Robust fault-recovery in Software-Defined Networks

IBSDN: IGP as a Backup in SDN

Master's thesis presented in partial fulfillment of the requirements for the degree of
Master in computer sciences and engineering,
with a specialization in networks and security.

Candidate:
Olivier Tilmans

Supervisor:
Olivier Bonaventure

Readers:
Stefano Vissicchio
Marco Canini

Academical year 2013-2014

Acknowledgments

I would like to thank the following people who helped me in the making of this document:

- Dr. Stefano Vissicchio for his numerous feedbacks as well as his constant support,
- Pr. Olivier Bonaventure for providing insightful comments,
- Chris Luke for releasing an Open vSwitch patch which saved me valuable amounts of time,
- Simon Knight for his work on AutoNetkit.

Contents

Chapter 1

Introduction

Once created to simply carry scientific data across space, today's networks are expected to provide more and more services, ranging from the access of one's mailbox from anywhere on the planet, to guaranteeing the isolation of traffic between virtual machines in virtualized environments.

Traditional, networks have been run using distributed protocols, such as BGP [1] and OSPF [2], directly running on the network nodes. These protocols are robust by design, and have been proved to be able to restore traffic in less than 50ms in the events of failure [3]. These protocols in conjunction with destination-based forwarding are efficient, and provide strong guarantees.

However, achieving more fine grained tasks such as traffic-engineering, or trying to setup arbitrary forwarding paths in the network is hard, resulting in additional protocols being used for these extra features.

Finally, interactions between these protocols are complex, and even using the best-practices to do some operation on one protocol does not ensure that it will not cause issues in another one [4]. The end result being that managing distributed network is intrinsically hard.

Software-Defined Networking on the other hand is a novel approach. It decouples the control-plane, the part of network defining its behavior, from the data-plane, the part that actually forwards the data. It then makes of a well-defined API to let the control-plane remotely setup the content of the Forwarding Information Base present on the elements of the data-plane. The SDN approach brings programmability to the control-plane, and breaks the tight integration between the forwarding hardware, and the routing behaviors that the network operators want to setup.

This approach has gathered multiple use-cases ranging from network visualization [5] to access-control management [6], including fine-grained traffic engineering [7]. However its real-world deployment are still close to noone, and mostly confined to campus networks. Plausible explanations include the fact that service providers are traditionally slow to adopt new disruptive technology or that this require massive economical investments.

One of the main concern with the SDN approach is however its ability to react quickly to failures. Indeed, as it decouples the two network-planes, it requires to perform a trade-off between the control-plane overhead resulting of the handling of the failure, as it has to recompute the new state of the network and the configure the data-plane according, and the additional entries which must be kept in the data-plane elements memory to support autonomous failover.

As a result, ensuring scalable and robust fault-recovery in pure SDN networks is hard.

On the other hand, traditional distributed networks benefit from inherent resiliency and have well-studied recovery mechanisms, both minimizing the time necessary for the traffic restoration as well as the load induced on the control-plane.

Current real-world networks infrastructure are all built upon distributed protocols. This motivates the fact that both networking paradigm are likely to have to co-exist, as the transition will not be instantaneous [8].

From that observation, this master's thesis present a network architecture attempting to use both networking paradigm at the same time by performing a separation of concerns between a long-term optimization problem and a short-term feasibility problem. This results in an architecture where a SDN control-plane is responsible for the majority of the traffic, and where an IGP is used to provide resiliency in the events of failures.

Finally, an experimental prototype is implemented and evaluated in order to show that this architecture is both practical and overcomes the main limitations of previous works.

This document is structured as follow.

The first part will attempt to present the general principles in order to manage the control-plane of a network. As such, Chap.2 will attempt to define what is the control-plane of a network, and what are some of its desirable properties. Chap. 3 will then present the general principles used in the distributed routing protocols and Chap. 4 will cover the ones of the SDN approach. Finally, Chap 5 will discuss the challenges of both paradigm and Chap. 6 will conclude that part.

The second part will focus on a critical aspect of network management, the handling of failures. Chap. 7 will present the general techniques used in distributed networks to deal with failures, which is a well understood topic. Chap. 8 will present the equivalent in the SDN world, although since it is a much younger paradigm, less work exists on the subject.

Finally the third part will present IBSDN, for IGP as a backup in SDN. This is a novel network architecture in which a distributed control-plane flanks a SDN in order to provide robust and scalable fault-management. The architecture in itself and its associated operational model will be presented in Chap. 9, as well as the theoretical guarantees brought by it. An implementation will be present in Chap. 10. Finally, Chap. 11 will present the results of benchmarks of IBSDN.

Part I

Control-Plane Management

Chapter 2

The Control-Plane as the brain of networks

Computer networks can be envisioned as a set of interconnected nodes, which allow data to be sent from one of these nodes towards another one. Nodes can have different roles in the network (e.g. routers, switches, servers, middleboxes, ...), and the means to connect them can vary throughout the network (e.g wireless, Ethernet links, optical fiber, ...).

Computer networks are logically made of two components:

The *data-plane* Also called the forwarding-plane, represents the set of network elements (e.g. switches, routers) that effectively carry data from a part of the network to another. As such, these elements query their forwarding table to determine the output port of each packet, possibly altering some part of it (e.g. decreasing TTL in IP networks). This can be seen as the *body* of the network;

The *control-plane* It is the set of all protocols and algorithms that are used to compute the contents of the forwarding tables of the network elements. While not performing any forwarding on its own, the control-plane will decide what and how data is forwarded. As such, it can be seen as the *brain* of the network.

The control-plane is thus the component that is responsible for the overall behavior of the network, which ranges from the basic forwarding of the traffic to optimizing the load on links as well as ensuring some peering agreements. As such, lots of different control-plane protocols exist, with different purposes (e.g. intra-routing vs inter-routing). It is not unusual to have multiple control-planes running at the same time in the network, and erroneous control-plane(s) can lead to undesirable network behaviors, such as black-holes and forwarding loops.

Control-plane protocols are expected to ensure a network-wide consistency of the forwarding behaviors, as well as to be highly resilient to network elements having an availability varying over time as these can be brought down for maintenance, or simply crash, and then come back online.

These protocols are the ones defining the range of services that a network can provide. They are thus highly diverse, and often interact with each other:

- in direct ways, e.g. exporting routes learned from one protocol over another one;
- in subtle ways, e.g. determining the next-hop via one protocol, and how to reach it via another one).

Understanding how the protocols operate and interact is thus mandatory to be able to provide the guarantees needed to meet the Service Level Agreements of the network (e.g. guarantees regarding the bandwidth usage of some links).

Two main paradigms of control-plane protocols exist, namely the distributed routing protocols and the software-defined networking approach. These are two completely different techniques to manage the control-plane of a network. Both will have their key principles detailed in the next chapters, followed by a discussion regarding the challenges of each paradigm.

Chapter 3

Distributed Routing Protocols

Distributed routing protocols are the traditional ways to run a control-plane, and still are the most used ones. Each node in the network that has forwarding responsibilities also takes part in network-wide distributed algorithms in order to infer its forwarding table.

These protocols rely on two principles:

1. Each node reports routing information in its neighborhood;

2. Upon receiving these information, each node can check if it needs to update its forwarding table to reflect the inferred change in the network status.

These protocols were historically favored in order to bring robustness and survivability to the control-plane, even in the events of massive outage of some of its elements.

There are two main categories of distributed routing protocols, illustrated by the Fig. 3.1:

Intra-domain routing These are the protocols responsible for the dissemination inside the AS of the available prefixes, in order to have every node on the network able to connect to any other one. On the Fig. 3.1, router R1–R2–R3–R4 are all running an Interior Gateway Protocol, which allows for example R2 to have a next-hop for the prefixes announced by R4, despite not having a direct link with it;

Inter-domain routing There is only one protocol in use in this category: the Border Gateway Protocol [1]. This protocol is in charge of advertising routes between AS'es. The Fig. 3.1 shows it running between R2 and Ra, thus potentially advertising to Ra a route towards R4's prefixes, if the export filters of R2 allow it. This particular aspect of the control-plane will not be detailed in this document.

The separation between the two types of routing helps to ensure some scalability. Indeed, as the number of advertised BGP prefix exceeds 500k entries [9], the memory requirements to operate a BGP router are huge. Splitting the two allows to have routers with a smaller memory, thus cheaper, doing the forwarding inside the network, while the core BGP-enabled routers of the AS forward the packets exiting the AS.

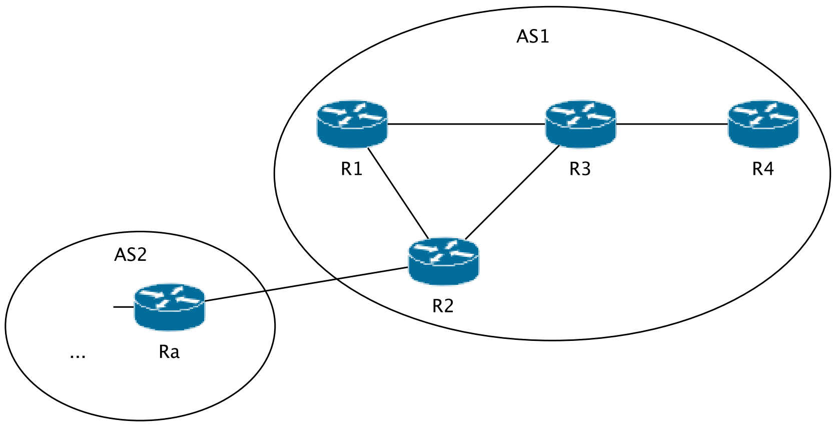


Figure 3.1: Two interconnected sample AS'es.

IGP protocols come in two families, distance-vector protocols and Link-state protocols. Both families perform destination-based forwarding, meaning that packet are forwarded only according to their destination address, but they differ on how they build their forwarding table:

- In distance-vector protocols, nodes maintain a table associating each destination in the network with its relative cost, and the router it learned the route from. Periodically, each node will send that routing table to all its neighbors. Upon receiving routes on an interface, a node will add the cost associated to that interface to the cost of the received routes. If the resulting cost of one of the new routes is lower than the one in the node's routing table, then that route is updated to the newly received-one. In these protocols, nodes have no knowledge about the network topology, they only have reachability information. As such, next-hop are decided as the result of a distributed computation process;
- Link-state protocols instead rely on each node reconstructing a complete directed graph of the whole network topology, and then computing a shortest-path tree with itself as the root [10]. The nodes then route packets according to the shortest-path they have computed for their destination address.

Link-state protocols are the most widely used IGP's. As such, the next section is devoted into detailing their working more precisely.

3.1 Link-state protocols

Link-state protocols (e.g. OSPF [2]) work in two phases.

First, when booting up, routers first start by discovering their neighborhood. They do so by sending `HELLO` messages periodically over their links, waiting for another router to answer. These messages serve to monitor the status of the interfaces of the router, thus its logical links towards other routers. They messages are never forwarded across routers.

Once a router has discovered a neighbor, it then proceeds to periodically send him Link State Packet. These LSP's contain the id of the router that created the packet, a sequence number, as well the list of links and their associated cost that the router has. Each router maintains a database of its received LSP.

When a router receives a LSP, it checks whether it has it already in its link state database or not. If it does not have it already and if its sequence number is newer, it registers it in its database then proceeds to flood it in the network by sending it over all interfaces except the one it has been received on. The transmission of these LSP's is done in a reliable way: when a router receives a message, it has to send back an acknowledgment otherwise the sender will attempt to retransmit it.

From this LSP database, each individual router is able to make a directed graph with the latest status of each link. A link between two router is considered to be up if it has two recent LSP associated with it (one for each of its end). Using this information regarding the liveness of the links, and the inferred network graph, nods can then compute their shortest-path trees to determine the content of their RIB. Upon processing new LSP's, if it detects a change in the graph, the node then proceeds to recompute its shortest-path tree, thus updates its forwarding table.

This family of protocols implicitly handles both nodes and link failures, as eventually this means that a router will stop receiving the `HELLO` messages answers from one of its neighbor. It will then proceed to update its link states, and thus initiate a flooding process, which will trigger a network-wide update of all shortest-path trees.

3.2 Distributed protocols in today's networks control-planes

Initially, traditional networks were designed to be as resilient as possible, and had the single purpose of relaying data. As such, they were using distributed protocols that allowed them to exchange information about the network topology between each node, eventually leading to each individual node deciding on how to forward a given packet based on the information it had received. One of the oldest of these protocols is RIP [11].

Networks in their current state are the result of a continuous evolution process. As time went by, the networks grew both in size and numbers, and some started to interconnect them, eventually leading to what we now know as the Internet. In order to avoid overloading each individual node of the network with information about the whole globally interconnected topology, BGP was developed and added to specifically control the exchange of routing information between networks. This led to the creation of different class of routers: those able to run BGP, and the others.

More and more data had to be carried, and the speed of the links increased. However, classical routers in IP networks could not perform the necessary table lookups to forward the traffic at line rate. As an answer to that problem, MPLS [12] was designed. However, it also required the introduction of additional protocols to support the distribution of labels (e.g. LDP [13]). Yet another class of routers was added to the network: the ones that could be used as Label Switched Router.

Since operating networks is not free, operators tried to reduce their costs. This kickedoff the era of traffic engineering, aiming for example at maximizing the usage of cheap links while avoiding more expensive ones, which led to various new protocol extensions, such as OSPF-TE, [14] RSVP-TE [15], ... and even new use-cases for MPLS as routers hardware had finally caught up with the link speed, thus making it no longer mandatory to operate at high speed.

Nowadays, beside carrying data, networks are also supposed to provide QoS, to meet security policies, to support concurrent addressing schemes (IPv4/6), ... leading to even more protocols and services running directly on the routers bare metal.

The aftermath. Networks have drastically evolved. Instead of having to setup a few nodes, and a control protocol, nowadays network operators, sometimes referred to as the *masters of complexity*, have to manage hundreds of nodes, with many different protocols, and dozens of extensions to them.

As everything is distributed, each protocol has to be configured separately, and on every node (with minor differences on each of them). Editing the configuration files of nodes is error-prone, and a single typographic error can cause unpredictable troubles in the network (at best, it might simply shutdown the particular protocol, more serious issues include exporting bogus routes over BGP). A recent study [4] has shown that in a Tier-1 ISP, more than 10k changes have been made in a span of 12 months, only in the BGP configuration files.

Furthermore, as there are many protocols running on the nodes with the sole purpose of synchronizing its FIB, the interactions between them can sometimes prove to be unpredictable [4].

Overall, *distributed networks have grown too complex*, and even making a simple change in one node can have disrupting effects on the whole Internet [16].

There exists softwares, such as AutoNetkit [17], that attempt to ease up the generation of configuration files, but these only solve a part of the problem of network management in a distributed environment.

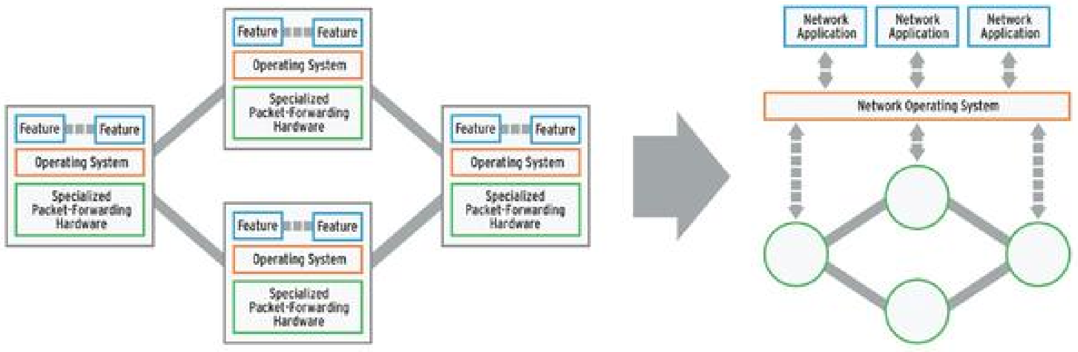
Finally, a recurring pattern can be observed throughout the evolution the networks. As everything is directly running on the routers, often tightly integrated with the hardware to maximize the performances, introducing new services often requires the service providers to acquire new hardware supporting these. This leads to heterogeneous networks, as well as a slower adoption of these services due to the non non-negligible costs induced.

Chapter 4

Software Defined Networking

Software Defined Networking is a network paradigm which drastically changes the way networks are managed. It revolves around two key principle:

1. It radically decouples the control-plane, from the data-plane;
2. Its control-plane is implemented in software and can be seen as a logically centralized component, controlling every data-plane elements in the network. It makes use for that of an API allowing to interact with the forwarding tables of the routers, switch, . . . of the network, thus bringing a level of programmability in these to define arbitrary rules in their FIB.



Picture credits: http://www.admin-magazine.com/Articles/Floodlight-Welcome-to-the-World-of-Software-Defined-Networking

Figure 4.1: Differences between the integration of features in traditional networks compared to the SDN approach.

With these two principle, the network is thus controlled by an element, decoupled from the data-plane itself, that has a full control on the way all traffic is handled. As a consequence, the hardware building the data-plane is no longer tied with the one of the control-plane. This breaks the tight link between the routing features that were supported by some router or switch, and the actual capabilities of the hardware, as illustrated on Fig. 4.1 . Indeed, whereas traditional hardware is sold with some packet forwarding capabilities tightly coupled with some particular supported protocols and RFC's, the SDN approach exposes well-defined hardware capabilities of the data-plane element to the control-plane. This allows the creation of a centralized control entity, a network OS, in charge of controlling the whole network, communicating with the switches via a standard API. This network OS can then expose logical views of the network to applications written by the network operators to implement given services, such as basic Layer-3 routing, access control [6], . . .

Because it is a software, running on external hardware, this approach can benefit from all the experience acquired in the software engineering field.

Indeed, while the networking world has been working for years by successive iteration of *protocol creation → standardization → vendor implementation → deployment*, before noticing that the said protocol was lacking of a feature or not covering a particular use-case, thus requiring the whole process to start over from scratch; the software engineering world has been able to quickly progress by leveraging on the use of abstractions. First to abstract the hardware details of the machines, then to abstract common tasks such as drawing geometrical figures, . . . up to being able to write complete GUI with few lines of code by leveraging those abstractions.

The SDN approach attempts render the overall management of a network less complex, by allowing abstractions layers to be built in the control-plane, similarly to what has been done in the software engineering field, instead of re-designing complete new protocols from scratch for each new task.

Finally, as the SDN paradigm decouples both planes, thus forces to design an API between the two, it also enables researchers to directly experiment within the network itself, with their current hardware. Indeed, all it would take to experiment with a given extension of a given network protocol would be to (i) implement the protocol in the control-plane (ii) patch it with the extension. This allows for a quicker feedback loop between protocol design and their validation. It also enables network operators to benefit from the latest development in networking, as all they would have to do would be to *patch* their control-plane instead of changing all data-plane elements.

The rest of this chapter will first start by a brief recap on the few attempts to bring programmability in the networks, which eventually led to this paradigm. It will then outline the de-facto standard SDN technology, OpenFlow.

4.1 A retrospective on networking management technologies

The whole Software-Defined-Networking paradigm did not come of thin air. Indeed, many of its ideas were spawned by earlier attempt at bringing programmability into the networks [18].

This section will outline the key ancestors to the SDN paradigm.

4.1.1 Active Networks

Active Networks [19] is an attempt during the 90's at bringing programmability into the data-plane. The idea was to use programmable switches, that would expose some parts of their hardware capabilities (FIB, storage, . . .) to user-specified programs. This was achieved in two different modes:

The Programmable Switch model In this approach, programs could be sent to programmable switch either in in-band or out-band messages. The processing the traffic would then be made on a per-packet basis, where the installed program would be able to determine the behavior that the switch should take by examining the packet header;

The capsule model This is a more extreme version, where the forwarding would be entirely driven by the traffic itself. Indeed, every packet would carry some instructions on how the switch should handle it.

While this technology was never widely used, it however brought up many interesting concepts. Indeed, the concept of a Node-OS was mentioned in [20] as the basis to start building abstraction layers to program the network. [20] and [21] also raised the point that specific languages had to be designed, to support this new paradigm, in order to provide abstractions over the fine-grained control introduced by active-network.

Finally, [21] attempted to present some temporary plans to integrate both classical distributed IP routing and active networks in the same network.

While this technology did not break through, probably because there was no real demand of from the industry at the time, it is important to notice that many of concept that SDN (and OpenFlow as covered in the next sections) raises was already pointed there. The main difference between active networks and SDN is the fact that active networks attempted to program the network directly in the data plane elements, thus forcing vendors to open their platform (similarly to what Juniper eventually did as show in Sec. 4.1.3), whereas SDN brings programmability only in the control-plane.

4.1.2 Early attempts to decouple the control plane and the data plane

The peak of the Internet bubble (around 2000) also marked an increase in the traffic volume transiting through the networks. This led operators to start working on traffic engineering, to be able to manage that increase in a more predictable way with hopefully better performances than plain destination-based forwarding.

This led to two technologies, that were designed to be quickly deployable:

The Forwarding and Control Element Separation Framework ForCES [22] consisted in a drastic separation of the forwarding elements and the control elements inside a network element. It then provided an API for the control elements to program and communicate with the forwarding elements;

The Routing Control Platform RCP [23] took an alternative approach. Instead of waiting for some new API to be defined and accepted by the vendors, it instead leveraged on an existing deployed standard (BGP), to install forwarding entries in the routers.

Although ForCES was standardized by the IETF, the major vendors did not adopt it, which eventually prevented it from being deployed. It however ended up spawning the Netlink [24] feature in the Linux kernel.

RCP on the other end was deployable immediately, as it was relying on no new element. However its expressiveness was quite limited as it was dependent on BGP.

It is interesting to note that these two attempts came as a result of the ever increasing complexity of managing networks. Both approaches attempted to provide a solution, by first decoupling the control-plane and the data-plane, and secondly by bringing programmability in the control-plane instead of in the data-plane directly.

These are the key principles that are at the basis of OpenFlow.

4.1.3 JUNOS SDK – Bringing power in the hands of network operators

The JUNOS SDK [25] is a recent attempt from Juniper Networks to provide an API to their customer to define custom services operating in the network, thus effectively opening their platform to external development.

It consists in exposing a set of C API's, bringing programmability in both the control-plane and the data-plane directly on the OS of the network elements. It first provides an SDK to extend the routing engine of JUNOS, thus allowing operators to implement e.g. custom signaling protocols. The second SDK is the service SDK, which allows to define custom packet processing behaviors. These two SDK's are complemented by libraries and inter-process communications, allowing operators to have modular custom applications.

While only available for Juniper hardware, this SDK is available on all of its product since 2007. This is thus the first widely available technology providing a direct access to extend the features present directly on the device. This SDK was used to add OpenFlow support on the Juniper routers, to showcase its power.

