch can not inform its descending nodes in case of failure, this work defines the *weight of a routing tree* as the sum of the weight of all its unprotected nodes. To increase the resiliency of a given network its weight must be minimized. This can be achieved by optimizing controller placement and/or optimizing the routing tree generation. A greedy placement algorithm is proposed, iterating over a list sorted in an order of decreasing degree of the nodes and choosing the location on node that has the maximum number of protected neighbors. A greedy routing tree algorithm is also presented, which starts from a shortest-path tree, checking nodes and changing its parent node if the new parent node increases its resiliency. This approach limits backup paths to neighbor switches and ignores that multiple switches connected to the same parent node can coordinate to find a new control path.

Ros et al. [159] aim to achieve high level of reliability in optimizing controller placement. To measure reliability authors use *k-terminal-reliability*, which is the probability that there is at least one operational path between a node and its controller. Based on the fault tolerant facility location problem [169] the *fault tolerant controller placement problem* is defined as a minimization of the cost of deploying controllers at a location in conjunction with the cost of all its neighbors contacting the controllers at their locations, with a k-terminal-reliability of at least β. The heuristic proposed ranks of possible controller locations in order of decreasing degree of node and iterate over them, calculating its lower bound reliability, and then, if it meets with the reliability requirements, it deploys the controller at that location and increases the controller rank according to the τ parameter, that helps controller reuse. Since it uses the reliability lower bound, this algorithm deploys a number of controllers that satisfies worst-case scenario. Experiments were performed in more than 100 topologies and in the majority of cases ten or less controllers were enough to achieve 99.999% of reliability.

Survivor [153] is a controller placement strategy that aims to optimize controller survivability. Survivability is strongly related to three aspects: connectivity, capacity and recovery. The proposed solution addresses these problems in two different parts, (i) a controller placement strategy to optimize connectivity and capacity; (ii) a generic heuristics framework, previously explained in the subsection VI-A2, is used to define an ordered list of backup controllers to increase recovery efficiency. The controller placement strategy is modeled as an integer linear programming problem. In order to maximize connectivity between the devices and the controllers, the linear function maximizes the average number of disjoint paths between the devices and controllers while guaranteeing that the controller capacity will not be exceeded.

Philip et al. [163] takes a similar approach but with a different metric. They argue that network resiliency is related to convergence delay. A minimum convergence delay would increase network because the network would return to a stable

TABLE VI: Controller placement methods comparison

Authors	Goal	Metric	Method
Zhang et al. [156]	Minimize total failure likelihood	Number of intercluster links Number of intra-cluster links	Min-cut for partitioning centroid for placement
Hu et al. [158]	Minimize control path loss likelihood	Percentage of control path loss	Greedy algorithm Simulated annealing
Beheshti et al. [162]	Minimize number of unprotected switches	Sum of the weight of all unprotected nodes	Greedy algorithm
Ros et al. [159]	Achieve 99.999% of reliability	k-terminal-reliability	Greedy algorithm
Muller et al. [153]	Maximize survivability	Number of disjoint nodes and controller capacity	Integer Linear Programming
Philip et al. [163]	Maximize resiliency	Convergence delay after a link failure	Integer Linear Programming
Guo et al. [157]	Minimize cascading link failures	Mutually connected cluster for each controller after a failure	Partitioning
Tam et al. [161], [164]	Maximize path reuse	Link reuse	Partitioning
Li et al. [160], [165].	Enable Byzantine fault tolerance assigning controllers efficiently with low latency	Controller capacity	Greedy algorithm
Yao et al. [166]	Minimize cascading controller failures	Controller capacity	Load balancing
Hock et al. . [167]	Optimize trade-off between different requirements	-	Pareto-optimality

state faster. An optimal placement would place controllers closer to faulty locations, leading to faster reactions. They also treat this problem as an ILP, where the goal is to minimize total convergence delay after a failure, under shortest path routing policy. A greedy algorithm is used to found a feasible solution.

Tam et al. [161], [164] propose a decentralized flow assignment that optimizes controller reuse. The network management is partitioned between controllers, where each controller only is aware of its partition. Partitioning is performed by precomputing all paths (represented in a multipath) to each (u, v) pair of nodes. Then, multipaths are distributed among controllers, which will be responsible for nodes in the links in its multipaths. Two distribution methods are presented: 1) Path-partition, a multipath between (u, v) is computed, then assigned to a controller considering link reuse and fair multipath distribution; 2)Partition-path, all links are partitioned equally between controllers and then multipaths are calculated using links already assigned to the controllers. The former method produces shorter paths and the latter assigns less links to controllers. Both methods, as well as any approach that uses devolved controllers, produce a link coverage overlap.

Fault tolerance is achieved by assigning the same multipath to more than one controller. The authors claim that the increase in the links covered by each controller will not be significant. Controllers exchange heartbeat packets and a failure is detected if some controller fails to reply. After failover, the new primary will keep sending heartbeats to the failed controllers, which will take control of the network in case of recovery.

One of the main obstacles to address byzantine failures in SDN is that most solutions require a high number of controllers and a significant communication overhead. For example, a PBFT [170] solution would require $3f+1$ controllers assigned to each switch to handle f faults (e.g. 4 controllers to handle 1 fault). In order to address this problem Li et al. [160] propose an efficient controller assignment algorithm. First, Li et al. assume that an efficient byzantine fault-tolerant SDN has two requirements: 1) Each switch must be managed by m controllers, and 2) communication between controllers must have a maximum latency of l. Considering that the switches may require different number of controllers, and controllers have the capacity to manage a limited number of switches, *Capacity First Allocation*(CFA) iteratively assigns controllers to switches, in a descending order of controllers required. Then, the algorithm assigns the controllers, first assigning the controllers with more residual capacity. At the end of each iteration, the algorithm checks if the solution meets the requirements. Li et al improved CFA algorithm proposing *Requirement First Assignment*(RQFA) [165]. Instead of choosing a single controller candidate in each iteration, RQFA chooses a candidate set of controllers in each iteration. Evaluation showed that RQFA outperformed CFA. However, RQFA and CFA were not compared with other controller assignment strategies [153], [164]. The authors propose to move the control plane to the cloud to use its resource provisioning flexibility, but the efficiency of this approach may not be suitable for time critical applications.

Yao et al. [166] explore the threat of controller cascading failures in SDN. When a controller fails, its switches must be reassigned to active controllers, however, if their residual capacity is not taken into consideration, an active controller with a high load might be overwhelmed, crashing and increasing the number of orphan switches, moving the problem forward. A failure model is proposed based on the Motter model [171], where the capacity of a controller is proportional to its initial load. To prevent cascading failures, a controller placement and flow reassignment strategies are proposed. Controller placement guarantees that the load is initially balance among controllers, and flow reassignment checks whether a reassignment causes the controller capacity to be exceeded.

In data plane, a link or a node failure frequently leads to cascading failures of other nodes, since they are often interdependent [172]. Guo et al. [157] investigate that interdependence network analysis can improve controller placement. First, they separate the network in two interdependent networks, *controller-switch(CS)* network , used for network

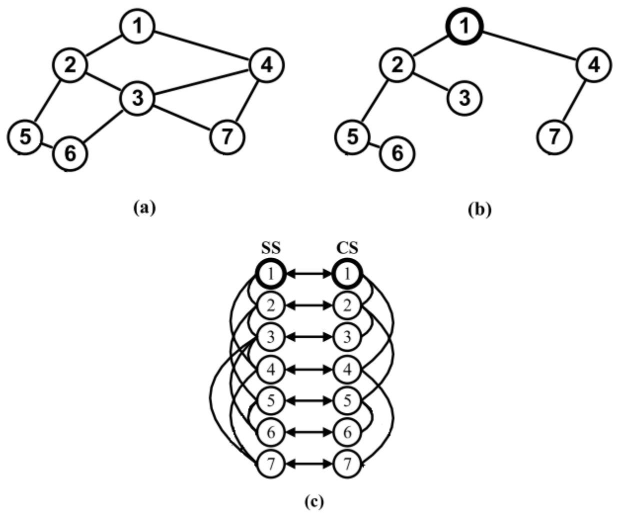


Fig. 9: (a) SS network, (b) CS network and (c) interdependence graph

control, and *switch-switch(SS)* network, used for data forwarding, and construct their interdependence graph, as depicted in figure 9. Since only nodes mutually connected and managed by a controller are potentially functional, Guo et al. define a resilience metric that measures the expected fraction of nodes that will survive a given cascading failure scenario in the mutually connected cluster for each controller.

The proposed controller placement method uses the greedy modularity optimization method [173] to partition the network, then, controllers are placed in each partition at the node with maximal closeness centrality. Experiments were performed over different topologies, however, proposed method was only compared to random placement algorithm.

As network requirements differ and different controller placement strategies can produce significantly different results, to evaluate trade-offs is of great importance. Hock et al. [167], [174] explore different resiliency issues with its associated metrics and trade-offs, and then, instead of optimizing resiliency metrics, it presents their pareto-optimal placements. The following trade-offs between latency and resiliency were considered: Optimize latency considering a failure free scenario or considering possibly failed controllers, the former spreads the controllers in the topology, and the latter keeps them closer; Place controllers in a way that minimizes the possibilities of controller-less nodes due link failures, or minimize latency, the former can greatly reduce the placement possibilities; Load balancing and minimum latency during failure-free and controller failures scenarios; Minimize latency between controller and nodes, or among the controllers. Experiments were performed in PlanetLab, in order to evaluate performance in dynamic network conditions and a toolset implemented in MatLab was made available [175].

Summary: Network requirements may vary greatly, prioritizing availability, minimum latency, or balanced workload. Even considering only resiliency requirements, optimization parameters can vary greatly such as, network reliability, failure isolation, control path protection and controller overload avoidance. Thus, a controller placement strategy must consider which metric may be influenced and which tradeoffs are being made, e.g. Zhang et al. [156] values a minimum network total failure likelihood over path diversity; Hu et al. [158] values control network reliability over data network reliability. Tools such as POCO [167] might provide insightful information and assist in network design.

Additionally, recovery and protection mechanisms should be aware or, at least, consistent with controller placement strategies to take advantage of its features. A placement strategy that maximizes number of disjoint paths might use a path protection that leverages on path diversity.

C. Control Traffic

In this subsection we discuss efforts that focus on improving control plane fault tolerance by quickly detecting and responding to control channel failures. Table VII summarizes these efforts.

1) Fault detection: Kotani et al. [69] propose a control channel failure detection method. Each switch sends probe messages to all controllers, and as soon as the first one responds, it sets its channel state to active, otherwise, it is set to inactive. In order to avoid unnecessary communication overhead, a switch sends echo request to control channels only after it sends a network event to the controller. The controller checks each switch individually sending an echo request and, upon reply notification, it updates other controllers with the information that the switch is alive.

Lee et al. [11] also propose a failure detection mechanism. Monitoring circle paths, composed by a subset of network links, are responsible for the detection of link failures. Initially, the controller forwards a packet to the circle and waits for a timeout period. If the controller receives a response within this period, links of this circle are considered as healthy, otherwise it starts a failure location mechanism that sends a packet along the suspected failed path, where each switch must send a copy directly to the controller. The node that does not send the packet is considered *faulty*. In in-band control an extra round is performed to check whether the failure affects control traffic. A probe is sent along the in-band control tree, and if one or more controllers fail to reply, the location is determined by sending a probe packet to each respective control tree node, and then, the root of the subtree or leaf that fails to reply is finalized as the failure location.

2) Fault recovery: Network control can be of two types, *out-of-band*, control is performed directly over dedicated communication channels, and *in-band*, the same network is used for data and control traffic. Usually, in in-band control, switches need to traverse intermediate devices to reach controller. Consequently, if one intermediate node fails, control traffic is lost.

Petry et al. [177] investigates how cellular data network can improve control plane resiliency providing out-of-band control. They added cellular link between forwarding planes and the controller that could be used in case of primary link failure. It was demonstrated that cellular network bandwidth was equivalent to ethernet network, while the delay increased.

