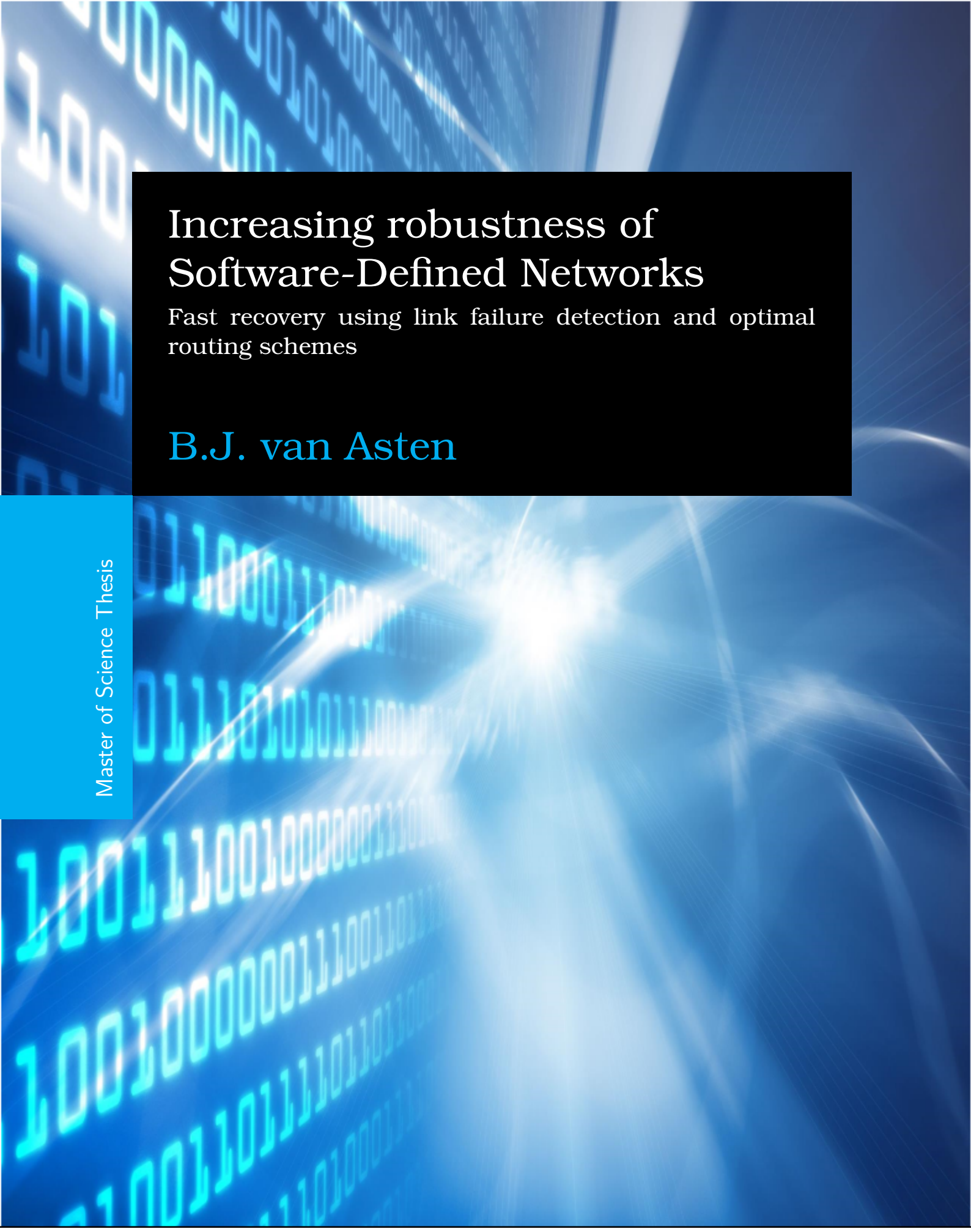


Increasing robustness of Software-Defined Networks

Fast recovery using link failure detection and optimal routing schemes

B.J. van Asten

Master of Science Thesis

Network Architectures and Services Group · Delft University of Technology

Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS)

. . .

Network Architectures and Services Group

Increasing robustness of Software-Defined Networks

Fast recovery using link failure detection and optimal routing schemes

B.J. van Asten
1279076

Committee members:

Supervisor: Dr. Ir. F.A. Kuipers
Member: Dr. R. Venkatesha Prasad
Member: Dr. Zaid Al-Ars
Mentor: Ir. N.L.M. Van Adrichem

September 12, 2014
M.Sc. Thesis No: PVM 2014-082

The work in this thesis was supported by the Ministry of Defense - Royal Netherlands Navy. Their cooperation is hereby gratefully acknowledged.

Abstract

Many real-time services, like voice and video conversations, have been adapted for transportation over Ethernet and Internet Protocol networks. The growth of these services resulted in over complicated and inflexible configurations at the control plane of routing and switching hardware, and high demands are set on the availability of the network infrastructure. The Software Defined Network paradigm is designed to abstract the available network resources and control these by an intelligent and centralized authority with the aim to optimize traffic flows in an flexible manner. Therewith the first problem is accounted for, but Ethernet IP infrastructures are not designed to meet carrier-grade and industrial network specifications on resiliency against failures. On these networks, failures must be detected and restored within millisecond order to not disturb provided network services. Currently, the OpenFlow protocol enables the SDN paradigm for Ethernet IP networks, but resiliency requirements can not be met without the application of purpose designed network infrastructures and protocols.

In this thesis, an overview on performed research is given to investigate possible enhancements and solutions to enable SDN as future network paradigm. Currently, beside robustness, problems exist on scalability and security with the application of SDN to current network infrastructures. On robustness, current research do not provide the necessary solutions to detect failures and activate protection schemes to failover to pre-configured backup paths within the set failover requirements. We will attempt to solve the problems to reduce the failover times on Ethernet IP networks with the application of active link monitoring and advanced capabilities of the OpenFlow protocol. To enable protection scheme, a routing algorithm is required that provides link-based protection. We propose a protection algorithm that guarantees protection, minimizes path cost on primary path and discovers protection paths for intermediate switches on the primary path with the main purpose to minimize failover times, optimize network traffic and reduce the need for crankback routing. *In short, we provide a complete solution to increase the robustness of Ethernet IP networks to the level of carrier-grade and industrial networks with the application of a link-based protection scheme and optimal routing algorithm, combined into a Software Designed Networking solution.*

Contents

List of Figures

List of Tables

Acknowledgments

At first I want to thank the Royal Netherlands Navy for providing me the opportunity and time to finish my masters at TU Delft. Second I want to thank the NAS group for support during master courses and execution of my thesis. A special thanks to Niels van Adrichem, who supported me during the complete thesis period and commented my (extensive) draft versions. I enjoyed our cooperation during experiments and writing of our paper *Fast Recovery in Software-Defined Networks.* Also a special thanks to Fernando Kuipers for commenting the draft version of my thesis and two papers. I would also thank my colleagues enrolled at TU Delft for cooperation and support. Last I want to thank my family, especially my wife Monique and daughter Ysabel, for supporting me during my study over the last two years.

Chapter 1

Introduction

1-1 Background

Since the adaptation of Ethernet and the Internet Protocol (IP) as network standards, many real-time services, like voice and video conversations, are adapted to be transported over IP networks. The low cost character and the ease of packet routing made operators to converge to IP networks. In order to transport and manage the enormous amounts of data in an efficient, robust and flexible manner, multiple networking protocols have been implemented in switching and routing network solutions. Generally, the switching and routing solutions can be split into a data and control plane. The data plane performs per-packet forwarding based on look-up tables located in the memory or buffer of the switch, where the control plane is used to define rules based on network policies to create the look-up tables. Due to the high demands on network performance and growing configuration complexity, the control plane has become over complicated, inflexible and difficult to manage. To solve this problem a new networking paradigm was needed, which was compatible with the widely used Ethernet switching and IP routing techniques.

The solution was found in virtualization techniques used in server applications, where an abstraction layer is positioned above the server hardware to allow multiple virtual machines to share the available resources of the server. Software Defined Networking (SDN) adopted this paradigm and introduced an abstraction layer in networking. By abstracting the network resources, the data and control plane are separated. The data plane is located at the switch hardware, where the optimized forwarding hardware is preserved and the control of the network is centralized into an intelligent authority with the aim to improve flexibility and manageability. A centralized authority provides the intelligence to instruct network switches to route and control the traffic through the network infrastructure. Optimal paths through the network can be provided by the central authority in advance or on demand. The current implementation of the SDN networking paradigm is found in the OpenFlow protocol [1] developed by Stanford University in 2008 and is currently under development with the Open Networking Foundation [2]. OpenFlow has attracted some big vendors in the networking community and became the most popular realization of the SDN networking paradigm.

For voice and video conversations high demand on latency exists, as a delay of 50 ms causes voice echo's. These echo's are experienced as inconvenient during conversations [3] and therefore high demands are set on the transportation network. If a failure in the network is recovered within the maximum accepted delay, the failure will not be noticed by users and applications. Before the convergence to Ethernet and IP networks, voice was transported over carrier-grade networks, which are designed for high availability and robustness. The physical layer of carrier-grade networks are designed to limit the delay and switch-over to pre-programmed backup paths after failure detection to 50 ms. Fast failure detection and failover mechanisms are implemented to switch over to pre-programmed backup paths. The researchers of Ghent University [4] have shown that Ethernet IP networks enabled with SDN can meet the failover requirements of carrier-grade networks. In industrial networks, expensive and complex systems exist to monitor and control automated processes. These systems and processes have even higher demands on availability and robustness. For time critical processes failover times of 20 ms [5] are required, where for machine control failover times below 10 ms [6] are necessary. The current industrial networks are build up from specialized network devices and routing protocols, such as the Resilient Packet Ring (RPR) protocol, to provide the necessary robustness. These routing protocols, however, require specific (dual) ring topologies and cannot be applied to each network topology.

1-2 Problem Description

Since the adaptation of SDN and OpenFlow in network infrastructures, much research has been performed on three different fields, being i) scalability and performance of the central authority in relation to the growth of network traffic and requests, ii) the robustness and resiliency of the network against link and switch failures in the network, but also failures of the central authority and iii) the application of known security, Quality of Service (QoS) and monitoring [7] mechanisms into the OpenFlow implementation. To make SDN a success candidate for robust and resilient industrial networks, two problems must be solved. The first problem is to reduce the failover time to industrial standards as the current SDN implementation of [4] does not meet these robustness standards. Application of standard network devices, without advanced failure detection and failover mechanisms, is encouraged to reduce complexity and costs. The second problem to resolve is the need for specific (ring) topologies and routing protocols. SDN has the possibilities to optimize traffic flows through the network, so the possible solution must optimize traffic flows through the network at all times without stringent requirements on the network. Both problems need to be resolved without compromising scalability and security issues in SDN.

1-3 Research Objective

The main research objective for this thesis is to show that SDN has the ability to provide robustness to IP networks with industrial standards on failure recovery. From this objective the following sub-objectives are derived:

1. Identify problems, solutions and enhancements with SDN and OpenFlow networking;
2. Identify failure detection and failover mechanisms which are available and can be applied to OpenFlow enabled copper Ethernet IP networks;
3. Implement failure detection and failover mechanisms to measure failover and recovery times in OpenFlow enabled networks with regular topologies;
4. Identify existing routing algorithms that provide solutions for path protection network failures;
5. When no protocol exists, develop a routing algorithm that provides path protection, optimizes traffic flow over the network and can be implemented to the central authority;
6. Show improvements of the protection algorithm in relation to existing mechanisms that provide path protection.

1-4 Research Questions

From the research objectives the main research question can be deduced: *Which operations are required to meet the industrial requirements on robustness with the application of SDN, while preserving optimal traffic flows over a regular network topology?* From this main question the research questions for this thesis are derived as:

1. Which frailties, solutions and enhancements come with the application of SDN to networks?
2. How are network failures detected on networks and how can detection times be minimized?
3. What recovery times are possible with existing protocols and *standard* network devices on regular networks?
4. Which routing algorithms exist that provide multiple paths over the network, while pursuing optimal traffic flow?
5. What is the performance gain of the protection algorithm in comparison to existing algorithms?

1-5 Thesis Structure

This thesis will answer the proposed research questions and is build up as follows. Chapter 2 discusses the SDN network paradigm, gives a review on the limitations and drawbacks of current data networks and shows how SDN networks can be beneficial. Also an insight of the OpenFlow protocol with its capabilities is given. Numerous solutions have been proposed to increase the scalability, performance, resiliency and security in SDN networks. Chapter 3 gives an overview of performed research on SDN, where a general approach is used to show the diversity of solutions to solve particular problems within the SDN and OpenFlow approach. In chapter 4 the basics of network survivability and failure detection are discussed, where after these basics are translated to OpenFlow implementations and a proposal to reduce the recovery times is given. Existing routing algorithms for path protection schemes and the development for a protection algorithm will be discussed in chapter 5. In chapter 6 simulations on the protection algorithm are performed to show the gain in comparison to existing path protection schemes. Experiments on different network topologies are performed on our proposal to reduce recovery times in chapter 7. Chapter 8 will conclude this thesis and proposes future work.