4.1.4 OpenFlow as the first successful SDN technology

OpenFlow [26] was created by a group of researchers at Stanford as a successor of their work on Ethane [27], which was an early attempt at bringing programmability in the networks through the use of simple programmable switches under the supervision of a remote controller.

The concept of fully programmable networks, as in networks where *anything* could be scripted to happen (e.g. programming switches to have DPI-like functionalities, to perform traffic encryption/decryption on the fly, . . .), has been around for a while as per the previous sections, however OpenFlow restricts what can be done to a subset of these functions in order to be deployable in real-world networks through a trade-off between network programmability and the capabilities of deployed hardware.

Indeed, what hindered the deployment of earlier attempts at providing network programmability, was either the issues with the approaches themselves or more-often the fact that these were complete disruptive technology, requiring existing hardware to be fully replaced if one wanted to use them. On the contrary, OpenFlow leverages on the fact that most modern Layer-2 routing hardware have a form of flow-table that is used to forward traffic at line rate, while also collecting statistics to perform QoS functions [26], etc.

It introduces flow-based programmable routing, by defining flows as packets sharing similarities in their headers as well as their desired forwarding behaviors, and by providing an open API to install those flows remotely in the switches. This can be seen as a generalization of MPLS networks, where packets falling under a certain class take a particular path in the network. The key difference being that where MPLS relies on managing a stack of labels to identify which path should be taken by a packet, OpenFlow switches are embedding a classifier matching virtually *any* part of the packet to identify the flow they are part of, thus allowing to handle packets as-is, regardless of their format or encapsulation.

Since the concept of flows was already exploited in Ethernet devices, and that it limits the operations it performs on packet to operations that were usually already performed (e.g. changing a value in the header, collecting statistics), support for OpenFlow can thus be realistically implemented on current hardware.

OpenFlow also re-uses the concept of having a Network-level OS, that is responsible for managing the switches in order to have an accurate view of the state of the network at all time similarly to the Node-OS of active networks. With this element, it enable operators to start adding abstractions on top of that OS, in order to reduce the overall complexity needed to manage a network.

An argument often pointed by its detractors (e.g. [28]) is that it does not solve any problem as-is. While this is true, it however gives new tools that can be used by operators, researchers, . . . Indeed, as it gives an open interface to the switches TCAM, it frees up from vendor lock-in, thus allows people to directly experiment with the network, with their current hardware. As it does not require a particular extension to be (i) defined (ii) standardized though RFC's (iii) supported by vendors, which is a rather slow process, it allows instead for fast-iterations directly in the network. Once these are completed, operators can then decide to implement them in their own network by the means of a software update instead of upgrading the hardware.

As a conclusion, it is important to understand that OpenFlow and SDN, while often linked together, do not refer to the same thing. Indeed, SDN as a whole refers to a paradigm where the control-plane and the data-plane are decoupled, and where arbitrary software running in the control-plane can define what is happening in the data-plane. OpenFlow is an instantiation of this paradigm, although with a strong emphasis on backward compatibility to ease-up its implementation thus limiting its programmability.

4.2 OpenFlow

OpenFlow is the current de-facto reference SDN technology. While not yet widely deployed in most physical networks, due to both economical reasons as well as lack of expertise in that area, it has already gained lots of traction in data-centers as well as is used together with network virtualization technologies [5]. This section is dedicated in providing the key concepts forming the OpenFlow components. It is based on the OpenFlow 1.4 Switch Specifications [29].

4.2.1 Overview of an OpenFlow network

An OpenFlow network, illustrated in 4.2 decouples the control plane and the data plane in a strict way. Two key components define the standard:

1. **The OpenFlow switch specifications** These define the essential components that must be present in an OpenFlow switch and how they interact with each other. This specifies the expected behaviors and features of the data plane elements, which are the five switches in the figure 4.2.

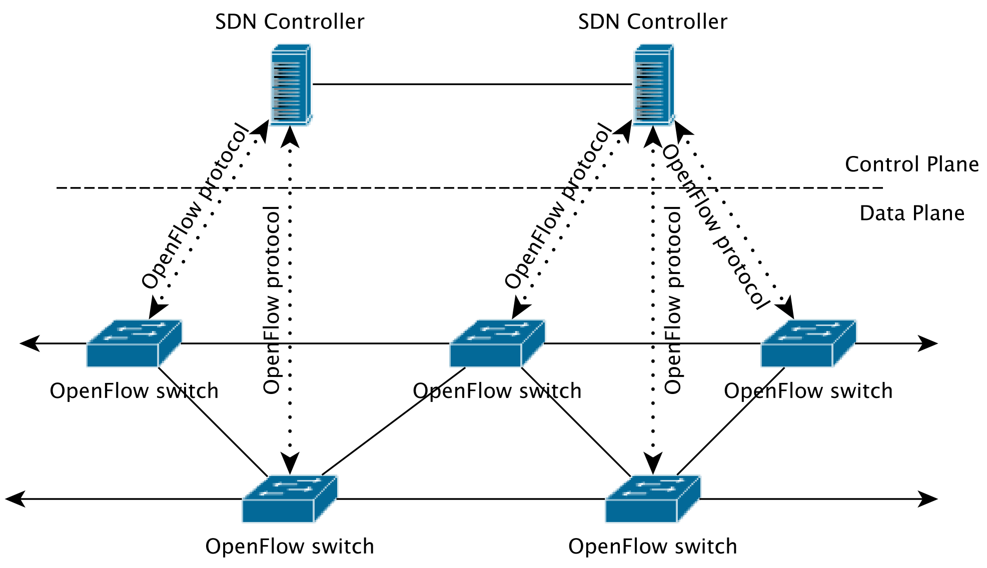


Figure 4.2: An OpenFlow network with two controllers

2. **The OpenFlow protocol** This is the control protocol which allow an external management station, which will be called a controller, to program the forwarding behavior of the switches. This protocol is the link between the control plane and the data plane.

With these two components, a third-party, usually called the controller can remotely program the FIB of the switches.

The fact that OpenFlow only define the minimal set of hardware capabilities (via the switch specifications), and how to access them (via the OpenFlow channel), but not what to do with these switches, is what allows network operator to define their own behaviors in their networks.

4.2.2 Components of an OpenFlow switch

4.2.2.1 The OpenFlow pipeline

The OpenFlow pipeline is defined as *"the set of linked flow tables that provide matching, forwarding, and packet modification in an OpenFlow switch"* [29]. Each packet that is received by an OpenFlow switch is first labeled by its ingress port and associated with an empty action set. It then enters the pipeline of flow tables, starting at table 0. In a flow table, the packet is matched against each flow entries.

If a matching flow is found, the packet action set is updated according to the flow entry, if applicable. The packet may then be sent to another flow table, if a `GOTO_TABLE` instruction is present, optionally carrying some metadata across tables. Otherwise, if the flow entry does not specify another flow table where the processing should continue, the action set is executed.

If the packet enters a tables where it cannot be matched against any flow, the flow-miss entry of that table is then executed. This can result in the packet being encapsulated and sent to the controller, or some other actions pre-programmed by the controller.

Packets can only be sent to flow tables with a strictly higher sequential number than the current one, and the last table of the pipeline cannot have flow entries with `GOTO_TABLE` instructions, as a result of this.

Possible actions are to update some value of the packet header, to add some local metadata which will not leave the pipeline,. . . or eventually to specify an OpenFlow port as output to forward the packet to as its next-hop. It is interesting to note that there is no explicit `DROP` action in the OpenFlow protocol. Instead, it is inferred from the fact that the action set of the packet does not contain any `OUTPUT` action.

4.2.2.2 Flow Tables

Flow tables are a list of flow entries installed by a `FLOW_MOD` message over the OpenFlow channel. Flows consists mainly of a set of match fields, which include all fields in the packet as well as its ingress port and the associated metadata if any, and a set of instructions to apply on matching packets. Unspecified match fields take a wildcard value (`ANY`).

By default, flows will timeout after a while (e.g. 30s) if no packets matches it, unless it was installed with a timeout of 0, thus as the equivalent of a static route. The controller can also send a `FLOW_REM` message at any time to remove a given flow from the switch table.

Finally, each flow entry has a set of counters to keep statistics regarding the packets that matched it. These can be queried by controller.

The matching of a packet against all flows in a table to find the set of actions to process it follows these rules:

1. The packet in its current state is matched against all possible fields, including its ingress port and its metadata;

2. If there are multiple matching flows, keep the only the most specific one(s) (the one with the least wild-carded matching fields);

3. Keep only the flow with the highest priority value (which ranges between 0 and 65535);

4. If there are still multiple flow, the behavior is undefined, as in any of remaining flows can be selected, or none at all. This is dependent on the implementation.

A flow entry with all field wild-carded and a priority of 0 is called the flow-miss entry, and is thus selected when not other flow can apply.

The specification (as of OpenFlow 1.4.0) do not specify what happens when a packet is corrupted/malformed.

This flow system is agnostic of the underlying datalink layer technology, and made to be easily extendable. However, while extremely powerful, it requires explicit actions from the controller to reproduce basic mechanism such as the `ICMP TTL_EXPIRED` message found in current Layer-3 routers.

Finally, in parallel with its flow tables, a switch maintain a meter table, containing per-flow meter. Such meter can be referenced to in the flow action set, to perform simple QoS operations (e.g. rate-limiting) and can be combined with per-port queues to build more complex QoS services for each flow.

Flow table nr: **0**

Match Fields	**Priority**	**Counters**	**Instructions**	**Timeouts**
ip, tcp, dst_port:80	65535	...	decrease_tll, output:3	0
eth_type:0x88cc	65535	0	output:`CONTROLLER`	0
...	...	...	...	...
ip, dst_ip:91.3.52.16, udp, dst_port:31	...	...	set_field:(dst_ip:10.0.0.5, dst_port:20), `GOTO_TABLE`:5	150
...	...	...	...	...
*	0	...	`DROP`	0

Table 4.1: A flow table with some installed flows.

The table 4.1 shows an example of an OpenFlow flow table, to illustrate some of the possible behaviors.

The first two flows are a static (timeout:0) entries. The first one decreases the IP TTL then forwards all http traffic over the physical port 3 as a classical router. The second one has never been used (counter:0) and will forward all LLDP traffic (eth_type:0x88cc) to the controller, for example as part of a topology discovery mechanism implemented on the controller.

The third visible flow replicates IPv4 NAT features found in some routers, where the packet header is rewritten. Furthermore, as the final instructions is a `GOTO_TABLE`, additional processing will be performed on the packet, after the fields have been rewritten, starting in the flow table 5. This table could for example then implements firewall-like functionalities. Finally, this flow is set to expire after 150s without encountering any packet match.

Eventually, the last rule is the flow-miss, which in this case discards every packet not matching any other flow. This particular flow entry has the side effect that the controller will never be able to react to new kind of traffic in the network at this switch. Another approach could be to send packets not matching any flow to the controller. However as this is the first flow table, this would expose the controller to a DDoS attack as it could be easily overloaded by `PACKET_IN:FLOW_MISS` messages. A better design would be to have the first flow table sending all 'likely desirable' packets to other tables, where flow-miss entries are set to encapsulate the packet for the controller, while keeping the flow-miss entry in the first table as a `DROP` as a protection.

4.2.2.3 Group Tables

An OpenFlow switch has one group table. A group has a unique identifier, pointing to a set of action buckets. It also has its set of counters as for the flow entries, and a group type to control its behavior. There are four possible types of groups:

`ALL` This group will execute all action buckets associated to it. This can be used e.g. to perform multicast forwarding;

`SELECT` This group will execute one action bucket. This bucket is chosen by an external, implementation specific, selection algorithm, e.g. to implement a round

Group Identifier	**Group Type**	**Counters**	**Action Buckets**
1	ALL	...	decrease_ttl, output:1 ;
			decrease_ttl, output:2
...	FF	...	[watch_port:1] output:1 ;
			[watch_port:`CONTROLLER`] output:`CONTROLLER`
...	...	...	...

Table 4.2: A group table in an OpenFlow switch.

robin queue between the action buckets and have the switch behave as a load-balancer;

`INDIRECT` This group can only have a single action bucket associated with it, and always execute it. This allows to add an indirection layer in the flow table, by having multiple flow pointing to the same group. Use case is for example to enable faster flow reconvergence when a next-hop for a given FEC changes as only one message from the controller is needed to change that group, thus effectively updating all flows using the group.

`FAST FAILOVER` This group will execute the first live bucket. Each action buckets of this group is associated to a port and/or a group which define whether the bucket is live or not. This requires the switch to support a liveness mechanism. This is used e.g. to execute a local-repair of a flow without any action from the controller.

In a flow table, the action set of a flow entry can have a specific instruction (`GROUP`) which allows a packet to be processed by a given group. This adds a layer of abstraction in the flow table, by adding an indirection between the flows FEC, and the forwarding action in itself. Furthermore groups can be chained through their action buckets, to allow for more complex behavior (such as flooding and applying a meter table to avoid flooding too often).

An example of a group table is visible on the table 4.2. Assuming that the switch in which this group table is installed has more than 2 OpenFlow-enabled physical ports, we can say that the first group entry denotes a multicast group (grouptype: `ALL`).

The second entry is more complex. It is a fast failover group, which attempts to forward the traffic to the physical port 1 as long as it is live. If the port goes down, it will then forward all traffic to the controller. It is worth noting that even the last action bucket needs to have a liveness constraint (in this case the liveness status of the `CONTROLLER` port), meaning that even the last remaining action bucket in the group will have its liveness evaluated. The fast-failover group is the only mechanism in OpenFlow that might decide to drop packets despite the fact that its set of programmed actions contains `output` actions, if all its buckets are considered as dead.

4.2.2.4 OpenFlow Ports

The Openflow ports of a switch are the interfaces allowing packets to transit between the OpenFlow processing pipeline and the rest of the network. A switch will expose a given set of ports to its OpenFlow pipeline. This set of ports might not include all hardware ports of the device, and can also include extra logical ports.

Three types of ports are defined in the specifications:

Physical ports These are the ports that are directly mapped to an hardware interface of the device.

Logical ports Unlike physical ports, these ports are an higher level of abstraction and have been defined in the switch using non-OpenFlow methods (e.g. loopbacks, . . .). Those ports are used by the OpenFlow pipeline in the exact same way than the physical one.

Reserved ports These are special port numbers defined by the OpenFlow standard. They refer to generic forwarding actions such as flooding (`ALL`, all ports excluding the ingress port of the packet) and sending the packet to the controller (`CONTROLLER`, associated with a specific reason). It is worth noting that the only way to do an U-turn (to send back a packet to its sender) is to use the reserved port `IN_PORT`. Among these reserved ports, there is also a special one (`NORMAL`) specified to let the switch process the packet in its non-OpenFlow pipeline if available, e.g. to behave as a traditional L2 switch (MAC learning switch).

4.2.3 The OpenFlow communication channel

The OpenFlow communication channel is the second component of the specification. It is the medium through which a remote entity, usually called a controller, can program an OpenFlow switch, thus control its forwarding behaviors.

It is interesting to note that the specification do *not* cover the initialization of the switch. They let the configuration of the initial flows in the switch (e.g. to be able to forward traffic towards the controller) at the discretion of the network operators.

4.2.3.1 Overview of the OpenFlow protocol

The OpenFlow protocol standardizes the way a switch and the controller communicates, allowing the latter to have a complete control over the FIB of the switch. It is designed to be secured with TLS, but supports a fall-back mode to use unencrypted TCP connections.

Because the protocol runs on top of (S)TCP, it provides reliable message delivery. However, by default the protocol does not provide any guarantees regarding the order in which the messages are processed nor does it provides acknowledgments that the message has been processed.

Upon opening a connection, the controller should inquire for the version the protocol that the switch is using, as well the particular features it supports. For example, supporting the `NORMAL` reserved port is an optional feature for the switches to implement.

A keep-alive mechanism is included in the protocol, in order to monitor the health of the OpenFlow channel, through the use of `ECHO` messages.

It is possible to have a switch having OpenFlow channels with multiple controllers at once. It must then be carefully configured, e.g. to know how which switch it should reach to when it faces a `FLOW_MISS`.

Finally, the protocol support the notion of transaction, via `BUNDLE` messages. These will instruct the switch to register all messages associated to a given bundle, in order to execute them later all at once if they are valid, or none of them if one is erroneous. This mechanism brings the notion of local all-or-nothing transactions, which are a key component to ensure consistency between what the controller thinks is the state of the switch and what it is in reality.

4.2.3.2 Main messages exchanged over the OpenFlow channel

There are three main message types defined in OpenFlow.

Controller-to-Switch message These are the messages sent by the controller to the switch, sometimes requiring an answer from it. Among this set of message, are all the messages modifying the state of the switch, such as `FLOW_MOD`, which alter the flow entries in the flow tables, `GROUP_MOD`, which does the same but for the groups, ...

Another important message in that category is the `PACKET_OUT` message. It allows the controller to instruct the switch to send arbitrary data over one of its port, thus allowing it to for example to actively send LLDP probes to discover its topology.

Finally, a `BARRIER` message is present. This message forces the switch to process every message it has received before the `BARRIER`, before being able to process newer one. This allows the controller to ensure that temporal dependencies between messages (such as removing and adding of flows) can be met.

Asynchronous messages These messages can be sent by the switches to the controller at any time.

The first one is the `PACKET_IN` message, which contains an encapsulated packet. This can be sent for three different reasons: an explicit instruction to send the packet to the controller was set in a flow, the packet triggered a flow-mis or the packet TTL expired.

The two other ones reflect change of status of the switch: the `FLOW_REM` message, notifying that a flow has been removed, and the `PORT_STATUS` message, which notifies the controlled that ports have been added/removed from the OpenFlow datapath, or that the status of a port has been modified (for example due to a liveness mechanism detecting that it is down).

Symmetric messages These can be sent by any party, without solicitation. They include the `ECHO` message, as well as the `ERROR` one among other. These messages are of no use to program a switch, but rather needed by the OpenFlow protocol itself to function properly.

Chapter 5

Challenges of both paradigms

5.1 Limitations of distributed routing protocols

5.1.1 Limited expressiveness

Distributed protocols lack of expressive power, as they make the next-hop decision for every packet according to the same algorithm and their forwarding table. This means that:

1. It is very hard to have certain packets following an arbitrary path in the network. Some workarounds exist (e.g. strict routing option in IP header or tunneling), but add yet another layer of complexity in the network and are hard to manage;

2. Because the decision algorithm is well-defined, usually linked with the destination IP, splitting traffic towards the same IP address based on an arbitrary field of the packet is hard.

Overall, these protocols have static behaviors, and if an operator wants something else he usually has to use an alternate protocol, e.g. encapsulate the packet and forward it using MPLS.

5.1.2 Traffic engineering is intrinsically complex

Ensuring that some link are favored over others, that only some particular routes are exported to other AS, that some links are always congestion-free, . . . requires the network operators to design complex traffic matrices to set the IGP properties, as well as complex configuration files for BGP, possibly also dealing with RSVP-TE and MPLS, . . .

The result being that in order to achieve what could be simple requirements requires the operators to manipulate multiple protocols and to individually configure each of those of every router of the network.

5.1.3 Interactions between protocols can hinder the overall network behavior

Due to the fact that some protocols exchange information with others, it is extremely hard to predict what will happen after a given event (e.g. a configuration change, . . .). This can results in transient inconsistencies, as presented in [4] where reconfigurations of BGP sessions are shown as potential source of forwarding loops and black-holes due to its interactions with the IGP.

This means that since operators have to use multiple protocols at the same time to achieve their overall goal, the interactions between those can also be the source of the violations of these same goals.

5.1.4 Transient undesirable effects can happen during the network convergence

Distributed protocols rely on each node building their own forwarding tables, meaning that there are no synchronization during the network convergence between the nodes. As routing update message (e.g. LSP's) can get lost during their transmission, some nodes might eventually receive them way after the other ones, thus update their tables later, which in some cases can result in forwarding inconsistencies such as routing loops. Example of such inconsistencies are shown in Sec. 7.2.1.

5.2 Main challenges of the OpenFlow SDN approach

5.2.1 Scalability issues

OpenFlow revolves around the decoupling of the control plane from the data-plane elements. This causes the data-plane to be made of programmable switches, responsible of the traffic forwarding according to the flows that have been installed on them by the OpenFlow controller. This implies that the switches are completely relying on the controller as they do not perform any reasoning as to how to forward a given packet. They only apply pre-installed rules. This design can lead to scalability issues in a network.

5.2.1.1 Switches have a limited memory

The memory available on the switches is limited [30]. This means that the controller has to carefully select which flows are installed on the switches at a given time, in order to avoid overloading their memory.

Flow aggregation on the switches in order to reduce the flow table size can only happen if the controller explicitly optimizes the flow tables of each switches. Furthermore, this is a costly process as it requires multiple Controller-to-switch messages to perform it. Indeed, as presented in 2, the switch will prefer more specific flows against more general ones in order to select which is the matching flow. This means that if the controller wants to aggregate two flows, it first has to send a `FLOW_MOD` message to create the aggregated flow from one of the existing one, then has to send a `FLOW_REM` for the other one in order to effectively stop using it.

5.2.1.2 Asynchronous messages could overload the controller

Another scalability issue arises from the fact that the switches are dependent on the controller to know how to forward the traffic. These have to actively communicate with it in order to learn the new flows by reporting `FLOW_MISS`. From there, the controller can choose to either setup a new flow entry on the switches, or to process the rest of that traffic itself through `FLOW_MISS` messages. As the number of switches in a network is likely to be much greater than the controllers, this means that the controller could be overloaded if the switches flood it with asynchronous messages.

5.2.2 Challenges for the controller

The controller in an OpenFlow network is a critical component. Indeed, it is the one exploiting the capabilities of the switch over the OpenFlow protocol, and can be custom-made specifically to meet the network needs. However, it is also expected to be always available, as nothing happens in a pure OpenFlow network if the controller is down.

Designing controllers also brings a lot of research challenges, in various areas.

5.2.2.1 The controller has to be scalable

One inherent issue with having a central point of decision for a whole network can be the scalability of the controller. Indeed, as it has to do everything, from setting up flows, to collecting statistics to perform network-wide traffic engineering, or simply interacting with other routing protocols such as BGP, not overloading the controller is a crucial need to ensure the proper behavior of the network. In order to meet this need as well as the high-availability requirement, OpenFlow supports running multiple controller in parallel. Those can for example be responsible of different part of the network, or be part of a slave-master relationship were a controller makes the global network-wide decision and delegates the fine-grained management of the switches to other controllers. Another way to help controller to scale better is advocated in [31], and consists in trying to reduce controller-switch interaction to a minimum, by aggressive use of wild-carded flows to reduce the number of `PACKET_IN` messages, by using proactive protection mechanism to handle failures without involving the controller, ...

A point especially critical when having multiple controller is the controller placement problem, for example studied in [32]. Indeed, knowing how many controller are needed for a given topology, or where to place them is an important topic. It directly has an impact on the latency of the controller-to-switch connections, as well as its resiliency.