TABLE VII: Control traffic fault management approaches

Authors	Goal	Method
Hu et al. [176]	Failure protection	Greedy algorithm that combines local rerouting and reverse forwarding.
Watanabe et al. [128]	Failure detection Failure recovery	OSPF Fast Hello packets to detect failures and OSPF to recover from failures.
Petry et al. [177]	Failure recovery	Cellular links that can be used to maintain control network connectivity in case of primary link failure
Lee et al. [11]	Failure detection Failure location	Monitoring cycle path
Kotani et al. [69]	Failure detection	Controller and switches monitoring through probe messages
Sharma et al. [178]	Failure protection Failure recovery	Precompute backup paths for control channels

Authors argue that further investigation and optimization might be necessary to guarantee that data cellular networks are robust enough to handle data and control plane failures.

Sharma et al. [178] provide restoration and protection mechanisms to control traffic in in-band networks. In control path restoration, the new path must be restored first on switches closer to the controller to guarantee that its descending switches will be able to receive restoration rules. The controller sends barrier requests, which force switches to process pending messages in order to guarantee the restoration of ordering. In control path protection, disjoint paths are pre-installed on switches in order to mitigate failures.

Leveraging on multiple controllers to decrease recovery time and increase robustness, Hu et al. [176] propose a protection mechanism for control traffic that combines local rerouting and constrained reverse forwarding. In local rerouting, control traffic is redirected to an upstream neighbor switch which has a reachable controller in its primary path, while in reverse forwarding, control traffic is redirected to a downstream switch which has a reachable controller in its primary path. Considering that reverse forwarding may lead to multiple hops until a controller is found the method limits the number of hops and prefer local rerouting. Also, it is always preferable that a switch connects to its original controller. If a switch can use one of the both methods to a restore its control channel, the switch is considered *protected*. A greedy algorithm is proposed, with a goal to maximize the number of switches protected while reducing latency as much as possible. First, an optimal shortest-path control network protection is computed. Then, the control network is modified to protect more switches iteratively, until convergence.

Watanabe et al. [128] propose to move intelligence to network devices in order to achieve fast failure detection and recovery of control channels. In *ResilientFlow*, Control Channel Maintenance Module(CCMM), a distributed failure detection mechanism, runs on each switch. CCMM uses OSPF Fast Hello packets to monitor control channel link failures, continuously collecting network information and exchanging network topology maps. Since this method is based on OSPF, it is capable of handling multi-link failure.

Summary: Due to decoupling of control plane and data plane, highly reliable communication between both planes must be provided. Control channel disruption can even prevent data traffic from being restored due to some other links. Thus, multiple strategies must be applied to avoid different issues: *fault detection* must monitor control channels constantly in order to quickly respond to failures [11], [69], [128]; *failure protection* can enable quick responses [176], [178], however its downsides must be considered (discussed in section V). Out-of-band control is an approach that enables simpler fault detection and protection [177], however its requirements may not be feasible in large networks.

VII. Application Plane Fault Management

Unlike control plane, where the network controller and its core services are arguably more homogeneous and less mutable after deployment, being designed following consistent project decisions and network requirements, application plane may be heterogeneous, highly mutable, and the network applications may be from different sources with conflicting requirements, being added and removed indefinitely after deployment. These characteristics lead networks applications to be potentially failure-prone.

Application plane fault tolerance can be addressed during any phase of network application right from the application design until post-deployment. Fault tolerance abstractions can be embedded in programming language enabling construction of fault tolerant programs and high-level policies [179]–[182]. In case of application failure its effects must be contained [47], [183], [184] and normal state recovered [183], [185]. Additionally, network administrators must be able to identify failure causes [58], [60], [186]–[191], verifying whether the applications are deploying policies correctly [192]–[194]. Applications must be tested extensively before and after deployment in order to ensure robustness and fault tolerance [195], [196]. In this section we group these efforts according to which issue they relate to and we discuss how they deal with these issues. Discussed methods are summarized from Table VIII to Table XI.

A. Network application design

Programmability allows developers to address fault tolerance issues during design of network functions. However, network abstractions and language support must be provided for this end. Fault tolerance of both network and applications may be supported at application plane.

TABLE VIII: Fault tolerant application design efforts

Authors	Fault Tolerance Attribute	Description	App plane	Ctrl plane	Data plane
Foster et al. [179]	Fault prevention	Allows modular design of OpenFlow applications in a run-time system that handles race conditions transparently to the programmer.	✓		
Williams et al. [180]	Fault protection	Supports high-level flow assignment and backup planning through fault-tolerant primitives.	✓		✓
Reitblatt et al. [181]	Fault protection	Allows specification of forwarding policies with different fault tolerance requirements using regular expressions.	✓		✓
Beckett et al. [182]	Fault troubleshooting	Provides assertion primitives that enable definition and verification of dynamic properties in network applications.	✓		✓

a) Network fault tolerance: In Rulebricks [180], Williams et al. aim to provide high availability to end applications by adding expressiveness to OpenFlow. Server applications are frequently replicated in order to provide high availability of its services, and flows must be assigned among replicas. To support high-level flow assignment and backup planning, three key primitives are proposed: 1) *drop*, forwards all traffic from various IP sources to a specified replica if they match a specific IP prefix; 2) *insert*, adds a rule that specifies the replica that must be assigned to an IP prefix in case of the failure of the currently responsible server for this IP prefix; 3) *reduce*, merge overlapping or redundant rules. These rules are translated into flow entries and installed on network devices.

FatTire [181] is a high-level programming language, based on NetCore language [197], that allows specification of forwarding policies with different fault tolerance requirements. FatTire syntax has primitives to represent header fields, matching patterns and number of failures tolerated. FatTire combines these primitives with a set of operations that enables programmers to reason about forwarding decisions without specifying low-level details. FatTire compilation is performed in four phases: 1) All non-overlapping forwarding policies are combined in a single broad policy. 2) A forwarding graph is generated for each policy and a breadth-first search is done, in order to find backup paths, until its fault tolerance requirement is satisfied. 3) Forwarding graphs are translated to NetCore policies. 4) NetCore compiler was extended to translate FatTire policies into OpenFlow rules using fast failover group types.

b) Application fault tolerance: Frenetic [179] is a high-level domain-specific language that allows construction of OpenFlow network applications using a declarative and modular design. Frenetic programs are compiled and passed to a run-time system that handles race conditions transparently for the programmer, preventing possible software bugs.

Beckett el al. [182] propose an assertion language that helps debugging and troubleshooting by allowing programmers to verify network dynamic properties. The language syntax supports: 1) assertion primitives, that checks a property at a given immediately (`assert_now`) or during a time interval (`assert_continuously`); 2) formulas, which are used to describe network properties that a set of network entities must keep and; 3) set modifications, which insert or remove network entities from a set. This approach is also related to network troubleshooting and network correctness.

B. Application fault tolerance

Upon an application failures two main concerns must be addressed: 1) Guarantee that its failure does affect other applications nor the controller and; 2) Restore application state to the last valid configuration, if any. State recovery is also related to control plane fault tolerance, however, in this section we only describe methods that focus on individual application state, whereas control plane state fault tolerance is discussed in section VI.

1) Failure isolation: LegoSDN [183] proposes a way to isolate application failures. Every network application is encapsulated in a single process and a proxy application running on top of controller performs communication between them and the controller. Applications use the proxy application to register for events. This approach guarantees that an application failure will affect only its own respective process. Also, according to the authors, LegoSDN handles fail-stop and byzantine failures but no experiment was presented in regard to these properties.

Shin et al. [47] argue that application resource consumption and network OS kernel modules also must be compartmentalized in order to achieve application isolation. They propose a resilient network OS called Rosemary. In Rosemary, similar to LegoSDN, each application is invoked as a new process which accesses network OS kernel through a generic inter-process communication (IPC) method. To diminish IPC overhead, Rosemary allows applications to communicate with service libraries independently. To guarantee that applications will not interfere in the well-functioning of Rosemary kernel a resource manager service monitors and limits application consumption of memory and CPU. Leveraging on microkernel architecture concepts, each Rosemary kernel module run as a different process, therefore, a failure of a kernel service does not affect other services. In case of failure of critical services, Rosemary restarts its services one at a time, running minimal services required for a *safe boot* mode.

Monaco et al. propose *yanc* [184], a network operating system that extends Linux with abstractions necessary to support network management, whereas taking advantage of the many implemented capabilities, such as failure isolation and support for multiple languages. They leveraged on the file system abstraction to represent and configure the network state. Network abstractions like switches, hosts and flows, are represented as directories and their internal state as files.

TABLE IX: Application fault tolerance efforts

Authors	**Fault Tolerance Attribute**	**Description**	**App plane**	**Ctrl plane**	**Data plane**
Chandrasekaran et al. [183]	Fault isolation Fault recovery	Each network application is encapsulated in a single process and a proxy application performs Inter-process communication (IPC). Rollback and recovery is performed upon failures.	✓	✓	
Shin et al. [47]	Fault isolation	Each application and kernel module is invoked as a new process and network OS kernel is accessible through a generic IPC method and shared libraries.	✓	✓	
Monaco et al. [184]	Fault isolation	Represents and manipulates the network through a file system built on Linux, allowing construction of network applications as independent programs.	✓	✓	
Zhang et al. [185]	Fault recovery	Checkpointing of both controller state and switches' state, which coordinate in order to rollback to latest valid state.	✓	✓	✓

Network applications are independent processes that read and manipulate the file system, thus, they might fail independently. Besides, Linux namespaces can also be used to enhance isolation between applications.

2) Application state recovery: Chandrasekaran et al. [183] argue that every operation has an equivalent that undoes its effect. Every action from a network transaction must be stored, along with relevant data (*e.g*, flow deletion must preserve flow statistics), and in case of failure, every action will be undone. Authors propose to handle associated actions as atomic network transactions in order to guarantee consistency after failures. Different fault correction policies are proposed to decide whether failures can be ignored in order to preserve network service availability. This paper assumes that the last event processed to be the most likely cause of failure, however Scott et al. [59] suggest otherwise.

NetRevert [185] performs checkpointing of both applications' state, using `unix` checkpointing tool *libckpt* [198], and of the switches' state, where a NetRevert agent running on each switch stores snapshots of its flow table. In case of failure, controller and switches coordinate among themselves to determine the most recent valid state and rollback to that state.

C. Network troubleshooting

a) Root cause analysis: Scott et al. [186] leverage SDN stack to provide insightful troubleshooting information that can assist developers to identify bugs in control plane. First, in order to identify errors, a correspondence checking is performed, verifying whether the application policies are correctly translated into network configuration by injecting packets in the network and comparing the propagation graph generated by the controller network view and physical network. When a policy violation is detected, a simulator explores the future behavior reproducing failure-free events to check whether the violation will persist. In case of persistent error, the mechanism generates a causal graph from the events and traverses the tree, pruning leaves and replaying resulting graph to check whether the violation still occur. The goal is to provide a *minimal causal sequence* to ease the troubleshooting process. Minimal causal sequence algorithm was later extended in [187] by adapting Delta debugging algorithm [199] to a distributed system and incorporating domain-specific knowledge.

b) Packet tracing: Handigol et al. [60] introduce `ndb`, a network debugger that provides two fundamental debugging primitives for SDN: breakpoints and packet backtraces. `ndb` configures switches to create, for each packet that traverses its output ports, a copy of the packet's header with the matching flow rule, switch ID and output port. This structure is called *postcard*. Postcards are sent to an external *collector*, so that when a packet breakpoint is defined, the *collector* can reconstruct its flow trace using its stored history.

Handigol et al. [188] extended packet history mechanisms to build a generic interface, called NetSight, that can be used to develop different network troubleshooting tools. NetSight provides *Packet History Filter* (PHF), a regular-expression-like language that allows to define *postcard filters*. Postcard filters allow to specify attributes (e.g. packet headers, switch ID) of packet histories of interest. Besides `ndb`, three other applications were developed using the PHF language: 1)`netwatch`, a live network invariant checker that uses PHF to specify packets that may violate network invariants, 2)`netshark`, a network analyzer tool where the users can interactively specify packet filters to see entire history of the packets and 3)`netprof`, a hierarchical network profiler that combines packet histories with topology information to monitor link utilization.

Zhang et al. [190] aimed to reduce overhead caused by path tracing leveraging on the concept of *pathlets*. In the proposed method, each pathlet is identified by an unique ID which is encoded on packet header as it traverses the network. At any point, it is possible to determine the packet history by decoding its header. The authors argue that performance of other methods is scales with the number of packers or flows, whereas using pathlets reduces overhead significantly.

c) Record and replay: Considering that control plane traffic accounts for less than 1% but is responsible of 95-99% of observed bugs [200], OFRewind [58] records control plane messages and selected data plane messages to provide a replayable and temporally consistent trace of controller. OFRewind allows the administrator to partition the trace and select which parts must be replayed, assisting the network administrator to identify the origin of a network failure or misbehavior.