Chapter 2

Software Defined Networking

In this chapter a brief introduction to Software Defined Networking is given. First, a description is given on current state switching techniques in section 2-1, where section 2-2 abstracts these concepts to an SDN philosophy. Section 2-3 gives insight in different SDN applications for networks, where section 2-4 gives a description of the popular OpenFlow protocol, which enables SDN concepts in Ethernet networks. Because SDN is a new networking concept and not all (managed) switches support OpenFlow or equivalent protocols, a software solution is needed for experimental testing with OpenFlow. Open vSwitch is an open-source virtual switch implementation used for OpenFlow experiments and measurements. A description of Open vSwitch is given in 2-6, where section 2-7 concludes this chapter by reviewing SDN with respect to enhancements to current state networking.

2-1 Current state in Ethernet switching

For this thesis the main objective is focused on switching Ethernet networks transporting Internet Protocol (IP [8]) data packets. The main purpose for Ethernet switches is to interconnect nodes to a local network, with the ability the exchange data packets. From the Open Systems Interconnection (OSI [9]) reference model, Ethernet switches are located at the bottom two layers, the physical and datalink layer. Ethernet is developed as a broadcasting mechanism to sense the broadcast channel and to transmit when the channel is not occupied. On the physical level connectivity can, among others, be provided by 1000BASE-T twisted pair copper cables for local data distribution at 1 Gbps with a maximal range of 100 meters. To increase transmission range, optical transmission lines can be applied in the form of 1000BASE-ZX single-mode fibers. Connectivity is not limited to physical transmission lines. In WiFi networks, Ethernet is also adopted as standard, so we can conclude that Ethernet can be utilized on a large variety of transmission mediums, which makes it the most popular protocol in current data networks. Ethernet, described in the IEEE 802.3 standard [10], fulfills the functionality of the datalink layer and is developed to transmit data packets. Without an addressing scheme, data packets would not arrive at the destination, assuming that the

source and destination are not directly connected. Ethernet provides an addressing scheme using the Media Access Control (MAC) protocol. Before describing the switching functions and workings, a typical layout for Ethernet switches is given in figure 2-1.

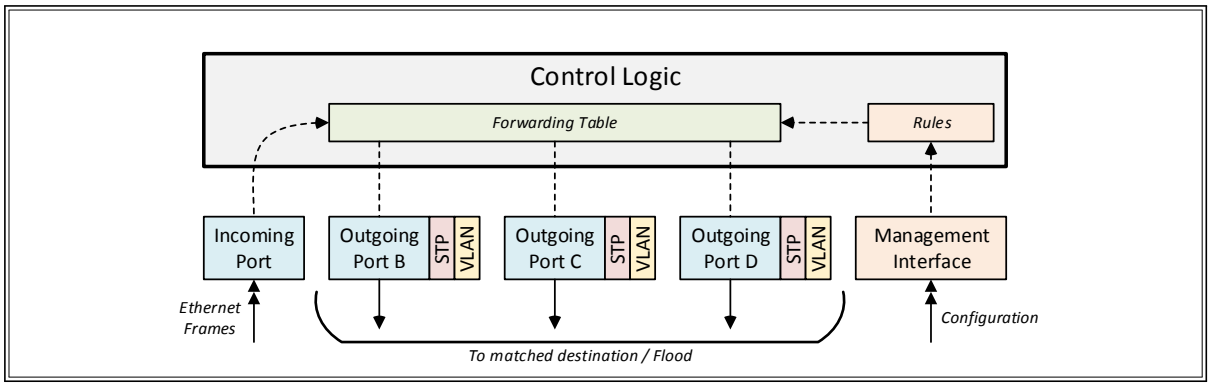


Figure 2-1: Current state of (managed) Ethernet switches - *The switch is build up from multiple Ethernet ports and a control logic. At the control logic the switch functions are performed and a forwarding table is present. The forwarding table contains MAC addresses of Ethernet nodes monitored at each port. Incoming traffic is matched at the forwarding table and forwarded to the corresponding outgoing port. When no match is found, the Ethernet frame is flooded over all outgoing ports. In more advanced and managed switches a management interface is present for configuration purposes.*

An Ethernet switch is built up from two planes. On the physical plane, multiple in- and outgoing ports are available. These port are controlled and configured by the control logic, containing the forwarding logic for the switch. The important part from the control logic is the Forwarding Table, which contains a list of MAC addresses coupled to the corresponding port. From incoming traffic the destination address is compared to the Forwarding Table and on a positive match, traffic is forwarded to the correct outgoing port. When no match is found, the packet is flooded over all ports, except the incoming port. From here, the four main functions of an Ethernet switch are described in more detail:

- *Learn MAC addresses* - The first function of an Ethernet switch is to learn MAC addresses from incoming traffic. Ethernet frames contain both the destination and source addresses. When the source address of incoming traffic is not present in the Forwarding Table, it is added to the table with corresponding incoming port;

- *Forward data packets* - The main function for a switch is to forward traffic to the correct outgoing port on a positive match at the Forwarding Table. On a negative match, the packet is flooded over all outgoing ports, with the aim to reach the destination. From this, we can learn that a switch only acts locally and switches do not have global information to optimize traffic flows;

- *Avoiding loops* - On a negative match, traffic is flooded over the network and will reach the destination. When loops exist in the network, the process of flooding continues, as forwarding tables of some switches in the loop will not be updated. The reverse path does not transverse all switches in the loop, so the non-updated switches will continue to flood unmatched data packets, leading to overloaded networks. To prevent switching

loops, more advanced and managed switches are often enhanced with the Spanning Tree Protocol (STP), standardized in IEEE 802.1D [11]. The protocol creates a spanning tree within a network, disabling ports whose outgoing link is not a part of the tree. Using this mechanism a single path between all nodes in the network exists without switching loops and the network is protected from errors introduced by loops;

- *Provide separation* - Networks are used by multiple groups and users. To separate these groups and create individual networks, one can install more switches and create multiple networks. Another way of separating data traffic is the application of the Virtual Local Area Network (VLAN) standard [12]. Ports of the switch can be assigned to a virtual network indicated with a VLAN indicator. Traffic from different virtual networks is separated, as forwarding is only possible when switch ports are assigned the same VLAN indicator. In this manner, a physical switch can be applied for multiple groups, removing the need for additional switches, assuming the network security policy allows VLAN applications.

With the switching functions described above, data packets can be forwarded over the network at high speed, without creating switching loops in the network. More advanced configurations on traffic separation are possible by the application of VLAN's. Beside these basic functions, a trend is noticed to manage different data streams over the network. Therefore, more protocols, such as the Simple Network Management Protocol (SNMP) [13] for management purposes, and standardizations are implemented at the switch control logic, which are often referred to layer-3 switches. These switches operate at the traditional layer-2, but also provide routing functionality on layer-3. Advanced switching functions, like STP and VLAN, require configuration at local switch level, resulting in a large number of complex configurations for large networks. Path selection on and routing over multi-layered networks, build up from multiple physical different topologies, is complex in current state networking [14]. On manageability level, local switch configurations are unwanted and inflexible, because small errors can lead to an overall networking failure and quick adaptation to changed traffic characteristics, such as high link utilization, is difficult.

2-2 Abstracting the network

As presented in the previous section, switches are configured locally, due to which real-time global optimizations in the network are difficult to perform. Nowadays networks form the backbone for social communication and entertainment services. Providing these services over an inflexible current state switching network is leading to non-optimal network utilization and over-dimensioned networks, but also the requested service is not guaranteed. A better solution is found in the SDN philosophy, where the network topology is configured based on requests from network services and QoS solutions [15] can be implemented. Services request connectivity to a network and if the request can be fulfilled, paths through the topology are provided to the service for the requested amount of time. To embrace this philosophy, switch configurations must be performed centralized and an abstraction interface is needed to translate the current network state to the network services [2]. In figure2-2 the SDN concept is presented.

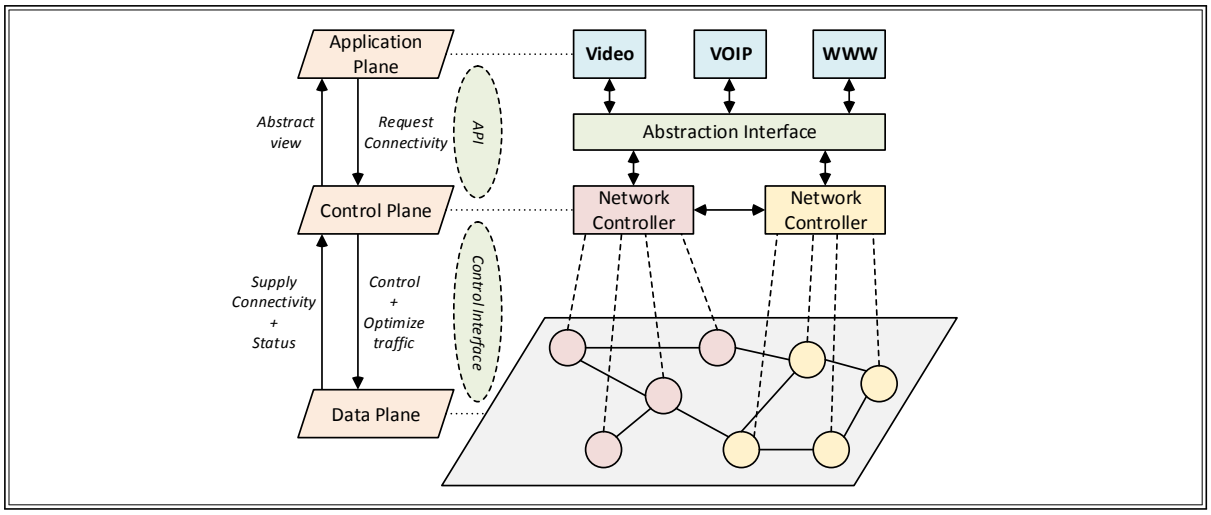


Figure 2-2: SDN concept of abstracting network view - *On the Data Plane network elements (switches) provide network connectivity and status to the Control Plane. The network elements are configured by Network Controllers via a Control Interface for global optimized configuration. An abstract view of the network is given to the Application Plane via a standardized interface. Network services request connectivity from the Network Controllers, where after the Network Elements are configured.*

The SDN concept speaks of three planes, which do not correspond directly with the OSI reference model. A short description of the planes is given below in relation with figure 2-2:

- *Data Plane* - The Data Plane is built up from Network Elements and provides connectivity. Network Elements consist of Ethernet switches, routers and firewalls, with the difference that the control logic does not make forwarding decisions autonomously on a local level. Configuration of the Network Elements is provided via the control interface with the Control Plane. To optimize network configuration, status updates from the elements are sent to a Network Controller;

- *Control Plane* - Network Controllers configure the Network Elements with forwarding rules based on the requested performance from the applications and the network security policy. The controllers contain the forwarding logic, normally located at switches, but can be enhanced with additional routing logic. Combined with actual status information from the Data Plane, the Control Plane can compute optimized forwarding configurations. To the application layer, an abstract view from the network is generated and shared via a general Application Programming Interface (API). This abstract view does not contain details on individual links between elements, but enough information for the applications to request and maintain connectivity;

- *Application Plane* - Applications request connectivity between two end-nodes, based on delay, throughput and availability descriptors received in the abstract view from the Control Plane. The advantage over current state networks is the dynamic allocation of requests, as non-existing connectivity does not need processing at local switch level. Also applications can adapt service qualities based on received statistics. For example reduce the bandwidth for video streaming applications on high network utilization.

By centralizing the control logic not only the management of switches simplifies, changes in network (security) policies only need to be changed on the Network Controller, preventing reconfiguration of switches locally by hand. Switching functions, like forwarding and separation, remain untouched, so no degradation in functionality is required for implementing SDN to the data planes in Ethernet networks. Besides that, SDN offers the possibility to optimize traffic over the network, increase robustness and enhance security with availability of up-to-date information of the network. Where current state switches traditionally act on local level and utilize MAC addresses for forwarding, a central control logic acts on global level handling higher layer properties, such as IP on the network layer and TCP on the transport layer, for forwarding decisions. This lifts the capabilities of current state switches to the network, transport and higher OSI-layers. Existing routing protocols can be implemented at the central control logic, resulting in reduced needs for expensive network middle boxes, such as routers, gateways and firewalls in networks. Centralization comes with a cost, as decisions to flood or forward data packets are not made locally. Header information from data packets at the switches must be transmitted to the central control logic for processing and configuration computations, which introduces an unwanted additional delay in packet forwarding. To indicate differences between an SDN enabled switch and a current Ethernet switch, an SDN switch is illustrated in figure 2-3.