5.2.2.2 Consistency requirements

A strong requirement for a controller is to have an accurate view of the network state, at all time. This implies the need to define ways to monitor the health of the network elements, to infer the network topology as new elements are added and removed, as well as to know how to infer the expected flows from these topologies and the observed traffic. One could for example run a variant of LLDP, implemented on the controller, to discover the available links.

Another key challenge linked with the previous one lies in the fact that the controller has to setup network-wide flows, by actually sending messages to each individual switch along these. This brings in the need to design mechanism allowing for consistent update of the network state, in order to avoid transient undesirable effect such as the ones observed during IP networks convergence. Such a mechanism is being heavily researched, as it is an highly desirable feature to have, and has been discussed for example in [33].

5.2.2.3 The controller as a software engineering project

As OpenFlow is a rather low-level API, it is highly desirable for a controller to be modulable, in order to be able to extract common frameworks that network operators could cherry-pick to build the controller best-suited for their needs. Example of these would be new languages created specially to manage SDN networks [34] [35], frameworks providing an abstraction layer on top of OpenFlow, frameworks to remove the need of the network operators to deal directly with failures [36], ...

Last but not the least, building an OpenFlow controller is part of a software development process. However, bugs in the controller can have disastrous effects, ranging from simply making forwarding mistakes to turning the whole network as routed by primitive switches, thus subject to congestion, ... As such, new tools have to be developed to ensure that the controller are as free of bugs as possible. Such tools include model-checker aware of the OpenFlow protocol [37], tools to test the implementation of the OpenFlow protocol [38]. ...

5.2.3 Impact of failures in a SDN

The OpenFlow approach, by decoupling the control-plane from the data-plane, introduces a new element in the networks: the controller. As with any other element, it is thus an additional source of failures.

The fact of having a clean separation between both planes can help to remove a long-lasting problem that was present distributed networks: transient inconsistencies (e.g. forwarding loops, ...). Indeed, if a mechanism to apply updates in the global network state in an atomic fashion is in use, it can allow the controller to bypass any inconsistencies due to the fact that different nodes in the network are running different version of the network state (have not yet updated their FIB while others did, resulting in inconsistent forwarding decision).

The main issue with failures in OpenFlow is the lack of any default behaviors. Tools exists to program some form of reaction to failures directly on the switches, however it is the network operator responsibility to properly set up those, for every network design. This is a major limitation of the protocol, has it forces operators to spend time into accounting for every failure scenarii and programming the reaction to it, even if the only thing they want is basic connectivity.

This limitation however could be solved by introducing general frameworks that one could use, which would automatically handle failure in both the reactive mode and the proactive on. Some of those will be covered in Chap. 8.

5.2.3.1 Failures related to the controller-to-switch connection

The first of these is the failure of the link between the controller and a switch. Indeed, if it loses its connection with the controller, the switch cannot learn any new flow nor react to any unplanned event. What happens in that case is left to the particular switch vendor implementation and on their configuration. For example, it could start behaving as a traditional Layer-2 device, it could stop forwarding any packet, . . .

Another one is the failure of the controller itself, causing every switch in the network to lose their connection with it. This would be a disaster, as the whole network could then potentially turn as a block-hole.

It is possible to mitigate these issues by having multiple controllers on the network, allowing for others to take the relay in case one of them crashes.

5.2.3.2 Link and node failures

SDN networks are obviously also affected by link or node failures, as any other networks. The effect however is different. Indeed, as the switches are not autonomous, they will at best be able to detect that a network element has failed. They will then report it to the controller, which will react to it and program the new network state resulting of that failure in all the affected switches.

This has two negative side-effects. The first one is that this process is slower than what has been achieved in traditional IGP's. Indeed, the network recovery time is constrained both by (i) the communication delay between the switches and the controller and (ii) the complexity of the computation on the controller.

The second delay is quite important. Indeed, instead of distributing the load of computing the new network state across multiple nodes, it is solely performed on the controller. If the failure affects multiple flows, the controller has to find new paths that will satisfy the network policies and send `FLOW_MOD` messages to each concerned switches in order to restore the traffic. This is a non-negligible load, especially since the controller would still be responsible for managing the rest of the network, not affected by the failure, thus possibly processing `PACKET_IN` messages from the switches, . . .

This drawback is a strong argument in favor of having a distributed controller, as this could help to split the load across multiple nodes (e.g. by assigning some controller to failure handling, others to new flow instantiation, . . .). This however increases the overall complexity of controller design.

Chapter 6

Conclusion

The road to having programmability in the network has been long and is certainly not finished, however the SDN paradigm brings interesting capabilities in the networks. OpenFlow especially seems to be a promising technology allowing network operators to define their own custom behaviors in their network, without waiting for the usual slow process of standardization and vendor adoption.

While not resolving any long-standing issues on its own, OpenFlow gives more flexibility to the network operators, while lowering the costs to change the way a network operates as it is mostly a matter of performing a software upgrade on the controller.

Being a kind of very low-level API, OpenFlow is however still quite complex to use if one wants to have high level functionalities in its network. Developing abstractions on top of it is a requirement to reduce complexity, which is one of the key feature enabled by OpenFlow as an API (compared to the multiple networking protocols all adding complexity on top of each others).

Some interesting challenges have been brought up, such has the ability to perform consistent updates over the whole network or distributing the controller, and are likely to be resolved thus bringing new frameworks taking of these low level details, eventually leading to network operators being able to express what they want in a high level fashion.

While already deployed in data-centers, OpenFlow still has issues that needs to be solved for deployment in service-provider networks. Indeed, the lack of built-in resiliency and the combination of small TCAM sizes makes it hard to meet the carrier-grade requirements of having sub 50ms restoration times when failures occur in larger networks with a high number of flows, especially if some backup policies must be enforced. Another obstacle to its deployment in physical networks is the fact that this is a very young protocol. Indeed, updates to it frequently break backward compatibilities and there is a lack of expertise with that protocol, might be a deterrent factor for service providers.

On the other hand, distributed control-planes protocols have been in use for decades. They are well-understood, stable, robust, and highly scalable. However, as we expect more and more from the networks, we end up running more and more distributed protocols in parallel. This results in an increasingly complex network, which is hard to operate and where the subtle interactions between all of them can lead to severe traffic disruptions.

Furthermore, distributed control-planes are historically highly tied to the hardware they are running on. This has the side-effect that adding new capabilities to existing protocols is a very slow process, and can result in a vendor lock-in.

The rest of this document will focus on failure recovery. Failures handling is a critical part of network management, as these directly impact its overall quality, thus the ability of service providers to meet their strict SLA's. This is known to be one of the bigger weaknesses of the SDN approach, whereas the handling of failure was one of the stronger points in distributed networks.

The next part will first explore how those were handled in a distributed environment, and then presents the recovery techniques currently developed to be used in OpenFlow. Finally the last part will present a network architecture advocating for the usage of distributed mechanism to handle failures. It leverages the fact that most networks, if they want to move to the SDN paradigm, will have some transition period during which both control-plane technologies will co-exist, which can have some benefits.

Part II

Handling failures in a network

Chapter 7

Failure recovery in distributed IP networks

Distributed routing protocols are the traditional way to control the forwarding of the packets in IP networks. Their usual mode of operation consists in each router exchanging some topology data in its neighborhood, eventually leading to each host having enough information about the state of the network to be able to autonomously decide of the forwarding of the packets, in a globally consistent way providing a best-effort delivery.

As these protocols are distributed, they benefit of some inherent resilience against failures, as long as a mechanism to notify the nodes that a network element is no longer available is present. Indeed, as each node has its own decision process, the failure of one of them will not impact the capabilities of the other nodes to make their forwarding decision. It might however cause some packet losses during the network convergence.

The handling of failures in IP networks is a well-known topic and has been studied for years, as well as improved over various RFC's.

This section will outline the general techniques used by these protocols to detect failures in the network, and how these protocols react in order to restore the traffic as fast as possible. The goal of this section is to understand the applicability of these techniques in the SDN paradigm, as failures handling are one of its weakest point in its current instantiation.

7.1 Failure detection

Failure detection is one the key component allowing for failure recovery. The speed at which a failure is detected will directly influence the overall restoration speed. They can be detected by different mechanisms, built within the network protocols themselves, or running beside those.

7.1.1 Using HELLO PDU's

Link state protocols (e.g. OSPF, IS-IS) make uses of HELLO PDU's in order to infer their neighborhood as well as their liveness status, as every message sent to another router has to be acknowledged by it. They will then flood the network with their LSP's to propagate the information. This is a primitive failure detection mechanism, embedded within the protocol itself.

The speed at which a failure is detected is directly related to the 3 timers linked with these messages:

1. The one controlling the rate at which these PDU's are sent;
2. The one controlling the retransmission delay to wait for the acknowledgment of sent messages;
3. The one controlling the delay to wait before considering a neighbor as failed if the router has not received HELLO from it in a while. This delay is heavily linked with the two previous ones.

The lower the value of these timers are, the faster the failure will be detected. However, this increases the number of control-plane messages sent, thus requires more processing power on the routers.

Distance-vector protocols (e.g. RIP) tend to not have this mechanism, but instead rely by default on periodical broadcast of each node routing table. If a node stops receiving updates from a neighbor, it infers that it has crashed and reacts.

BGP uses HELLO messages, on top of its TCP session and thus can detect when the session is down similarly than with link-state protocols, although it only needs the timers 1 and 3, as the retransmission is handled by TCP.

7.1.2 Using dedicated protocols to monitor the link health

There exists dedicated protocols to monitor the state of the network elements.

BFD, for Bidirectional Forwarding Detection, is a protocol dedicated to monitoring the health of tunnels, with a low overhead. These tunnels can span over a single link, thus monitoring the link health, or across multiple router (e.g. monitoring a MPLS tunnel). These tunnels to monitor are configured on each separate routers. The general idea of the protocol is to actively probe the tunnel health. Either by sending special packets and then waiting for the reply of the router at the other end of the tunnel, which is best suited at times where the traffic going through it is low. Or by piggybacking data in the forwarded packets themselves.

This protocol has been designed to be efficiently implemented in hardware, which allows it to be run with very short timers without increasing the load on the router CPU, unlike the HELLO PDU's.

Other protocols can be used, e.g. LLDP, however these might not run on hardware, thus suffer from the same drawbacks than using the HELLO PDU's were applicable.

7.1.3 Hardware-based detection

Depending on the underlying physical medium, there can exist some form of signaling mechanism which notify a router when it loses a physical link. These mechanisms, upon a failure detection, can for example bring down the interface at the OS level of the router, which will trigger a reaction from the control-plane protocols.

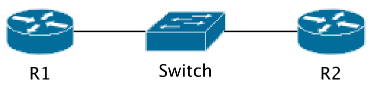


Figure 7.1: A sample Ethernet network connecting two routers

Example are Loss-of-light signals in optical networks, or more advanced technique such as signal degrade [39] in SONET/SDH networks.

However, these techniques might not be enough, especially in the case of Ethernet networks. For example, if we add a switch in between two routers, as seen in the figure 7.1, and if only the link between R1 and the switch fails, then R1 will be notified of the failure via a hardware mechanism. It will then be able to act on it and update its forwarding table. However R2 will not, as it does not use the physical link that has failed. It is thus necessary to use some Layer-3 protocol to monitor the health of the link between R1 and R2.

7.2 Reactive recovery techniques

This is the set of mechanisms that are built-in the distributed routing protocols. The execution of those on each router triggers what is called the network convergence. They ensure that:

1. The traffic is eventually restored;
2. The FIB of each routers in the network are eventually consistent after the failure has been detected with regards to the routes selected for each prefix, thus removing all transient issues such as black-holes, forwarding loops, loss of visibility.

MPLS has no default reaction mechanism in case of failures. Indeed, as it relies on an external protocol, such as RSVP-TE or LDP, to perform path signaling, Label Switching Routers do not build a full RIB covering all prefix in the network. They are thus unable to react to failures using only MPLS information. It is the responsibility of the ingress/egress routers to ensure that the path they are using is actually working, by example by using a probing protocol to monitor the health of the tunnel such as BFD or LLDP. MPLS can however use proactive techniques, as presented in section 7.3

The next section will outline how IGP's react to failures

7.2.1 Reaction of link-state routing protocols

Upon detecting a change of connectivity with one of its neighbors, the router will send Link-State Packets to each of its neighbor reflecting its new connectivity status. These will then performing flooding of that LSP. Upon receiving this message, routers will recompute their shortest-path tree and update their RIB if needed.

As there is no synchronization between the routers, it is possible to observe some transient forwarding loops [40] in the network during the flooding, as routers along a forwarding path might not all have updated their RIB when forwarding the traffic, resulting in transient inconsistencies.

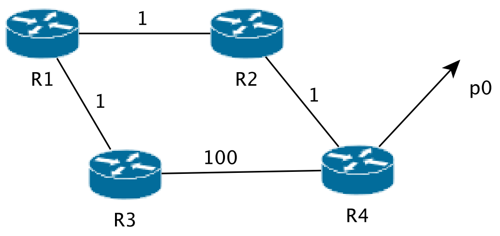


Figure 7.2: A sample network were transient forwarding loops can happen.

An example is illustrated by the network on Fig 7.2. According to the costs, R1 and R3 will route traffic towards prefix p0 via R2, as the link R3-R4 is costly. If the link between R2 and R4 fails, then R2 and R4 will send a LSP notifying the change of connectivity to their neighbors, instructing them to update their RIB. However, as R1 keeps receiving packets towards p0, it will during a short time frame keep forwarding it to R2, as it is in the process of computing its shortest-path tree. This will create a transient loop between R1 and R2, lasting as long as R1 has not updated its RIB. Moreover, when it has done so, a loop could still happen between R1 and R3 if R3 takes more time to update its RIB than R1 did.

These transient loops could cause packet losses, as they might cause the IP TTL of the packets to expire. Furthermore, they can induce temporary congestion on the links, as traffic enters that portion of the network but does not leave it. Thus it also affects routes towards other prefixes that should not have been affected by the failure.

The bottleneck of the convergence time after the failure has been detected and advertised is the computation time on each router to update their SPT, as the Dijkstra algorithm is in $O(\texttt{\#Links}^2)$. In order to improve it, routers usually implements incremental SPT algorithms.

As these mechanisms are executed *after* the failure has been detected, and take some time to complete, packets can be lost during the network convergence. However, it has been proved that the traffic can be restored in less than 50ms, thus meeting carrier-grade requirements (having a very high service availability to meet their SLA's). This result is achieved by fine tuning the failure detection mechanism, and also using protection mechanism to speed up the traffic restoration during the network convergence, as covered in sec 7.3.

7.2.2 Reaction of distance-vector protocols

Distance-vector protocols maintain a table linking prefixes to their outgoing interface and the relative cost from the current router as well as the router they learned the route from. They however do not compute the path that the packet will follow in the network.

When a failure is detected by a router, it will send a route update message mentioning that it has no longer a route towards the failed network element, thus setting its cost to infinity.

A well-known issue with this is the count-to-infinity problem. Indeed, as it has no knowledge on the path selected towards a prefix, when a router receives route updates including the failed element, it does not known if the route advertised goes through itself (thus is no longer valid), or goes through another path in the network (is thus a valid route).

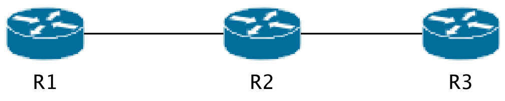


Figure 7.3: A sample network causing distance-vector protocols to suffer from the count-to-infinity problem.

For example, if we take as metric the number of hop towards a prefix and for the topology in Fig 7.3, if the link between R1 and R2 fails, and if R2 relies on the broadcast of route update messages to detect the failure, then it will receive a route from R3 towards R1 with a cost of 2, as R3 has not yet been informed of the failure. Has it received a route, and knows that it cannot reaches R1 by itself, R2 will then send a route update to R3 saying that it is at a cost of 3 from R1. R3 however knows that it reaches R1 through R2, so it will update its cost and advertise R1 at a cost of 4, causing R2 to react again, ...

This issue has been limited in RIP by using two techniques:

1. Split-horizon: If a router updates its RIB upon receiving a route update message, then it will not send its own route update message on the interface from which it received the initial message;
2. Reverse-poisoning: RIP defines a maximum cost for a route to be considered valid which can be used to directly advertise the loss of connectivity to neighbors. Those can then check whether their route is still valid or also remove it and advertize that infinity cost as well.

While not removing every possible occurrence of count-to-infinity phenomenons, these workarounds help to mitigate the issue and its effects.

7.2.3 Reaction of BGP

BGP will interact with failures in two different ways.

If the failure happens inside the AS, the IGP will converge and cause RIB updates within all routers in the network. As border routers run both BGP and an IGP, this will cause them to send UPDATE message over the BGP session if needed. For example, as the underlying IGP paths have changed, the egress point for some prefix could have changed, potentially affecting their overall AS-PATH. Furthermore, if the egress point changes, then depending on the policies of the border routers, some prefix could no longer be advertised resulting in route removal messages.

The other case is when the failure directly impacts a BGP session. The border router detecting the failure will send UPDATE message to all other BGP routers it is connected to (thus possibly other border-routers, route-reflectors, . . .), removing all the routes it had received over the failed BGP session. Those other BGP routers will then rerun their decision process and advertise alternate routes towards the lost prefixes if applicable.

7.3 Proactive recovery techniques

These techniques work in two phases:

1. The offline phase involves generating backup configurations in order to handle some failure cases;
2. The online phase involves using the backup configurations as *short term* solutions in order lose as few packets as possible, while in the background the full protocol convergence process is started in order to eventually return in an optimal configuration.

7.3.1 IP Fast ReRoute extensions to routing protocols

The IETF has standardized some techniques in order to restore connectivity in case of failure directly at an IP level. These are all part of the IP Fast ReRoute framework, standardized in [41]. Experiments have shown that these technique can protect between 40 and 90 % of a network links and speed up the traffic restoration process [42].

These are best suited for use in link-state protocol, as they require knowledge of the path long which packets get forwarded.

7.3.1.1 ECMP – Equal Cost MultiPath

This extension allows routers to store multiple next-hop for a given prefix, if the result of their SPT computation yields multiple next-hops with the same overall path cost towards the destination.

The router then selects one primary path to use, and keeps the other ones in order to immediately switch to them if the selected path fails. Forwarding loops are not possible with this restoration scheme if assuming a non-zero value for the links. Indeed, when a packet reaches the ext-hop of a particular path, the overall remaining cost of that chosen path is strictly less than going back to the path origin.

This extension is also frequently used for traffic engineering, in order to perform load balancing across links sharing the same cost. In that case, link selection is for example performed by `hash(packet) modulo #paths` which allow one to pick links at random with uniform probability. Other alternative exists, such as scheduling.

This has been discussed in [43] and introduced as part of the IEEE 802.1 standard [44].

7.3.1.2 LFA – Loop-Free Alternate

This technique, proposed as IETF standard in [45], also known as Feasible successors in Cisco's iOS implementation, protect a router's link by computing a backup next-hop to be used in order to proceed to a local repair of the link.

As illustrated on Fig. 7.4, a link I–J used by a source S towards destination D has

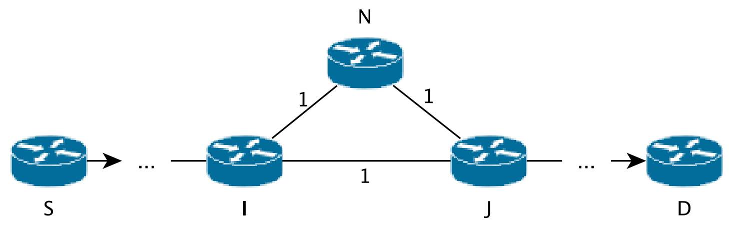


Figure 7.4: A sample network where router I has an LFA protecting the link I–J.

a loop free alternate router N if it does not introduce loops, thus if it satisfies the triangular inequalities (a packet sent to a LFA by router Icannot be sent back to I):

$$Cost(N \rightarrow \ldots D) < Cost(N \rightarrow \ldots I) + Cost(I \rightarrow \ldots D)$$

As this may yields lots of alternates, reducing the set of all LFA's we can add the downstream criterion (a packet can go through a LFA only if makes it *advances* towards its destination):

$$Cost(N, D) < Cost(S, D)$$

This will furthermore limit the amount of transient loop that could happen if multiple LFA's are used along the path.

7.3.1.3 U-turn alternate

This is not yet standardized, the current draft is available at [46] and adds customs behaviors to routers as well as requires modification to the existing routing protocols in order to support it.

It states that if a router R receives traffic from a router N towards a destination D for which its closest next-hop is N, then it has detected a U-turn. If the router R has a LFA L for that destination D, it will forward the traffic to it. That LFA is then called a U-turn alternate, and R will set its next-hop for that destination to L.

This technique is thus dedicated to breaking forwarding loops as soon as they are detected, leveraging the Loop-Free Alternates.

7.3.1.4 Not-via addresses

This was also introduced in the IPFRR framework [41] and later developed in [47].

It requires the usage of 2 addressing spaces within the network, the *classical* one, e.g. the IP's belonging to the prefixes owned by the AS, and the *not-via* one, e.g. local IP addresses. When a router S detects that it cannot reach D_k (router D on his port k) it will encapsulate the packet and send it to the *not-via* D_k address to one of its neighbor.

This can be done via tunneling (e.g. using MPLS tunnels solely for protection), and thus require the participation of the other routers along the recovery path. This can also be coupled with ECMP and LFA's, in order to cover all prefixes in the network, for both addressing spaces.

7.3.2 Tunneling – MPLS Fast Restoration schemes

Despite not including built-in reactive resiliency, MPLS offers a few standardized techniques to provide protection against link and node failures. These consists in pre-installing backup tunnels on the routers, to protect their tunnels from failures. It is thus coupled with external failure detection mechanism as covered in 7.1.3 and in 7.1.2.

This comes however at a price, as the space required to store all backup tunnels explodes with regard to the number of simultaneous failures one wants to protect from. However, as [48] has shown that more than 70 % of the failures in a network at a given time are usually due to only one network element (link or node), protecting against single link/node failure already provides valuable coverage.

MPLS has two main protection schemes, both subdivided in variants providing different levels of protection and other drawbacks. They are all described in the MPLS-TP Survivability Framework [49] as well as in [50].

7.3.2.1 Global Path Protection

These protection schemes focus on protecting a complete LSP at once, through the use of another backup tunnel. This is available both for unidirectional tunnels and bidirectional ones. In the latter, it requires the usage of an additional protocol to coordinate which tunnel is in use at a given time by both routers. The common flavors providing global path protection are described in the next paragraphs.

1:n Linear Protection Schemes The 1:1 protection scheme creates one backup tunnel per LSP. This implies that the backup path is actively reserving the needed resources to be able to immediately switch between paths if the primary one fails.