Besides tracing network state, Durairajan et al. [189] also aimed to provide *application tracing* in order to tie network behavior with program execution. Authors extended the *fs-*

TABLE X: Application and network troubleshooting efforts

Authors	Goal	Description	App plane	Ctrl plane	Data plane
Scott et al. [186], [187]	Root cause analysis	Determines minimal causal sequence that leads to an error.	✓		
Handigol et al. (2012) [60]	Packet tracing	Provides breakpoints and packet backtrace primitives that allows programmers to debug an application interactively.	✓		✓
Handigol et al. (2014) [188]	Packet tracing	A regular-expression-like language that allows to define packet history filters through a generic interface that can be used to develop different network troubleshooting tools.	✓		
Zhang et al. [190]	Packet tracing	Encodes pathlets traversed in packets' header.	✓		✓
Wundsam et al. [58]	Replay events	Allows developers to replay network events that lead to a failure.	✓		
Durairajan et al. [189]	Replay events Testing environment	A testing environment with debugging capabilities that allows developers to tie application behavior with network state and generate reports for each testing run.	✓		
Pelle et al. [191]	Tool integration	Defines a common input/output interface for troubleshooting tools, allowing pipelining of actions.	✓		

sdn [201] testing environment with debugging capabilities, by creating a library called *OFf*. Beyond tracing, *OFf* also provides regular debugging features (e.g. track variables, breakpoints) and a *diff report generator* that detects changes in different *fs-sdn* simulation runs.

d) Integration between tools: Heller et al. [202] argue that SDN architecture allows more sophisticated and automated network troubleshooting, integrating SDN layers to systematically detect mistranslations between them. They propose a fault localization workflow to determine which layer the fault belongs and discuss how current SDN solutions can be integrated to realize such a workflow.

Towards the realization of a systematic troubleshooting environment, Pelle et al. propose Epoxide [191], a modular framework that allows integration between troubleshooting tools in a single platform. Pelle et al. represent a combinations of tools as a *troubleshooting graph*, where each node is a troubleshooting tool and directed links indicate the flow of data. Epoxide implements such vision by defining a common input/output syntax that is used to communicate tools and assigning a Emacs text editor buffer to each tool, which uses it to write and read data. This operation is analogous to how Unix pipes connect the programs. However, in order to realize fully automated troubleshooting, the developer must implement decision nodes that will analyze intermediate outputs to guide the sequence of actions.

D. Application and network correctness

Configuration errors are the most common case of operator errors, which are the largest contributor to failures and network downtime [203]. In SDN, network applications are among the responsible entities for network configuration. Correctness verification may be performed with respect to the network application by exploring its behavior and its actions on network state in order to find possible policy violations. During network operation, network state may also be verified by checking whether its configuration corresponds to desired properties and policies. Here we present a subset of correctness verification efforts, describing methods that are representative of most popular approaches. In table XI we summarize these efforts, in the column *verification type* we indicate if they verify application or network correctness.

1) Application correctness: Two most popular methods of application verification are *model checking* and *first-order logic verification*. Canini et al. propose NICE [195], a testing tool for OpenFlow applications that uses model checking to explore state space of the network and to detect violations of network correctness properties. Canini et al. define a transition model for controllers, switches and end hosts, that is used in state space exploration. In order to avoid state space explosion due to high number of possible packet inputs, Canini et al. use symbolic execution to explore possible code paths. Streams of packets are generated to trigger events that are handled in applications being tested. State space is explored through Openflow-specific search strategies that reproduce critical scenarios while limiting state space. If the model checker detects a correctness property violation, NICE outputs the error along with its trace.

Kuzniar et al. propose OFTEN [204], a testing tool to validate integration between network applications and real switches. OFTEN provides an interface between NICE model checker and real network devices by associating a physical interface of the real switch with the switch model. During NICE state exploration, OFTEN sends the network events to the real switch and upon switch state change, it updates the switch model state. If OFTEN detects the same policy violations at both switch model and real device, then it is very likely that the network application contains an error. If there are differences between the switch model and the real device, further investigation is needed to determine the root cause (it may be an bug in the switch software, for example).

Ball et al. [205] argue that model checking is not well suited for large networks, because it is difficult to scale model checking methods. They also argue that model checkers can only identify errors, but cannot guarantee the absence of errors. The authors propose Vericon, a verification tool that ostensibly evaluates application correctness at compile-time using deductive program verification. Network applications are written in a imperative programming language called CSDN, whereas

TABLE XI: Application and network correctness verification efforts

Authors	Verification type	Description	App plane	Ctrl plane	Data plane
Canini et al. [195]	Application	A testing tool for OpenFlow applications that uses model checking to explore the state space of the network and detect violations of network correctness properties.	✓		
Kuźniar et al. [204]	Application	Extends NICE by interfacing the model checker to real network devices, and vice-versa, in to support testing with real switches and capture a wider range of faults.	✓		✓
Ball et al. [205]	Application	Converts applications into first-order formulas and checks them against network-wide invariants, considering all possible sequence of events and topologies.	✓		
Nelson et al. [206]	Application	Converts applications and properties into Alloy specifications, which performs bounded correctness verification.	✓		
Al-Shaer and Al-Haj [193]	Network	Performs identification of misconfiguration at inter-switch and network-wide level using binary decision diagrams and model checking.	✓		✓
Zeng et al. [207]	Network	Checks network correctness in a distributed fashion using MapReduce [208] to distribute network state.	✓		✓
Khurshid et al. [192]	Network	Divides the network into a set of forwarding equivalence classes (FECs) and verifies affected FECs upon each modification.	✓		✓
Kazemian et al. [194]	Network	Converts network properties into reachability assertions and uses header space analysis [209] to send test packets and observe network behavior.	✓		✓
Zeng et al. [6]	Network	Generates a minimal set of packets necessary to test all possible rules.	✓		✓
Shelly et al. [196]	Network	A post-deployment failure injector to systematically introduce failures in the network.	✓		✓

network-wide invariants and topology constraints are defined using first-order logic. Vericon converts network applications into first-order formulas and uses a theorem prover to test application correctness under any sequence of events and with all admissible topologies. Even though the theorem prover used has no termination guarantee, evaluation indicated that many of the verification conditions generated are easy to verify and results are found at sub-second verification time.

Similar to Ball et al., Nelson et al. [206] propose FlowLog, a tierless language for developing SDN applications with first-order relational semantics, and use Alloy Analyzer [210] to check application correctness. FlowLog programs and desired properties are converted into Alloy specifications. The use of a tierless language is an advantage over Vericon, since it provides a robust and simplified approach for developing SDN applications, however, Alloy does not offer completeness, because it is necessary to bound the exploration depth.

2) Network correctness: FlowChecker [193] performs misconfiguration identification at inter-switch and network-wide level. FlowChecker encodes flow table configuration using *binary decision diagrams* and global behavior using state-machines. Network correctness properties are defined using *computational tree logic*, and then, similar to NICE, symbolic model checking is used to search the property violations.

Veriflow [192] is a tool used to check network correctness in real-time. Veriflow is implemented as a layer between the controller and the switches, which monitors communication in both directions. Veriflow divides the network into a set of equivalence classes (ECs), which represent sets of packets on which a forwarding action has the same effect. For each EC, Veriflow computes a graph representing how EC's packets will be forwarded. For each modification, Veriflow verifies affected ECs in order to check whether some network-wide correctness property (e.g., loops, black-holes) was violated.

NetPlumber [194] is a real-time policy checker based on header space analysis [209]. Header space analysis (HSA) is a protocol agnostic framework that defines a geometric model of packet processing. HSA allows the definition of a subspace, in the space of possible headers, and a set of operations that can be performed over an input header space and output another header space. NetPlumber builds a forwarding graph representing network reachability based on installed flow rules. Policies are defined using a language based on FML [211] and then, are converted into reachability assertions using HSA (e.g., after a set of transformations, an input header space must output a specific header space). In order to check policy violations, probe nodes placed in the network send and receive test packets and check them against policies. A distributed version of NetPlumber is also implemented, where forwarding ECs are assigned to different NetPlumber instances.

Kazemian et al. propose the *Automatic Test Packet Generation* (ATPG) framework [6], which generates a minimal set of packets that can be used to test network liveness, data plane misconfiguration and performance assertions. ATPG uses HSA to generate a set of packets to check all-pairs connectivity, then it selects one packet from each forwarding equivalence class and, finally, it models the problem as a min-set cover problem and uses an approximation to calculate the minimal set of test packets.

Zeng et al. [207] argue that most of policy checkers do not scale to large data centers with thousands of switches and millions of flow rules. They propose Libra, a policy checker that aims to improve scalability using parallel processing. Libra retrieves all flow tables and construct a forwarding graph which represents the network snapshot state. Then, Libra builds an application using MapReduce [208], a parallel programming model, to statically check network correctness. First, the forwarding graph is partitioned into a number of slices, one for each subnet. Then, the slices are assigned to multiple controllers which analyzes each slice in parallel, checking its properties with a graph library.

Notwithstanding the comprehensive testing routine which the controllers must undergo before deployment, frequently, there are complex errors that can only arise in the post-deployment scenarios. Chang et al. [196] propose Armageddon, a post-deployment failure injector to systematically introduce failures in the network. Armageddon pipeline is divided in the following steps: 1) Failure scenarios are defined, potentially using probabilistic failure models; 2) Failures are injected in the network following specified requirements; 3) A module checks network invariants to verify whether a policy was violated.

Summary: In this section we discussed efforts that focused on fault management issues of application plane and/or fault tolerance solutions placed on application plane: fault tolerant network application design, fault tolerance support at application level, application failure isolation, preservation of application state during failures and network applications that provide troubleshooting and correctness verification capabilities. One of the main benefits of SDN is the ability to create high-level network abstractions and manage the network through them, instead of specifying physical addresses and point-to-point links. FatTire [181] and RuleBricks [180] are important steps towards high-level fault tolerance management. However, many other fault tolerance aspects remain to be explored.

Recently, an increasing number of domain-specific languages were proposed for SDN [212], each with different goals. These languages provide features such as modularity and declarativity to assist the design of network applications. Even though such features might assist the construction of more secure applications, it is still necessary that fault tolerance and debugging capabilities are provided to achieve the same level of reliability and fault tolerance of cloud and big data applications.

Application failure isolation still demands improvement in order to achieve desired levels of performance. Presented solutions [47], [183] implement each application as a different process, however, the interface with the controller is a bottleneck that hampers scalability. Since this is a well-known issue in operating systems design, lessons learned from approaches such as *microkernel*, *exokernel* and *hybrid kernel* may be used and adapted in order to meet network performance requirements.

VIII. Open Issues and Lessons Learned

Analyzing interactions between layers, surveyed efforts, and ongoing networking trends, we identified a set of challenges and possibilities that future work in SDN may develop. These challenges are mainly related to (1) providing the same level of fault tolerance and performance that those found in legacy networks, (2) exploration of new possibilities that SDN brings, and (3) integration of SDN with new technologies. Open issues in data plane are mostly of the category (1), whereas open issues and challenges of control and application plane are more related to (2), and support for (3) require integration between SDN layers.