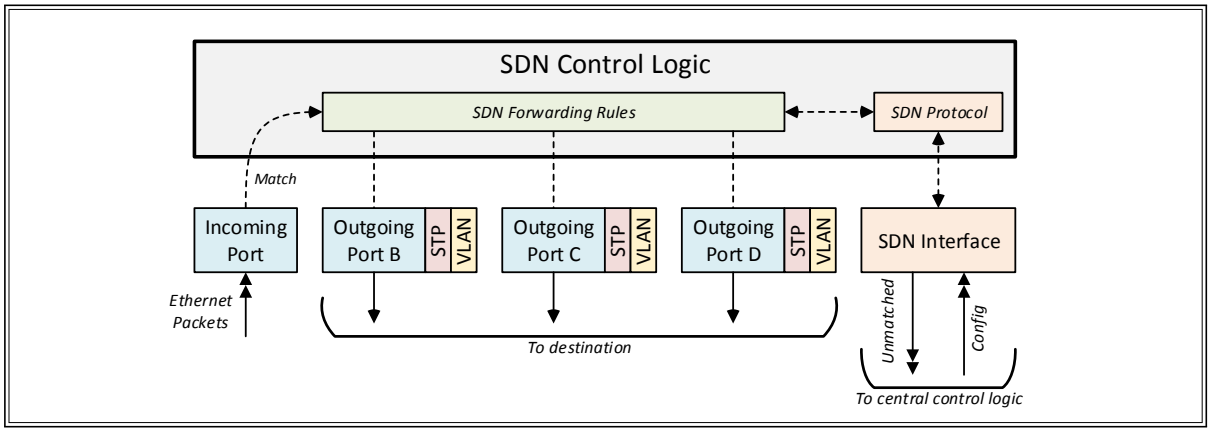


Figure 2-3: SDN enabled Ethernet switch - *Incoming traffic is matched to SDN Forwarding Rules and on positive matching traffic is forwarded as normal. Information of unmatched packets is send to the central control logic (Network Controllers), where new SDN forwarding rules are computed and configured at the SDN switches involved for transporting the data packets. The local control logic is enhanced with a configuration interface (SDN protocol) to communicate with Network Controllers.*

As seen in figure 2-3, the basic functionality of the switch remains untouched, as header information of incoming packets is matched and processed accordingly. Header information is matched to the configured SDN Forwarding Rules and on positive matching packets are forwarded to the configured outgoing port(s). Unmatched header information is sent to the central control logic via the SDN control interface. Thus, for communication between the SDN switch and the centralized controllers an additional protocol is needed. This protocol must contain the functionality to configure forwarding rules and ports, as well as collect and transmit switch status and statics to the central control logic.

2-3 Applications of SDN

To give an indication how SDN can be implemented in real-life, some examples from possible implementations will be discussed. In [16] four SDN applications are discussed. The first application is found in resource allocation (separation) and security policies on enterprise network environments where different network requirements are controllable with SDN technology, without the need for separate hardware. SDN can provide the functionality of classic middle boxes, making firewalls, load balancers and network access controllers obsolete. But the main improvement for enterprise networks is the ability to control and manage the network from a centralized and unified control authority. Configuration and security policy changes in the network do not have to be configured locally, but can be assigned globally to the network controller(s).

The second application identified is energy conserving techniques in data centers. Physical hardware needed for the emerging cloud computing services are housed in large data centers. Research in [17] has shown that 70% of all energy in data centers is consumed for cooling purposes, where the network infrastructure accounts for $10 - 20\%$ of the total energy cost. By smartly redirecting traffic over the network using SDN network controllers and advanced algorithms, switch load can be decreased and switched off for power saving. Results in [17] show energy savings between $25 - 62\%$, which means lower operating costs for data centers.

In previous examples and the explanation of the SDN concept, current state Ethernet switches are used for illustration, but the centralized control philosophy of SDN is also expandable to infrastructure based wireless access networks (WLAN). For WLAN it is common to enter password credentials and other security measures before access is gained to the network. The OpenRoads project [18], showed that the possibility to seamlessly roam over multiple wireless infrastructures, controlled by different providers, without changing and entering credentials. Users identify at the application plane, where after the centralized controllers configure the wireless infrastructure and provide handover between the different providers located at the control layer.

The fourth application for SDN is found in home and small business networks. Information security is a hot topic, as users demand more insight in the events on the network. Creating log files and implementing additional security modules to current state switches and routers is not an easy task and that is where SDN can fill the gap. Extended modules with user friendly interfaces can be installed on the application plane, which can secure the complete network from a single point of control. Logging modules can gain statistics from the network and new computer hardware is easily integrated into an existing network. For home users and small businesses, which often do not have affinity with computer networks, the management and control of the modules on the application plane can be outsourced to third party experts [19].

From the four applications in different environments, we have seen that SDN can improve networks. The SDN philosophy does not imply new network protocols and standards, but provides a framework wherewith existing network technology is deployed in a different manner. More effective deployment of network hardware will result in better management, less devices and increased efficiency. Unfortunately, a complete and integrated SDN solution does not yet exist. Much research has been performed on the control interface between the data and control plane, where a clear standard rises in the form of the OpenFlow protocol.

2-4 Description of OpenFlow protocol

Implementations of SDN can be found, among others, in OpenFlow [1] or ForCES [20]. More protocols exist, however OpenFlow is the most popular and it performs the control interface tasks listed in section 2-2. OpenFlow has attracted many researchers, organizations and foundations, so a wide collection of open-source software is available in the form of OpenFlow controllers (section 2-5), as well as physical and virtual switch implementations (section 2-6). The OpenFlow protocol describes and couples switching hardware to software configurations, such as incoming and outgoing ports, as well as an implementation of the SDN Forwarding Rules. In figure 2-4 a flow diagram is given of the matching process of an incoming packet in an OpenFlow switch enabled with protocol version 1.3. A detailed survey on the OpenFlow specification is given in [21] by Lara et al.

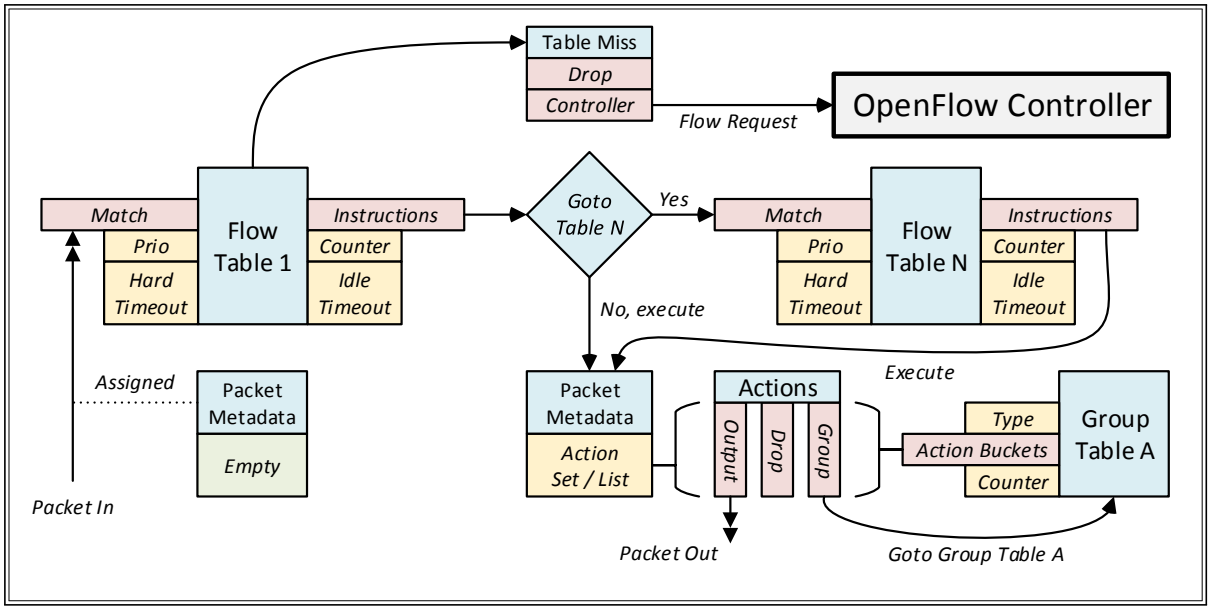


Figure 2-4: Flow diagram of OpenFlow protocol - *An incoming packet is assigned with Packet Metadata and is matched to Flow Rules in Flow Tables. Each Flow Rules contains instructions, which are added as Actions to the Packet Metadata. Instructions can include forwarding to other Flow Tables. If all Flow Tables are passed, the Actions in the Metadata are executed. Actions define the outcome for the packets, such as Output or Drop traffic. If a group of Flow Rules requires the same processing, Group Tables are applied. Group tables also contain Actions. When a packet does not match, a Table Miss is initiated and the packet can be forwarded to the OpenFlow controller for further processing or the packet is dropped.*

The SDN Forwarding Rules, called Flows in OpenFlow, are stored in one or more Flow Tables. For each incoming packet, a meta data set is created, containing an Action List, Action Set or both. In the Action List and Set actions are added for each Flow Table the packet transverses, where the Actions define the appropriate operations for the packet. Examples of Actions are forward (Output) the packet to port X, drop the packet, go to Group Table A or modify the packet header. The main difference between a List and Set is the time of execution. Actions added to a List are executed directly after leaving the current Flow Table, where the Actions defined in the Set are accumulated and executed when all Flow Tables are processed. Each Flow Table contain Flow Entries with six parameters [1]:

- *Match* - The criteria where the packets are matched against. Criteria include parameters of the datalink, network and transport layer of data packet headers and optionally meta data from previous tables. A selection of criteria used to match incoming packets is given in table 2-1;

Match Field	*Layer*	*Description*
Ingress Port	Physical	Incoming ports and interfaces
Ethernet Address	Datalink	Source and destination MAC-address
VLAN	Datalink	VLAN identity and priority
MPLS	Network	MPLS label and traffic class
IP	Network	IPv4 / IPv6 source and destination address
Transport	Transport	TCP/UPD, source and destination port

Table 2-1: Selection of fields for Flow Rules to match incoming packets [1]

- *Instructions* - When a packet matches, instructions are added to the meta data to direct the packet to another Flow Table or add Actions to the Action List or Set;
- *Priority* - The packet header can match to multiple Flow Entries, but the entry with highest priority determines the operations;
- *Counter* - Every time a packet has matched and is processed by a Flow Entry, a counter is updated. Counter statistics can be used by the OpenFlow controller, Application Plane to determine network policies or for network monitoring [7];
- *Hard Timeout* - A Flow Entry is added by an OpenFlow controller, where the maximum amount of time this entry may exist in the Flow Table before expiring is defined by the Hard Timeout. The Hard Timeout can be used to limit network access for a certain node in the network and automatic refreshing of the Flow Table to prevent large tables;
- *Idle Timeout* - The amount of time a Flow Entry is not matched is defined as the idle time. Idle Timeout defines the maximum idle time and is mainly used for refreshing Flow Tables.

From OpenFlow protocol version 1.1 and onwards, Group Tables have been defined. Group Tables allow more advanced configurations and consist of three parameters:

- *Action Buckets* - Each bucket is coupled to a switch port and contains a set of Actions to execute. The main difference with Instructions from the Flow Table is that Action Buckets can be coupled to counter and meter parameters. Based on values of these parameters a bucket is valid or not.
- *Type* - Defines the behavior and the number of Action Buckets in the Group Table. Multiple Action Buckets can be used for, i) multi-cast and broadcast applications, where the incoming packet is copied over the Action Buckets (multiple ports), ii) load sharing applications, where a selection mechanism selects the Action Bucket to execute and iii) failover applications, where from the available Action Buckets the first live is selected to execute. Assigning a single Action Bucket to a Group Table is useful for defining Actions for a large number of Flow Entries with the same required forwarding policy;

- *Counter* - The number of times the Group Table have been addressed.

With the descriptions of the Flow and Group Table we can follow the incoming packet from figure 2-4. At entry, a meta data set is assigned to the data packet and the packet header is matched to the Flow Entries in the first Flow Table. On match, instructions are added to the Action Set and the packet can be processed further. When the instructions include forwarding to next Flow Tables, the packet with meta data is processed in a similar way and instructions are added to the Set. When no forwarding to other Flow Tables is instructed, the Action Set from the meta data is executed. Actions from the Set and / or Group Table determine the process of the packet. In switching operation, the MAC address is matched in the first Flow Table and the Action Set define to output the packet on a specified outgoing port. When none of the Flow Entries match, a Table Miss is initiated. Depending on the configuration by the OpenFlow controller, the packet is dropped or transmitted to the controller for a Flow Request. At the controller, new Flow Entries are computed and added to Flow Tables of involved switches.

2-5 Description of OpenFlow Controller

OpenFlow controllers come in many variations and all share the same goal of controlling and configuring switches. In [16] a list of 15 controllers is given, all OpenFlow compliant. Differences are found in programming languages and support for different OpenFlow protocol versions. Popular implementations, like NOX [22] and the Open vSwitch (OVS)-controller from Open vSwitch [23] use the C/C++ language, where POX [24] and Ryu [25] are Python based controllers. Java based controllers are found in FloodLight [26] and OpenDayLight [27]. Only Ryu, OpenDayLight and unofficial ported versions from NOX support OpenFlow protocol version 1.3 so far. For more advanced configuration purposes and the use of Group Tables, we advise the Ryu and OpenDayLight controller. Both FloodLight and OpenDayLight offer web browser based configuration tools instead of command line interfaces and are therefore more user friendly. NOX, POX and Ryu share a similar building structure and this shared structure is used to give a generic overview of an OpenFlow controller in figure 2-5.