The impact of that can be lowered by setting QoS rules, such that packets can be forwarded along reserved backup tunnel, but with lower priority. This allows keeping the guarantees of having a protection path capable of supporting the whole traffic from the protected LSP, while not forcing network elements to stay idle, not forwarding any traffic, because *a failure might happen.*

Another side effect is that core LSR's are required to know twice as many paths, thus requires routers to have more memory.

A variant of this is the 1:n protection scheme, were one backup path is created for a set of backup tunnels. This allows to lower the memory footprint on the LSR. However, it might not be possible for the backup path to reserve enough resources to cover the needs of all the LSP's it is protecting at once. A more reasonable resources constraint for the backup path is to be able to reserve the resources needed by the most expensive path.

1+1 Linear Protection This scheme has each Label Switch Path associated with a backup LSP. The traffic is then duplicated and forwarded through both tunnels. If the two paths do not have network elements belonging to the same Shared Risk Link Groups, this provides instant recovery from failures without any packet losses.

However, this wastes lots of bandwidth, as it uses at least 50 % in extra in order to transmit the data twice. Furthermore, this means that unlike in the 1:n approaches, the backup paths cannot be used for other traffic under normal mode of operation. If the backup tunnel is longer than the one it protects, which is likely, then the bandwidth consumption of the backup path even exceeds the one from the original tunnel.

7.3.2.2 Local Path protection

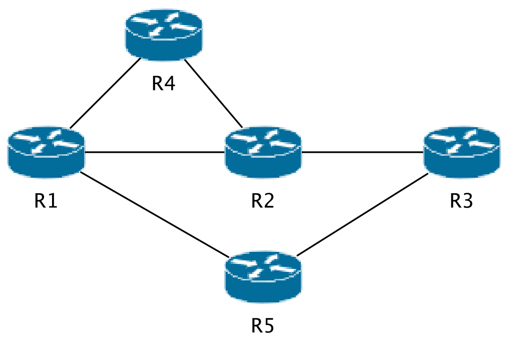


Figure 7.5: A network were R1 can both have NHOP and NNHOP protection tunnels.

These protection schemes attempt to proceed to local repair of the overall LSP. This means that a LSR, upon detecting a failure, will attempt to use an alternate neighbor in order to get around the failed element. This is similar to LFA's in IPFRR.

Two kinds of protection can be established, and are illustrated in Fig 7.5:

NHOP protection Also known as link protection, a backup tunnel is made to provide resiliency against the failure of a link. This usually implies using another router as next-hop, which will immediately send the packet to the original next-hop to circumventing the failed link. On Fig 7.5, R1 has a NHOP protection for the link (R1, R2) if it sets up a tunnel doing R1–R4–R2.

NNHOP protection Also known as node protection. This time, the backup tunnel is attempting to bypass a node as a whole, as it is suspected to have failed. On Fig 7.5, R1 has a NNHOP protection towards R3 against the failure of R2 thanks to R5.

Similarly to the 1:1 and 1:n global linear protections, local protection can be set in two flavors.

Local protection: Detour – One-to-One Backup A backup LSP towards the destination is created at each hop for each LSP that has to be fast re-routable.

Reduction rules exist in order to reduce the total number of LSP's this generates, for example by taking advantages of Forwarding Equivalence Classes. The overall footprint however stays quite high is those are computed for every LSP on every LSR.

Local protection: Bypass – Facility Backup This is the closest to the 1:n protection scheme. A single backup tunnel is created for each possible next-hop on each LSR. This backup tunnel will then be shared with all LSP's.

When a packet has to take that backup tunnel, it will typically have an extra label pushed on top of its stack, which will get popped when it reaches the original desired next-hop of the LSP.

7.3.3 Other approaches

While not all standardized or in use, lots of other techniques providing fast restoration have been proposed over the years.

This part will outline some of the more interesting ones.

7.3.3.1 Virtual Protection Cycles

This is an attempt to port the p-cycle protection scheme from the SONET-SDH physical layer [51], to the IP layer and is presented in [52].

It consists in creating virtual circuit between routers and links in the network. If a router detects a failure when it wants to use a link to forward data, it then encapsulates the packet and sends it in reverse over the p-cycle containing that link. The packet will leave the cycle when its current cost towards the original destination will be lower than the cost it had in the router that sent it in the p-cycle.

This approach has the benefit over not-via addresses in that it creates explicit set of protection cycle, and that if those cycles are well chosen, they can provide protection against multiple logical link breakdowns (if for example a single physical link was used for multiple IP-level links).

7.3.3.2 Failure insensitive routing

This approach [53] claims that since most of the failures are short lived (as seen in [48]), flooding the network with link updates can induces more instability in the network due to the convergence process than needed, especially since some protocols implements mechanism such as route dampening to penalize routers often going down. Instead, it favors an approach where successive local repairs are performed by the nodes, as they learn about failures.

It does so by assigning a FIB to each line card. When a packet for a given destination arrives by an unusual interface, the node can infer that a link has failed, and thus proceed to select a next-hop avoiding that failed element. As it does not perform a global flooding, it avoids all transient issues caused by network convergence with standard protocols.

The key concept of this approach is the fact that the forwarding is no longer only conditioned by the destination of a packet, but also by its input interface. This allows routers to update their topology of the network directly from the data-plane, which runs at higher speeds than the control-plane.

The main drawback of this approach is that it has to have pre-computed SPT for each destination as well as the reverse ones from which they infer the failures when they receive a packet on an unusual interface.

7.3.3.3 Resilient Routing

This approach, presented in [54], attempts to provide a congestion-free routing as well as performance predictability in case of an arbitrary number of failures.

In order to do so, it moves the topology uncertainty which comes with traditional protection mechanisms, into uncertainty in rerouted traffic. This means that instead of having to deal with failures of various elements in the network which directly affect the routes taken by the packets, it instead attempts to model the increased load on other links due to the re-routing of the traffic. From that model, it can then preemptively set the weights on the links to avoid congestion once the failures have been detected.

It operates in two phases:

Off-line Phase It computes a set of linear constraints on the amount of traffic which will have to be rerouted on the other links in case of failure. It then computes the routing configuration according to a traffic matrix made of the normal condition to which it add the virtual demand that would happen in case of failure. This ensures that the link weights are set such that no congestion will happen as a result of the failures, if possible.

On-line Phase When a link fails, the router simply rescales the traffic load to the other links, for example by using tunneling. This traffic off-loading will not cause congestion, as it is only a small part of the total virtual demand that has been computed during the off-line phase, and thus of the pre-allocated bandwidth.

7.3.3.4 Multiple Configuration Routing

The key concept of this approach [55] is to have multiple sets of link weights on each routers. One to be used under normal condition, and the other ones to use in case of one or more failure(s).

The configurations used upon failures are generated in order to isolate links and/or nodes, by setting high weight values. Those network elements are considered isolated when the only traffic going trough them is when they are the source or the destination of the packet. Once every link or node in the network has been isolated in a configuration, these can be installed on the routers. Configurations can be merged together, provided that in each configuration, the subgraph containing all nodes and links that are not isolated in that particular configuration stays connected.

When a router detects that a link or node has failed, it attempts to find a configuration in which that failed elements is isolated. It then routes the packet according to that particular configuration, marking in the DSCP byte of the IP header which configuration it is using in order to let know the other routers that something went wrong.

When a router receives a packet, it checks the DSCP byte to infer which configuration it should use to forward that packet. This ensures that the overall forwarding of a packet is consistent across the whole network, thus preventing transient effects such as loops, . . . ingThis is yet another example of a technique avoid the whole network convergence process/SPT re-computation.

One big advantage is that this does not depend on the failure type (e.g. if it is a node

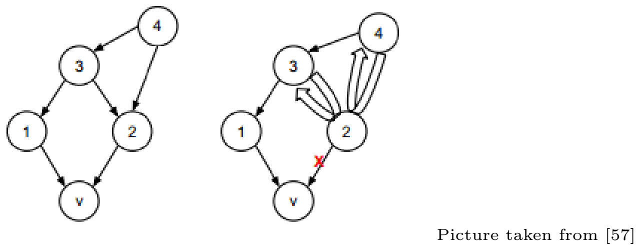


Figure 7.6: Illustration of the forwarding DAG of DDC. (i) Before a failure (ii) After a link failure

or a link) nor on the failure count.

7.3.3.5 Delegating connectivity insurance to the data-plane

Recently, a completely different solution was presented by [56]. It advocates for the relegation of failure recovery to the data plane. The main benefit of this lies in the fact that the data plane operates at line rate, which is much faster than the control plane. While the link connectivity itself is maintained by the data plane, the control plane is still responsible for the optimality of the paths that the data will follow across the network.

This approach makes uses of the link reversal routing algorithm originally presented by [57], adapted for the data plane (thus without the control messages). It lets switches build a directed acyclic graph where the only node without outgoing link is the packet destination.

It makes use of a single bit of state carried by the data packet, which can be implemented either by piggybacking or by using two virtual links on top of a single physical interface. That bit is then used in order to propagate whether a link has been reversed or not.

The rest of the algorithm is then quite simple: upon detecting a failure preventing from forwarding the incoming packet, the switch reverses the incoming link and bounces back the packet.

This is illustrated in Fig. 7.6. The left part of the figure shows the original DAG towards destination node v which is in used during normal operation. When the link between node 2 and v fails, as visible on the right part of the figure, the node 2 invert its links, thus bounces back the packets towards v using the introduced state bit of the approach. This allows node 3 and 4 to also take note of this, thus to also reverse their link towards node 2 in their respective DAG towards v.

This approach is independent of the routing protocol used, as it behaves by handling failures on its own, thus ensuring connectivity. As it is operating at line rates, it requires the use of as fast as possible detection mechanism. Finally, the main issue regarding the deployment of this technique is the fact that this requires changes directly in the hardware operating in the network. As such, there are no physical implementation of this.

7.4 Applicability of these techniques in SDN's

The key difference between IP routing protocols and SDN is the fact that the former are completely distributed processes instantiating the control-plane directly on the nodes doing the forwarding, while the latter is a centralization of the control-plane, completely decoupled from the data-plane.

This means that whereas IP network can benefit from some inherent resiliency due to the fact that each node can make its own decisions, sometimes with undesirable transient side effects. SDN nodes on their side need to wait for instructions coming from the controller if they want to react to a failure.

Regarding the detection of failures, as the SDN nodes do not run any routing daemons, they cannot use HELLO PDU's to infer the liveness of their neighborhood. However, all the other means of failure detection can still be used, especially if those can be implemented in hardware(e.g. built-in the physical link medium, BFD, . . .). When a failure has been detected, the switch can then signal it to the controller to trigger a reaction from it, instead of reacting by itself as IP routers do.

Most of the techniques present in the IPFRR framework can be translated in the SDN paradigm, mostly as set of proactive backup rules, updated when the controller sees a change in the network topology. For example, LFA's for a given next-hop could be implemented in OpenFlow as fast-failover groups where each action bucket would contain one LFA.

The SDN approach, especially in its OpenFlow version, shares lots of similarities with MPLS (although SDN switches are likely to have less memory than core LSR). Indeed, it shares the notion of global arbitrary paths across the network, spanning over multiple network elements. Furthermore, as with MPLS, even though it has no resilience built-in, proactive recovery can be setup to protect the established paths. Unlike MPLS however, it does not require the use of external protocols to perform signaling or resources reservation. These aspects are all directly handled by the controller. Furthermore, as mentioned in Section 4.2.2.3, OpenFlow has the notion of Fast-Failover groups, allowing one to switch between pre-configured next-hops as soon as a failure has been detected.

The alternative approaches such as resilient routing or failure insensitive routing can also be implemented on SDN nodes, as these techniques can be split in two phases, the offline computation phase being performed on the controller, while the online one happens directly on the switch (for example in OpenFlow by using matches on the input port, by using groups to perform load balancing as fast-failover action, . . .).

As a conclusion, IP routing has explored various techniques to provide resiliency against failures. Many of them can be directly applied within the SDN paradigm, especially those involving a proactive protection mechanism. Purely reactive techniques are unlikely to work by design, as these require distributed processing to happen on the nodes.

Chapter 8

Recovering from failures in an OpenFlow network

As outlined in section 5.2.3, failures in OpenFlow networks is one of the main weaknesses of the paradigm.

Indeed, the OpenFlow Switch Specifications [29] do not define any default behavior in case of failure, meaning that the network could turn itself into a black-hole when one occurs, or worse, depending on the implementation of the protocol on the switches.

Furthermore, as OpenFlow introduces a centralized, globally visible element in the network, namely the controller, this is also another point succeptible to failures. This last type of failure is especially important considering that the switches are not autonomous concerning their forwarding behavior, as they explicitely need actions from the controller to learn how to handle new traffic.

Whereas traditional network protocols rely on their distributed algorithm running on the network nodes to achieve resiliency, OpenFlow, as a consequence of the separation between the control-plane and the data-plane, cannot do so. Instead, in current OpenFlow networks, the operator has to explicitely define what happens when a particular failure occurs in the network, in order to restore the affected flows.

This section will outline how failures can be dealt with in OpenFlow networks, and what are the limitations of these approaches. Namely, it will start by presenting how to detect failures in an environement where switches are not running routing protocols. From there, it will present the two main modes of recovery, the proactive approach and the reactive one, and finally it will present how failures of the controller-to-switch connection affect the network, and how these can be dealt with.

8.1 Detecting failures

Detecting that *something went wrong* is the first step towards ensuring that failures eventually get dealt with. Several kind of failures can affect a network state. Nodes can crash, links can break, either as a result of a physical issue (e.g. submarine cable disruption) or of a logical issue (e.g. bringing an interface down on a router's administrative itnerface), resulting in flows which are no longer behaving properly and

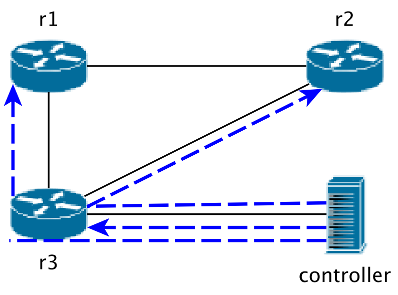


Figure 8.1: A controller having in-band connections with the switches it is monitoring.

are losing packets. The detection speed at which failures are detected directly affects the overall performance of the network in terms of traffic restoration time.

The failure detection can be achieved by any agent in the network, depending on the desired performance (e.g. detection speed, load on the controller), the available hardware capabilities (of both on the switches and on the controller) and level of failures (how many simulatneous failures do we want to be able to identify at once).

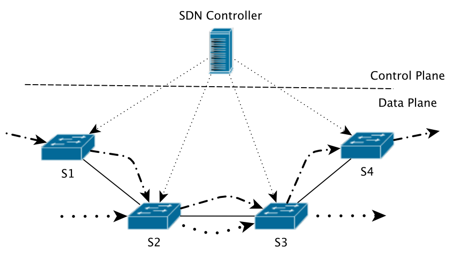
Detection mechanisms exists for both the controller, and the switches.

8.1.1 Controller based detection

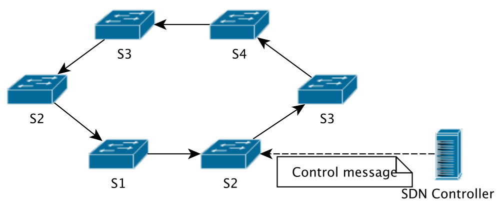
A first approach is to assign the responsibility of the detection to the controller. A naive and inefficient solution would be to have it actively probe each switch (on each link) in the network using some kind of hello messages (e.g. LLDP[58], . . .).

This solution suffers from multiple issues. First of all, it might not be possible to detect with precision which network element has failed. Indeed, assuming that the controller has an out-of-band connection with each switch is not reasonible for real-world deployments, as adding dedicated links for the control-plane is very expensive. As a consequence the controller-to-switch connection is in-band, alongside the regular traffic, and possibly crossing multiple nodes before reaching the other end of the connection as illustrated in Fig. 8.1. This means that if the controller sees that a connection with a switch is down, for example its connection towards r1, it has to check all other OpenFlow channel sharing that path, in order to infer which element has failed, which in this case could only be the link between r3 and the controller. This gets even more complicated in the events of multiple simultaneous failures (e.g. due to SRLG's), as these can partition the network thus prevent the controller from monitoring a whole partition.

Beside not providing a complete detection mechanism, this solution does not scale well at all[59], as the controller has to send an increasing number of messages directly proportional to the link count and the node count to perform a single round of checking to see if the network elements are still alive. Since the speed of failure detection is critical to ensure fast restoration, this means that these control message should be sent at high rate. The combination of these two points makes the simple task of monitoring the network health a task imposing a significant work load on the controller, which might cause it to have poorer performances for its normal tasks e.g. responding to the `PACKET_IN` messages sent by the switches to learn new flows.



(a) A simple SDN network with two active flows.



(b) An Eulerian cycle across all physical links of the network.

Figure 8.2: A sample network where the links health is monitored according to [60]

A more scalable approach is proposed by [60]. It revolves around the controller computing an Eulerian cycle across all links under its responsibility. If there are no such cycle in the network, it uses a workaround by using each link twice, by making them directed, thus forcing said cycle to exist as every node in the network has then an even number of edges. The controller then proceeds to install flows in each switches, thus instantiating that cycle. A possible Eulerian cycle for the SDN in Fig.8.2(a) can be seen on the Fig.8.2(b).

From there, it can attach itself to one of the switches, and send a control messages over that cycle. In the example on the Fig.8.2(b), we see that the controller has attached itself to the switch S2 and is sending control messages from there and waits for those to loop back in order to assess the network health.

As soon as that control message fails to come back, it can perform a binary search using another set of static rules in order to locate the link or node that has failed. In terms of OpenFlow rules, the binary search can be understood by the controller sending two different control messages: one over the link S2–S1 and one over the link S2–S3. It also setups S4 for example to send the control messages it receives as `PACKET_IN` to controller. This allows it to detect in which set of links is the failure, repeating the process until the set is reduced to a single element.

There are two main limitations with that approach. The first one is that it cannot handle more than a single link or node failure, as this would break the cycle in multiple places. This would render the binary search unable to locate all failures (as for example the first two sets of links it would create could both contain failed elements).

The second one relates to the speed of the detection of the failure itself. Indeed, if it uses only one control message at a time, this means that the time needed to detect the failure is proportional to potentially twice the sum of the latency over each link. This means that while this approach scales well with regards to the load on the controller, it performs badly on larger topology. This scaling effect could be avoided by sending multiple control messages with delays between them shorter than the time they take to travel over the whole cycle. However, it spawns a new issue, as one as to be able to estimate the delay on each link in order to setup how long it should wait between the receiving of two control messages befored declaring that a network element has failed. Finally, in this framework, detecting that a failure happened is only part of the detection time. Indeed, after that, it needs to perform a search over the network elements to find the one that has failed in order to be able to react to that failure and restore the traffics.

Overall, controller-based detection solution do not perform well if one wants to detect failures both as fast as possible and without overloading the controller in larger networks. This motivates the need to still keep some mechanisms directly implemented on the switches, in order to quickly detect failures.

8.1.2 Switch based detection

Assigning the responsibility of detecting the failure to the nodes themselves is not new, and is what was done in traditional networks. While these methods are usually much faster, the controller still has to be notified that a failure happened, to let it respond to it and update the affected flows.

Such system is presented in [59]. This paper presents a way to monitor the health of a given tunnel at the egress side, thus in one direction. It advocates that fault monitoring is a function that should be put close to the data-plane. As such, it adds generators of OAM messages on each switches. These will send control message over the installed flows to monitor their health. The switches along the flows are then able to receive these packets, and through the use of static flow table entries can process them accordingly. It is worth noting that this paper presents a solution pre-dating OpenFlow 1.1. As such, it has to implements state tables and timers by itself.

Similarly, especially since the introduction of the fast failover group type, BFD[61], as presented in Sec 7.1.2 could be used on the switches (either by piggybacking it in the data packets, or by generating new packets) as the liveness mechanism of the switches.

As these methods are performed at the speed of the data plane, either by piggybacking control message or by generating packets if the traffic is too low, the detection itself is very fast. As it makes use of dedicated protocols optimised for that purpose, the overhead on the network itself stays low and as it does not involve the controller until the failure is localised, it does not add any load on it, regardless of the timers used.

8.2 Handling link and node failures

Once a failure has been detected, the network has to react in order to be able to fulfil its role again, withstanding the failure. There are two kind of reactions which are possible, depending on the timescale at which we want the recovery process to happen, and overall optimality of the new state of the network.

8.2.1 Proactive techniques – Using path protection

These techniques relies on using the fast failover group, coupled with e.g. BFD, to quickly switch between forwarding actions depending on the liveness of the output ports, without any intervention from the controller at the time at which the failure occurs. These are very similar in essences to the protection techniques more detailled in Sec. 7.3.2.

With these techniques, when the controller installs a flow in the network, it will also pro-actively install backup flows entries to protect it from a set of link or node failure(s). According to [62], using this Openflow functionality can already provide sub 50ms recovery time although this was only done in simulations, and only covers failures of a single element at a time, thus is not covering SRLG's.

The main issue with these techniques lies in the fact that the switch memory dedicated to the flow tables is rather small. It thus limits the overall numbers of flows that a given switch can have at any time, thus forcing the network operators to a trade-off between the number of flows in-use and the number of backup flows they can setup to cover a predetermined set of failures.

Backup flows number can be optimized down. For example, CORONET[63] reduces the overall number of backup entries via the following technique: the network operator will express its desired flows in terms of segments (a set of one or more link). Each of these segments is implement as fast failover groups on the switches. This allows the operator to setup a high number of flow entries, by decomposing them in simpler elements, common to multiple flows, which can then be individually protected.

While this approach allows to tune down the number of backup flows to support, it causes one issue: as backup flows are aggregated, this means that they lose the granularity of their original flows, which was the result of enforcing network-wide policies. This means that during a failure, some packets could take forwarding path no longer respecting these policies.

This leads to a solution called FatTire[64]. It proposes a way for the network programmer to express policies at a path level, which must be enforced at all time, even in case of failure, e.g. enforcing that all outgoing traffic must go through the firewall except for the one originating from the DMZ. This framework then computes all possible backup paths meeting these invariants for the flows needing protection, and aggregates those that have non conflicting policy-requirements. In the end, this reduces the overall number of backup paths which have to be installed on the switches, while still metting network-wide invariants.

Finally, an orthogonal approach is presented in [65]. This paper argues that load balancing and path protection should be treated as one single problem. In this approach, the controller has to compute and install backup paths for each flows. All of these are then in-use all the time, achieving some overall load-balancing. Upon a failure, the switches will split the traffic that should have taken the failed path over the remaining ones. It also notifies the management system about it, which can then react to provide a more optimal routing solution in the network, but the connectivity is preserved meanwhile. This paper is independent from the routing paradigm used, but using it within SDN's is straightforward. It sounds also quite promising when combined with the approach presented in Sec 7.3.3.3

Overall, proactive techniques have the key benefit of allowing the network to restore connectivity as soon as the failure has been detected. However, it is hard, both memory-wise and computationally-wise, to proactively protect a given path against simultaneous arbitrary link failure(s). These techniques should be seen as *temporary* solutions, while the controller computes a globally optimal solution to handle all the traffic in network, given the failed elements.