A. Data plane

Similarly to what we found in issues (Section IV) and efforts (Section V), open issues in data plane are mainly related to legacy networks. The main challenge is to match the performance and scalability of legacy fault recovery solutions.

a) Efficient fault recovery: SDN allows the development of novel fault recovery solutions, however, the need for coordination between multiple layers and network entities increases the overall recovery time. Even though legacy solutions have problems (e.g., slow convergence), they are widely implemented and were thoroughly tested and optimized through the years, enhancing their robustness and efficiency. Whereas, the majority of SDN efforts is deployed in experimental testbeds. Real deployments and validation, in conjunction with some of the future directions discussed later in this section (e.g., move more intelligence to data plane, integration between layers), may result in more efficient fault recovery mechanisms.

b) Standardization of fault tolerance features: In SDN, many solutions extend the southbound protocol behavior and/or repurpose header fields (e.g. to use priority field to assign backup paths) in order to support fault recovery mechanisms. However, the lack of standardization limits the adoption and practical use of these solutions. Newer OpenFlow specifications provide features related to QoS and traffic monitoring, which can be used to support novel fault tolerance mechanisms, even though these features are not directly related to fault recovery. Additionally, many switch vendors and network controllers only implement older versions of OpenFlow protocol. In summary, there is a gap between fault tolerance efforts and southbound protocol standardization, and between protocol specifications and available implementations. Future specifications of OpenFlow and other southbound protocols, such as OpFlex [29], may open new possibilities to fault tolerance research.

c) Additional capabilities in data plane: The initial proposal of SDN advocated the complete separation of control logic from network devices. However, real deployments, experiments, and many efforts suggest moving more intelligence back to devices, due to diverse reasons, from decrease recovery time to give more independence to switches. Some works propose simple modifications, such as the ability to detect traversed paths [78], [98], while others propose new abstractions to support stateful data forwarding [91]. This raises some research questions: How much, if any, intelligence should be placed in the network devices? Which trade-offs are involved? In which cases is this suited? Initiatives like P4 [213], a high-level language for programming switches' packet processing, give more independency to data plane and allow more complex logic to be placed in the network devices.

d) Integration with other infrastructure aspects: A limitation that we identified in some data plane efforts is how they restrain their use of SDN features to rework old solutions without considering integrating innovation with other infrastructure aspects, such as topology. For instance, it is possible to design fault management solutions that take advantage of a novel network organization hierarchy.

B. Control plane

As discussed before, logically centralized management may be physically distributed among multiple controllers. Most efforts that use physical distribution leverage current techniques, such as distributed filesystems, to achieve fault tolerance. However, a gap that we identified in this approach is that it does not fully take advantage of SDN capabilities. Approaches more specific to SDN and networking may open new possibilities and achieve better results than more generic methods.

a) Fault tolerant programming platforms: Most distributed control plane architectures share their *state* among replicas. This approach, at most, allows applications to configure consistency level desired. A more flexible programming platform could give control to network applications define different fault tolerance policies for different events and types of traffic. Additionally, the programming platform may support the definition of high-level fault tolerance objectives. For instance, latest versions of ONOS [55] allow programmers to create *intents*, which represent high-level control desires (e.g., connectivity between two hosts) that are translated and constantly enforced through low-level rules.

b) Geo-distributed recovery: Local fault tolerance may be achieved in a domain through the partitioning of management among independently managed clusters, where controllers serve as backups to others in case of failure. However, a failure that affects the whole domain (e.g. a disaster event) would require geo-distributed state redundancy in order to recover functionality. Authentication, latency, and inter-domain management are some of the issues that must be addressed in order to achieve efficient geo-distributed recovery. SDN facilitates coordination between different domains with different policies and access policies. Some efforts proposed inter-domain failure management mechanisms [117], [151], however, experiments showed performance issues, demonstrating that there is much to be improved. Gramoli et al. [148] showed that the performance can be improved by extending SDN with inter-domain communication primitives, indicating that southbound protocol support may help to achieve higher efficiency.

C. Application plane

The idea of programming network policies, services, and even switch packet processing, brings many new possibilities to networking. However, we identified that few efforts explore what these possibilities bring to fault management.

a) Fault tolerance abstractions: One of the main benefits of SDN is the possibility of network management through high-level terms. Many works propose abstractions for multiple aspects [214], such as network structure, modular composition of applications, and virtualization. However, few works explore the possibility of providing abstractions suited for fault management issues. Fault tolerance abstractions can be used to specify high-level fault tolerance strategies and fault tolerant constructs. Additionally, many fault tolerance mechanisms present some kind of trade-off, e.g. network resilience *vs* performance, network redundancy *vs* scalability. Abstractions can be provided to allow specification of different policies that would enforce different levels of fault tolerance, according to the desired trade-off. Some efforts already proposed methods to specify network failure planning at a high-level [180], [181], however other aspects of network fault management are yet to be explored.

D. Integration between layers

In this survey, a research gap that we identified in current efforts is that few works explore the possibilities that integration between multiple layers may provide. Instead, many efforts focus on exploration some specific issue of a layer. We argue that such integration is essential to network automation and comprehensive fault diagnosis.

a) Multi-layer coordination: In SDN, failures may appear at multiple and different layers. Failures manifest themselves in each layer through different symptoms and are detected by different mechanisms. This may mislead network administrators about the failure location, making failure diagnosis difficult. SDN's integrated resource management along with a consistent network view might be used to reason failure location and diagnosis. Besides, mechanisms at different layers might coordinate in order to mitigate failures. Heller et al. [202] propose a workflow to systematically integrate tools at different layers in order to troubleshoot errors, however, no implementation was presented.

b) Integration and coordination between approaches: Many fault management issues in SDN are related because they involve similar trade-offs and/or affect the same metric. Thus, in order to consistently address the network requirements, different approaches must be in tune. SDN programmability and logical centralization ease the integration of the whole failure management process, whereas in legacy networks it would be difficult to have such an approach because of their heterogeneous and closed nature.

c) Self-healing: Accurately determining a failure cause and providing a solution may take a non-negligible amount of time. Network administrators reported that most network failures require from about 30 minutes to more than an hour to solve and a month may have more than 100 error tickets [6]. One approach to address this issue is to provide self-healing capabilities [215] to the network in order to improve network fault management by automation of network recovery, reducing network administrator interference. Early discussion can be found in [216], where Sanchez et al. proposed a self-healing framework based on bayesian networks, but there is much to explore in order to achieve a robust self-healing SDN.

E. Integration with different requirements and scenarios

Since its beginning, SDN is associated with other network trends, such as network virtualization and data center management. A natural direction for future work is to explore intersections of SDN with other mechanisms and network requirements.

a) Network function virtualization integration: Concurrently with SDN, *network function virtualization* (NFV) [217] has gained momentum in the recent years as an efficient way to reduce costs and increase flexibility using network softwarization, with an increasing adoption by the industry. One popular approach is to use SDN's centralization to assist NFV orchestration [218], [219]. This adds a virtualization layer on top of SDN architecture, which also must meet reliability requirements [220]. SDN and NFV development and integration are still in their early stages, with growing number of works exploring their possibilities and full potential integration.

b) Quality of Service: Bandwidth guarantees are an important aspect of fault tolerance, particularly in environments with strict requirements, such as service providers and for time-critical applications. Even though OpenFlow protocol specifications [51], [221] improved QoS support by providing expressive QoS features (e.g., queues and meters) few fault tolerance techniques explore them in order to guarantee minimum bandwidth during failures.

c) Innovation on different scenarios: SDN can be integrated thereby providing programmability and logical centralization for a great diversity of environments, such as 5G infrastructure, Internet-of-Things management, virtual networks, wireless networks. It is also possible to use other approaches to leverage SDN capabilities. For example, Cui et al. [38] argue that big data processing can be used to improve network performance by extracting meaningful information from thousands of network devices. Ongoing research is already investigating possible directions [222], [223] to fully explore SDN potential.

IX. Conclusion

This work presents a comprehensive view of fault management in SDN. Our goal was to identify which fault management issues are present in SDN, how current efforts address those issues, what are the major contributions of those efforts and what are the major gaps in the ongoing researches.

We have identified fault management issues of each layer/interface. We have observed that most of fault management issues raised by SDN are related to its layered architecture and logical centralization of control. Faults in each layer may affect other layers in different aspects, for example, a faulty application may cause a black hole in the network, as well as failures in communication between layers (e.g., controller-switch communication). A logically centralized control plane is radically different from legacy networks, raising new issues, such as controller placement, control channel reliability and controller failure.

Surveyed efforts were classified according to their planes, issues, approaches and features. We discussed trade-offs of different approaches and their suitability to different scenarios. It was demonstrated that ongoing research has addressed most of the fault management issues presented by a split-architecture, demonstrating that SDN can be made robust enough to meet high-demand requirements, such as large data-centers deployment [224] and carrier networks [74].

Nevertheless, besides solving problems of SDN architecture and providing flexibility to handle issues already addressed by current methods in legacy networks, SDN's features may be the bridge to close the gap between open research problems and novel solutions. For example, logical centralization and programmability enabled real-time policy checking [192], [194] and fine-grained network debugging [60], [188].

Many efforts put SDN in a central role for the management of multiple networked environments, such as multi-tenant cloud networks, 5G mobile networks [223], wireless networks [225] and optical networks [226]. With the introduction of new elements that must be integrated in the network, such as NFV, and new requirements that must me met, fault management research has much to explore.

Acknowledgement

We are most grateful to Prof. Dr. Eduardo Feitosa for review, and Anibrata Pal for proofreading an early draft of this paper. The author Paulo Fonseca thanks the brazilian institution FAPEAM for student funding received during the development of this work.

References

[1] N. Feamster, J. Rexford, and E. Zegura, "The Road to SDN," *Queue*, vol. 11, pp. 20:20–20:40, Dec. 2013.

[2] M. F. Bari, R. Boutaba, R. Esteves, L. Z. Granville, M. Podlesny, M. G. Rabbani, Q. Zhang, and M. F. Zhani, "Data center network virtualization: A survey," *IEEE Communications Surveys & Tutorials*, vol. 15, pp. 909–928, Sept. 2013.

[3] S. Yu, M. Liu, W. Dou, X. Liu, and S. Zhou, "Networking for Big Data: A Survey," *IEEE Communications Surveys & Tutorials*, vol. PP, Sept. 2016.

[4] P. K. Agyapong, M. Iwamura, D. Staehle, W. Kiess, and A. Benjebbour, "Design considerations for a 5G network architecture," *IEEE Communications Magazine*, vol. 52, pp. 65–75, Nov. 2014.

[5] G. Maurice, D. Felipe, C. Camille, C. Christophe, S. Kazuhiko, Y. Xu, D. Pierre, S. Jean-Paul, L. Jonathan, and L. Stephen, "Downtime statistics of current cloud solutions," Tech. Rep., International Working Group on Cloud Computing Resiliency, 2013.

[6] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown, "Automatic test packet generation," in *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, (New York, NY, USA), pp. 241–252, ACM, 2012.

[7] A. Basu and J. Riecke, "Stability Issues in OSPF Routing," in *ACM SIGCOMM Computer Communication Review*, vol. 31, (New York, NY, USA), pp. 225–236, ACM, Aug. 2001.

[8] T. Benson, A. Akella, and D. A. Maltz, "Unraveling the Complexity of Network Management," in *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '09, (Boston, MA, USA), pp. 335–348, USENIX Association, Apr. 2009.

[9] U. C. Kozat, G. Liang, and K. KÃűkten, "On diagnosis of forwarding plane via static forwarding rules in Software Defined Networks," in *IEEE Conference on Computer Communications*, INFOCOM '14, (Toronto, Canada), pp. 1716–1724, April 2014.

[10] S. S. Lee, K.-Y. Li, K.-Y. Chan, G.-H. Lai, and Y.-C. Chung, "Path layout planning and software based fast failure detection in survivable OpenFlow networks," in *Design of Reliable Communication Networks (DRCN), 2014 10th International Conference on the*, pp. 1–8, IEEE, 2014.

[11] S. S. Lee, K.-Y. Li, K. Y. Chan, G.-H. Lai, and Y. C. Chung, "Software-based fast failure recovery for resilient openflow networks," in *Reliable Networks Design and Modeling (RNDM), 2015 7th International Workshop on*, (Munich,Germany), pp. 194–200, IEEE, Oct. 2015.

[12] S. Scott-Hayward, S. Natarajan, and S. Sezer, "A survey of security in software defined networks," *IEEE Communications Surveys & Tutorials*, vol. 18, pp. 623–654, First Quarter 2016.

[13] I. Ahmad, S. Namal, M. Ylianttila, and A. Gurtov, "Security in Software Defined Networks: A Survey," *IEEE Communications Surveys & Tutorials*, vol. 17, pp. 2317–2346, Fourth Quarter 2015.

[14] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, “Software-defined networking: A comprehensive survey,” *Proceedings of the IEEE*, vol. 103, pp. 14–76, Jan 2015.

[15] D. Kreutz, F. M. Ramos, and P. Verissimo, “Towards secure and dependable software-defined networks,” in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN ’13, (New York, NY, USA), pp. 55–60, ACM, Aug. 2013.