The example controller is built around a Core application, which acts as an backbone in the controller. To communicate with the OpenFlow switch, a translation module added to translate OpenFlow protocol messages to controller parameters. Other modules, like a layer-2 switch module (L2-switch) in for example Ryu, can advertise themselves to the Core application and register on specific switch parameters or events. The mentioned controllers supply the Core application with translation modules for OpenFlow protocol version 1.x. Depending on the requirements on the controllers, one can construct and program modules and advertise these to the controller. In figure 2-5 examples are given, such as a topology module, for maintaining an up-to-date status of the network infrastructure to feed routing modules for calculation of optimal and backup paths through the network. Also modules for redundancy purposes can be added, to synchronize information with other controllers at the control plane.

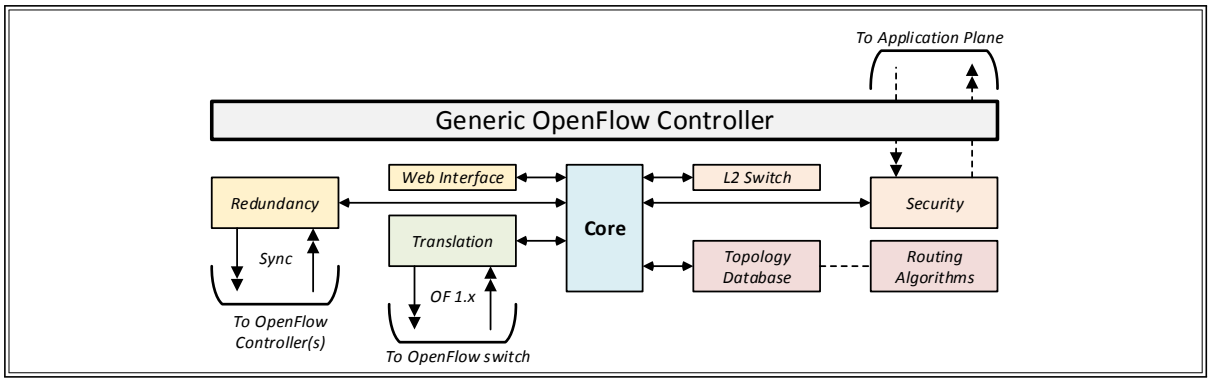


Figure 2-5: Generic description of an example OpenFlow Controller - *The controllers mentioned are built around a Core application, which act as a backbone in the controller. To communicate with OpenFlow switches, a translation module translates the OpenFlow protocol parameters to the "language" used inside the controller. Additional modules can advertise itself to the Core application to receive switch events. Based on the application, flow rules can be calculated or notification are send to the application plane.*

2-6 Description of Open vSwitch

Although Open vSwitch is not specifically designed to enable the SDN philosophy, it is widely used by researchers and organizations to test OpenFlow implementations and benefit from flexible SDN configurations. OVS can be configured to turn regular servers and computers with multiple physical network interface into a virtual OpenFlow switch, as shown in figure 2-6. Many Linux distributions, such as the Ubuntu OS, support OVS from their repositories, which make installation of the virtual switch an easy task.

Depending on the configuration, OVS can be configured as layer-2 switch (controlled by local OVS-controller) or as generic OpenFlow switch. Configuration can be supplied by an external OpenFlow controller or Flow Rules are supplied by hand via the command line interface. External plug-ins can also configure the OpenFlow module of OVS. An example of such an plug-in is the Quantum plug-in from OpenStack [28]. OpenStack dynamically assigns virtual Ethernet interfaces on Open vSwitch to virtual machines in the cloud. Migration of virtual machines over different servers in the network is enables due to automatic configuration of the OpenFlow module by the Quantum plug-in. The testbed used for experiments in Chapter 7 is constructed from multiple Linux Ubuntu servers configured with Open vSwitch to enable a network topology with OpenFlow capabilities.

2-7 Conclusion on Software Defined Networking

In this chapter the transition is described from current switching networks to SDN enabled networks, where the four main functions for switches are distributed among the switch data plane and controlled by a centralized logic. SDN decouples the control logic from the forwarding layer and centralizes network control. This enables better management for switches, as well as the possibility to optimize traffic flows through the network. The next step is to abstract the network view to the application plane, resulting in a framework for requesting

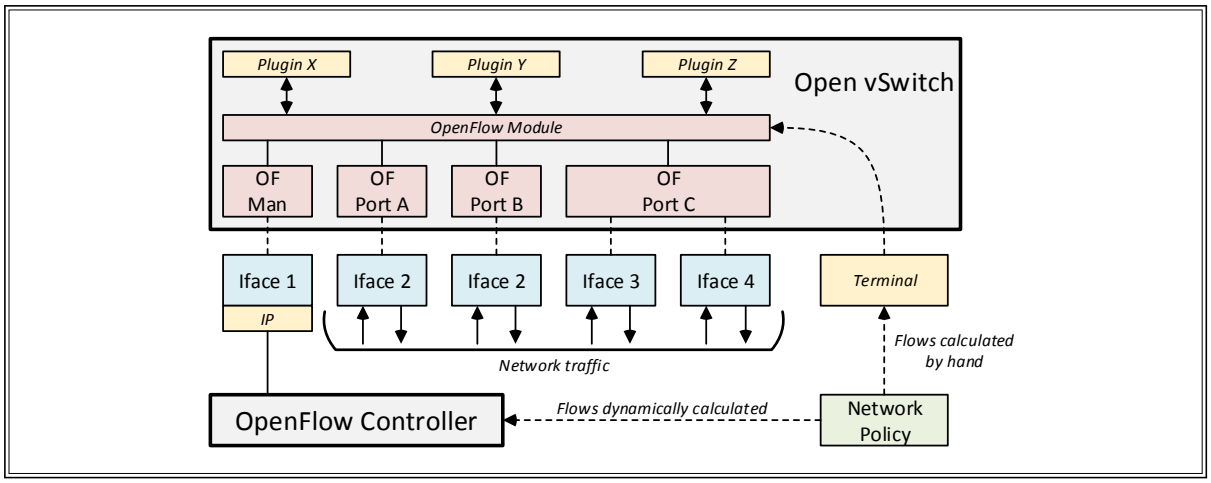


Figure 2-6: Description of Open vSwitch in relation to OpenFlow - *With Open vSwitch physical network interfaces from servers and computer can be assigned to an OpenFlow switch. To connect the virtual OpenFlow switch to a remote controller via an IP connection, a management interface (OF Man) with IP address is assigned. The other option to configure the OpenFlow module is via a command line interface. Other physical interfaces can be assigned directly to an OpenFlow port, where it is also possible to bind multiple physical interfaces to a single OpenFlow port. With other Open vSwitch plug-ins it is possible to control the OpenFlow module.*

connectivity by applications based on delay and throughput. With the application of SDN a switching network is transformed from an inflexible configuration to an adaptable ad-hoc network. All these benefits come with a price, as extra configuration interfaces are needed, that can introduce unwanted latency. Currently, OpenFlow and Open vSwitch provide the possibility to experiment with SDN. In the next chapter, an overview is given on research performed on SDN concepts between the infrastructure and control plane. We will show that SDN enables the possibility to enhance networking on layer-2 and lift switching networks to layer-3 and higher, but it is also shown that SDN contains frailties itself, that needs solving.

Chapter 3

Overview of SDN research

Since the introduction and the acceptance of OpenFlow as enabler of the SDN network paradigm, much research has been performed and published. Most of the research is performed to address and solve specific problem areas in software defined networking and, in some cases, networking in general. The main problem and research areas identified are scalability of control traffic in high performance data networks, the resiliency against network infrastructure failures and enabling security solutions on network level. To differentiate and compare the proposed solutions in the appointed research areas a general method is needed. Section 3-1 describes the developed framework, where sections 3-2 to 3-4 will discuss and review the proposed solutions in the identified research areas. This chapter concludes with section 3-5, where a short summary and the focus for the remaining part of this thesis is given.

3-1 General framework

In order to make a good comparison between proposed solutions, we developed a global SDN framework. Within the graphical framework multiple layers are defined, which indicates the hierarchy level of components within SDN networks (see Figure 3-1).

The identified layered structure of performed research on SDN and OpenFlow solutions is given below. For our graphical framework we define that controllers perform the computations and tasks to control traffic and additional layers can be added for administrative and synchronizing purposes.

- Level-0 - *Switch Layer* - The lowest layer identified in the OpenFlow structure is the switch layer, with the main purpose to deliver data plane functionality. Data plane functions are performed at the *Switch / Open vSwitch* sublayer, where the two additional sub-layers, being the *Additional Switch Layer* and *Local OpenFlow Controller*, add additional functionality to perform minor control plane tasks. Additional layers are added to enhance control capabilities at the switch layer in the network;

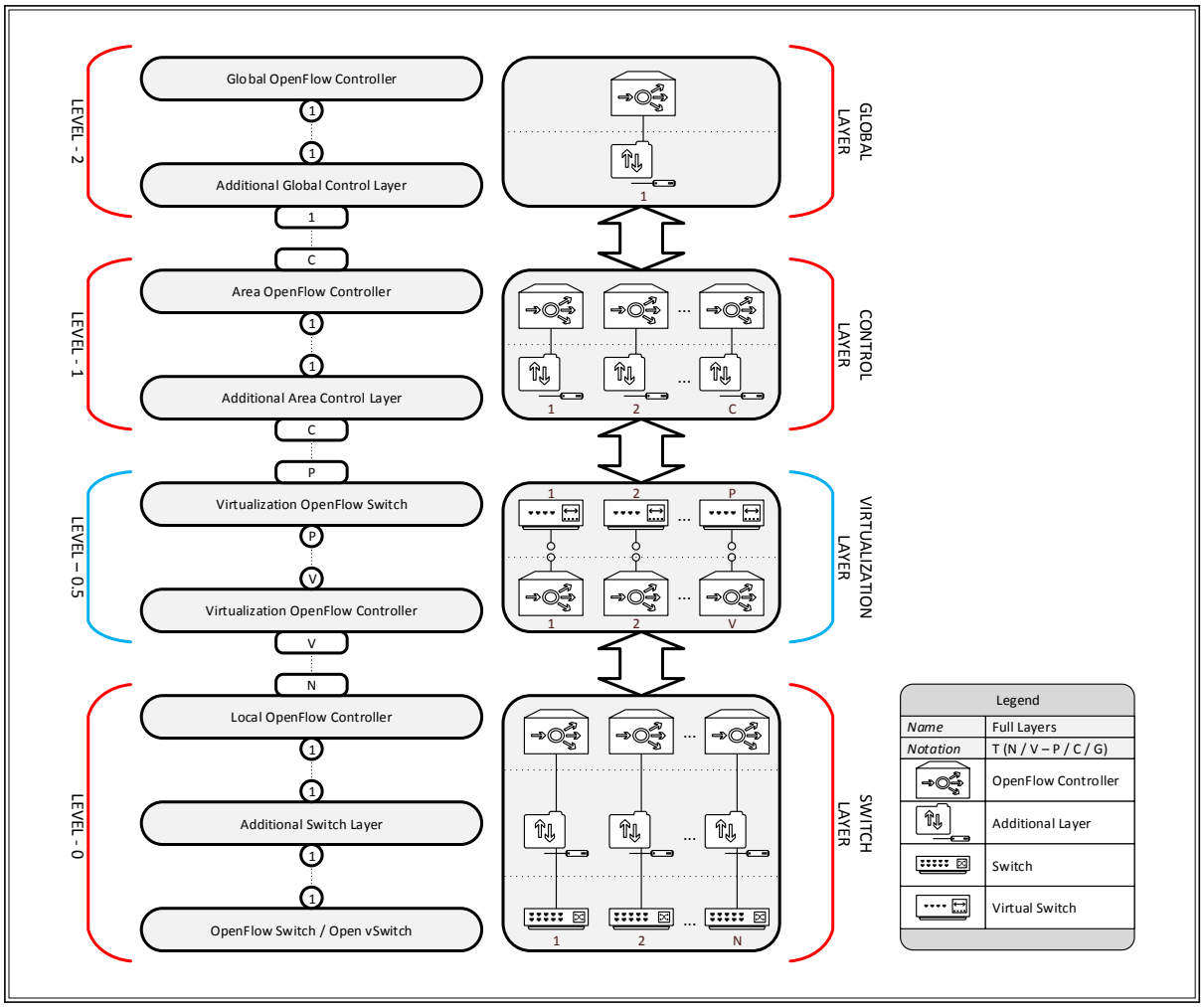


Figure 3-1: Graphical Framework for OpenFlow differentiation and comparison - *On the left the numerical UML relationship between the components and layers of an OpenFlow network topology is visible. On the right a physical decomposition of the same configuration is given to clarify the UML relationship.*

- Level-0.5 - *Virtualization Layer* - On top of the switch layer, the *Virtualization Layer* can be placed with the main function to divide and share the switch resources over multiple OpenFlow controllers. It enables multiple virtual network topologies on top of a single physical infrastructure. Resources of physical switches are virtualized by this layer and presented to the *Control Layer* as multiple virtual switches;

- Level-1 - *Control Layer* - The functionality of the control layer is to perform the tasks of the SDN control plane in a defined area of the network topology for a number of switches. Decisions made at this layer influence only a part of the network and are locally optimal. In regular OpenFlow configurations, only a single *Area OpenFlow Controller* is present. Solutions have been proposed to enhance the control layer with additional area OpenFlow layers to extend functionality, such as synchronization of Flow Rules with other controllers;

- Level-2 - *Global Layer* - The top layer of the OpenFlow layered structure in the framework has the functionality to control network topology at global level, where forwarding and routing decisions influence the whole topology. A *Global OpenFlow Controller* can thus compute globally optimal routes through the network, as it controls all switches. The structure of the global layer is similar to the control layer, so an OpenFlow controller and an additional layer.