8.2.2 Reactive techniques – Using the controller to restore connectivity

Restoration techniques, albeit usually slower as on-the-fly computation is performed, provide at least locally-optimal routing configuration once a failure has occurred.

There are no default restoration behavior in Openflow: once a path has failed, the packets eventually get lost in the network at the point of failure, causing a blackhole. If no protection has been setup for a flow, it needs the controller to define a new flow to use as backup and to install it on the switches in order to restore connectivity. This is a very slow process, compared to the speed at which the data-plane runs, especially since the controller has to modify at least all the flows impaired by the failure.

As no default failure recovery mechanism is specified, it is thus the network operator's task to implement error recovery in its network controller. This usually implies at least two components:

1. A way to infer all the current flows in the network;
2. An heuristic to repair the failed flows, according to some policies.

Having to implement this is precisely what [36] argue against. It claims that error handling code and network policies instantiation are two separate tasks, and that mixing both will increase code complexity thus increase the risk of bugs. It advocates for a cleaner architecture, where both tasks are completely separated. As such, it proposes a generic module for the POX controller which will provide failure handling on its own, without any special intervention from the network operator, thus letting him focus on implementing the normal mode of operation of the network.

The principle is to record all events happening on the controller in-between failures, such as `PACKET_IN`, `FLOW_REM`, ... Once a failure occurs, it spawns a new controller instance in a virtual machine, with the same topology as before minus the failed elements. It then proceeds to replay all events it has recorded in that new controller instance. Finally, it swaps the state of the controller with the one of the virtual machine, and pushes all new flows to the switches.

This approach brings two key benefits: (i) The resulting network state is a *global* optimum given the network policies (ii) It brings a clean separation of concerns while writing the controller code, thus reduces the risks of bugs and the overall compelxity of the controller code. It however induces a non-negligible load on the controller during the reaction triggered by the failure, which might prove to be too heavy and disturb the overall network if failures happen too often.

Another approach which has a much lower footprint on the controller is presented in [66]. Unlike the previous solution, when the controller has to perform the restoration, it does not perform a complete recomputation of the flows. Instead it requires from the controller to remember all established paths in the network, and selectively repairs the ones directly impacted by the failure.

This trades global path optimality for local one. Indeed, the obtained solution might not be the optimal one as only few flows are adapted to the new network configuration. An example of this can be found on the figure 8.3. Supposing the network policy is optimal load balancing, if the link S1-S3 fails, [66] will only change the flow S1–S5 to either S1–S2–S5 or S1–S4–S5. This could result in congestion on the link S1–S2, or on

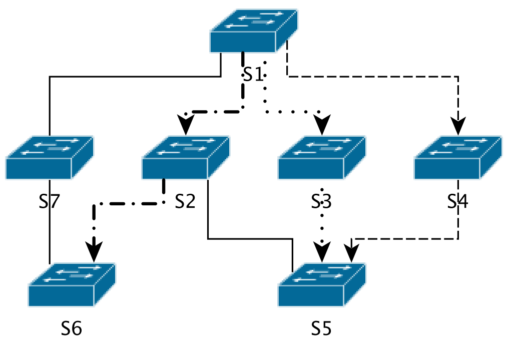


Figure 8.3: An example SDN whith 3 established flows where [66] could provide a sub-optimal recovery.

the link S1–S4, which would violate the network policy. On the opposite, [36] would have also changed the S1-S6 and make it go through S7 instead of S2. Performing multiple successive local repair like this could result in a poor usage of the network resources.

Both restoration types have their pros and cons, and there is certainly a right balance to achieve between the two, e.g. by first reacting using a local approach, and by running a daemon in the background that periodically checks whether the network could be optimised better or not.

8.3 The controller as an additional failure source

This additional element in OpenFlow networks is critical.

Without the controller, switches are dependent on their implementation and their configuration to handle the traffic. Two modes of operation [29] are available for switches to enter as soon as they lose their connection with the controller:

fail-secure mode With this failover mode, the switch keeps using all the flows it has learned until then. They will however drop every packet that should have been sent to the controller. Flows still expire as in normal mode of operations when their timeout expires. Using this mode means that the network operator expects that the controller will come back online in a short amount of time. Indeed, after a while, due to the variation of the traffic and the fact they no longer can learn new flows, the flow tables might become empty, resulting in switches behaving as black-holes;

fail-standalone mode This mode requires the switch to support the reserved `NORMAL` port. Upon entering this mode, the switch forward every packet to that port. This means that this could result in a whole network being operated by Layer-2 MAC learning switch (which would be terrible from a traffic engineering point of view).

There are three main sources of failures tied to the controller, that would cause the switches to enter one of these modes.

The controller can crash. Either as a result of a bug, or because of an hardware failure, the controller availability can be compromised. This can be prevented by using multiple controllers, for example in a slave-master relationship, or as a part of a distributed controller [67]. This would bring some resiliency to the control-plane in case one of them crashes. Furthermore, using tools from the software-engineering world such as model checkers [37] could also help to prevent bugs, thus potential crashes.

The controller-to-switch connection can fail Either because link fails, or because the controller-to-switch connection was crossing another switch such as in Fig.8.1. Protection can be achieved by allowing controllers to be multi-homed, thus allowing it to use another interface if it detects than its primary one is down. Distributing the controller is once again a solution bringing resiliency to the control-plane.

The switches can overload the controller This is a more subtle interaction. Indeed, given the fact that a single controller has to manage multiple data-plane elements, and that it is globally visible to them and can receive asynchronous messages, switches can possibly overload their controller [31].

This overloading can be due for example to the default behavior of flow-miss entries in the flow tables which is to encapsulate the packet and send it to the controller. If the rules in the flow table matches only partially the whole traffic of the network, then the switch will receive lots of `PACKET_IN` message for all that unhandled traffic until it setups the proper flow entries.

Protecting against this requires more than just adding more controller. Indeed, distributed controllers need to share some state information regading the network. However, the state of the network is very likely to change after `PACKET_IN` messages for example. This means than even distributing the controller is not a solution. It could actually even make it worse, as all controllers would broadcast their local state-change to all the other ones, thus generating lots of control-plane traffic.

Instead, it requires the controller to aggressively use wildcards in the flows where possible, instead of specific values, to decrease the number of `PACKET_IN`, to make active use of fast failover groups to reduce the number of link down event, ... For example, if one wants to program his network to do some IP destination based routing, then wilcarding all the flows specific to the transport protocols would help to reduce the number of `PACKET_IN` messages.

8.4 Conclusion

Handling failures in OpenFlow is a tricky task. Fast recovery can be achieved by separating the handling of failure in two consecutive steps:

1. A transient connectivity restoration performed by the switches themselves, provided as a set of backup rules, allowing the traffic to be restored as fast as possible, possibly providing some policy guarantees;

2. A long-term optimization problem left for the controller, to compute the new network state given the current set of failed elements and the full network policies.

As OpenFlow encourages to have one single control-plane element it also introduces another source of service disruption. The controller, as any software, can contains bugs eventually leading to crashes, and the connection it maintains with each switches can fail. Protecting this aspect of the protocol requires the usage of distributed controllers cooperating. Furthermore, as that same controller is also globally visible to the switches, it is important to reduce to a minimum the interactions it has with them, in order to avoid overloading it.

These are the key principles currently driving failure handling in OpenFlow. Another important one introduced by [36] is the separation of concerns while writing the controller, by cleanly separating all failure recovery code from the policies managmement one.

Part III

Preserving connectivity in Hybrid networks

Chapter 9

Design of a robust hybrid SDN architecture

IBSDN is an hybrid architecture in which an IGP and a SDN collaborate, in order to provide programmability in the network as well as effective and scalable failure handling. It brings resiliency in SDN's against an arbitrary number of failures, without causing any overhead on the controller.

On one hand, failure recovery in networks is a critical component that heavily influences its performance. As shown in the previous parts, this is a subject that has been heavily studied in distributed routing protocols, and as such is now well-understood and efficiently dealt with. However, failure handling in the SDN paradigm is more scarce. Operators are required to define by themselves recovery mechanisms, and the overall resulting robustness of the network is directly constrained by both the size of the memory on the switches as well as the desired reaction time.

On the other hand, existing real-world networks are all using distributed routing protocols, tightly integrated with hardware equipments. This implies that the deployment of SDN technologies in current network is likely to require both paradigm to co-exist in so called Hybrid networks [8].

IBSDN leverages the fact that hybrid networks have both a control-plane using distributed routing protocols, as well as a SDN control-plane. It then uses the best-suited paradigm for each task to be achieved in the overall resulting control-plane. It assigns the responsibility of traffic engineering to the SDN control-plane, as it brings more powerful tools for that purpose due to the fine-grained control it has over the forwarding tables across the whole network, and lets the distributed control-plane deal with the preservation of connectivity. It thus benefits from the 30+ years of researches performed in distributed routing paradigm to achieve scalable and robust fault management, while still enabling the new use-cases brought by the SDN approach.

This chapter will focus on the general principles behind IBSDN, as well as the guarantees brought by its architecture. The next ones will detail an implementation of such network, as well as evaluate its overall performance and assess how it fares against the two main practices in SDN, namely the reactive or proactive approaches.

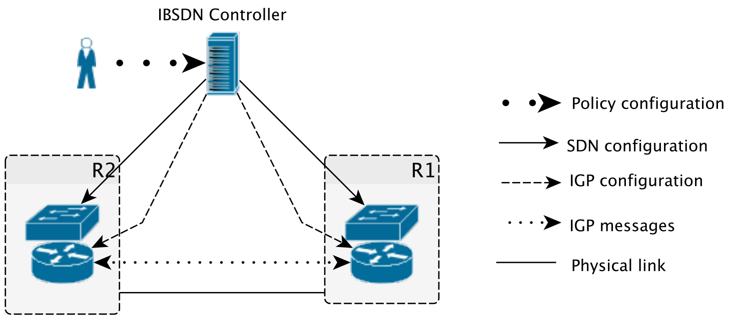


Figure 9.1: A high level view of the architecture of an IBSDN network

9.1 Architecture

IBSDN establishes a strict separation of concern when managing the network. It decouples the overall process of achieving optimal forwarding, which is assigned to a SDN control-plane, from the handling of failures, which is performed by an auxiliary IGP. As it uses concurrently the two network management technologies for specific purpose, IBSDN thus qualifies as a service-based hybrid SDN architecture [8].

From an high-level point of view, as illustrated on Fig. 9.1, IBSDN has two main components:

1. Network nodes running at the same time an SDN control protocol and a local agent which act as routing daemon for an IGP. This is why the routers on the Fig. 9.1 are represented as an aggregation of two datapaths, with one of them exchanging routing messages between the routers.

2. An IBSDN controller, that is in charge of both programming the FIB's of the nodes, as well as configuring the IGP daemons on each node.

Conceptually, an IBSDN network is used similarly by the network operators than a standard SDN, as depicted on Fig. 9.1. Network operators will first specify to the IBSDN controller the policies that should be enforced in the absence of failures. These policies are the backbone of SDN routing, as they allow operators to specify arbitrary forwarding behavior, e.g. forcing some paths to go through some middleboxes [68].

From these *primary policies*, the controller will then computes the required FIB entries that should be present in the switches forwarding tables, in order to meet said policies. The controller will then install these rules in the switches, either pro-actively (thus programming static routing entries across the whole network) or reactively to the traffic entering the network (thus reacting to switches querying it because they received unknown data), which will be called *primary rules.*

In parallel to that, the controller will also configure the IGP daemons running on each node in order to have them starting to exchanging routing information via a link-state protocol. This will allow the local agents to compute their shortest-path tree, thus to build their Routing Information Base, independently form the IBSDN controller. This RIB is then kept on each node, separated from the FIB containing the primary rules, as it contains the forwarding rules enforced by the IGP, which are called *backup rules*. Packets forwarded according to those backup rules are then called *IGP-forwarded packets*.

Finally, in order to define which control-plane (with its respective set of rules) should be used to handle a given packet, a third set of *control rules* are set in the primary datapath of the switches. These control rules are made of:

Next-hop control rules This is the set of rules that are applied if the next-hop of a packet is not reachable, e.g. because the link towards it has been detected as down;

IGP-path control rule This rule detects IGP-forwarded packets, and as such enforces a consistent forwarding for those packets.

All these control rules have the same effect, they instruct the node to use the backup rules to forward the packet. The number of these control rules set on any switch should thus always be lesser or equal to (the number of ports + 1), which is a very limited overhead.

Beside these three sets of rules, IBSDN also requires a mechanism to identify IGP-forwarded packets. Possibilities include tagging, encapsulation, tunneling, ... This mechanism is tightly coupled the IGP-path control rule setup on the switches.

An implementation of these rules is covered in Sec. 10.1.

9.2 Operational model

In the absence of failures, nodes in the network only use their set of primary rules to handle the traffic. This implies, assuming the correctness of the IBSDN controller, that the primary policies expressed by the network operator are thus enforced network-wide during normal operation. From an external point of view, the network thus behave as a pure SDN. However, as local agents on each nodes are running an IGP daemon, routing information are periodically exchanged between them. This allows the local agents to always have up-to-date backup rules in their RIB.

Upon the occurrence of a failure, packets will at first be forwarded according to their matching primary rules. When reaching a node that is adjacent to a failure, the node will use its next-hop control rules to decide whether the packet can still be forwarded according to its primary path, or if it has to be forwarded using the backup rules. If the node decides to use the backup rules, it then proceeds to mark the packet as IGP-forwarded.

Thanks to the IGP-path control rule present on every node, all nodes will then be able to detect packets whose primary path have failed, and thus will be able to route them according to the IGP paths. This allows IBSDN to avoid packet losses, as these are rerouted as soon as the failure is detected. An operational requirement for IBSDN is the fact that nodes must have a liveness mechanism to monitor the status of their interfaces, such as BFD [69]. This allow them to transition between primary rules usage and backup rules without requiring the involvement of the IBSDN controller and is nearly lossless if this liveness mechanism operates at line rate.

Fig. 9.2 and Fig. 9.3 illustrate how IBSDN works. As in Fig. 9.1, it represents the node as an aggregation of local agent and SDN datapath. The dotted links represent the

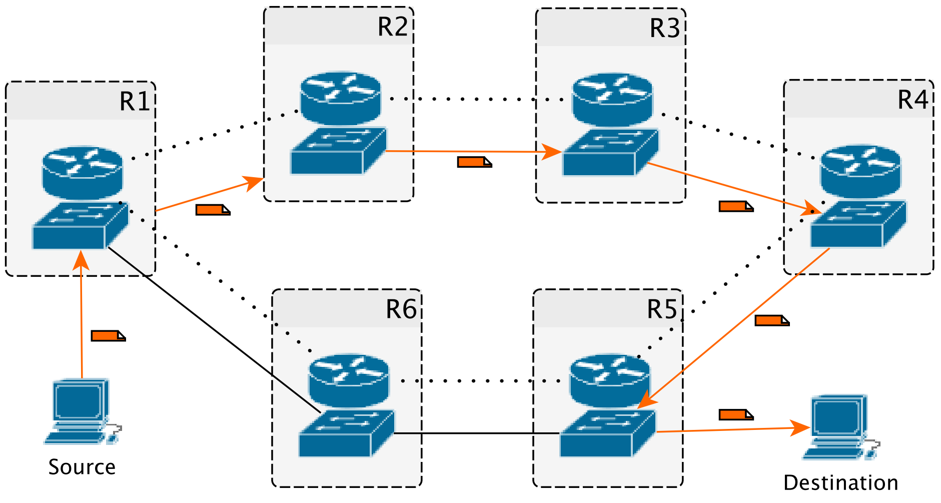


Figure 9.2: Forwarding of packets from Source to Destination in the absence of failure.

IGP links established between the local agents running on those nodes. The network operator and the IBSDN controller are not present on the figures because they do not play any role when failures occur.

Fig. 9.2 describes the initial state of the network, where all orange packets from the Source node to the Destination follow the primary orange path. As there are no failed element yet, the packets strictly follow the orange path, thus are handled via the primary policies of the network. Meanwhile, local agents sit there, exchanging their routing information.

Fig. 9.3 shows the reaction of the network to the failure of the link between R3 and R4 fails. Packets are forwarded using the primary rules until they reach a node that is adjacent to a failure, in this case R3. Using its next-hop control rules, R3 will then mark the packet as IGP-forwarded, hence the change of color, and send it towards its IGP next-hop. Due to the presence of the IGP-path control rules on every other router, the packet is then routed using the IGP backup rules until it reaches its destination.

This means than the initial path

$$\text{Source} \to R_1 \to R_2 \to R_3 \to R_4 \to R_5 \to \text{Destination}$$

Now becomes

$$\text{Source} \to R_1 \to R_2 \to R_3 \to R_2 \to R_1 \to R_6 \to R_5 \to \text{Destination}$$

Without any action from the controller.

9.3 IBSDN guarantees

This section will show that IBSDN is safe and guarantees maximal robustness.

From its architecture and operational model presented in Sec. 9.1 and Sec. 9.2, IBSDN has the following properties:

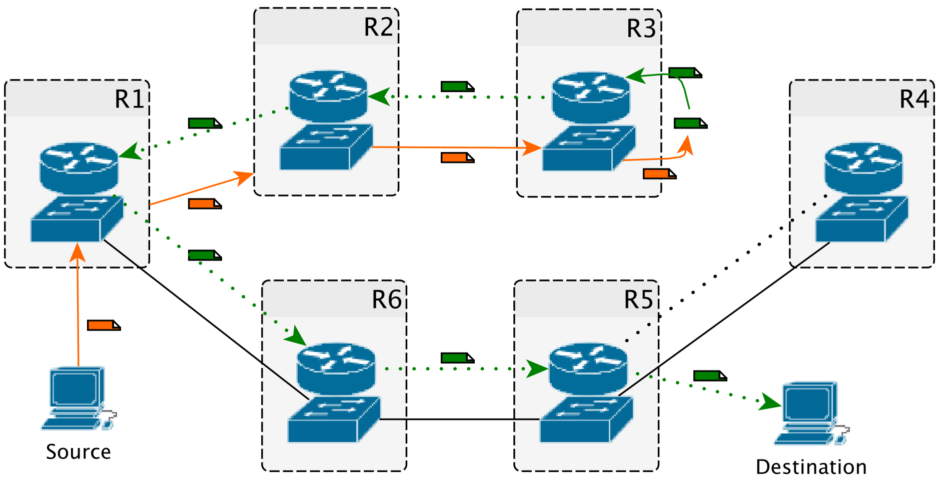


Figure 9.3: Forwarding of packets using IBSDN when the link R3–R4 has failed.

P1 Nodes adjacent to a failed network element rely on the IGP backup rules to forward packets whose corresponding primary rules was using the failed element;

P2 Packets are marked as being IGP-forwarded by the first hop that makes use of backup rules to forward it;

P3 Any packet marked as being IGP-forwarded will be processed on every node by their backup rules, thus forwarded to the IGP next-hop.

The first two properties are ensured by the next-hop control rules present on every node, and the last one is a consequence of the IGP-path control rule.

These properties ensure that:

P4 IBSDN preserves connectivity in the network, for any combination of failure not causing a network partition;

P5 The re-establishment of the connectivity is performed independently from the controller.

These can be proved easily.

For any combination of failures not inducing network partitions, local agents will exchange IGP information resulting in a network-wide convergence process, if needed, to recompute their RIB. This means that all nodes will have paths between each of them according to the newly-computed IGP backup rules.

Let P be a packet that is being forwarded through the network, after the occurrence of the failure. Let $\pi_{s,d}$ be the primary path that P is following in the network from any source s to any destination d, according to the network primary policies. Two cases are possible:

- $\pi_{s,d}$ contains no failed network element (node or link). As a result, P is then forwarded only according to the primary rules setup on each switches. This means that it strictly follows $\pi_{s,d}$, and is thus eventually delivered to its destination;

- $\pi_{s,d}$ contains at least one failed network element. Let n be the first node in $\pi_{s,d}$ which precedes a failed network element. P will be forwarded according to the primary rules until it reaches n. From there, due to P1 and P2, P will be marked as being IGP-forwarded and as such be forwarded according to n's IGP next-hop towards d. From there, due to P3, every other node of the network will forward P according to the IGP backup rules.

This proves P4. Indeed, since there are no partition, the IGP will always have next-hops for P to reach d. The property P5 on the other hand is inherent to the fact that this connectivity restoration mechanism is the result of local decision made by the nodes, thus not involving the controller, using the property P1.

Additional properties follow this:

P6 Packets never loop indefinitely in the network;

P7 IBSDN provides maximal robustness.

The sixth property is due to the fact that packets can forwarded according to two scenarii:

1. Packets are forwarded only according to the primary rules, thus are unaffected by the failures. They will not loop unless the initial primary policies were inducing those loops.

2. Packets are first forwarded by the primary rules until they reach a network element that start forwarding them using the IGP backup rules. As a result, the packets are then forwarded according to those rules until they reach their egress node. Due to the IGP-path control rule, packets cannot loop between the two sets of forwarding rules. Transient loops can occur as a result of the IGP convergence process, but these will eventually disappear, proving P6.

Finally, property P7 is the direct result of P4 and P5. Indeed, P4 ensures that IBSDN is resilient to any combination of node and/or link failures, while P5 ensures that IBSDN can react to those failures without involving the controller. This effectively means that IBSDN is resilient to all three types of failure that can happen in a SDN, as the failure of the controller can be dealt with by using the IGP backup rules for all traffic, thus providing complete connectivity in the network, until the controller comes back online.

9.4 IBSDN Benefits

IBSDN is an architecture that decomposes network management into two distinct problems, namely a long-term optimization and a short-term feasibility one. This distinction lets IBSDN uses the networking paradigm that is best-suited for each task.

The SDN approach allows network operators to express fine-grained arbitrary policies on how they want the network to behave. This is a very powerful tool, and as such is the primary one used in an IBSDN. However, in the events of failures, due to its logically centralized control-plane decoupled from the data-plane, this approach suffers from a few drawbacks as summarized in Sec. 8.4.

IBSDN attempts to solve this limitation by using a distributed algorithm as a way to preserve connectivity when failures occur. As such, it makes use of an IGP, that allows each node to have an additional RIB, to always have up-to-date reachability information. IGP's are robust by design, as covered in Sec. 7.2, and provides a strong protection against the failure of the controller-to-switch connection, which is the Achille's heel of SDN, by making the nodes able to share topological information and thus forward traffic properly.