[16] A. S. da Silva, P. Smith, A. Mauthe, and A. Schaeffer-Filho, “Resilience support in software-defined networking: A survey,” *Computer Networks*, vol. 92, Part 1, pp. 189 – 207, Oct. 2015.

[17] J. Chen, J. Chen, F. Xu, M. Yin, and W. Zhang, “When Software Defined Networks Meet Fault Tolerance: A Survey,” in *Algorithms and Architectures for Parallel Processing: 15th International Conference, ICA3PP 2015*, (Zhangjiajie, China), pp. 351–368, Springer International Publishing, Nov. 2015.

[18] J. P. Sterbenz, D. Hutchison, E. K. Çetinkaya, A. Jabbar, J. P. Rohrer, M. SchÃűller, and P. Smith, “Resilience and survivability in communication networks: Strategies, principles, and survey of disciplines,” *Computer Networks*, vol. 54, no. 8, pp. 1245 – 1265, 2010.

[19] “Software-defined networking: The new norm for networks,” ONF White Paper, Open Networking Foundation, 2012.

[20] “OpenFlow Switch Specification Version 1.1.0.” http://archive.openflow.org/documents/openflow-spec-v1.1.0.pdf. Accessed: 2016-06-30.

[21] J. E. van der Merwe, S. Rooney, L. Leslie, and S. Crosby, “The Tempest-a practical framework for network programmability,” *IEEE Network*, vol. 12, pp. 20–28, May 1998.

[22] L. Yang, R. Dantu, T. Anderson, and R. Gopal, “Forwarding and Control Element Separation (ForCES) Framework,” RFC 3746, RFC Editor, April 2004.

[23] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, “Can the Production Network Be the Testbed?,” pp. 365–378, October 2010.

[24] S. Sezer, S. Scott-Hayward, P. K. Chouhan, B. Fraser, D. Lake, J. Finnegan, N. Viljoen, M. Miller, and N. Rao, “Are we ready for SDN? Implementation challenges for software-defined networks,” *IEEE Communications Magazine*, vol. 51, pp. 36–43, July 2013.

[25] C. Chaudet and Y. Haddad, “Wireless software defined networks: Challenges and opportunities,” in *Microwaves, Communications, Antennas and Electronics Systems (COMCAS), 2013 IEEE International Conference on*, (Tel Aviv, Israel), pp. 1–5, Oct 2013.

[26] S. Vissicchio, L. Vanbever, and O. Bonaventure, “Opportunities and research challenges of hybrid software defined networks,” *SIGCOMM Computer Communication Review*, vol. 44, pp. 70–75, Apr. 2014.

[27] N. Mckeown, T. Anderson, H. Balakrishnan, G. M. Parulkar, L. L. Peterson, J. Rexford, S. Shenker, J. S. Turner, and S. Louis, “OpenFlow: enabling innovation in campus networks,” *ACM SIGCOMM Computer Communication Review*, vol. 38, pp. 69–74, Mar. 2008.

[28] A. Doria, J. H. Salim, R. Haas, H. Khosravi, W. Wang, L. Dong, R. Gopal, and J. Halpern, “Forwarding and control element separation (forces) protocol specification,” RFC 5810, RFC Editor, March 2010. http://www.rfc-editor.org/rfc/rfc5810.txt.

[29] M. Smith, M. Dvorkin, Y. Laribi, V. Pandey, P. Garg, and N. Weidenbacher, “OpFlex Control Protocol,” Internet-Draft draft-smith-opflex-00, IETF Secretariat, April 2014. http://www.ietf.org/internet-drafts/draft-smith-opflex-00.txt.

[30] Y. Jarraya, T. Madi, and M. Debbabi, “A Survey and a Layered Taxonomy of Software-Defined Networking,” *IEEE Communications Surveys & Tutorials*, vol. 16, pp. 1955–1980, Fourth Quarter 2014.

[31] R. Braga, E. Mota, and A. Passito, “Lightweight ddos flooding attack detection using nox/openflow,” in *35th IEEE Conference on Local Computer Networks*, LCN ’10, (Denver, CO, USA), pp. 408–415, Oct 2010.

[32] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, “FlowVisor: A Network Virtualization Layer,” Tech. Rep. OPENFLOW-TR-2009-1, Deutsche Telekom Inc. R&D Lab, Stanford, Nicira Networks, 2009.

[33] G. Wang, T. E. Ng, and A. Shaikh, “Programming Your Network at Run-time for Big Data Applications,” in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN ’12, (Helsinki, Finland), pp. 103–108, ACM, Aug. 2012.

[34] W. Xia, Y. Wen, C. H. Foh, D. Niyato, and H. Xie, “A survey on software-defined networking,” *IEEE Communications Surveys & Tutorials*, vol. 17, pp. 27–51, First Quarter 2015.

[35] B. A. A. Nunes, M. Mendonca, X. N. Nguyen, K. Obraczka, and T. Turletti, “A survey of software-defined networking: Past, present, and future of programmable networks,” *IEEE Communications Surveys & Tutorials*, vol. 16, pp. 1617–1634, Third Quarter 2014.

[36] R. Jain and S. Paul, “Network Virtualization and Software Defined Networking for Cloud Computing: A Survey,” *IEEE Communications Magazine*, vol. 51, pp. 24–31, November 2013.

[37] L. E. Li, Z. M. Mao, and J. Rexford, “Toward Software-Defined Cellular Networks,” in *2012 European Workshop on Software Defined Networking*, (Hague, Netherlands), pp. 7–12, Oct 2012.

[38] L. Cui, F. R. Yu, and Q. Yan, “When big data meets software-defined networking: Sdn for big data and big data for sdn,” *IEEE Network*, vol. 30, pp. 58–65, January 2016.

[39] W. Xia, P. Zhao, Y. Wen, and H. Xie, “A survey on data center networking (dcn): Infrastructure and operations,” *IEEE Communications Surveys & Tutorials*, vol. 19, pp. 640–656, Firstquarter 2017.

[40] D. B. Rawat and S. R. Reddy, “Software defined networking architecture, security and energy efficiency: A survey,” *IEEE Communications Surveys & Tutorials*, vol. 19, pp. 325–346, Firstquarter 2017.

[41] M. Walraed-Sullivan, A. Vahdat, and K. Marzullo, “Aspen Trees: Balancing Data Center Fault Tolerance, Scalability and Cost,” in *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT ’13, (Santa Barbara, California, USA), pp. 85–96, ACM, 2013.

[42] M. Al-Fares, A. Loukissas, and A. Vahdat, “A Scalable, Commodity Data Center Network Architecture,” in *Proceedings of the 2008 ACM Conference on SIGCOMM*, SIGCOMM ’08, (New York, NY, USA), pp. 63–74, ACM, 2008.

[43] V. Liu, D. Halperin, A. Krishnamurthy, and T. Anderson, “F10: A Fault-Tolerant Engineered Network,” in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, NSDI ’13, (Lombard, IL, USA), pp. 399–412, USENIX Association, Apr. 2013.

[44] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu, “DCell: A Scalable and Fault-tolerant Network Structure for Data Centers,” in *Proceedings of the 2008 ACM Conference on SIGCOMM*, SIGCOMM ’08, (New York, NY, USA), pp. 75–86, ACM, 2008.

[45] M. Walraed-Sullivan, K. Marzullo, and A. Vahdat, “Scalability vs. fault tolerance in Aspen trees,” Technical Report MSR-TR-2013-21, 2013.

[46] R. d. S. Couto, S. Secci, M. E. M. Campista, and L. H. M. K. Costa, “Reliability and survivability analysis of data center network topologies,” *Journal of Network and Systems Management*, vol. 24, no. 2, pp. 346–392, 2016.

[47] S. Shin, Y. Song, T. Lee, S. Lee, J. Chung, P. Porras, V. Yegneswaran, J. Noh, and B. B. Kang, “Rosemary: A robust, secure, and high-performance network operating system,” in *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, (Scottsdale, Arizona, USA), pp. 78–89, ACM, 2014.

[48] N. Katta, H. Zhang, M. Freedman, and J. Rexford, “Ravana : Controller Fault-Tolerance in Software-Defined Networking,” in *Proceedings of the ACM SIGCOMM Symposium on SDN Research*, SOSR ’15, (Santa Clara, CA, USA), June 2015.

[49] M. Desai and T. Nandagopal, “Coping with link failures in centralized control plane architectures,” in *Communication Systems and Networks (COMSNETS), 2010 Second International Conference on*, (Bangalore, India), pp. 1–10, IEEE, Jan. 2010.

[50] B. Heller, R. Sherwood, and N. McKeown, “The controller placement problem,” in *Proceedings of the First Workshop on Hot Topics in Software Defined Networking*, HotSDN ’12, (New York, NY, USA), pp. 7–12, ACM, Aug. 2012.

[51] “OpenFlow Switch Specification Version 1.2.0.” https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.2.pdf. Accessed: 2016-06-30.

[52] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, “NOX: towards an operating system for networks,” *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, pp. 105–110, 2008.

[53] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker, “Onix: A Distributed Control Platform for Large-scale Production Networks,” in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI’10, (Berkeley, CA, USA), pp. 1–6, USENIX Association, 2010.

[54] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann, “Logically Centralized?: State Distribution Trade-offs in Software Defined Networks,” in *Proceedings of the First Workshop on Hot Topics*

in Software Defined Networking, HotSDN '12, (New York, NY, USA), pp. 1–6, ACM, Aug. 2012.

[55] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, W. Snow, G. Parulkar, B. O'Connor, and P. Radoslavov, "ONOS: Towards an Open, Distributed SDN OS," *Proceedings of the third workshop on Hot topics in software defined networking*, pp. 1–6, Aug. 2014.

[56] "Floodlight, Open SDN controller." http://www.projectfloodlight.org/. Accessed: 2016-06-30.

[57] A. S. Tanenbaum, J. N. Herder, and H. Bos, "Can we make operating systems reliable and secure?," *Computer*, vol. 39, pp. 44–51, May 2006.

[58] A. Wundsam, D. Levin, S. Seetharaman, A. Feldmann, *et al.*, "OFRewind: Enabling Record and Replay Troubleshooting for Networks," in *USENIX Annual Technical Conference*, (Portland, OR, USA), June 2011.

[59] C. Scott, A. Wundsam, S. Whitlock, A. Or, E. Huang, K. Zarifis, and S. Shenker, "How did we get into this mess? Isolating fault-inducing inputs to SDN control software," Tech. Rep. UCB/EECS-2013-8, EECS Dept., Univ. California, Berkeley, 2013.

[60] N. Handigol, B. Heller, V. Jeyakumar, D. Maziéres, and N. McKeown, "Where is the debugger for my software-defined network?," in *Proceedings of the first workshop on Hot topics in software defined networking*, pp. 55–60, ACM, Aug. 2012.

[61] L. L. Pullum, *Software fault tolerance techniques and implementation*. Artech House, 2001.

[62] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "OpenFlow: Meeting carrier-grade recovery requirements," *Computer Communications*, vol. 36, no. 6, pp. 656–665, 2013.

[63] D. Katz and D. Ward, "Bidirectional Forwarding Detection (BFD)." RFC 5880 (Proposed Standard), June 2010. Updated by RFC 7419.

[64] N. L. van Adrichem, B. J. Van Asten, and F. A. Kuipers, "Fast recovery in software-defined networks," in *Software Defined Networks (EWSDN), 2014 Third European Workshop on*, (Budapest, Hungary), pp. 61–66, IEEE, Sept. 2014.

[65] J. Kempf, E. Bellagamba, A. Kern, D. Jocha, A. Takács, and P. Sköldström, "Scalable fault management for OpenFlow," in *Communications (ICC), 2012 IEEE International Conference on*, pp. 6606–6610, IEEE, 2012.

[66] R. Aggarwal, K. Kompella, T. Nadeau, and G. Swallow, "Bidirectional Forwarding Detection (BFD) for MPLS Label Switched Paths (LSPs)." RFC 5884 (Proposed Standard), June 2010. Updated by RFC 7726.

[67] D. Gyllstrom, N. Braga, and J. Kurose, "Recovery from link failures in a Smart Grid communication network using OpenFlow," in *Smart Grid Communications (SmartGridComm), 2014 IEEE International Conference on*, pp. 254–259, IEEE, 2014.