Between the layers, a differentiation is made in how the network is managed. At the global level (domain), the controllers or applications in the additional layer have a global view of the network, with the status of all switches in the controlled infrastructure and the decisions to install flows to the assigned switches are based on the status of the complete network under control. The global level is expected in large data networks with a high number of switches to control. One level down is defined as the area level, where only a subset of the total number of switches is assigned to an OpenFlow controller. This indicates directly that more controllers are present in the network infrastructure. In the case that all switches are assigned to a single OpenFlow controller, decisions automatically are made at global level. At local level, decisions are made based on traffic present at the switch, which indicates that the size of the subset of switches at local level is equal to one. The virtualization layer has no particular role in the hierarchy of the OpenFlow layered structure, as it is designed to be transparent to controllers and it is used to share switches resources over multiple network services or users.

To indicate the numerical relationship between the layers, the UML-standard, as used in object oriented programming, is used as guidance in the framework in figure 3-1. The relationship states that the components (sub-layers) at each level share a one to one relationship (1..1) with each other. From the switch level a many to many or many to few relationship exists with the virtualization layer ($N..V$) or control layer ($N..C$), when no virtualization is applied. In the case of virtualization, a many to few or many to many relation ($P..C$) indicates P virtual switches controlled by C OpenFlow controllers. Within a domain, multiple area controllers can be controlled by a single centralized controller with the global view of the network. In case of an inter-domain network infrastructure, global layers can be interconnected.

In order to differentiate multiple network topologies using the UML relationships, we developed a notation. The notation simplifies the identification of proposed OpenFlow solutions and for network topology T the notation is given as $T(N/V - P/C/G)$, where the description of the used symbols is given in table 3-1. The notation with the defined symbols of table 3-1 would not cover the entire framework, as additional sub-layers or applications are not indicated. Therefore an extra indicator is added to the OpenFlow notation, to indicate if an enhancement is added and for which enhancement area (security, scalability and / or resiliency) the enhancement is added. Beside additional components to the layers, it is possible from OpenFlow protocol version 1.2 to add redundant controllers to the control plane. Therefore the backup indicator is defined. When an enhancement overlaps multiple areas or when a component is applied redundantly, it is possible to combine indicators. The controller C^{+SB} indicates a security enhanced controller which is redundantly applied.

Before multiple OpenFlow enhancement proposals will be discussed, a reference OpenFlow network configuration is given in the framework (see figure 3-2). The reference configuration is indicated by $T(N/-/C/0)$, where the infrastructure is build up from N switches controlled by C general OpenFlow controllers. If the reference configuration is more generalized, it

simplifies to the shown $T(N/-/1/0)$ network. Hereby we denote that no virtualization layer is applied.

Symbol	*Description*	*Relation*
N	Number of switches	$\in (1, 2, .., N)$
V	Number of virtual controllers	$\in (1, 2, .., V), V \leq N$
P	Number of virtual switches	$\in (1, 2, ..P)$
C	Number of area OpenFlow Controllers	$\in (1, 2, .., C), C \leq N$, $C \leq V$ or $C \leq P$
G	Global OpenFlow controller enabled	$\in (0, 1)$
X^{+s}	Layer enhanced for security	
X^{+p}	Layer enhanced for scalability	
X^{+r}	Layer enhanced for resiliency	
X^{+b}	Backup component available	

Table 3-1: Explanation for developed OpenFlow notation standard

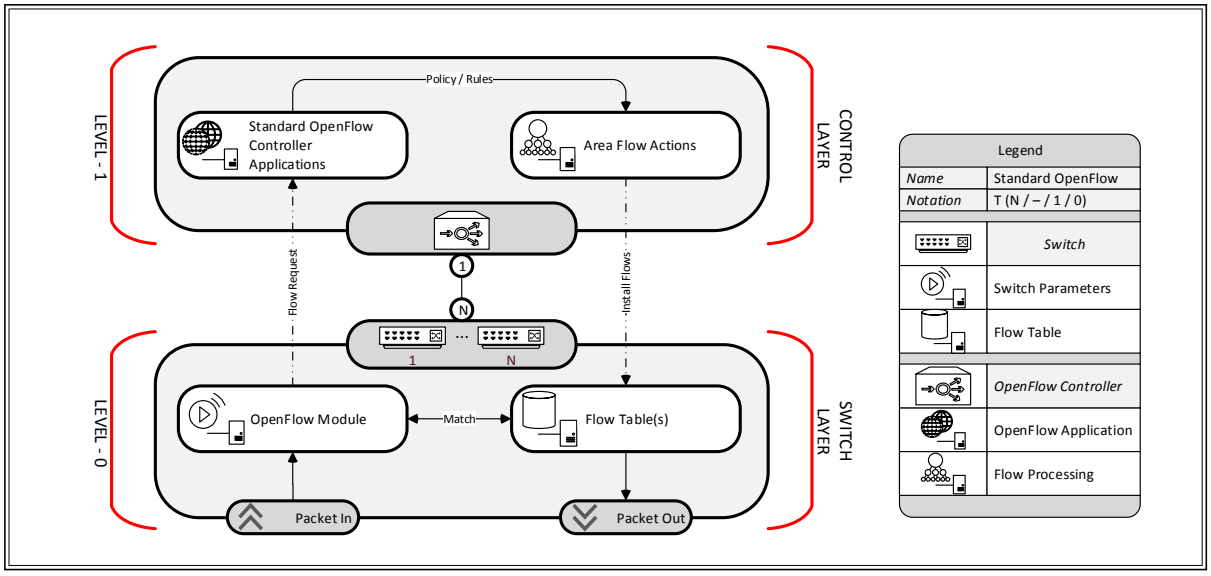


Figure 3-2: Standard OpenFlow configuration projected to framework - *Arriving network traffic is processed and when no flow is present in the flow table(s), a flow request is sent to a standard OpenFlow controller, where network routing applications determine a new flow based on set "Policy and Rules". The new flow is installed by the controller in the N-assigned switches and network traffic can transverse the network*

The flow of actions in figure 3-2 is as follows. Network traffic, in the form of data packets, arrive at the switch data plane, where the packets destination is matched to a set of Flow Rules stored in the flow tables. When no match is found, the OpenFlow module of the switch will initiate a Flow Request to the assigned controller. The controller will process the event and install the needed Flow Rules into the switches at the designated route through the network, based on the policy and rules defined in the forwarding and routing applications of the controller.

3-2 Scalability in SDN

Scalability concerns in SDN are generally identified by the community. These concerns are justified, as by introducing a centralized authority to control the network traffic over a large network, the growth of network traffic will not scale with the performance of the controller [29]. The last scalability concern is the performance in large production networks, as huge amount of Flow Requests and network events have to transverse the network and need to be processed by a single instance, which will lead to long flow setup latencies. Multiple proposals have been introduced by [30, 31, 32, 33, 34] to create a distributed centralized authority, to solve the concerns on scalability and forwarding performance. These proposals will be discussed and projected to the developed framework in paragraphs 3-2-1 to 3-2-5. This subsection will conclude with a summarizing conclusion, where advantages and disadvantages of the proposals are discussed for future implementations in SDN networking.

3-2-1 HyperFlow

The proposed solution [31] by Tootoonchian et al. to solve the scalability problem in SDN and OpenFlow networks is to deploy multiple OpenFlow controllers in the network infrastructure and implement a distributed event-based control plane, resulting in a $T(N/-/C^{+p}/1)$ structure. Designers of HyperFlow aim to keep the network control authority centralized and provide the distributed controllers with a global view of the network. The improvement in scalability and performance is found in the placement of multiple controllers close to the switches, reducing flow setup times, while each controller is provided with the global view of the network. Another advantage of this approach is the resiliency against network partitioning (disconnected network) in case of network failures, as the views of the connected switches and controllers are synchronized. In figure 3-3 the HyperFlow implementation is projected on the graphical framework.

HyperFlow is build up from three layers, where no virtualization is applied. The architecture at the switch level (data plane) and the OpenFlow specification are unmodified, which makes the HyperFlow concept applicable to current SDN capable networks. At the control layer the HyperFlow application is connected to a standard NOX-controller. To enable communication with the global layer, a publish-subscribe system is placed at the additional control layer to locally store the network status and view. The HyperFlow distributed global layer consists of three parts: the data channel, control channel and the distributed controllers. On the data channel network events are distributed by the HyperFlow application which contribute to the global status of the network. These events are stored in the publish-subscribe system, which is set to synchronize the global view of the controllers, but can operate independently in case of partitioning. The control channel is used for monitoring the status of the controllers. In case of failure of one or more controllers, the remaining controllers will not consider flow installs in switches under control by the failed controllers. The remaining functionality of the global layer are found in the distributed controllers. As each controller has a global view of the network and its status, it has capabilities to install flows into all switches. All required flows for a particular data stream are published to the distributed controllers. On synchronization by the publish-subscribe systems, the HyperFlow application on the controllers read out the requested flows for the switches assigned to it and will install the flows.

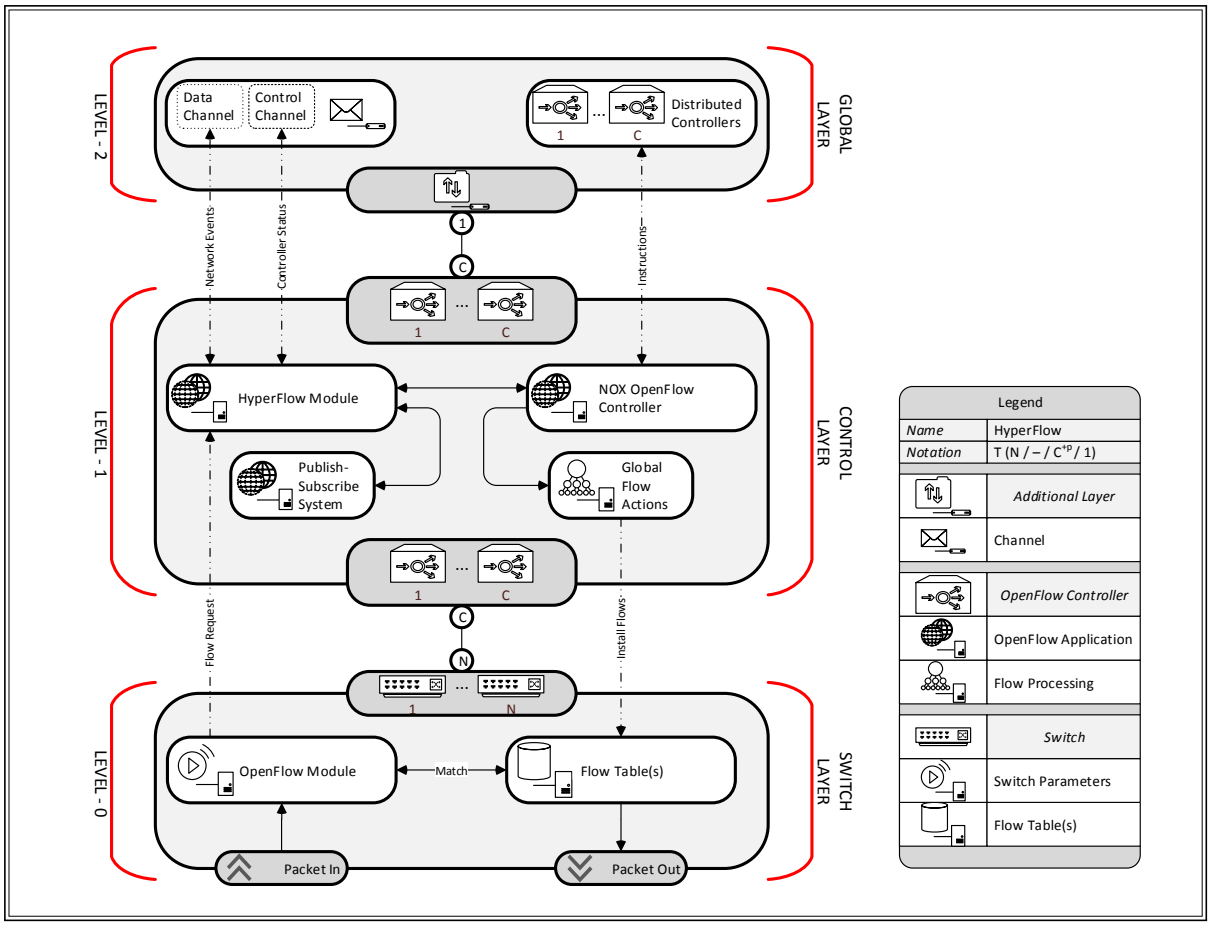


Figure 3-3: HyperFlow implementation projected to framework - *An additional global layer and extra OpenFlow applications at the control layer provide area controllers with additional information, to increase decision making capabilities to global level for optimal data packet forwarding*

The use of HyperFlow enables the possibility to operate a network domain with multiple controllers, without losing the centralized paradigm of SDN. In [31] no extensive measurements are performed on different network topologies and traffic loads, thus no conclusion can be drawn from the HyperFlow approach to improve scalability in real-life SDN networks. Besides that, there are some limitations in the HyperFlow design. The first limitation was found by the authors themselves in the performance of the publish-subscribe system (WheelFS). All network events, flow installs and status information needs to be synchronized between the multiple controllers, which requires a fast distributed storage system. In HyperFlow the performance of the publish-subscribe system was limited to approximately 1000 events per second. This does not indicate that the controllers are limited in processing, but the global view of the network controllers may not quickly converge. A second limitation is the lack of a management application in the global layer. In [31] no distinction is made between the functionality of the switches and controllers. This assumes that a global policy for Flow Rule installations must be configured in all assigned controllers. The last limitation is found in the performance of HyperFlow. The load on controllers can be reduced by assigning less switches to a controller. In the extreme case, a system can be designed, where a single

high priority switch is assigned to an OpenFlow controller. If the load on the controller still exceeds its capacity, flow requests will be dropped and network services may fail. We think a solution where Flow Requests can be forwarded to neighbor controllers for processing or smart applications at the switch layer to off-load controllers could be another solution.