IBSDN allows a cleaner design of the control-plane by enforcing a separation of concern in it, which translate in simpler components to setup. Indeed, the SDN controller does not need to deal with failure and as such benefits both from simpler programming, thus reducing the number of bugs, and from a lower load when something goes wrong. At the same time, configuring the IGP is also easier, as there is no longer a need to use complex traffic-engineering extensions, as this task is delegated to the SDN control-plane.

Beside lowering the load on the controller, using an IGP also allows to reduce the memory footprint on the switches. Indeed, as these are keeping an up-to-date RIB at all time, they can deal with an arbitrary number of concurrent failure, thus protecting affecting any number of SDN paths with a constant memory consumption.

IBSDN is an architecture performing a trade-off between the memory requirements of the switches, and the speed at which the traffic is restored in the worst case (e.g. failure triggering a network reconvergence). As an hybrid technique, this allows it to scale better in large networks, where purely proactive approaches have to deal with huge numbers of backup paths and where reactive approaches are too slow. This particular point will be evaluated in Sec 11.1.2.

9.5 Limitations

IBSDN makes use of an IGP to handle failures. This comes at a price. Indeed, as IGP protocols proceed to destination-based forwarding, they do not provide the same expressive power than SDN. As such, it is not possible to enforce arbitrary policies with this backup scheme. The IGP can be tuned to provide guarantees regarding some policies (e.g. avoiding particular links) but some cannot be achieved at all (e.g. taking alternate paths based on the IP protocol number of the packet).

Furthermore, even with a well-chosen set of link weights for the IGP, it is still possible that a network convergence might be triggered as a result of failures. This process has however been proven to be quite fast, as seen in Sec. 7.3.1. Network convergence can also induce some transient undesirable behaviors such as forwarding loops, but techniques exist to mitigate those as much as possible [40].

Similarly to the proactive recovery techniques performing local-repairs, packet are forwarded according to the primary policies until they reached a node that has detected a failure, before starting to follow the IGP paths from that node. This might induce some patch stretch, and packets could even have to perform some kind of u-turn as exhibited on Fig. 9.3.

Chapter 10

Implementing an IBSDN prototype

This chapter will describe the implementation of an IBSDN testbed, as well as show how the abstract behaviors described in the previous chapter are instantiated in practice. This is a three steps process.

First, in order to meet the guarantees shown in Sec. 9.3, some requirements for the overall architecture of the testbed have to be expressed and defined, such as what mechanism should be used to mark IGP-forwarded packets, ...

With this general architecture, it is possible to individually implements IBSDN nodes according the specifications defined earlier. The implementation of these nodes is the core of our testbed. Indeed, these nodes need to be able to have two concurrent forwarding datapath, as well as be able to run the two chosen control-plane protocols. Furthermore, they also need to be able to reason on a per-packet basis on what datapath should be used. The next component of an IBSDN network, the controller, will then be implemented, finalizing the implementation of the main components.

Finally, the process of creating the testbed in itself, matching a pre-defined topology, will be covered, as this is the final requirement to be able to evaluate the performance of an IBSDN network as presented in Chap. 11

10.1 Implementation overview

The first step for the implementation of an IBSDN is to define the two control-planes that will form this hybrid architecture. In this case, OpenFlow will be used, as it is the de-facto standard SDN protocol, in conjunction with OSPF for the IGP. OSPF, being a link-state protocol, can benefits from all the IP Fast-ReRoute framework presented in Sec. 7.3.1 and is widely used in enterprise networks. For ease of use, OPSFv2 thus IPv4 addresses will be used. There are however no compelling reasons preventing a switch of the testbed to OSPFv3/IPv6. Finally the IBSDN nodes will be devices running a Linux kernel, and will be more detailed in Sec. 10.2.1.

From this high-level choices, the key components of the IBSDN architecture as presented in Sec. 9.1 must be specified.

10.1.1 IGP-forwarded packets

The marking of the IGP-forwarded packets will be implemented via a tagging mechanism. In this case, packets will be explicitly tagged inside their IP header, by using different values of the TOS byte. The convention used in the implementation is that IGP-forwarded packets will have their TOS byte set to 32, any other value thus refers to the SDN-forwarded packets (or left for future usage).

As a consequence, implementing the matching clause of the IGP-path control rule is straightforward and can be as simple as: `eth_type:0x800 (IPv4), tos: 32`, where all non-specified fields are set to `ANY`.

10.1.2 Collaboration between the control-planes

The second step is to define how these two control-planes will collaborate. IBSDN assigns different function to each control-plane, by setting the OpenFlow one as the primary control-plane to use and the OSPF one as the backup one, which implies a hierarchy between the two. As a result, packets entering a switch will first enter the OpenFlow datapath. There, either the packet will match the IGP-path control rule, and thus be sent over to the OSPF datapath, or it will be processed according the the flows currently in the flow tables of the switch.

The transfer of packets from the OpenFlow datapath to the OSPF has however to be specified. As it is a one-way transfer between the two control-plane, it is implemented in OpenFlow by the use of the `NORMAL` output port, as presented in Sec. 4.2.2.4. Indeed, the OpenFlow specifications states that the behavior of that port is left to the implementors. In the case of IBSDN, the `NORMAL` port refers to traditional Layer-3 routing, using the OSPF control-plane.

With this specification, it is now possible to write down the complete OpenFlow version of the IGP-path control rule:

```
match=eth_type:0x800, tos: 32; actions=output:NORMAL; priority=65535
```

The priority clause ensures that this rule will have precedence over any other, thus ensuring the property P3 of Sec. 9.3.

10.1.3 Next-hop control rules

These rules are the central element of IBSDN, allowing its nodes to react to a failure without the involvement of the controller.

Their first requirement, is the presence of a liveness mechanism directly present on the nodes. Example of these have been cited in 8.1.2. In order to simplify the setup, and as everything is software-based, the failure of a link in this implementation will be simulated by bringing one of its bounding interface down. As such, monitoring the status of each interface is enough, especially since Linux provides ways for programs to be notified about a change in an interface status asynchronously, e.g. via netlink/`NETLINK_ROUTE` [24].

The next part of these control rules is the ability to dynamically switch between the control-planes, depending on the liveness status of the next-hop, or more precisely in this case, the output interface for that next-hop. This mode of operation can be implemented in two steps in OpenFlow:

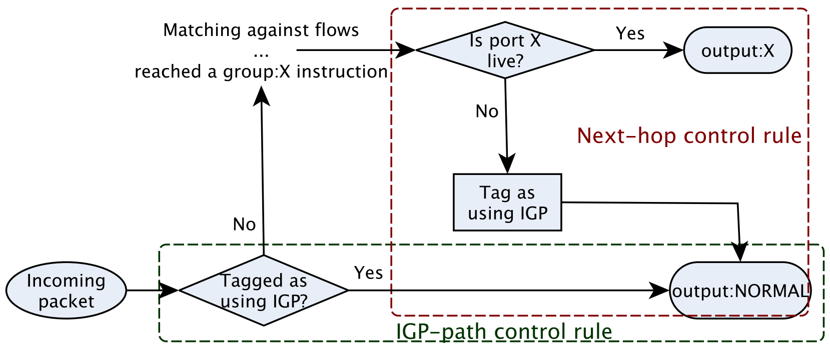


Figure 10.1: The decision proces for each IBSDN node to determine the next-hop of a packet.

1. It requires the presence of fast failover groups, each of those representing an output port. These groups, depending on the liveness status of the port, will then either forward the packet to said port, or send it over the `NORMAL` one, thus over the IGP;

2. It requires all the `output:X` actions on the flows to be replaced by groups actions (`group:X`), to be able to dynamically switch of control-plane in case of failure.

Finally, if the packets are forwarded on the `NORMAL` port, thus when a failure occurs, packet must also be tagged in their TOS byte as being IGP-forwarded.

A summary of the interactions between these components is shown on the Fig. 10.1. The figure explicits the interactions between the control rules, as the IGP-path control rule is always executed first. It also explicitly shows the fact that packets can only be tagged as IGP-forwarded if they are forwarded using the `NORMAL` port due to a next-hop control rule. This lets the network operators the freedom to define additional usage of the TOS byte in the absence of failures.

10.2 Implementation of IBSDN components

10.2.1 Implementing an IBSDN node

An IBSDN node is implemented as a Linux machine, which has two main components:

1. An OpenFlow datapath, in this case Open vSwitch [70];

2. A local agent which has an IGP datapath, which in this case is decomposed in two parts, the kernel IP forwarding engine, and the routing daemon BIRD [71].

As mentioned earlier, there is a hierarchy between these components, as the SDN datapath is the primary one. As a result, all physical ports of the node are under the responsibility of Open vSwitch and thus added to its virtual bridge, meaning that Open vSwitch is the only entity on the node handling the traffic, more detail on its behavior is presented in Sec. 10.2.2.

On startup, the node will first start Open vSwitch.

IPv4 forwarding by the kernel will then be enabled, by overwriting a kernel setting:
```
# sysctl net.ipv4.ip_forward=1
```
This setting will allow the Linux machines to act as Layer-3 routers if they have an IGP next-hop in their RIB for the packet entering that part of the kernel networking stack.

Finally, the BIRD routing daemon is started. It makes use the configuration that the controller has installed on the file-system of the node.

Fig. 10.2 presents the processing pipeline of the incoming traffic. At first, the packets get processed by the Open vSwitch datapath, following the decision process presented in Fig. 10.1.

If there is a matching flow, and if it contains forwarding instructions using physical ports, and if said ports are alive, the packet is immediately sent over the requested port. Otherwise, upon hitting a `NORMAL` output port, the packet is then passed towards the Linux kernel IP forwarding datapath where it is matched against the RIB of the kernel, and is forwarded according to a matching next-hop. If the destination of the packet is the node itself, it gets send further down and consumed by the process it is destinated to, e.g. in the Fig. 10.2 the BIRD daemon for OSPF messages.

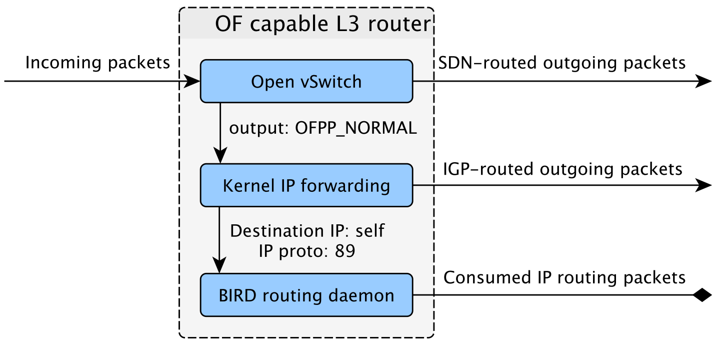


Figure 10.2: Processing of the incoming traffic on an IBSDN node.

With these three components running and organized as on Fig. 10.2, two issues preventing the node from operating properly remain:

1. IGP links between the local agents must be established. However, as all traffic, including the IGP messages, first go though Open vSwitch, these control messages never reach the routing daemon. Indeed, as there are no matching flows for that particular class of traffic, these IGP messages generate table-miss behavior on the switch, resulting in `PACKET_IN` messages on the controller, which is not the intended behavior;

2. The default behavior of Open vSwitch when hitting the `NORMAL` output port is to act as a Layer-2 learning switch, instead of letting the packet fall deeper in the Linux networking stack as expected in the IBSDN architecture in order to act as a Layer-3 router.

The first issue can be solved by adding a static OpenFlow rule. Similarly to the IGP-path control rule, it will match a class of traffic, in this case OSPF messages, and then directly send it to the local agent via the `NORMAL` port. This results in the

following rule:
`match=eth_type:0x800, ip_proto:89; actions=output:NORMAL; priority=65535`

The second issue is more complex. It requires to patch Open vSwitch, in order to achieve the expected behavior. This patching process is covered in Sec. 10.2.2.1.

With these issues solved, the node is then fully operational and can be used in an IBSDN.

10.2.2 Overview of Open vSwitch

Open vSwitch is made of three main components:

1. A Linux kernel module that provides an high-performance software switch through the use of a small and optimized flow table;
2. The ovs-vswitchd daemon, which implements management protocols such as OpenFlow and configures the kernel module to reflect the content of the OpenFlow flow tables;
3. The Open vSwitch database, which stores all the configuration information regarding ovs-vswitchd.

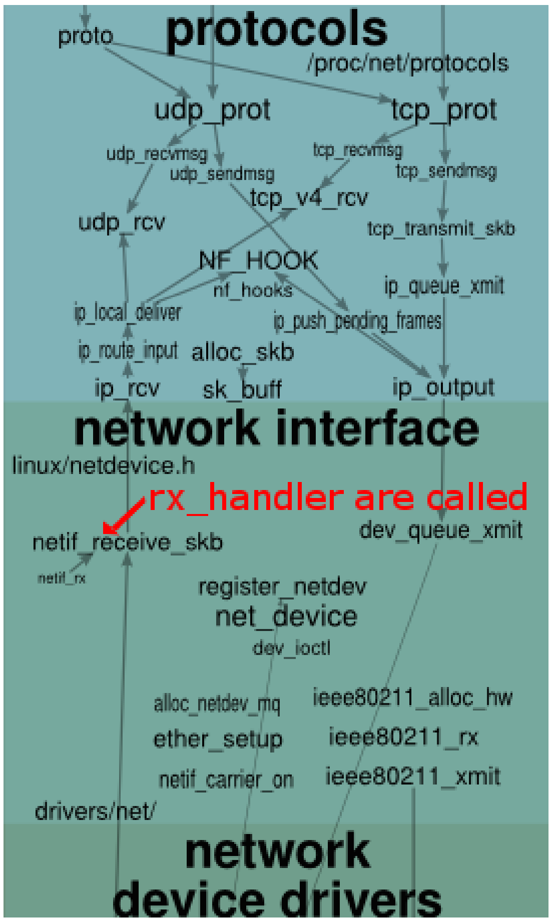
Changing the behavior happening when a packet is forwarded towards the `NORMAL` port thus consists mainly in altering the kernel module. In order to grasp what has to be done, it is useful to first understand how the switch, and more generally the Linux networking stack, operates.

First, the user will create a virtual bridge via ovs-vswitchd. From there, he will add interfaces to that bridge, either logical or physical. When an interface is added to an Open vSwitch bridge, the bridge instance in the kernel module will register itself as the receiving handler (`rx_handler`) of the added interface.

When data arrives on an interface, the driver handling it will read it and place the data in a socket buffer representing the packet content, before triggering a software interruption to notify the kernel that it has received data. At that point, the receiving handlers associated to that interface are called first, receiving the full packet data (thus including the complete Ethernet frame if this was an Ethernet port). In this case, this is the Open vSwitch kernel module, which can then process the packet, thus apply the rules from its flow table.

The relative location of the receiving handlers in the networking stack can be seen on the Fig. 10.3. The handlers can choose to either consume the packets (these do not go any further in the stack than the handlers), or to let the rest of the kernel process them. These handlers can be seen as hooks in the kernel which allow a developer to specify arbitrary behaviors as to how packets are handled, bypassing completely the networking stack or enhancing it (e.g. one could implement packet filters at that level protecting one interface).

For performance reasons, the flow table in the kernel module is very small, and has expiration timers set to typically 5s. If the kernel module hits a flow-miss, it then proceeds to pass the packet to the user-space daemon. This one will then perform a



Credits: `http://www.makelinux.net/kernel_map/`

Figure 10.3: An excerpt of the Linux kernel networking stack.

lookup in the complete OpenFlow flow tables. If it finds a matching flow, it installs it in the kernel module and notifies to reprocess the packet, otherwise it executes the OpenFlow flow-miss action, which by default instructs the switch to: (i) Register the packet in a packet; (ii) Send it to the controller as the payload of a `PACKET_IN` message; (iii) Possibly retrieve the packet from its buffer and send it according to the instructions of the controller.

10.2.2.1 Patching Open vSwitch `NORMAL` output action

By default, the Open vSwitch `rx_handler` always consumes the packets it receives. This means that the packets effectively never go passed that point in the kernel networking stack. This is completely logical with regard to the expected behavior of the switch, which is to either forward the packet or drop it. However, a third case is introduced in IBSDN. Indeed, IBSDN requires packets forwarded to the `NORMAL` port to be processed by components lower further up in the kernel networking stack. In this case, the desired component to reach is the IP stack, which is located way higher, in the protocol segment of the networking stack whereas Open vSwitch lies in the network interface segment.

Changing the behavior of Open vSwitch when hitting a `NORMAL` port would then be to return the `RX_HANDLER_PASS` special value, in order to let the packet be processed by he rest of the kernel. This change is at the core of a patch proposed in December 2013 by Chris Luke, which, among other things, adds a custom OpenFlow action to explicitly send packets back to the kernel and allows the user to redefine the `NORMAL` action to that behavior. While the patch itself never made it to the main tree, it is publicly available at `http://patchwork.openvswitch.org/patch/2651/`.

This patch was released for Open vSwitch pre-1.9. However, due to the fact that IBSDN requires OpenFlow 1.1+, and that its support is experimental with older version of OVS (and still is with newer one), a few adjustments are needed to apply it on newer version of Open vSwitch. The version used by the testbed is available there: `https://github.com/oliviertilmans/openvswitch`, where all patching conflicts have been cleaned up.

10.2.3 Implementing an IBSDN controller

The implementation makes use of the Ryu [72] controller framework. It is written in Python, thus allowing for fast prototyping, and supports OpenFlow 1.1+, which is a requirement to use the fast failover group type as required by IBSDN.

In this implementation, the IBSDN controller is specifically made to perform the evaluations whose results are detailed in Chap. 11. However, it still features the key parts common to every IBSDN controller, regardless of their goal.

An IBSDN controller has 4 mandatory responsibilities, detailed in the following subsections.

10.2.4 Setting up static flow entries

When a node connects to the controller after it has booted, the controller first installs static flow entries on the node ensuring that it behaves properly. The minimal set of flows is given in Table 10.1 and simply ensures the presence of the IGP-path control rule as well as allowing the IGP links between the local agents to be established.

Match	**Action**
eth_type=0x800, ip_proto=89	output:NORMAL
eth_type=0x800, tos=32	output:NORMAL

Table 10.1: Minimal set of flow entries for IBSDN.

In addition to these, the implemented controller also adds rules to:

1. Send ARP packet to the `NORMAL` port, easing up the usage of tools like `ping` without having the controller maintaining ARP tables. Indeed, this feature is already implemented on the nodes since they are running full Linux kernel;

2. Send LLDP packets back to the controller, as it has a built-in link discovery mechanism, allowing it to link the observed topology information to data it had about (e.g. linking the MAC addresses of the switch ports defining the boundaries of a link with the logical representation of that link it had in memory);

10.2.5 Setting up the fast failover groups

As discussed in Sec. 10.1.3, these groups implement the Next-hop control rules. As such, the controller has to setup one group per physical port. These have the following action buckets:

1. The simple `output:X` instruction, where X is the number of the port we want to protect. This bucket liveness status is defined by the liveness of the port X itself

2. First a `set_field:tos=32` instruction to tag the packet as IGP-forwarded, then an `output:NORMAL` instruction, which thus forward the packet according to the IGP backup path.

The implementation uses a slightly modified first action bucket.

Indeed, it precedes the output action by first updating the Ethernet header of the packet, thus changing the source and destination mac addresses to reflect the used link. Then, it adds an IP TTL decrease operation.

These additional actions ease up the visualization of the path followed by the packets while being forwarded by OpenFlow rules when capturing the traffic. They provide a strict ordering of the links taken by the packets (via its decreasing TTL), as well as allow to detect the direction in which the packet is going on the link (by checking the source and destination mac addresses).

10.2.6 Configuring the IGP

The controller sets up the IGP daemon by generating a configuration file for the BIRD daemon prior to the node startup. This file is copied on the node file-system, allowing the daemon parse it at startup time and then to start advertising its prefixes over its OSPF links.

This configuration generation process is more detailed in Sec. 10.3.

10.2.7 Installing flows matching the network policies

This part is specific to each controller, as it is related to the process of translating expected policies into OpenFlow rules on each switches.

IBSDN controller only have one difference with classical SDN controllers: every `output:X` action in the flows are replaced `group:X` actions, in order to protect them.

10.3 Emulating an IBSDN network

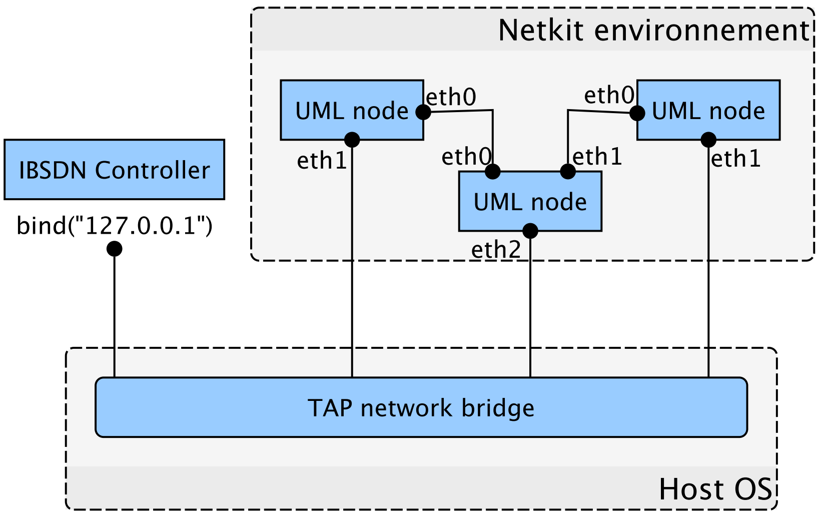


Figure 10.4: An high-level overview of the testbed architecture

In order to ease up the instantiation of the network topology, the testbed makes use of Netkit [73] which relies on the User Mode Linux virtualization technology. These Netkit nodes are Linux kernel compiled to be run as process, thus in user-space. The Netkit framework takes care of creating the virtual interfaces for each node, and bind them according to a configuration file specified at launch. Each node also has an interface that is bound to the TAP [74] Ethernet bridge of the host machine, allowing them to communicate with processes running on the host OS, in our case: the IBSDN controller. An alternative to Netkit could have been Mininet [75], which, instead of running complete kernel instances, uses lightweight process containers such as network namespaces.

The version of Netkit available for download however comes with an outdated kernel, not able to run the Open vSwitch kernel datapath. Furthermore, the default filesystem does not contain the required package required by an IBSDN node. Scripts are however available at `https://github.com/oliviertilmans/netkit` to setup a Netkit environment in which a recent kernel version is in use and where all the required packages (such as BIRD) to run an IBSDN node on top of Netkit VM's are download and installed.

The last step is then to compile the patched Open vSwitch version introduced in Sec. 10.2.2.1, and to install in on the node file-system.

Spawning a testbed for a given input topology works in two phases.