[68] L. Liu, H. Y. Choi, T. Tsuritani, I. Morita, R. Casellas, R. Martínez, and R. Muñoz, "First proof-of-concept demonstration of OpenFlow-controlled elastic optical networks employing flexible transmitter/receiver," in *Proc. Int. Conf. Photon. Switching*, vol. 3, (Corsica, France), p. 5, Sept. 2012.

[69] D. Kotani and Y. Okabe, "Fast failure detection of openflow channels," in *Proceedings of the Asian Internet Engineering Conference*, pp. 32–39, ACM, 2015.

[70] D. Ahr and G. Reinelt, "A tabu search algorithm for the min–max k-chinese postman problem," *Computers & operations research*, vol. 33, no. 12, pp. 3403–3422, 2006.

[71] A. Sgambelluri, A. Giorgetti, F. Cugini, F. Paolucci, and P. Castoldi, "OpenFlow-based segment protection in Ethernet networks," *Journal of Optical Communications and Networking*, vol. 5, no. 9, pp. 1066–1075, 2013.

[72] A. Sgambelluri, A. Giorgetti, F. Cugini, F. Paolucci, and P. Castoldi, "Effective flow protection in OpenFlow rings," in *National Fiber Optic Engineers Conference*, Optical Society of America, 2013.

[73] D. Staessens, S. Sharma, D. Colle, M. Pickavet, and P. Demeester, "Software defined networking: Meeting carrier grade requirements," in *Local & Metropolitan Area Networks (LANMAN), 2011 18th IEEE Workshop on*, pp. 1–6, IEEE, 2011.

[74] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "Enabling fast failure recovery in OpenFlow networks," in *Design of Reliable Communication Networks (DRCN), 2011 8th International Workshop on the*, pp. 164–171, IEEE, 2011.

[75] P. Goncalves, A. Martins, D. Corujo, and R. Aguiar, "A fail-safe SDN bridging platform for cloud networks," in *Telecommunications Network Strategy and Planning Symposium (Networks), 2014 16th International*, pp. 1–6, IEEE, 2014.

[76] R. M. Ramos, M. Martinello, and C. Esteve Rothenberg, "Slickflow: Resilient source routing in data center networks unlocked by openflow," in *Local Computer Networks (LCN), 2013 IEEE 38th Conference on*, pp. 606–613, IEEE, 2013.

[77] A. Atlas and A. Zinin, "Basic Specification for IP Fast Reroute: Loop-Free Alternates." RFC 5286 (Proposed Standard), Sept. 2008.

[78] W. Braun and M. Menth, "Scalable resilience for software-defined networking using loop-free alternates with loop detection," in *Network Softwarization (NetSoft), 2015 1st IEEE Conference on*, (London, UK), pp. 1–6, IEEE, 2015.

[79] M. Kuźniar, P. Perešíni, N. Vasić, M. Canini, and D. Kostić, "Automatic failure recovery for software-defined networks," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pp. 159–160, ACM, 2013.

[80] J. Liu, A. Panda, A. Singla, B. Godfrey, M. Schapira, and S. Shenker, "Ensuring connectivity via data plane mechanisms," in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '13, (Lombard, IL, USA), pp. 113–126, USENIX Association, Apr. 2013.

[81] B. Charron-Bost, J. L. Welch, and J. Widder, "Link reversal: How to play better to work less," Technical Report MIT-CSAIL-TR-2011-xxx, 2009.

[82] M. Borokhovich, L. Schiff, and S. Schmid, "Provable data plane connectivity with local fast failover: Introducing openflow graph algorithms," in *Proceedings of the third workshop on Hot topics in software defined networking*, (Chicago, USA), pp. 121–126, ACM, 2014.

[83] M. Borokhovich and S. Schmid, "How (not) to shoot in your foot with SDN local fast failover," in *Principles of Distributed Systems*, pp. 68–82, Springer International Publishing, 2013.

[84] S. Sharma, D. Staessens, D. Colle, D. Palma, J. Goncalves, M. Pickavet, L. Cordeiro, and P. Demeester, "Demonstrating resilient quality of service in software defined networking," in *Computer Communications Workshops (INFOCOM WKSHPS), 2014 IEEE Conference on*, (Toronto, Canada), pp. 133–134, IEEE, 2014.

[85] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "In-band control, queuing, and failure recovery functionalities for OpenFlow," *Network, IEEE*, vol. 30, no. 1, pp. 106–112, 2016.

[86] K. Phemius and M. Bouet, "Implementing OpenFlow-based resilient network services," in *Cloud Networking (CLOUDNET), 2012 IEEE 1st International Conference on*, pp. 212–214, IEEE, 2012.

[87] H. H. Liu, S. Kandula, R. Mahajan, M. Zhang, and D. Gelernter, "Traffic engineering with forward fault correction," in *ACM SIGCOMM Computer Communication Review*, vol. 44, pp. 527–538, ACM, 2014.

[88] S. Paris, G. Paschos, and J. Leguay, "Dynamic control for failure recovery and flow reconfiguration in SDN," in *Design of Reliable Communication Networks (DRCN), 2016 International Workshop on the*, (Paris, France), IEEE, 2016.

[89] S. Nelakuditi, S. Lee, Y. Yu, Z.-L. Zhang, and C.-N. Chuah, "Fast local rerouting for handling transient link failures," *IEEE/ACM Transactions on Networking (ToN)*, vol. 15, no. 2, pp. 359–372, 2007.

[90] C. Cascone, L. Pollini, D. Sanvito, A. Capone, and B. Sansò, "SPIDER: fault resilient SDN pipeline with recovery delay guarantees," *CoRR*, vol. abs/1511.05490, 2015.

[91] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "OpenState: programming platform-independent stateful OpenFlow applications inside the switch," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 44–51, 2014.

[92] A. Capone, C. Cascone, A. Q. Nguyen, and B. Sanso, "Detour planning for fast and reliable failure recovery in SDN with OpenState," in *Design of Reliable Communication Networks (DRCN), 2015 11th International Conference on the*, (Kansas City, USA), pp. 25–32, IEEE, 2015.

[93] N. Sahri and K. Okamura, "Fast failover mechanism for software defined networking: OpenFlow based," in *Proceedings of The Ninth International Conference on Future Internet Technologies*, (Tokyo, Japan), p. 16, ACM, 2014.

[94] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, (Melbourne, Australia), pp. 267–280, ACM, 2010.

[95] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: scaling flow management for high-performance networks," in *ACM SIGCOMM Computer Communication Review*, vol. 41, pp. 254–265, ACM, 2011.

[96] N. Kitsuwan, D. B. Payne, and M. Ruffini, "A novel protection design for OpenFlow-based networks," in *Transparent Optical Networks (ICTON), 2014 16th International Conference on*, (Graz, Austria), pp. 1–5, IEEE, 2014.

[97] H. Kim, J. R. Santos, Y. Turner, M. Schlansker, J. Tourrilhes, and N. Feamster, "Coronet: Fault tolerance for software defined networks,"

in *Network Protocols (ICNP), 2012 20th IEEE International Conference on*, (Austin, Texas, USA), pp. 1–2, IEEE, 2012.

[98] B. Stephens, A. L. Cox, and S. Rixner, "Scalable multi-failure fast failover via forwarding table compression," in *Proceedings of the ACM SIGCOMM Symposium on SDN Research*, SOSR '16, (Santa Clara, CA, USA), June 2016.

[99] B. Stephens, A. L. Cox, and S. Rixner, "Plinko: building provably resilient forwarding tables," in *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, p. 26, ACM, 2013.

[100] P. M. Mohan, T. Truong-Huu, and M. Gurusamy, "TCAM-Aware Local Rerouting for Fast and Efficient Failure Recovery in Software Defined Networks," in *2015 IEEE Global Communications Conference (GLOBECOM)*, pp. 1–6, IEEE, 2015.

[101] Y. Yu, C. Shanzhi, L. Xin, and W. Yan, "A framework of using OpenFlow to handle transient link failure," in *Transportation, Mechanical, and Electrical Engineering (TMEE), 2011 International Conference on*, pp. 2050–2053, IEEE, 2011.

[102] P. Zeng, K. Nguyen, Y. Shen, and S. Yamada, "On the resilience of software defined routing platform," in *Network Operations and Management Symposium (APNOMS), 2014 16th Asia-Pacific*, pp. 1–4, IEEE, 2014.

[103] M. R. Nascimento, C. E. Rothenberg, M. R. Salvador, C. N. Corrêa, S. C. de Lucena, and M. F. Magalhães, "Virtual routers as a service: the RouteFlow approach leveraging software-defined networks," in *Proceedings of the 6th International Conference on Future Internet Technologies*, pp. 34–37, ACM, 2011.

[104] D. Kim and J.-M. Gil, "Reliable and Fault-Tolerant Software-Defined Network Operations Scheme for Remote 3D Printing," *Journal of Electronic Materials*, vol. 44, no. 3, pp. 804–814, 2015.

[105] C.-Y. Chu, K. Xi, M. Luo, and H. J. Chao, "Congestion-aware single link failure recovery in hybrid sdn networks," in *Computer Communications (INFOCOM), 2015 IEEE Conference on*, pp. 1086–1094, IEEE, 2015.

[106] M. Markovitch and S. Schmid, "SHEAR: A Highly Available and Flexible Network Architecture Marrying Distributed and Logically Centralized Control Planes," in *2015 IEEE 23rd International Conference on Network Protocols (ICNP)*, pp. 78–89, Nov 2015.

[107] O. Tilmans and S. Vissicchio, "IGP-as-a-Backup for Robust SDN Networks," in *Network and Service Management (CNSM), 2014 10th International Conference on*, pp. 127–135, IEEE, 2014.

[108] J. Li, J. Hyun, J.-H. Yoo, S. Baik, and J. W.-K. Hong, "Scalable failover method for data center networks using openflow," in *Network Operations and Management Symposium (NOMS), 2014 IEEE*, pp. 1–6, IEEE, 2014.

[109] J. T. Araújo, R. Landa, R. G. Clegg, and G. Pavlou, "Software-defined network support for transport resilience," in *Network Operations and Management Symposium (NOMS), 2014 IEEE*, pp. 1–8, IEEE, 2014.

[110] D. Kotani, K. Suzuki, and H. Shimonishi, "A design and implementation of OpenFlow controller handling IP multicast with fast tree switching," in *Applications and the Internet (SAINT), 2012 IEEE/IPSJ 12th International Symposium on*, pp. 60–67, IEEE, 2012.

[111] D. Kotani, K. Suzuki, and H. Shimonishi, "A Multicast Tree Management Method Supporting Fast Failure Recovery and Dynamic Group Membership Changes in OpenFlow Networks," *Journal of Information Processing*, vol. 24, no. 2, pp. 395–406, 2016.

[112] T. Pfeiffenberger, J. L. Du, P. Bittencourt Arruda, and A. Anzaloni, "Reliable and flexible communications for power systems: Fault-tolerant multicast with SDN/OpenFlow," in *New Technologies, Mobility and Security (NTMS), 2015 7th International Conference on*, pp. 1–6, IEEE, 2015.

[113] H. Takahashi and A. Matsuyama, "An approximate solution for the Steiner problem in graphs," *Math. Japonica*, vol. 24, no. 6, pp. 573–577, 1980.

[114] M. Charikar, C. Chekuri, T.-y. Cheung, Z. Dai, A. Goel, S. Guha, and M. Li, "Approximation algorithms for directed Steiner problems," *Journal of Algorithms*, vol. 33, no. 1, pp. 73–91, 1999.

[115] V. R. Raja, A. Pandey, and C.-H. Lung, "An Openflow-Based Approach to Failure Detection and Protection for a Multicasting Tree," in *Wired/Wireless Internet Communications*, pp. 211–224, Springer, 2015.

[116] J. Nagano and N. Shinomiya, "A failure recovery method based on cycle structure and its verification by OpenFlow," in *Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on*, pp. 298–303, IEEE, 2013.

[117] M. Gerola, M. Santuari, E. Salvadori, S. Salsano, P. L. Ventre, M. Campanella, F. Lombardo, and G. Siracusano, "ICONA: Inter Cluster ONOS Network Application," in *Network Softwarization (NetSoft), 2015 1st IEEE Conference on*, pp. 1–2, IEEE, 2015.

[118] L. Liu, R. Muñoz, R. Casellas, T. Tsuritani, R. Martínez, and I. Morita, "OpenSlice: an OpenFlow-based control plane for spectrum sliced elastic optical path networks," *Optics express*, vol. 21, no. 4, pp. 4194–4204, 2013.