3-2-2 ONIX

The developers of ONIX, Koponen et al. [33], found that there was no general control paradigm to control large-scale production networks and therefore a control platform was developed, using the general SDN paradigm. ONIX is not build on-top of an OpenFlow controller, but can be classified as a *General SDN Controller*. The approach the developers took to abstract the network view and overcome scalability problems, was to abstract the network view by separating the data and control plane and add an additional distributed control platform between these planes. By adding an additional platform, the management and network control functions from the control plane are separated from each other. The management functions, used for link status and topology monitoring, are performed by the ONIX control platform and reduce the workload for a general flow controller. To partition the workload at the control platform, multiple ONIX instances can be installed to manage a subset of switches in the network infrastructure. The available information is synchronized between the ONIX instances, forming a distributed layer as seen with HyperFlow in section 3-2-1, but now at the controller layer. A network controller assigned to a part of the network, can subtract information from the distributed layer to compute area flows. In order to compute global flows, ONIX has the capability to aggregate information from multiple area platforms to a global distributed platform. The aggregation of information and the calculation of global optimal forwarding rules is similar to the routing of Internet traffic over multiple autonomous systems (AS's), where the route between AS's is globally determined, but the optimal route inside the autonomous system is computed locally. With this general introduction, we can conclude that ONIX can be classified as a $T(N/-/C^{+p}/1^{+p})$ SDN topology[1]. With the use of figure 3-4 more insight is given in the the design and functioning of ONIX at area level, while figure 3-5 shows how global flows are computed using the distributed control platform.

At switch layer no modifications are required for ONIX and two channels connect to the switch with the *Import-Export* module at the distributed platform. The ONIX platform is an application which runs on dedicated servers and requires no specific hardware to function. According to the load and traffic intensity, multiple switches are assigned to a ONIX instance. Multiple ONIX instances combined form the distributed control platform. The first channel connects the configuration database and is used for managing and accessing general switch configuration and status information. To manage the forwarding, Flow Tables and switch port status, the second channel is connected to the OpenFlow module of the switch. The information and configuration status collected by the import-export module represents the network state of the connected switches and is stored into the Network Information Base (NIB) as a network graph. To store the network state, the NIB uses two data stores, being a SQL-database for slow changing topology information and a Dynamic Hash Table (DHT-table) for rapid and frequent status changes (such as actual link utilization and round trip times). The application of two data stores overcomes the performance issues where HyperFlow

[1]In this thesis the network scope is limited to a single domain, but the capabilities of ONIX can reach beyond that scope as it is designed for large-scale production networks.

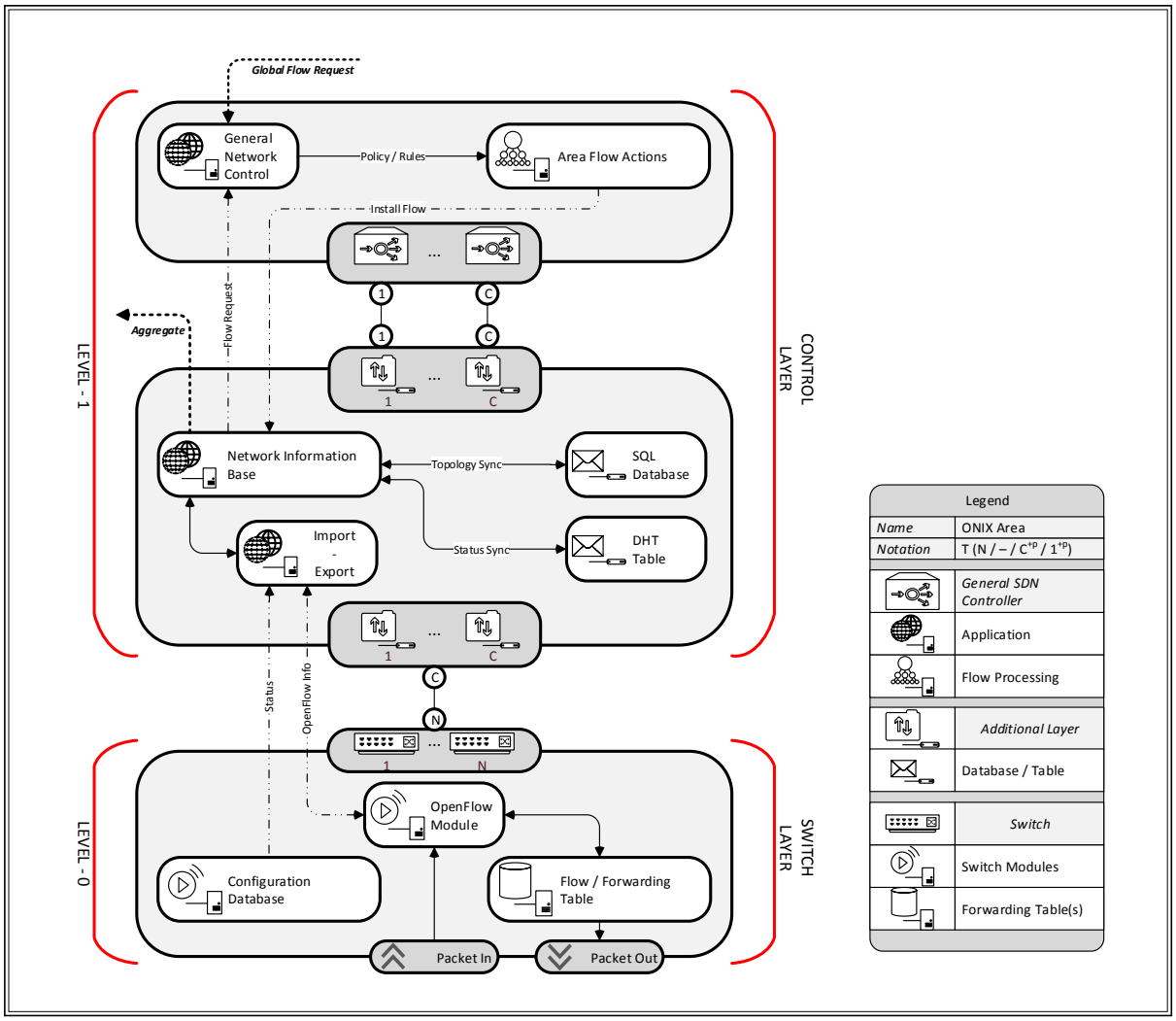


Figure 3-4: ONIX distributed control platform at area level - *ONIX adds a distributed control platform between the data and control plane to reduce the workload on flow controllers. Switch management is performed by multiple ONIX instances which are synchronized via two databases. Forwarding decisions are made at area level by a general SDN flow controller*

(WheelFS) deals with. Reliability and durability for topology information is provided by a persistent SQL-database, where availability of status information is guaranteed by a memory-only (non-persistent) DHT-table. From the NIB a general flow controller or control logic can receive Flow Requests, switch status and network topology and compute paths based on requirements throughout the network. In comparison to the reference OpenFlow controller, the ONIX platform has the availability of switch and link status information. Computation of paths is thus not limited by the information provided by the OpenFlow protocol, which leads to optimal routing solutions. The last step in the flow setup process is to install the computed flows in the switch forwarding table via the ONIX distributed platform.

To extend the functionality of ONIX to the global layer, the network state from multiple ONIX instances can be aggregated into a single global distributed platform with a single global controller. The global controller has a single ONIX instance available to extract the

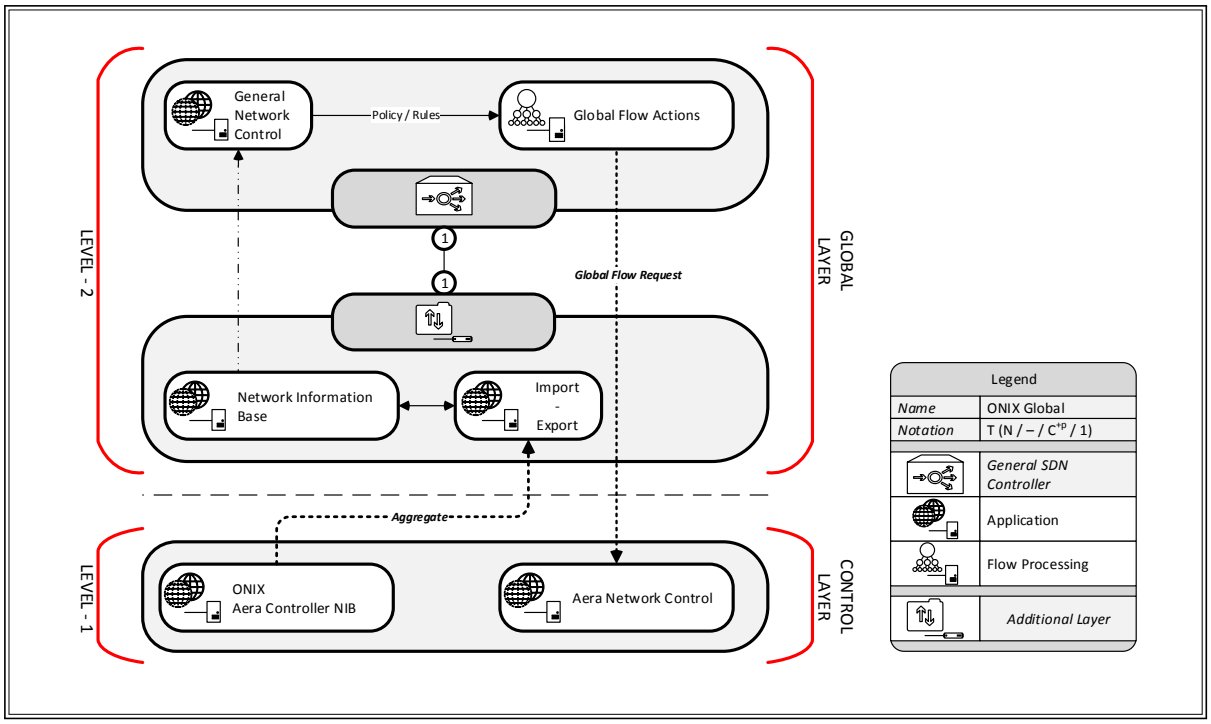


Figure 3-5: ONIX distributed control platform at global level - *Aggregate information from multiple area controller are combined at the ONIX global layer, where from the global controller can determine optimal routing and request the level-1 controllers for paths within assigned area*

required network information. As mentioned earlier, the global controller will not compute the entire path throughout the network topology, but it determines which areas are required to transverse the domain and requests area controllers to install the required flows. Area controllers itself will determine the optimal path and configure the flows into the assigned switches.

In [33] evaluation results are shown of measurements on the ONIX distributed layer to determine the performance of a single ONIX instance, multiple instances, replication times between the data stores and recovery times on switch failures. Unfortunately no information is present of the used control logic and the performance gain in comparison with a regular OpenFlow controller. The advantage of the ONIX distributed control platform is the partitioning of the workload over multiple instances. So, if ONIX instance is limiting traffic throughput (dropping flow requests) due to high workload, assigned switches can be reassigned to other ONIX instances. Also the distribution of switch status information is not limited to the OpenFlow protocol. Information available at the switch can be distributed over the control layer for optimal route calculations. As seen with HyperFlow, ONIX can extend to the global layer, enabling optimal routing in large production networks. The cost for these advantages is an increase of complexity and the widely available OpenFlow controllers need drastic modifications to work with ONIX.