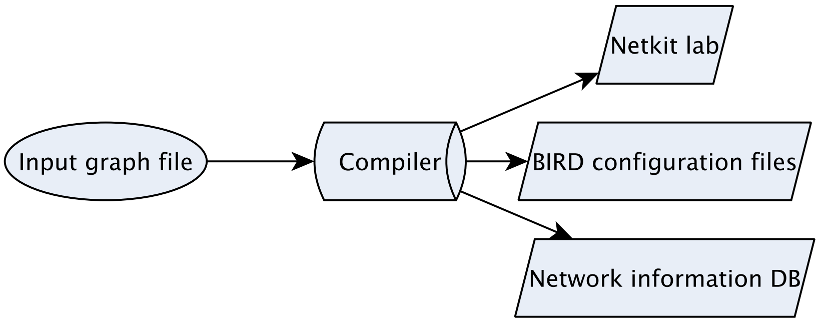


Figure 10.5: The compilation phase to create a testbed

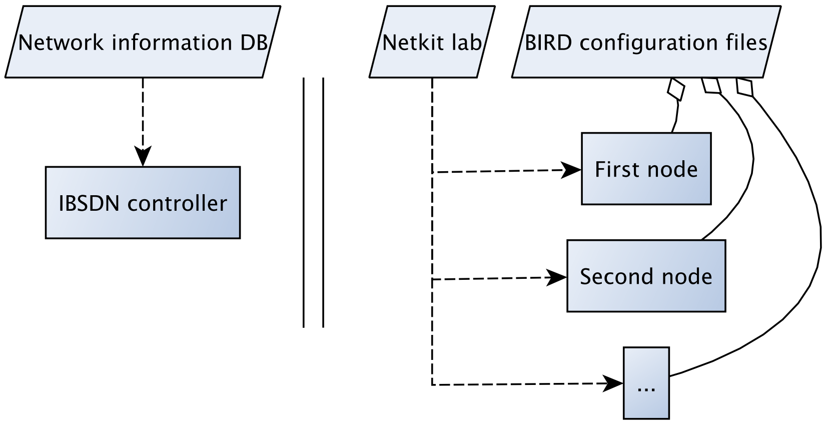


Figure 10.6: The spawning phase to create a testbed

The first phase, illustrated on Fig. 10.5 consists in creating the Netkit lab files, the configuration files for the BIRD daemon routing on each node, as well as an object registering various properties about the network which is called the Network Information Database.

It is based on work from the AutoNetkit [17] project, and integrated within the IBSDN controller as a `compiler` module. This lets the experimenter to specify the desired topology in a high-level fashion (e.g. a graphical graph). From there, the network compiler will proceed to create the OSPF topology as well as the IP allocation schemes, then will create the required files. As AutoNetkit did not support the BIRD daemon nor Open vSwitch, support to setup these two components was added to it.

Once these objects have been created, the testbed can be spawned. In order to do so, as visible on the Fig. 10.6, the user has to start the IBSDN controller on the host machine, feeding it the generated NIDB to let it know the expected topology of the network, such as the links between the nodes or the IP addresses. In parallel, the can spawn the Netkit lab, which will spawn all the nodes required by the topology as well as establishes the link as was mentioned in the Fig. 10.4.

When all nodes have finished their boot process, they connect to the IBSDN controller via the TAP bridge, thus completing the process of spawning a IBSDN, since the nodes are now handled by the controller.

In its current implementation, once the controller has noticed that all the switches are online, it will start running the micro-benchmark, which is presented in the next chapter.

Chapter 11

Evaluation

This chapter will present the evaluations performed on the implementation of an IB-SDN testbed present in Chap. 10.

This evaluation is split in three parts.

First, the performance of the prototype will be evaluated on small topologies, aiming at confirming the practical viability of the architecture, as well as its ability to avoid packet losses. In order to do so, the packet loss induced by failures in the data-plane will be measured.

Second, the post-failure patch stretch induced by using IBSDN will be evaluated on real-world topologies.

Third IBSDN results will be compared against related work in the SDN world, namely purely reactive or proactive techniques.

11.1 Benchmark description

11.1.1 Micro benchmark

The micro-benchmarks consisted in running the IBSDN prototype against simple topologies such as the one visible in Fig. 9.2, or the one on the Fig. 11.1. The two topologies model corner-cases for IBSDN, exhibiting different behavior in the events of failure.

In these topologies, flows were defined as the non shortest-path between between the source and the destination. Namely:

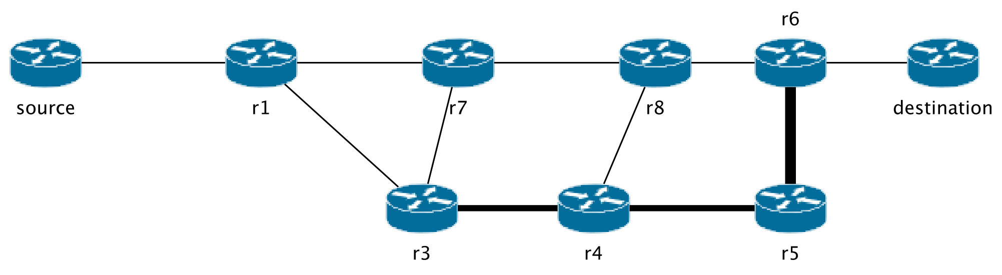


Figure 11.1: A simple topology used for the micro benchmark.

- On the Fig. 11.1, the flows were set up to follow the path: source → r1 → r2 → r4 → r5 → r6 → destination.
- One the Fig. 9.2, the used path was: Source → R1 → R2 → R4 → R5 → Destination.

Each experiment spawned a number of flows along those paths between 10 and 1000. In each of those, after a few secs, the last link of the primary SDN path was brought down, unless it caused a network partition. In the case of the topology on Fig. 11.1, this was the link r5–r6, whereas it was the link R4–R5 for the topology on Fig. 9.2.

Weights on the IGP topology were set to force the fact that the IGP had already converged and was using a path as disjoint as possible from the SDN one, even before the failure. For example, on the Fig. 11.1, the weights set are directly related to how thick the links are.

Failure of these links cause two heavily different response in IBSDN. In the case of Fig. 9.2, the flows would have to make a U-turn after reaching the router R4, looping all the way back to R1, thus causing a very high path stretch. On the opposite, on the other topology, the backup path would simply backtrack to r4 before taking the r4–r8 link, thus generating nearly no path stretch.

During the whole experiments, traffic matching the flows was sent in burst from the source towards the destination, at a rate of 10 or 20 bursts of packets per seconds per flow. The goal of the benchmark was to assess the probe loss rate occurring when using IBSDN to preserve the connectivity.

In order to track those probes, every node in the network was creating packet capture files allowing to dissect those after the experiments, in order to compute the overall results as well as verify that the paths taken by the packets where sensible and followed the theoretical model of IBSDN.

The result of this benchmark is explained in Sec. 11.2.1 and a comparison with the SDN reactive approach is made in Sec. 11.3.1

11.1.2 Macro benchmark

The macro benchmark attempted to measure the path stretch induced by IBSDN over real-world topologies. These topologies were gathered using the RocketFuel tool [76], and consist in weighted IGP topologies inferred from measurements. The results of these measurements are publicly available at `http://www.cs.washington.edu/research/projects/networking/www/rocketfuel`.

This benchmark makes the assumption that the preferred policy in case of failure is to re-route the failed paths in order to be as short as possible, and as a result all shortest-path computations take into account the weight set on the links in these topologies.

The first step of the benchmark was to compute the set of all disjoint paths in each network, thus the primary paths, between each pair of host, using the algorithm presented in Fig. 11.2. This algorithm takes as parameter the network graph, as well as a source node and a destination node. It then attempts to successively find the shortest

```
1: all_paths($G$, $V_1$, $V_2$)
2: paths $\leftarrow$ list()
3: while there exists a path $G$ between $V_1$ and $V_2$ do
4:   path $\leftarrow$ shortest_path($G$, $V_1$, $V_2$)
5:   for all edge in path do
6:     $G$.remove_edge(edge)
7:   end for
8:   paths.append(path)
9: end while
10: return paths
```

Figure 11.2: Finding all disjoint paths between a pair of hosts.

path between the two node, and then to remove all the link from the graph belonging to that path. This results in a set of path between the two hosts, which are all disjoints. Repeating this over all pairs of hosts in the network yields a set of disjoint paths over the whole the network. This is thus a reduced set of all the possible simple paths in a network, but is a sensible constraint to reduce the simulation space due to the fact that there can be up to n^2 paths in a graph and that some topologies have over 1000 nodes.

The next step is to compute all the backup path used in IBSDN, for each possible failures of these primary paths. These paths are the result of the start of the original path, up to the node adjacent to the failed element, to which the shortest-path between that adjacent node and the destination of the path is appended, after removing the failed link from the network graph.

Similarly, for each possible failure on each path, the IGP backup path can be computed as the shortest-path between the source and the destination, after removing the failed link from the network graph.

Finally, the path stretch induced by IBSDN for a particular failure for a particular path can be computed as the difference between the length of the IBSDN backup path and the length of the plain IGP backup path.

For example, the expected path stretch for the two failure scenarii presented in Sec. 11.1.1 are: (i) A path stretch of 3 for the failure of the link r5-r6 in the Fig. 11.1 (ii) A path stretch of 6 for the failure of the link R4-R5 in the Fig. 9.2

Knowing the importance of the path stretch matters because if this value gets too high, this increases the chances that IBSDN will cause congestion on some links.

The result of these simulation is presented in Sec. 11.2.2.

11.2 Benchmark results

11.2.1 Packet loss ratio

Since the IGP weights were set such that the IGP has already converged and had selected the backup path, the experiments reported nearly *no packet losses*, regardless of the tested topology. It is important to note that if that had not been the case, IGP convergence would have caused some packet losses. However it has also been shown in Sec. 7.3.1 that the convergence process can be quite fast using various techniques, even in larger networks.

Performing an evaluation where the IGP convergence process is triggered would be hard with the current prototype, as every node is sharing the same CPU. However, since they all have to recompute their SPT, which is a costly algorithm, this means that shared CPU would introduce an unquantifiable variance to the observed results.

The few packets that were lost during the experiment were due an unresolved bug between Open vSwitch and the User-Mode Linux VM's. Indeed, it has been observed that the first packet of a flow that is processed by the user-space daemon of Open vSwitch loses its Ethernet header when forwarded, which ultimately makes it be discarded at the next router. Indeed, a what is parsed as the Ethernet header of the packet on the receiving host is actually the first 22 bytes of the packet, thus the IP header and the start of the payload. Eventually resulting in the node parsing an incomplete IP header thus dropping the packet.

The analysis of the packet trace confirmed the expected paths taken by the packets.

For the Fig. 9.2, the observed paths were:

Primary path Source → R1 → R2 → R4 → R5 → Destination.

IBSDN backup path Source → R1 → R2 → **R4** → R2 → R1 → R6 → R5 → Destination.

For the Fig. 11.1, the observed paths were:

Primary path source → r1 → r2 → r4 → r5 → r6 → destination.

IBSDN backup path source → r1 → r2 → r4 → **r5** → r4 → r8 → r6 → destination.

11.2.2 Path stretch evaluation

The Fig. 11.3 shows a cumulative distribution function computed over the path stretch induced by each individual possible backup path. The graph plots one curve per RocketFuel topology. A particular point(X, Y) on the curve gives the percentage of backup paths (the Y value) that were experiencing a path stretch of at most X.

Negative path stretches result from the fact that the primary path was taking links with high IGP weights, but with less hops and that the backup path chosen by IBSDN were also using high weight links, thus resulting in overall shorter backup paths.

Two topologies define the boundaries of the results of this static evaluation:

(i) AS1221 exhibited the least overall path stretch induced by IBSDN. Indeed, about 55% of the backup paths had no path stretch at all. Finally, about 95% of the backup paths had a path stretch of at most 3 hops. The maximal value observed was a path stretch of 11;

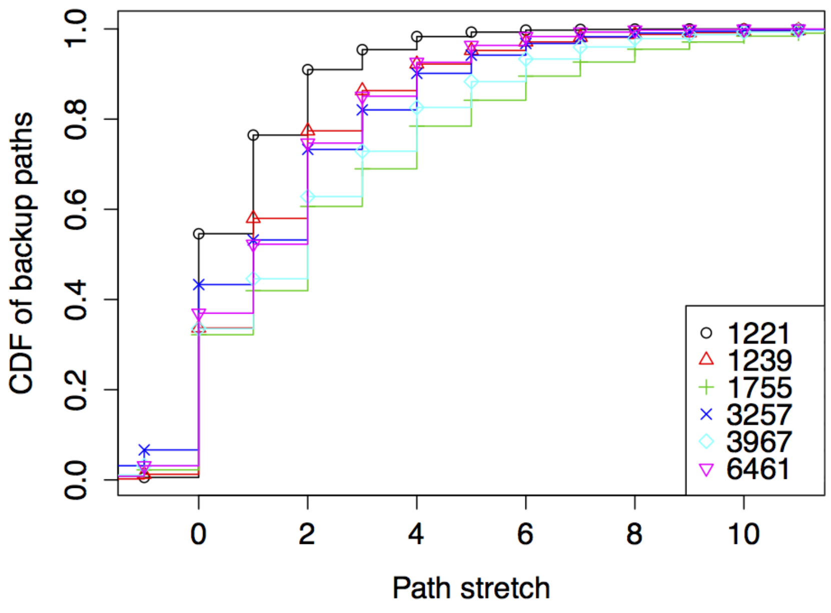


Figure 11.3:

(i) AS1755 on the contrary only had about 35% of its backup paths that were without path stretch, and about 95% of the backup paths had a path stretch of at most 8 hops. 1% of the paths had a path stretch greater than 10, and the maximal value observed was 21.

While some paths are not impacted at all by IBSDN, others suffer from a significant post-failure path stretch. The most likely cause is the appearance of traffic following a U-turn pattern, as shown in Fig. 9.3.

11.3 Comparison with pure SDN solutions

11.3.1 Comparison with a reactive SDN controller

When a failure occurs, a reactive SDN controller has be notified by a switch about it, or infer it itself via some kind of monitoring mechanism as covered in Sec. 8.1.1. Once it knows about the failure, it can then proceed to react to it, which implies computing the backup paths for all affected flows and then configuring all the switches to meet this new network state.

The testbed includes a primitive reactive controller, which attempts to reroute failed flows via their shortest-path. It does so in two steps:

1. For each each affected flow, it computes a backup flow by computing the shortest-path between its source and destination in a graph where the failed element is removed;

2. It then computes the minimal set of `FLOW_MOD` messages that have to be sent to recover that flow in the network and sends those.

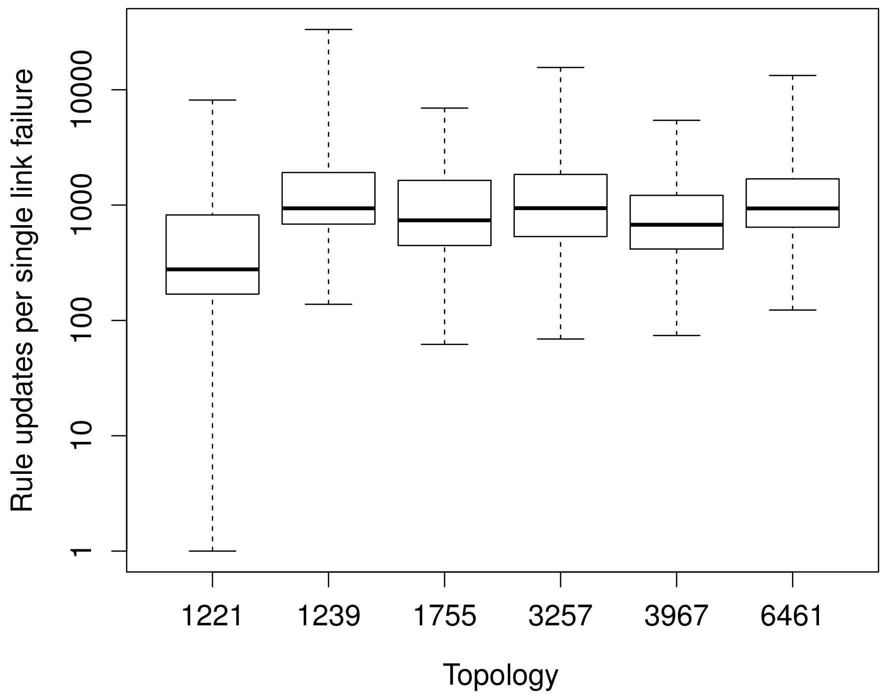


Figure 11.4: Number of messages to be sent by a reactive controller in case of a single link failure.

Two evaluations were performed between the IBSDN controller and that reactive one.

The first one evaluates the number of messages that have to be sent by the reactive controller when a single link failure happens. This was computed on the RocketFuel topology, using the same set of initial disjoint paths as presented in Sec. 11.1.2.

The results can be seen on the Fig. 11.4. Minimum and maximum values are represented by the whiskers, and the box represents the location of the values between the 25th percentile and the 75th. There is one boxplot per AS topology. The number of message to send is presented on a logarithmic scale.

Most median values were around 1000 messages, but some extrema are worth noting. First, the AS1239 has some failures which induce over 30k messages to be sent to repair the flows going through that link. These are very high values and induce a consequence load on the control-plane. Indeed, even if these messages are pre-computed and available in some form of cached memory, sending them to the switches generates a burst of control-plane messages. Furthermore, ensuring that all these messages do not induce transient effect, thus that the flow modification are atomic (e.g. [33]) requires the addition of some delays or additional control messages as well as an extremely good connection between the controller and the switches. Finally, computing those change on-the-fly could possibly overload the controller, as other asynchronous messages could come from other switches.

The second evaluation attempted to compare the traffic disruption caused by a failure in a network where a reactive SDN controller was in-use. The experiment consisted in running again the same benchmarks than in Sec. 11.1.1, this time observing the loss rate.

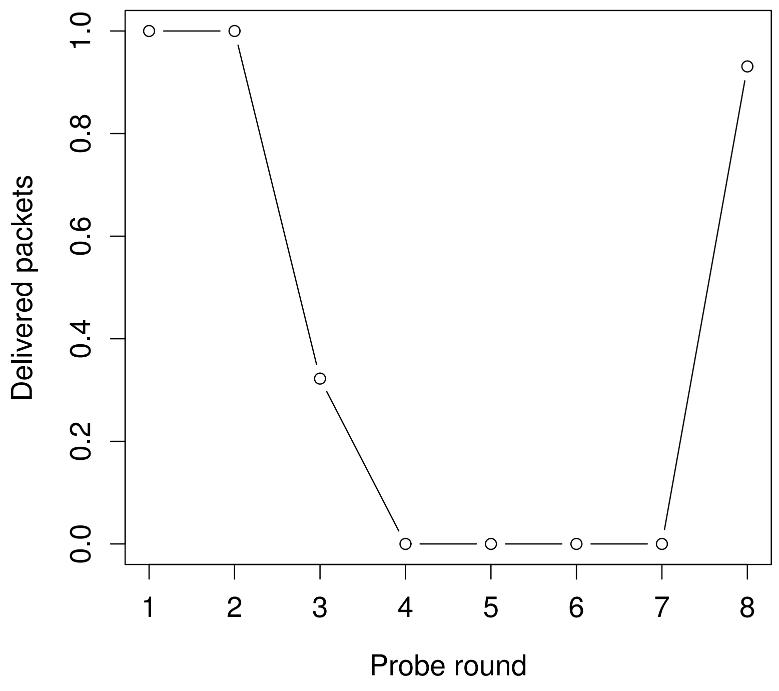


Figure 11.5: Loss rate observed with a reactive SDN controller

The Fig. 11.5 shows the most extreme variation of the transmission rate observed. It was observed on the topology shown on Fig. 11.1, when 1000 flows were instantiated and were probes waves were sent every 1/20th of a second. It plots the variation of the transmission rate over time (the probe waves).

As the link failure happened during the transmission of a probe wave, the 3rd of probes was only partially transmitted. However, the next 4 waves of probes were completely lost due the fact that the controller had to react to the failure.

11.3.2 Comparison with a proactive SDN controller

Conceptually, both the IBSDN controller and a proactive SDN controller have the play role regarding failures: they do not play any.

The difference lies in the way the switches store the data to be able to react autonomously to it. Whereas the IBSDN nodes use an IGP RIB, switches in a proactive SDN are required to store in the FIB both the primary flows as well as the backup flows protecting the established flows in the network.

Fig. 11.6 shows an evaluation performed on the RocketFuel topologies of that memory overhead introduced by proactive approaches. It plots the number of entries on the switches for the two topologies which produced extreme cases, both for the case when only the primary paths are setup, as well as the case were backup paths are also preemptively installed in the FIB.

The paths computation was performed as presented in Sec. 11.1.2, and the backup paths only protect against *single* link failures. It is important to understand that these entries represent only paths. Indeed, if one wanted to account for the possible matching clauses distinguishing flows, he would have to multiply by a certain factor the number of primary paths stored. However, depending on the strategy used to compute the backup paths (similarly to the 1:1 or 1:n approaches with MPLS backup

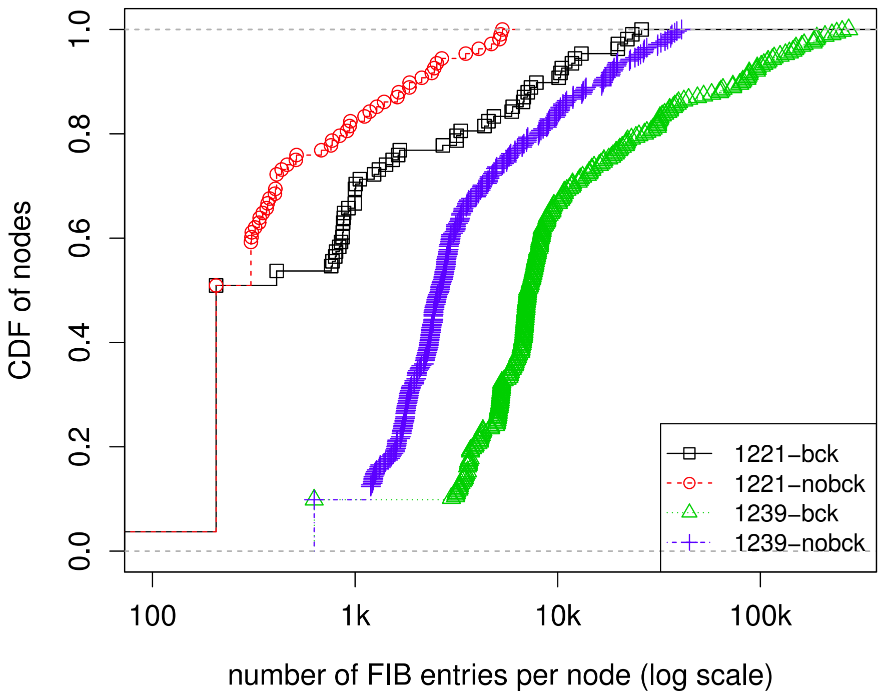


Figure 11.6: CDF of FIB entries on the switches, with or without backup paths.

tunnels presented in Sec. 7.3.2) the number of backup path is not always dependent on that factor. As a result, this figure shows the lower bounds of FIB entries.