[119] A. Patel, P. Ji, and T. Wang, "QoS-aware optical burst switching in OpenFlow based software-defined optical networks," in *Optical Network Design and Modeling (ONDM), 2013 17th International Conference on*, pp. 275–280, IEEE, 2013.

[120] L. Liu, W.-R. Peng, R. Casellas, T. Tsuritani, I. Morita, R. Martinez, R. Muñoz, M. Suzuki, and S. B. Yoo, "Dynamic OpenFlow-Based Lightpath Restoration in Elastic Optical Networks on the GENI Testbed," *Journal of Lightwave Technology*, vol. 33, no. 8, pp. 1531–1539, 2015.

[121] A. Giorgetti, F. Paolucci, F. Cugini, and P. Castoldi, "Fast restoration in SDN-based flexible optical networks," in *Optical Fiber Communication Conference*, pp. Th3B–2, Optical Society of America, 2014.

[122] H. Yang, L. Cheng, J. Yuan, J. Zhang, Y. Zhao, and Y. Lee, "Multipath protection for data center services in OpenFlow-based software defined elastic optical networks," *Optical Fiber Technology*, vol. 23, pp. 108–115, 2015.

[123] Y. Sone, A. Watanabe, W. Imajuku, Y. Tsukishima, B. Kozicki, H. Takara, and M. Jinno, "Bandwidth squeezed restoration in spectrum-sliced elastic optical path networks," *Journal of Optical Communications and Networking*, vol. 3, no. 3, pp. 223–233, 2011.

[124] A. Detti, C. Pisa, S. Salsano, and N. Blefari-Melazzi, "Wireless Mesh Software Defined Networks (wmSDN)," in *2013 IEEE 9th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pp. 89–95, Oct 2013.

[125] S. Salsano, G. Siracusano, A. Detti, C. Pisa, P. L. Ventre, and N. Blefari-Melazzi, "Controller selection in a wireless mesh SDN under network partitioning and merging scenarios," *CoRR*, vol. abs/1406.2470, 2014.

[126] Z. Wang, J. Wu, Y. Wang, N. Qi, and J. Lan, "Survivable Virtual Network Mapping using optimal backup topology in virtualized SDN," *Communications, China*, vol. 11, no. 2, pp. 26–37, 2014.

[127] M. Yu, Y. Yi, J. Rexford, and M. Chiang, "Rethinking virtual network embedding: substrate support for path splitting and migration," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 17–29, 2008.

[128] T. Watanabe, T. Omizo, T. Akiyama, and K. Iida, "ResilientFlow: Deployments of distributed control channel maintenance modules to recover SDN from unexpected failures," in *Design of Reliable Communication Networks (DRCN), 2015 11th International Conference on the*, (Kansas City, USA), pp. 211–218, IEEE, 2015.

[129] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[130] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems.," in *USENIX Annual Technical Conference*, vol. 8, p. 9, 2010.

[131] M. Johns, *Getting Started with Hazelcast*. Packt Publishing Ltd, 2015.

[132] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, (Philadelphia, PA), pp. 305–319, USENIX Association, 2014.

[133] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat, "Providing high availability using lazy replication," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 4, pp. 360–391, 1992.

[134] "HP SDN Controller Architecture," tech. rep., Hewlett-Packard Development Company, L.P., September 2013. Available on https://goo.gl/9knpJT.

[135] O. Alliance, *OSGI service platform, release 3*. IOS Press, Inc., 2003.

[136] S. Vinoski, "Advanced message queuing protocol," *IEEE Internet Computing*, no. 6, pp. 87–89, 2006.

[137] E. S. Spalla, D. R. Mafioletti, A. B. Liberato, C. Rothenberg, L. Camargos, R. d. S. VillaÃğa, and M. Martinello, "Resilient Strategies to SDN: An Approach Focused on Actively Replicated Controllers," in *Computer Networks and Distributed Systems (SBRC), 2015 XXXIII Brazilian Symposium on*, pp. 246–259, May 2015.

[138] "OpenReplica Coordination Service." http://openreplica.org/. Accessed: 2016-06-30.

[139] L. Lamport, "Paxos Made Simple," *ACM SIGACT News*, vol. 32, no. 4, p. 34, 2001.

[140] F. Botelho, A. Bessani, F. M. V. Ramos, and P. Ferreira, "On the Design of Practical Fault-Tolerant SDN Controllers," in *2014 Third European Workshop on Software Defined Networks*, pp. 73–78, Sept 2014.

[141] A. Bessani, J. Sousa, and E. E. P. Alchieri, “State machine replication for the masses with bft-smart,” in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 355–362, June 2014.

[142] P. Fonseca, R. Bennesby, E. Mota, and A. Passito, “A replication component for resilient openflow-based networking,” in *Network Operations and Management Symposium*, NOMS ’12, (Maui, HI, USA), pp. 933–939, IEEE, 2012.

[143] V. Pashkov, A. Shalimov, and R. Smeliansky, “Controller failover for SDN enterprise networks,” in *International Science and Technology Conference (Modern Networking Technologies) (MoNeTeC)*, pp. 1–6, Oct 2014.

[144] A. Tootoonchian and Y. Ganjali, “HyperFlow: A Distributed Control Plane for OpenFlow,” in *Proceedings of the 2010 Internet Network Management Conference on Research on Enterprise Networking*, INM/WREN’10, (Berkeley, CA, USA), pp. 1–6, USENIX Association, 2010.

[145] K. Phemius, M. Bouet, and J. Leguay, “Disco: Distributed multi-domain SDN controllers,” in *Network Operations and Management Symposium*, NOMS ’14, (Istambul,Turkey), pp. 1–4, IEEE, 2014.

[146] B. M. Oki and B. H. Liskov, “Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems,” *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, vol. 62, pp. 8–17, 1988.

[147] P. Fonseca, R. Bennesby, E. Mota, and A. Passito, “Resilience of SDNs based on active and passive replication mechanisms,” in *Global Communications Conference (GLOBECOM), 2013 IEEE*, (Atlanta, GA, USA), pp. 2188–2193, IEEE, Dec. 2013.

[148] V. Gramoli, G. Jourjon, and O. Mehani, “Disaster-tolerant storage with SDN,” in *International Conference on Networked Systems*, pp. 293–307, Springer, 2015.

[149] Y.-C. Chan, K. Wang, and Y.-H. Hsu, “Fast controller failover for multi-domain software-defined networks,” in *Networks and Communications (EuCNC), 2015 European Conference on*, pp. 370–374, IEEE, 2015.

[150] M. Obadia, M. Bouet, J. Leguay, K. Phemius, and L. Iannone, “Failover mechanisms for distributed SDN controllers,” in *Network of the Future (NOF), 2014 International Conference and Workshop on the*, pp. 1–6, IEEE, 2014.

[151] K. Kuroki, N. Matsumoto, and M. Hayashi, “Scalable OpenFlow controller redundancy tackling local and global recoveries,” in *The Fifth International Conference on Advances in Future Internet*, pp. 61–66, Citeseer, 2013.

[152] K. Kuroki, M. Fukushima, M. Hayashi, and N. Matsumoto, “Redundancy Method for Highly Available OpenFlow Controller,” *International Journal on Advances in Internet Technology Volume 7, Number 1 & 2, 2014*, 2014.

[153] L. F. Muller, R. R. Oliveira, M. C. Luizelli, L. P. Gaspary, and M. P. Barcellos, “Survivor: an enhanced controller placement strategy for improving SDN survivability,” in *Global Communications Conference (GLOBECOM), 2014 IEEE*, pp. 1909–1915, IEEE, 2014.

[154] D. Li, L. Ruan, L. Xiao, M. Zhu, W. Duan, Y. Zhou, M. Chen, Y. Xia, and M. Zhu, “High Availability for Non-stop Network Controller,” in *Proceedings of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks*, Oct. 2014.

[155] K. Nguyen, Q. T. Minh, and S. Yamada, “A Software-Defined Networking Approach for Disaster-Resilient WANs,” in *2013 22nd International Conference on Computer Communication and Networks (ICCCN)*, (Nassau, Bahamas), pp. 1–5, July 2013.

[156] Y. Zhang, N. Beheshti, and M. Tatipamula, “On resilience of split-architecture networks,” in *Global Telecommunications Conference*, GLOBECOM ’11, (Houston, TX, USA), pp. 1–6, IEEE, Dec. 2011.

[157] M. Guo and P. Bhattacharya, “Controller Placement for Improving Resilience of Software-Defined Networks,” in *Networking and Distributed Computing, 2013 Fourth International Conference on*, (Los Angeles, CA, USA), pp. 23–27, IEEE, Dec. 2013.

[158] Y. Hu, W. Wendong, X. Gong, X. Que, and C. Shiduan, “Reliability-aware controller placement for Software-Defined Networks,” in *Integrated Network Management, 2013 IFIP/IEEE International Symposium on*, IM 2013, (Ghent, Belgium), pp. 672–675, IEEE, May 2013.

[159] F. J. Ros and P. M. Ruiz, “Five nines of southbound reliability in software-defined networks,” in *Proceedings of the Third ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN ’14, (New York, NY, USA), pp. 31–36, ACM, Aug. 2014.

[160] H. Li, P. Li, S. Guo, and S. Yu, “Byzantine-resilient secure software-defined networks with multiple controllers,” in *2014 IEEE International Conference on Communications*, (Sydney, Australia), pp. 695–700, June 2014.

[161] A. S.-W. Tam, K. Xi, and H. J. Chao, “Scalability and resilience in data center networks: Dynamic flow reroute as an example,” in *Global Telecommunications Conference*, GLOBECOM ’11, (Houston, TX, USA), pp. 1–6, IEEE, Dec. 2011.

[162] N. Beheshti and Y. Zhang, “Fast failover for control traffic in software-defined networks,” in *Global Telecommunications Conference*, GLOBECOM ’12, (Anaheim, CA, USA), pp. 2665–2670, IEEE, Dec. 2012.

[163] V. D. Philip and Y. Gourhant, “Cross-control: A scalable multi-topology fault restoration mechanism using logically centralized controllers,” in *IEEE 15th International Conference on High Performance Switching and Routing (HPSR)*, (Vancouver, Canada), pp. 57–63, July 2014.

[164] A.-W. Tam, K. Xi, and H. Chao, “Use of devolved controllers in data center networks,” in *Computer Communications Workshops (INFOCOM WKSHPS), 2011 IEEE Conference on*, pp. 596–601, April 2011.

[165] H. Li, P. Li, S. Guo, and A. Nayak, “Byzantine-resilient secure software-defined networks with multiple controllers in cloud,” *IEEE Transactions on Cloud Computing*, vol. 2, pp. 436–447, Oct 2014.

[166] G. Yao, J. Bi, and L. Guo, “On the cascading failures of multi-controllers in software defined networks,” in *Network Protocols (ICNP), 2013 21st IEEE International Conference on*, pp. 1–2, IEEE, 2013.

[167] D. Hock, M. Hartmann, S. Gebert, M. Jarschel, T. Zinner, and P. Tran-Gia, “Pareto-optimal resilient controller placement in SDN-based core networks,” in *Proceedings of the 2013 25th International Teletraffic Congress (ITC)*, pp. 1–9, IEEE, sep 2013.

[168] W. L. Goffe, G. D. Ferrier, and J. Rogers, “Global optimization of statistical functions with simulated annealing,” *Journal of Econometrics*, vol. 60, no. 1, pp. 65–99, 1994.

[169] C. Swamy and D. B. Shmoys, “Fault-tolerant facility location,” *ACM Transactions on Algorithms (TALG)*, vol. 4, no. 4, p. 51, 2008.

[170] M. Castro, B. Liskov, *et al.*, “Practical byzantine fault tolerance,” in *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, vol. 99 of *OSDI ’99*, pp. 1–6, USENIX Association, 1999.

[171] A. E. Motter and Y.-C. Lai, “Cascade-based attacks on complex networks,” *Physical Review E*, vol. 66, no. 6, p. 065102, 2002.

[172] S. V. Buldyrev, R. Parshani, G. Paul, H. E. Stanley, and S. Havlin, “Catastrophic cascade of failures in interdependent networks,” *Nature*, vol. 464, no. 7291, pp. 1025–1028, 2010.