3-2-3 DevoFlow

The developers of DevoFlow, Curtis et al. [30], took another approach to solve the scalability problem in large data networks. Where HyperFlow and ONIX are developed to enable more controllers, find a global routing optimum and divide the workload over a distributed control platform, the approach of DevoFlow is different in two ways. First, the amount of traffic between the data and control plane needs to be reduced, because the current hardware OpenFlow switches is not optimized for inter-plane communication and therefore the number of processed flows per second is not sufficient for large data networks. Second, the high number of flow requests must be limited, because the processing power of a single OpenFlow controller is not sufficient and does not scale with network traffic. To limit the traffic and flow requests to the control plane, traffic must be categorized into *micro* and *elephant* flows. In earlier research by Benson et al. [35] is shown that a large percentage of data conversations between servers and end-users occur at high rate, but are relative short in duration. These traffic conversations are labeled as *micro flows* and require a lot of communication between the data and control plane (OpenFlow controller) for flow request processing. In DevoFlow micro flows will be processed at the switch layer, without the need of the logic at the control layer. The DevoFlow philosophy is that full control over and a global routing optimum for all traffic flows is not required and that only *heavy* traffic users, *elephant flows*, need flow management. This because elephant flows have the most influence on the performance of the network and often come with extra QoS requirements. By limiting the management to elephant flows, only a single controller is needed in the network topology, shifting the forwarding complexity to the switch layer. With this information we classified the DevoFlow solution as a $T(N^{+p}/-/1/0)$ SDN topology. In figure 3-6 DevoFlow is projected to the framework.

In the framework the DevoFlow solution is drawn as an additional layer on top of the physical switch to simplify the representation on the framework, but the actual implementation is performed at the soft- and firmware of the switch. This indicates that modifications to the standard switch are required. To ease integration of DevoFlow to other SDN concepts, no modifications are made to the OpenFlow protocol and controller. The header of arriving packets is compared to Flow Rules installed in the switch table. When no match is found, a Flow Request must be initiated by the *Traffic Measurement* module in the switch. The Traffic Measurement module monitors the incoming flow of data packets for traffic statistics collecting. At the start of a flow, each flow is marked as a micro flow and existing Flow Rules from the switch forwarding table are cloned and modified by the *Local Flow Scheduler*. Modifications can include Flow Rules to allow multi-path routing and re-routing. In case that multiple paths exist between the incoming switch and destination, the Local Flow Scheduler selects one of the possible outgoing ports and the micro flow rule for that port is cloned. Multiple routing strategies, like equal-cost multi-path (ECMP) routing [36], uniform port selection or round-robin, are present at the Local Scheduler to spread traffic over the network. Re-routing is applied when one of the available outgoing ports is down and traffic needs alternative paths through the network.

To detect and route elephant flows through the network, the Area Scheduler can apply four different schemes:

- *Wild-card routing* - The switch pushes the available traffic statistics to the controller with a specified time interval. The scheduler pro-actively computes unique spanning

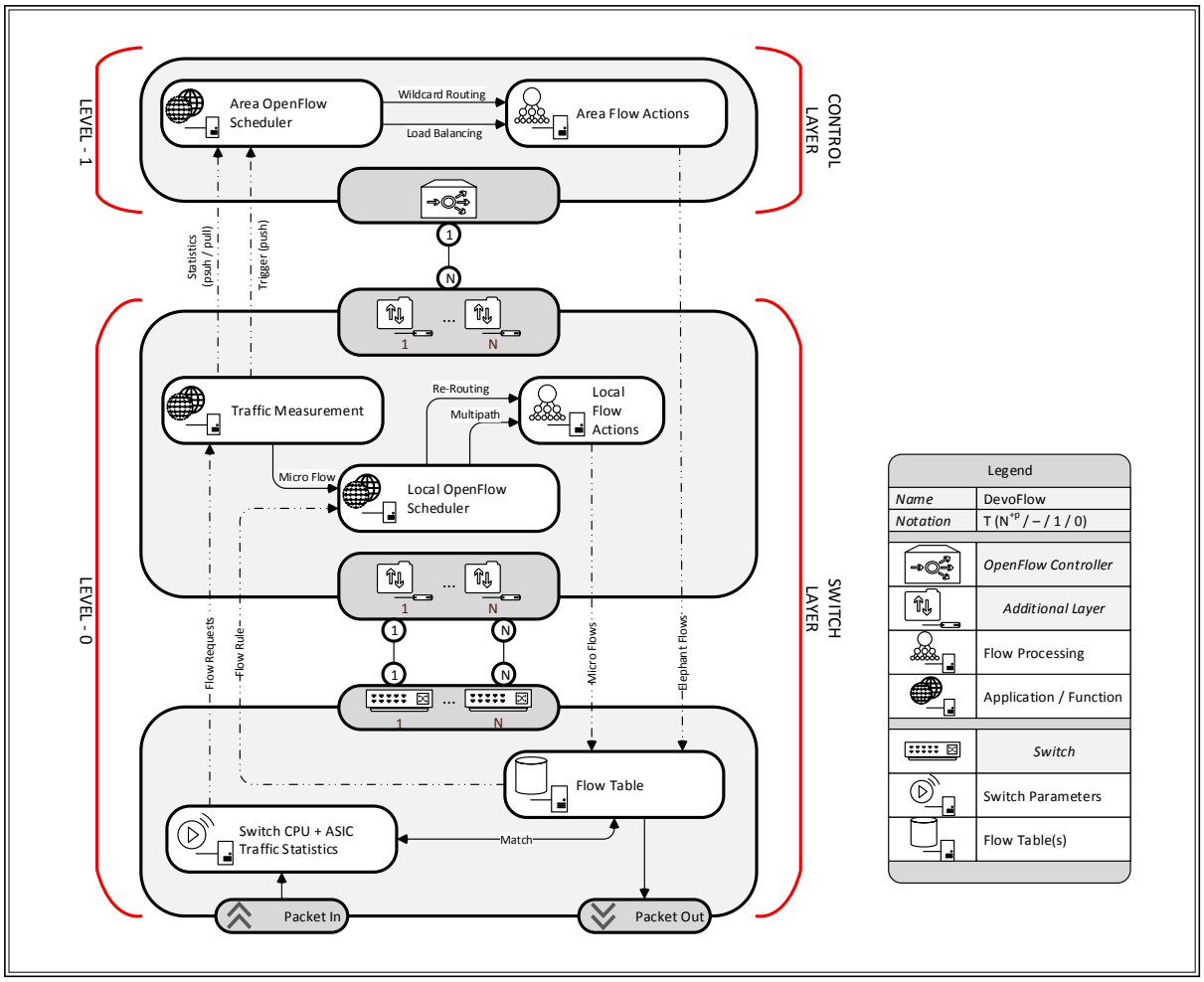


Figure 3-6: DevoFlow implementation projected to framework - *Scalability enhancements made at switch layer are shown as additional layer in the framework, but are implemented as modifications to the soft- and firmware in the switch ASIC / CPU. Routing decisions can be made at the switch layer traffic parameters as input to reduce workload at OpenFlow controller. Elephant flows invoke the use of the area controller for optimal routing, where micro flows are routed using cloned flow rules.*

trees for all destinations in the network topology using the least-congested route and install the trees as flows in the switch flow tables. For each destination in the network a Flow Rule is present at the switches forwarding table and no Flow Requests are sent from the switch to the OpenFlow controller;

- *Pull-based statistics* - The scheduler regularly pulls the traffic statistics from the switch and determines if elephant flows are present in current data flows. Once an elephant flow is detected, the scheduler determines the least congested path for this flow and install the required Flow Rules at the switches;

- *Sampled statistics* - This method is very similar to the pull-based scheme, but instead of pulling traffic statistics every time period, the switch samples traffic statistics into a bundle and pushes the bundle to the scheduler. At the scheduler, again is determined

if any elephant flows are present and on positive identification Flow Rules are installed, as described in the pull-based scheme;

- *Threshold* - For each flow at the switch, the amount of transferred data is monitored. Once a flow exceeds a specified threshold, a trigger is send to the scheduler and the least congested path is installed into the switches.

All schemes are based on traffic statistics, where flows are only installed if identified as elephant flow in a reactive manner. In [30] multiple simulations have been performed on a large data network simulator to capture the behavior of flows through the network and measure data throughput, control traffic and the size of Flow Tables in the switches. The results show that the pull-based scheme with a short update interval maximizes the data throughput in the network. This performance comes at a price, as much traffic is initialized between the switch and controller and the size of the Flow Table is significantly large in comparison with the other schemes. The threshold scheme is identified as most optimal, as the data throughput is high, less control traffic is required between the switch and the controller and the size of the Flow Table is minimal. Another advantage is the required workload on the scheduler in the controller, as no traffic statistics have to be monitored and processed.

3-2-4 Kandoo

In [32] Hassas et al. presented a framework with similar goals and philosophy as DevoFlow, namely limit the overhead of events between the data and control plane. Where DevoFlow limits the overhead by processing frequent events in the data plane and need complicated modifications to the switch software architecture, the developers of Kandoo aim to achieve the same results by applying more controllers in a network topology. Kandoo differentiates two layers of controllers, namely *Local Controllers*, which are located close to the switches for local event processing, and a *Root Controller* for network wide routing solutions. In practice this indicates that Local Controllers will process the micro flows and a trigger from the Local Controller must inform the Root Controller about the presence of an elephant flow. The number of Local Controllers is dependent on the amount of traffic (local events) and the workload on the controller, which is somewhat similar to the approach of ONIX and the distributed control platform. In an extreme case, every switch in the network topology is assigned to a single Local Controller. Translating the Kandoo terms to the graphical framework, the Root Controller is located at the global layer and the Local Controllers can be found on the area or switch layer. If the extreme case is valid, the Local Controller can be seen as an extension of the switch layer and the topology can be classified as $T(N^{+p}/-/1/0)$. In a regular architecture, where more than one switch is assigned to a Local Controller, a $T(N/-/C^{+p}/1)$ SDN topology is found. Figure 3-7 shows the regular case of Kandoo projected to the graphical framework.

Kandoo leaves the firmware in the switch unmodified and shifts the processing of events to Local Controllers. The Local Controllers are general OpenFlow controllers extended with a Kandoo module. This approach keeps the communication between the switch and controller standardized and provides the possibility to utilize standard OpenFlow applications. The Kandoo module intercepts OpenFlow traffic and monitors it for elephant flows using the *App*

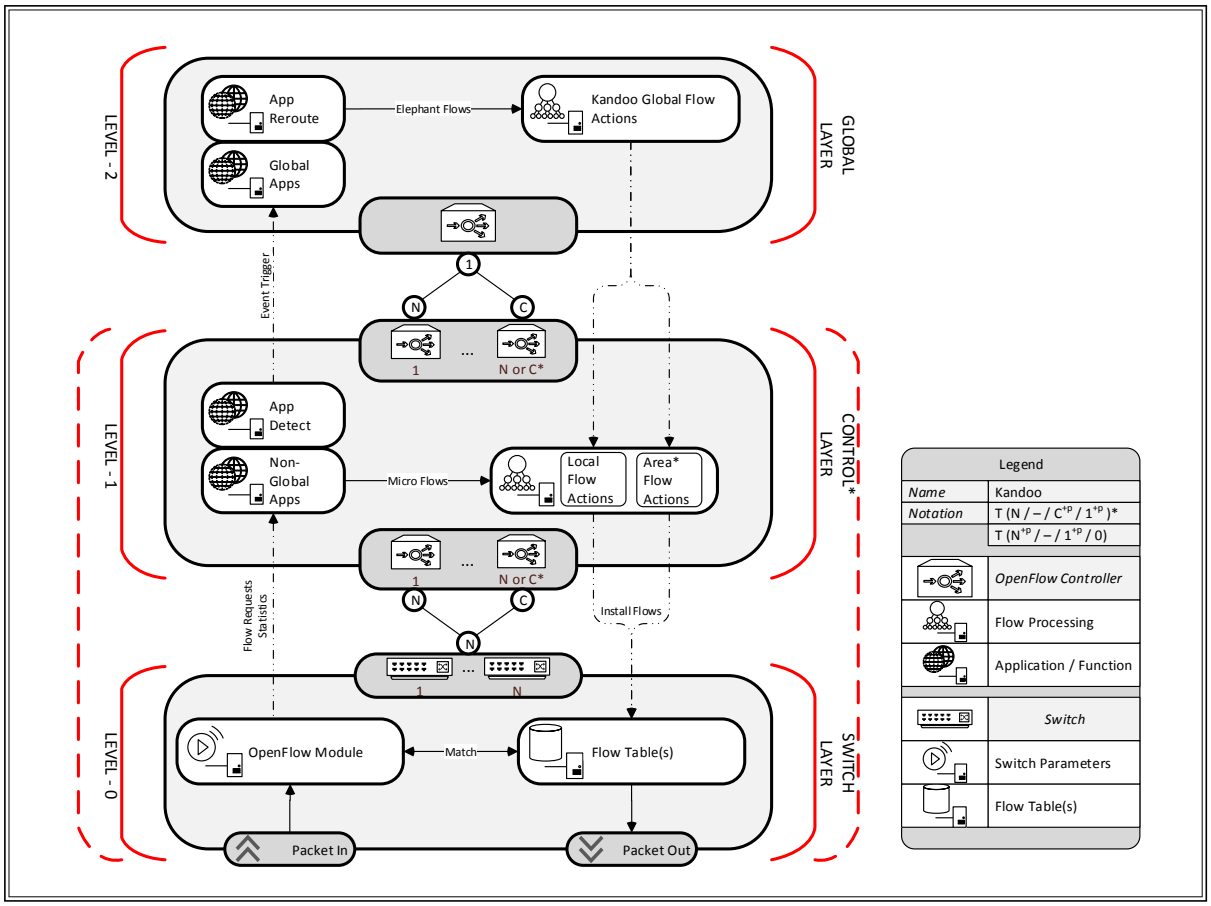


Figure 3-7: Kandoo implementation projected to framework - *An standard switch sends flow requests and statistics to the local controllers. The Non-global applications process local OpenFlow events, micro flows and the "App Detect" application detects elephants flows using a threshold. Elephant flows are processed by the root / global controller, where after the flow rules are send to the local controller to install at the switches*

Detect application. As long as no elephant flow is detected, the Local Controller processes Flow Requests as micro flows. Here we assume that processing of micro flows is performed by general OpenFlow applications and no additional routing schemes are applied for multi-path routing and re-routing. Processed micro flows are installed in the switch in a regular way, resulting in area optimal forwarding of the micro flows. Elephant flows are detected using the threshold scheme from DevoFlow [30], which relays an event to the Kandoo module of the root controller in case of positive detection.