The CDF's whose names finish in *-bck are the CDF of the FIB entries when switches store both primary paths as well as backup paths, which the CDF's ending in *-nobck show the case where only the primary paths are setup in the switches.

The CDF's exhibits a non negligible increase in the number of FIB entries stored on each switches when pre-preemptively installing the backup paths.

11.3.3 Benchmark conclusion

These evaluations confirmed that IBSDN presents interesting properties to react to failures.

Indeed, it was shown to be lossless if its underlying IGP was already converged, unlike the reactive approach which already exhibits losses on a small topology without any delay on the links. IGP convergence is known to be the source of packet losses, however techniques exist to make it under 50ms, even in very large network.

IBSDN is as robust by design as the reactive approaches against simultaneous failures of multiple elements, however, whereas the reactive cannot cope with the failure of the controller-to-switch connection, IBSDN provides a way to maintain connectivity while that happens.

Finally, IBSDN does not suffer from the FIB overhead induced by proactive SDN approaches.

The benchmark however revealed that IBSDN could induce non negligible post-failure path stretches. These are likely to be caused by local U-turn traffic, and such further work could add behaviors in the local agents to detects this and reduces its effect.

For example, local agents could analyze the IGP-forwarded packets to detect those U-turns. Then, they could cause the eviction of the flow responsible, directly sending the packet of that flow on the IGP control-plane in order to cut the U-turn.

Chapter 12

Conclusion

By design, SDN and traditional networks have different strengths and weaknesses. On one hand, traditional networks benefit from an inherent resiliency, while on the other hand SDN offers programmability in the control-plane, allowing network operators to express arbitrary policies on how hey want the network to behave

This master's thesis presented IBSDN, an hybrid network architecture attempting to use both network paradigm for the task they are best-suited for. It advocates for the separation of concerns when writing control-plane softwares, by splitting the management of the control-plane in two distinct tasks:

1. A long-term optimization problem, tackled by a pure SDN controller, to solve problems such as globally optimal traffic engineering;

2. A short-term feasibility problem, where local agents running on the nodes run an IGP in order to preserve connectivity in the events of failures.

In that context, a theoretical model of IBSDN network was presented, describing all required components and the interactions between them to behave properly. The model was then proved to be safe, providing maximal robustness independently from the network controller.

An implementation of a testbed was presented, describing how to implement all mechanisms required by IBSDN in order to behave properly.

OpenFlow was used as SDN technology allowing the network operators to express fine-grained primary policies enforced network-wide. In parallel, the IBSDN controller sets up an OSPF control-plane flanking the SDN one, in order to act as backup in the evens of failures. Tight collaboration between the two control plane was achieved by the use of OpenFlow fast failover groups coupled with static flows ensuring that the nodes could infer which control-plane to use in order to forward an incoming packet.

Finally, the implementation was evaluated, in order to assess its feasibility. It was also compared against common practice in pure SDN networks, and was shown to outperform them on some key points.

Namely, IBSDN is able to provide resiliency against arbitrary failure scenarii as long as no network partition occur, without overloading the switches memory.

Improvements can be made. For example, IBSDN can induce some significant path stretch when failures occur, as the switching between control-plane can only happen at nodes adjacent to failures. Implementing a mechanism to progressively lower this path stretch would help to prevent congestion on some links.

Another issue is the lack of expressiveness offered by the backup control-plane. Indeed, as it uses an IGP to preserve the connectivity, it is not possible to preserve all network policies one could enforce via a pure SDN network. However, maintaining the connectivity in itself should be the primary concern for most cases. Finding the right trade-off between the dedicated protection offered by SDN backup flows, coupled with an IGP to preserve the rest of the traffic with a low footprint on the memory of the nodes as well as fast restoration time could prove to offer a near optimal solution.

As a final conclusion, this work has shown that while logically centralized control brings huge benefits from a management point of view, keeping some aspect of the control-plane distributed brings scalability and robustness, which are also great assets to have when managing a network. Balancing the two is likely to bring more benefits than focusing only on one paradigm.

Bibliography

[1] K. Lougheed and Y. Rekhter, "Border Gateway Protocol 3 (BGP-3)," RFC 1267 (Historic), Internet Engineering Task Force, Oct. 1991. [Online]. Available: http://www.ietf.org/rfc/rfc1267.txt

[2] J. Moy, "OSPF Version 2," RFC 2328 (INTERNET STANDARD), Internet Engineering Task Force, Apr. 1998, updated by RFCs 5709, 6549, 6845, 6860. [Online]. Available: http://www.ietf.org/rfc/rfc2328.txt

[3] O. Bonaventure, C. Filsfils, and P. Francois, "Achieving sub-50 milliseconds recovery upon bgp peering link failures," *Networking, IEEE/ACM Transactions on*, vol. 15, no. 5, pp. 1123–1135, 2007.

[4] S. Vissicchio, L. Vanbever, C. Pelsser, L. Cittadini, P. Francois, and O. Bonaventure, "Improving network agility with seamless bgp reconfigurations," *IEEE/ACM Trans. Netw.*, vol. 21, no. 3, pp. 990–1002, Jun. 2013. [Online]. Available: http://dx.doi.org/10.1109/TNET.2012.2217506

[5] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Can the production network be the testbed?" in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–6. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924969

[6] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. Gude, N. McKeown, and S. Shenker, "Rethinking enterprise network control," *IEEE/ACM Trans. Netw.*, vol. 17, no. 4, pp. 1270–1283, 2009.

[7] R. Wang, D. Butnariu, and J. Rexford, "Openflow-based server load balancing gone wild," in *Hot-ICE*, 2011.

[8] S. Vissicchio, L. Vanbever, and O. Bonaventure, "Opportunities and research challenges of hybrid software defined networks," *Comput. Commun. Rev.*, vol. 44, no. 2, Apr. 2014.

[9] Hurrican Electric Internet Services, "Bgp prefix report." [Online]. Available: http://bgp.he.net/report/prefixes

[10] E. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959. [Online]. Available: http://dx.doi.org/10.1007/BF01386390

[11] G. Malkin, "RIP Version 2," RFC 2453 (INTERNET STANDARD), Internet Engineering Task Force, Nov. 1998, updated by RFC 4822. [Online]. Available: http://www.ietf.org/rfc/rfc2453.txt

[12] L. Andersson and G. Swallow, "The Multiprotocol Label Switching (MPLS) Working Group decision on MPLS signaling protocols," RFC 3468 (Informational), Internet Engineering Task Force, Feb. 2003. [Online]. Available: http://www.ietf.org/rfc/rfc3468.txt

[13] L. Andersson, I. Minei, and B. Thomas, "LDP Specification," RFC 5036 (Draft Standard), Internet Engineering Task Force, Oct. 2007, updated by RFCs 6720, 6790. [Online]. Available: http://www.ietf.org/rfc/rfc5036.txt

[14] P. Srisuresh and P. Joseph, "OSPF-xTE: Experimental Extension to OSPF for Traffic Engineering," RFC 4973 (Experimental), Internet Engineering Task Force, Jul. 2007. [Online]. Available: http://www.ietf.org/rfc/rfc4973.txt

[15] R. Aggarwal, D. Papadimitriou, and S. Yasukawa, "Extensions to Resource Reservation Protocol - Traffic Engineering (RSVP-TE) for Point-to-Multipoint TE Label Switched Paths (LSPs)," RFC 4875 (Proposed Standard), Internet Engineering Task Force, May 2007, updated by RFC 6510. [Online]. Available: http://www.ietf.org/rfc/rfc4875.txt

[16] RIPE NCC, "Youtube hijacking: A ripe ncc ris case study." [Online]. Available: http://www.ripe.net/internet-coordination/news/industry-developments/youtube-hijacking-a-ripe-ncc-ris-case-study

[17] S. Knight, A. Jaboldinov, O. Maennel, I. Phillips, and M. Roughan, "Autonetkit: Simplifying large scale, open-source network experimentation," in *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '12. New York, NY, USA: ACM, 2012, pp. 97–98. [Online]. Available: http://doi.acm.org/10.1145/2342356.2342378

[18] N. Feamster, J. Rexford, and E. Zegura, "The road to sdn: An intellectual history of programmable networks," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, pp. 87–98, Apr. 2014. [Online]. Available: http://doi.acm.org/10.1145/2602204.2602219

[19] D. L. Tennenhouse and D. J. Wetherall, "Towards an active network architecture," *SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 5, pp. 81–94, Oct. 2007. [Online]. Available: http://doi.acm.org/10.1145/1290168.1290180

[20] K. L. Calvert, S. Bhattacharjee, E. Zegura, and J. Sterbenz, "Directions in active networks," *Comm. Mag.*, vol. 36, no. 10, pp. 72–78, Oct. 1998. [Online]. Available: http://dx.doi.org/10.1109/35.722139

[21] J. M. Smith and S. M. Nettles, "Active networking: One view of the past, present, and future," *Trans. Sys. Man Cyber Part C*, vol. 34, no. 1, pp. 4–18, Feb. 2004. [Online]. Available: http://dx.doi.org/10.1109/TSMCC.2003.818493

[22] L. Yang, R. Dantu, T. Anderson, and R. Gopal, "Forwarding and Control Element Separation (ForCES) Framework," RFC 3746 (Informational),

Internet Engineering Task Force, Apr. 2004. [Online]. Available: http://www.ietf.org/rfc/rfc3746.txt

[23] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe, "Design and implementation of a routing control platform," in *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, ser. NSDI'05. Berkeley, CA, USA: USENIX Association, 2005, pp. 15–28. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251203.1251205

[24] A. Kleen, "netlink - communication between kernel and user space (af_netlink)." [Online]. Available: http://man7.org/linux/man-pages/man7/netlink.7.html

[25] J. Kelly, W. Araujo, and K. Banerjee, "Rapid service creation using the junos sdk," in *Proceedings of the 2Nd ACM SIGCOMM Workshop on Programmable Routers for Extensible Services of Tomorrow*, ser. PRESTO '09. New York, NY, USA: ACM, 2009, pp. 7–12. [Online]. Available: http://doi.acm.org/10.1145/1592631.1592634

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

[27] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: Taking control of the enterprise," *SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 4, pp. 1–12, Aug. 2007. [Online]. Available: http://doi.acm.org/10.1145/1282427.1282382

[28] D. Gourlay, "Will openflow solve the financial crisis?" [Online]. Available: http://www.networkworld.com/community/blog/will-openflow-solve-financial-crisis

[29] Open Networking Foundation, "Openflow switch specification, version 1.4.0." [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf

[30] B. Owens, "Openflow switching performance: Not all tcam is created equal." [Online]. Available: http://packetpushers.net/openflow-switching-performance-not-all-tcam-is-created-equal/

[31] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: Scaling flow management for high-performance networks," *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 254–265, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2043164.2018466

[32] B. Heller, R. Sherwood, and N. McKeown, "The controller placement problem," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, ser. HotSDN '12. New York, NY, USA: ACM, 2012, pp. 7–12. [Online]. Available: http://doi.acm.org/10.1145/2342441.2342444

[33] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid, "Software transactional networking: Concurrent and consistent policy composition," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '13. New York, NY, USA: ACM, 2013, pp. 1–6. [Online]. Available: http://doi.acm.org/10.1145/2491185.2491200

[34] A. Voellmy, H. Kim, and N. Feamster, "Procera: A language for high-level reactive network control," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, ser. HotSDN '12. New York, NY, USA: ACM, 2012, pp. 43–48. [Online]. Available: http://doi.acm.org/10.1145/2342441.2342451

[35] N. Foster, A. Guha, M. Reitblatt, A. Story, M. Freedman, N. Katta, C. Monsanto, J. Reich, J. Rexford, C. Schlesinger, D. Walker, and R. Harrison, "Languages for software-defined networks," *Communications Magazine, IEEE*, vol. 51, no. 2, pp. 128–134, February 2013.

[36] 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*, ser. HotSDN '13. New York, NY, USA: ACM, 2013, pp. 159–160. [Online]. Available: http://doi.acm.org/10.1145/2491185.2491218

[37] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A nice way to test openflow applications," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=2228298.2228312

[38] M. Kuzniar, M. Canini, and D. Kostic, "Often testing openflow networks," in *Proceedings of the 2012 European Workshop on Software Defined Networking*, ser. EWSDN '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 54–60. [Online]. Available: http://dx.doi.org/10.1109/EWSDN.2012.21

[39] Cisco, "Sonet triggers." [Online]. Available: http://www.cisco.com/c/en/us/support/docs/optical/synchronous-optical-network-sonet/62622-sonetrig.html#topic1

[40] P. Francois and O. Bonaventure, "Avoiding transient loops during the convergence of link-state routing protocols," *Trans. on Netw.*, vol. 15, no. 6, pp. 1280–1932, 2007.

[41] M. Shand and S. Bryant, "IP Fast Reroute Framework," RFC 5714 (Informational), Internet Engineering Task Force, Jan. 2010. [Online]. Available: http://www.ietf.org/rfc/rfc5714.txt

[42] P. Francois and O. Bonaventure, "An evaluation of ip-based fast reroute techniques," in *Proceedings of the 2005 ACM Conference on Emerging Network Experiment and Technology*, ser. CoNEXT '05. New York, NY, USA: ACM, 2005, pp. 244–245. [Online]. Available: http://doi.acm.org/10.1145/1095921.1095962

[43] D. Thaler and C. Hopps, "Multipath Issues in Unicast and Multicast Next-Hop Selection," RFC 2991 (Informational), Internet Engineering Task Force, Nov. 2000. [Online]. Available: http://www.ietf.org/rfc/rfc2991.txt

[44] IEEE, "802.1qbp - equal cost multiple paths." [Online]. Available: http://www.ieee802.org/1/pages/802.1bp.html

[45] A. Atlas and A. Zinin, "Basic Specification for IP Fast Reroute: Loop-Free Alternates," RFC 5286 (Proposed Standard), Internet Engineering Task Force, Sep. 2008. [Online]. Available: http://www.ietf.org/rfc/rfc5286.txt

[46] A. K. Atlas, R. Torvi, G. Choudhury, C. Martin, B. Imhoff, and D. Fedyk, "U-turn alternates for ip/ldp fast-reroute." [Online]. Available: http://tools.ietf.org/html/draft-atlas-ip-local-protect-uturn-03

[47] S. Bryant, S. Previdi, and M. Shand, "Ip fast reroute using not-via addresses." [Online]. Available: https://tools.ietf.org/html/draft-ietf-rtgwg-ipfrr-notvia-addresses-09

[48] G. Iannaccone, C.-n. Chuah, R. Mortier, S. Bhattacharyya, and C. Diot, "Analysis of link failures in an ip backbone," in *Proceedings of the 2Nd ACM SIGCOMM Workshop on Internet Measurment*, ser. IMW '02. New York, NY, USA: ACM, 2002, pp. 237–242. [Online]. Available: http://doi.acm.org/10.1145/637201.637238

[49] N. Sprecher and A. Farrel, "MPLS Transport Profile (MPLS-TP) Survivability Framework," RFC 6372 (Informational), Internet Engineering Task Force, Sep. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6372.txt

[50] J.-P. Vasseur, M. Pickavet, and P. Demeester, *Network recovery: Protection and Restoration of Optical, SONET-SDH, IP, and MPLS.* Elsevier, 2004.

[51] R. Asthana, Y. N. Singh, and W. D. Grover, "p-cycles: An overview," *Commun. Surveys Tuts.*, vol. 12, no. 1, pp. 97–111, Jan. 2010. [Online]. Available: http://dx.doi.org/10.1109/SURV.2010.020110.00066

[52] D. Stamatelakis and W. Grover, "Ip layer restoration and network planning based on virtual protection cycles," *Selected Areas in Communications, IEEE Journal on*, vol. 18, no. 10, pp. 1938–1949, 2000.

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

[54] Y. Wang, H. Wang, A. Mahimkar, R. Alimi, Y. Zhang, L. Qiu, and Y. R. Yang, "R3: Resilient routing reconfiguration," *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 4, pp. 291–302, Aug. 2010. [Online]. Available: http://doi.acm.org/10.1145/1851275.1851218

[55] A. Kvalbein, A. Hansen, T. Cicic, S. Gjessing, and O. Lysne, "Fast ip network recovery using multiple routing configurations," in *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, 2006, pp. 1–11.

[56] 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 conference on Networked Systems Design and Implementation*, ser. nsdi'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 113–126. [Online]. Available: http://dl.acm.org/citation.cfm?id=2482626.2482639

[57] E. Gafni and D. Bertsekas, "Distributed algorithms for generating loop-free routes in networks with frequently changing topology," *Communications, IEEE Transactions on*, vol. 29, no. 1, pp. 11–18, 1981. [Online]. Available: http://dx.doi.org/10.1109/TCOM.1981.1094876

[58] "Ieee standard for local and metropolitan area networks - port-based network access control," *IEEE Std 802.1X-2010 (Revision of IEEE Std 802.1X-2004)*, pp. C1–205, 2010. [Online]. Available: http://dx.doi.org/10.1109/IEEESTD.2010.5409813

[59] J. Kempf, E. Bellagamba, A. Kern, D. Jocha, A. Takacs, and P. Skoldstrom, "Scalable fault management for openflow," in *Communications (ICC), 2012 IEEE International Conference on*, 2012, pp. 6606–6610. [Online]. Available: http://doi.acm.org/10.1109/ICC.2012.6364688

[60] U. C. Kozat, G. Liang, and K. Kokten, "On Diagnosis of Forwarding Plane via Static Forwarding Rules in Software Defined Networks," *ArXiv e-prints*, Aug. 2013. [Online]. Available: http://adsabs.harvard.edu/abs/2013arXiv1308.4465K

[61] D. Katz and D. Ward, "Bidirectional Forwarding Detection (BFD)," RFC 5880 (Proposed Standard), Internet Engineering Task Force, Jun. 2010. [Online]. Available: http://www.ietf.org/rfc/rfc5880.txt

[62] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "Openflow: Meeting carrier-grade recovery requirements," *Comput. Commun.*, vol. 36, no. 6, pp. 656–665, Mar. 2013. [Online]. Available: http://dx.doi.org/10.1016/j.comcom.2012.09.011

[63] H. Kim, M. Schlansker, J. R. Santos, J. Tourrilhes, Y. Turner, and N. Feamster, "Coronet: Fault tolerance for software defined networks," in *Proceedings of the 2012 20th IEEE International Conference on Network Protocols (ICNP)*, ser. ICNP '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 1–2. [Online]. Available: http://dx.doi.org/10.1109/ICNP.2012.6459938

[64] 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*, ser. HotSDN '13. New York, NY, USA: ACM, 2013, pp. 109–114. [Online]. Available: http://doi.acm.org/10.1145/2491185.2491187

[65] M. Suchara, D. Xu, R. Doverspike, D. Johnson, and J. Rexford, "Network architecture for joint failure recovery and traffic engineering," in *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, ser. SIGMETRICS '11. New York, NY, USA: ACM, 2011, pp. 97–108. [Online]. Available: http://doi.acm.org/10.1145/1993744.1993756

[66] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "Enabling fast failure recovery in openflow networks," in *2011 8th International Workshop on the Design of Reliable Communication Networks (DRCN 2011)*. IEEE, 2011, pp. 164–171. [Online]. Available: http://dx.doi.org/10.1109/DRCN.2011.6076899

[67] A. Tootoonchian and Y. Ganjali, "Hyperflow: A distributed control plane for openflow," in *Proceedings of the 2010 Internet Network Management Conference on Research on Enterprise Networking*, ser. INM/WREN'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 3–3. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863133.1863136

[68] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "Simple-fying middlebox policy enforcement using sdn," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13. New York, NY, USA: ACM, 2013, pp. 27–38. [Online]. Available: http://doi.acm.org/10.1145/2486001.2486022

[69] E. Baccelli and M. Townsley, "IP Addressing Model in Ad Hoc Networks," RFC 5889 (Informational), Internet Engineering Task Force, Sep. 2010. [Online]. Available: http://www.ietf.org/rfc/rfc5889.txt

[70] "Open vswitch, an open virtual switch." [Online]. Available: http://www.openvswitch.org

[71] CZ.NIC Labs, "The BIRD Internet Routing Daemon," available at http://bird.network.cz/.

[72] "Ryu sdn framework." http://osrg.github.io/ryu.

[73] M. Pizzonia and M. Rimondini, "Netkit: easy emulation of complex networks on inexpensive hardware." in *TRIDENTCOM*, M. P. de Leon, Ed. ICST, 2008, p. 7. [Online]. Available: http://dblp.uni-trier.de/db/conf/tridentcom/tridentcom2008.html#PizzoniaR08

[74] M. Krasnyansky, "Universal tun/tap device driver." [Online]. Available: https://www.kernel.org/pub/linux/kernel/people/marcelo/linux-2.4/Documentation/networking/tuntap.txt

[75] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, ser. Hotnets-IX. New York, NY, USA: ACM, 2010, pp. 19:1–19:6. [Online]. Available: http://doi.acm.org/10.1145/1868447.1868466

[76] N. Spring, R. Mahajan, and D. Wetherall, "Measuring ISP topologies with rocketfuel," in *SIGCOMM*, 2002.

[77] C. Huang, V. Sharma, K. Owens, and S. Makam, "Building reliable mpls networks using a path protection mechanism," *Communications Magazine, IEEE*, vol. 40, no. 3, pp. 156–162, 2002.

[78] M. Shand, S. Bryant, S. Previdi, C. Filsfils, P. Francois, and O. Bonaventure, "Framework for Loop-Free Convergence Using the Ordered Forwarding Information Base (oFIB) Approach," RFC 6976 (Informational), Internet Engineering Task Force, Jul. 2013. [Online]. Available: http://www.ietf.org/rfc/rfc6976.txt

[79] P. Francois, C. Filsfils, J. Evans, and O. Bonaventure, "Achieving sub-second igp convergence in large ip networks," *SIGCOMM Comput. Commun. Rev.*, vol. 35, no. 3, pp. 35–44, Jul. 2005. [Online]. Available: http://doi.acm.org/10.1145/1070873.1070877

[80] M. Gjoka, V. Ram, and X. Yang, "Evaluation of ip fast reroute proposals," in *Communication Systems Software and Middleware, 2007. COMSWARE 2007. 2nd International Conference on*, 2007, pp. 1–8.

[81] S. Rai, B. Mukherjee, and O. Deshpande, "Ip resilience within an autonomous system: current approaches, challenges, and future directions," *Communications Magazine, IEEE*, vol. 43, no. 10, pp. 142–149, 2005.