[173] A. Clauset, M. E. Newman, and C. Moore, “Finding community structure in very large networks,” *Physical review E*, vol. 70, no. 6, p. 066111, 2004.

[174] D. Hock, M. Hartmann, S. Gebert, T. Zinner, and P. Tran-Gia, “POCO-PLC: Enabling dynamic pareto-optimal resilient controller placement in SDN networks,” in *Computer Communications Workshops (INFOCOM WKSHPS), 2014 IEEE Conference on*, (Toronto, Canada), pp. 115–116, IEEE, 2014.

[175] “POCO-framework for Pareto-optimal resilient controller placement in SDN-based core networks,”

[176] Y. Hu, W. Wendong, G. Xiangyang, C. H. Liu, X. Que, and S. Cheng, “Control traffic protection in software-defined networks,” in *Global Communications Conference*, GLOBECOM ’14, (Austin, TX, USA), pp. 1878–1883, IEEE, Dec. 2014.

[177] T. Petry, R. da Fonte Lopes da Silva, and M. P. Barcellos, “Off the wire control: Improving the control plane resilience through cellular networks,” in *Communications, 2015 IEEE International Conference on*, ICC ’15, (London, UK), pp. 5308–5313, IEEE, June 2015.

[178] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, “Fast failure recovery for in-band OpenFlow networks,” in *Design of reliable communication networks (DRCN), 2013 9th international conference on the*, pp. 52–59, IEEE, 2013.

[179] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, “Frenetic: A Network Programming Language,” in *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP ’11, (New York, NY, USA), pp. 279–291, ACM, Mar. 2011.

[180] D. Williams and H. Jamjoom, “Cementing high availability in OpenFlow with RuleBricks,” in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pp. 139–144, ACM, Aug. 2013.

[181] M. Reitblatt, M. Canini, A. Guha, and N. Foster, “Fattire: declarative fault tolerance for software-defined networks,” in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pp. 109–114, ACM, Aug. 2013.

[182] R. Beckett, X. K. Zou, S. Zhang, S. Malik, J. Rexford, and D. Walker, "An Assertion Language for Debugging SDN Applications," in *Proceedings of the Third ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, (New York, NY, USA), pp. 91–96, ACM, Aug. 2014.

[183] B. Chandrasekaran and T. Benson, "Tolerating SDN Application Failures with LegoSDN," in *Proceedings of the Third ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, (New York, NY, USA), pp. 235–236, ACM, Aug. 2014.

[184] M. Monaco, O. Michel, and E. Keller, "Applying Operating System Principles to SDN Controller Design," in *Proceedings of the Twelfth ACM SIGCOMM Workshop on Hot Topics in Networks*, HotNets-XII, (New York, NY, USA), pp. 2:1–2:7, ACM, Aug. 2013.

[185] Y. Zhang, N. Beheshti, and R. Manghirmalani, "NetRevert: Rollback recovery in SDN," in *Proceedings of the third ACM SIGCOMM workshop on Hot topics in software defined networking*, (New York, NY, USA), pp. 231–232, ACM, 2014.

[186] R. C. Scott, A. Wundsam, K. Zarifis, and S. Shenker, "What, where, and when: Software fault localization for SDN," Tech. Rep. UCB/EECS-2012-178, EECS Department, University of California, Berkeley, 2012.

[187] C. Scott, A. Wundsam, B. Raghavan, A. Panda, A. Or, J. Lai, E. Huang, Z. Liu, A. El-Hassany, S. Whitlock, *et al.*, "Troubleshooting blackbox SDN control software with minimal causal sequences," in *ACM SIGCOMM Computer Communication Review*, vol. 44, pp. 395–406, ACM, 2014.

[188] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown, "I know what your packet did last hop: Using packet histories to troubleshoot networks," in *11th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '14, (Seattle, WA, USA), pp. 71–85, USENIX Association, Apr. 2014.

[189] R. Durairajan, J. Sommers, and P. Barford, "Controller-agnostic SDN Debugging," in *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, (New York, NY, USA), pp. 227–234, ACM, Dec. 2014.

[190] H. Zhang, C. Lumezanu, J. Rhee, N. Arora, Q. Xu, and G. Jiang, "Enabling layer 2 pathlet tracing through context encoding in software-defined networking," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, (New York, NY, USA), pp. 169–174, ACM, Aug. 2014.

[191] I. Pelle, T. Lévai, F. Németh, and A. Gulyás, "One Tool to Rule Them All: A Modular Troubleshooting Framework for SDN (and Other) Networks," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, (New York, NY, USA), pp. 24:1–24:7, ACM, June 2015.

[192] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '13, (Lombard, IL, USA), pp. 15–27, USENIX Association, Apr. 2013.

[193] E. Al-Shaer and S. Al-Haj, "FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures," in *Proceedings of the 3rd ACM workshop on Assurable and usable security configuration*, pp. 37–44, ACM, 2010.

[194] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI '13, (Berkeley, CA, USA), pp. 99–112, USENIX Association, Apr. 2013.

[195] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A NICE way to test OpenFlow applications," in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '12, (San Jose, CA, USA), pp. 127–140, USENIX Association, Apr. 2012.

[196] N. Shelly, B. Tschaen, K.-T. Förster, M. Chang, T. Benson, and L. Vanbever, "Destroying networks for fun (and profit)," in *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, p. 6, ACM, 2015.

[197] C. Monsanto, N. Foster, R. Harrison, and D. Walker, "A compiler and run-time system for network programming languages," in *ACM SIGPLAN Notices*, vol. 47, (New York, NY, USA), pp. 217–230, ACM, Jan. 2012.

[198] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent Checkpointing Under Unix," in *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, (Berkeley, CA, USA), pp. 18–18, USENIX Association, 1995.

[199] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *Software Engineering, IEEE Transactions on*, vol. 28, no. 2, pp. 183–200, 2002.

[200] G. Altekar and I. Stoica, "Focus Replay Debugging Effort on the Control Plane," in *Proceedings of the Sixth International Conference on Hot Topics in System Dependability*, HotDep'10, (Berkeley, CA, USA), pp. 1–9, USENIX Association, 2010.

[201] M. Gupta, J. Sommers, and P. Barford, "Fast, Accurate Simulation for SDN Prototyping," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, (New York, NY, USA), pp. 31–36, ACM, 2013.

[202] B. Heller, C. Scott, N. McKeown, S. Shenker, A. Wundsam, H. Zeng, S. Whitlock, V. Jeyakumar, N. Handigol, J. McCauley, *et al.*, "Leveraging SDN layering to systematically troubleshoot networks," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pp. 37–42, ACM, 2013.

[203] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why do internet services fail, and what can be done about it?," in *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems*, USITS'03, (Berkeley, CA, USA), pp. 1–15, USENIX Association, 2003.

[204] M. Kuzniar, M. Canini, and D. Kostic, "OFTEN Testing OpenFlow Networks," in *2012 European Workshop on Software Defined Networking*, (Darmstadt, Germany), pp. 54–60, Oct 2012.

[205] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky, "Vericon: Towards verifying controller programs in software-defined networks," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, (New York, NY, USA), pp. 282–293, ACM, June 2014.

[206] T. Nelson, A. D. Ferguson, M. J. Scheer, and S. Krishnamurthi, "Tierless programming and reasoning for software-defined networks," in *11th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '14, (Seattle, WA), pp. 519–531, USENIX Association, Apr. 2014.

[207] H. Zeng, S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, and A. Vahdat, "Libra: Divide and conquer to verify forwarding tables in huge networks," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, (Berkeley, CA, USA), pp. 87–99, USENIX Association, 2014.

[208] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Commun. ACM*, vol. 51, pp. 107–113, Jan. 2008.

[209] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, (Berkeley, CA, USA), pp. 9–9, USENIX Association, 2012.

[210] D. Jackson, *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

[211] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker, "Practical declarative network management," in *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*, WREN '09, (New York, NY, USA), pp. 1–10, ACM, 2009.

[212] C. Trois, M. D. D. D. Fabro, L. C. E. de Bona, and M. Martinello, "A Survey on SDN Programming Languages: Towards a Taxonomy," *IEEE Communications Surveys & Tutorials*, vol. PP, no. 99, pp. 1–25, 2016.

[213] 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 Computer Communication Review*, vol. 44, pp. 87–95, July 2014.

[214] M. Casado, N. Foster, and A. Guha, "Abstractions for software-defined networks," *Communications ACM*, vol. 57, pp. 86–95, Sept. 2014.

[215] M. H. Behringer, M. Pritikin, S. Bjarnason, A. Clemm, B. E. Carpenter, S. Jiang, and L. Ciavaglia, "Autonomic Networking: Definitions and Design Goals." RFC 7575, Oct. 2015.

[216] J. M. Sanchez Vilchez, I. Grida Ben Yahia, and N. Crespi, "Self-healing Mechanisms for Software Defined Networks," in *8th International Conference on Autonomous Infrastructure, Management and Security (AIMS 2014)*, (Brrno, Czech Republic), June 2014.

[217] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, "Network function virtualization: Challenges and opportunities for innovations," *IEEE Communications Magazine*, vol. 53, pp. 90–97, Feb 2015.

[218] K. Yamazaki, T. Osaka, S. Yasuda, S. Ohteru, and A. Miyazaki, "Accelerating SDN/NFV with Transparent Offloading Architecture," in *Open Networking Summit 2014 (ONS 2014)*, (Santa Clara, CA), USENIX Association, 2014.

[219] R. Cannistra, B. Carle, M. Johnson, J. Kapadia, Z. Meath, M. Miller, D. Young, C. DeCusatis, T. Bundy, G. Zussman, K. Bergman, A. Carranza, C. Sher-DeCusatis, A. Pletch, and R. Ransom, "Enabling autonomic provisioning in SDN cloud networks with NFV service chaining," in *Optical Fiber Communications Conference and Exhibition (OFC), 2014*, pp. 1–3, March 2014.
[220] D. Cotroneo, L. De Simone, A. Iannillo, A. Lanzaro, R. Natella, J. Fan, and W. Ping, "Network function virtualization: challenges and directions for reliability assurance," in *Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium on*, pp. 37–42, IEEE, 2014.
[221] "OpenFlow Switch Specification Version 1.3.0." https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf. Accessed: 2016-06-30.
[222] S. Watts and O. G. Aliu, "5G resilient backhaul using integrated satellite networks," in *2014 7th Advanced Satellite Multimedia Systems Conference and the 13th Signal Processing for Space Communications Workshop (ASMS/SPSC)*, pp. 114–119, Sept 2014.
[223] A. Hakiri and P. Berthou, "Leveraging SDN for the 5G networks: Trends, prospects and challenges," *CoRR*, vol. abs/1506.02876, 2015.
[224] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: Experience with a Globally-deployed Software Defined WAN," *SIGCOMM Computer Communication Review*, vol. 43, pp. 3–14, Aug. 2013.
[225] I. T. Haque and N. Abu-Ghazaleh, "Wireless software defined networking: a survey and taxonomy," *IEEE Communications Surveys & Tutorials*, vol. PP, no. 99, pp. 1–1, 2016.
[226] A. S. Thyagaturu, A. Mercian, M. P. McGarry, M. Reisslein, and W. Kellerer, "Software Defined Optical Networks (SDONs): A Comprehensive Survey," *IEEE Communications Surveys & Tutorials*, vol. 18, pp. 2738–2786, Fourth Quarter 2016.

Paulo César da Rocha Fonseca received his B.S. degree in Computer Sciente and M.Sc. degree in Informatics from Federal University of Amazonas, Manaus, Brazil, in 2011 and 2013, respectively. He is currently pursuing the Ph.D. degree at the same university. His current research interests include software-defined networks, fault management and machine learning.

Edjard Souza Mota has been, since 1989, with the Institute of Computing (former Computer Science department) at the Federal University of Amazonas, Manaus, Brazil where he was head of the post-graduation program in informatics from 1999 to 2003, and is currently an Associate Professor. He holds a Ph.D. degree in Artificial Intelligence from The University of Edinburgh, Edinburgh, Scotland, and an M.Sc. degree in Computer Science from the Federal University of Minas Gerais, Belo Horizonte, Brazil in 1998 and 1993, respectively. His current research interests include the application of AI techniques to Software-Defined Networks, and neural-symbolic integration and cognitive reasoning