To propagate events from Local Controllers to the Root Controller, a messaging channel is applied. The Root Controller must subscribe to this channel in order to receive events. After receiving an elephant flow detection trigger, the *App Re-Route* module computes an optimal path and requests the local controllers to install this path. Hassas et al. [32] have not given any information on how the re-routing process is executed and which routing schemes are applied. Also no information is presented about traffic statistics relaying from Local Controllers to the Root Controller to optimize paths using link utilization information to meet QoS requirements for elephant flows.

Some measurements have been performed in [32] on a (small) tree topology with the Kandoo framework installed. Results show comparisons between the topology in a general OpenFlow and Kandoo configuration. As expected, less events are processed by the Root Controller, but no information is given about the workload and performance indicators of the Local Controllers. Overall we can state that the experiments and measurements are too limited to give a good indication of the performance enhancement provided by Kandoo. The limiting factor in the Kandoo configuration is the control interface between the switch and the local controller, as in DevoFlow [30] is shown that this interface is a limiting factor in current SDN network implementations. While the limit on this interface is not fully reached, the layered controller solution is an interesting concept which can also be useful to solve security and resiliency problems.

3-2-5 FlowVisor

Computer virtualization is a widely applied technique allowing multiple instances to share hardware resources. Hardware resources are abstracted by an abstraction layer and presented to a virtualization layer. The abstraction layer communicates with the hardware resources and process communication flows for the virtualization layer. On top of the virtualization layer, instances are presented with virtualized hardware resources and control them as without virtualization. This approach is roughly similar with the SDN philosophy, but in FlowVisor [34] by Sherwood et al. the virtualization approach is reapplied, where OpenFlow compliant switches are offered to the FlowVisor abstraction layer. FlowVisor offers slices of the network topology to multiple OpenFlow *Guest Controllers*, where the slices are presented as virtual OpenFlow switches. The Guest Controllers control the slices, where FlowVisor translates the configurations and network policies from each slice to Flow Rules on the physical OpenFlow switches. If this approach is applied to large networks, scalability problems can be resolved as control of the OpenFlow switches is divided over multiple controllers. To distinguish network slices, four dimensions are defined in FlowVisor:

- *Slice* - Set of Flow Rules on a selection of switches of the network topology to route traffic;
- *Separation* - FlowVisor must ensure that guest controllers only can control and observe the partition of the topology assigned;
- *Sharing* - Bandwidth available on the topology must be shared over the slices, where minimum data rates can be assigned for each slice;
- *Partitioning* - FlowVisor must partition the Flow Tables from the hardware OpenFlow switches and keep track on the flows of each guest controller.

With these four dimensions and definitions, OpenFlow resources can be virtualized and shared over multiple instances. Applications for FlowVisor and sharing network resources can be found in business networks, where each department (for example financial, sales, development, etc.) is assigned a partition of the network capacity, where separation or other network policies are required. Other methods to provide separation are multiple separated physical network topologies or application of VLAN's. FlowVisor differentiates from these methods, as slices

may overlap and same physical links can be utilized by multiple guest controllers. This means that a single hardware OpenFlow switch can be controlled by multiple Guest Controllers. To provide more details on the functioning of network virtualization in OpenFlow, FlowVisor is projected to the general framework in figure 3-8.

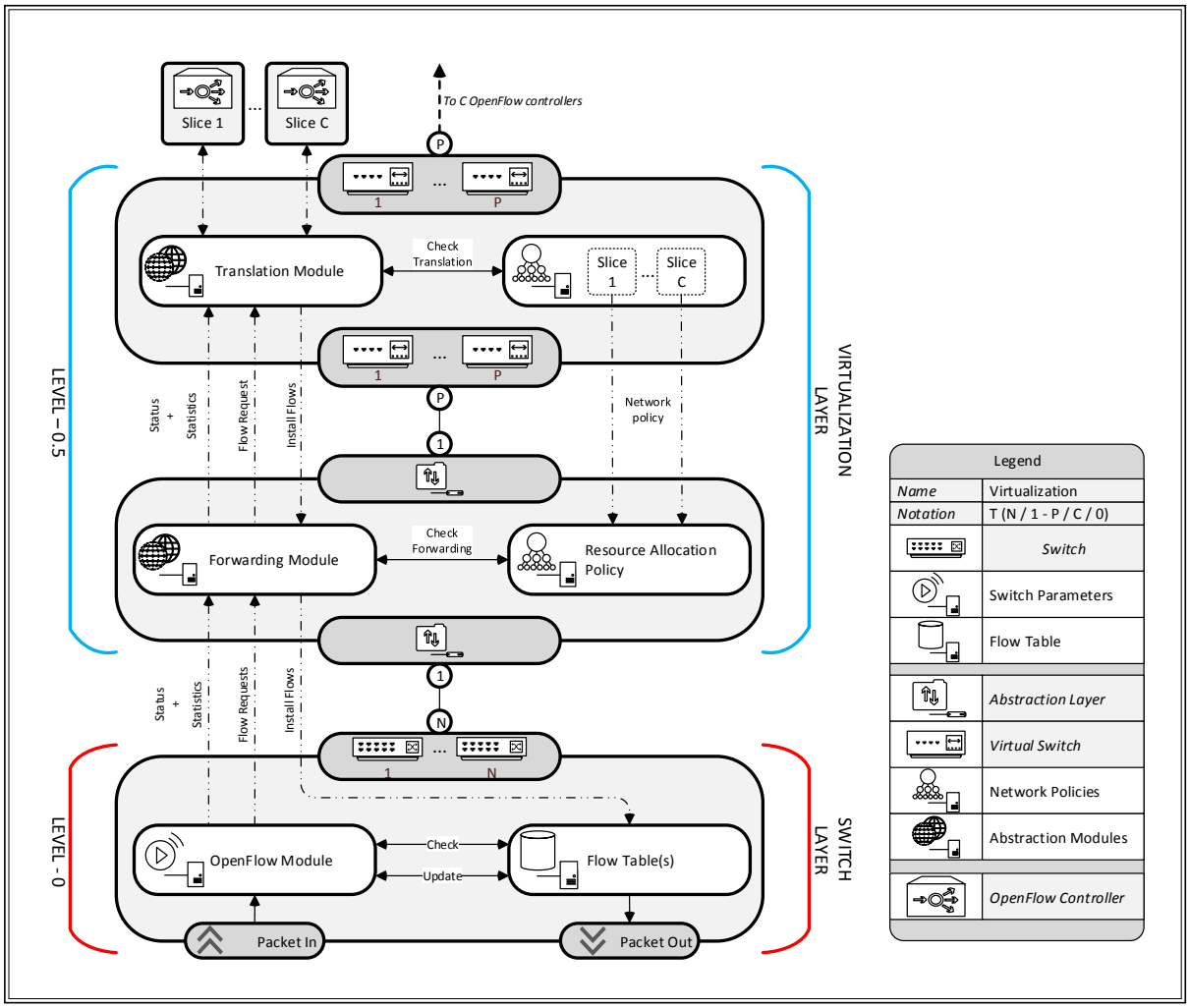


Figure 3-8: Network virtualization on OpenFlow network topology projected on the graphical framework - *Regular OpenFlow traffic (Flow Requests, status and traffic statistics) from the OpenFlow module is send to the Flow Visor Forwarding Module. Depending on the slice configuration and slice policy, OpenFlow traffic is forwarded to the Translation Module of the "virtual" switches. Guest controllers (which can be any OpenFlow controller) communicate with the translation modules, where Flow Rule are translated, forwarded and installed if they not interfere with the network policy for that slice.*

As illustrated in figure 3-8, FlowVisor acts like a transparent layer between hardware switches and the controllers. OpenFlow switches are assigned to the virtualization layer, where FlowVisor advertises itself as a controller. OpenFlow traffic is transmitted to FlowVisor at the virtualization layer, where the switch resources are sliced and divided over multiple users. The FlowVisor Forwarding Module checks on policy violations, before it is sent to the Translation Module at the virtual OpenFlow switch. At the translation module, the traffic is translated

and sent to the Guest Controllers, assigned to the corresponding slices. Flows coming from the Guest Controllers are checked on interference with the slice policy and translated to Flow Rules for the OpenFlow switches. Via the Forwarding Module the guest flows are installed at the switches. The projection shows a straight forward example of a $T(N/1 - P/C/0)$ configuration, where a single FlowVisor instance performs all virtualization tasks, but more complex configurations are possible, where multiple FlowVisor instances intermediate with physical switches and the virtual switches itself are assigned for virtualization. In figure 3-9 a $T(5/3 - 6/4/0)$ topology is given, with 5 OpenFlow switches, 3 FlowVisor instances and 6 slices, controlled by 4 Guest Controllers.

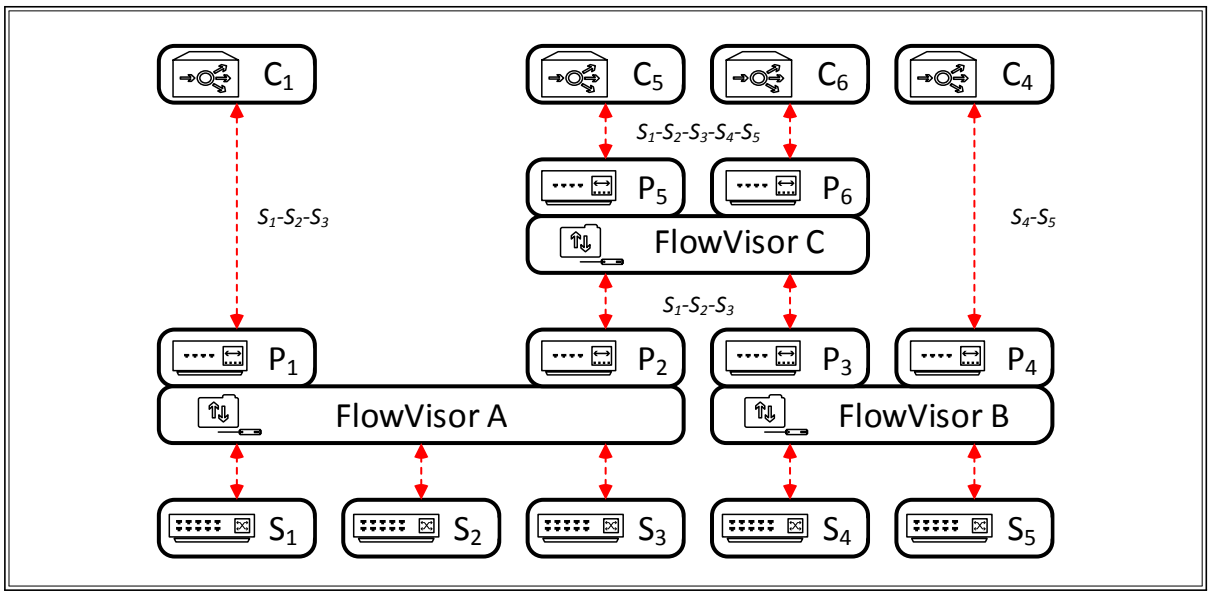


Figure 3-9: Example topology with multiple FlowVisor instances resulting in a hierarchical structure - *Switches S_1 to S_5 are assigned to two FlowVisor instances ($A + B$). On each FlowVisor instance, two slices are created (P_1 to P_4). Slices P_1 and P_4 are respectively controlled by C_1 and C_4, where slice P_5 and P_6 are assigned to FlowVisor C. Both controller C_5 and C_6 control switch S_1 to S_5 indirectly, but due to FlowVisor's dimensioning mechanisms the controllers do not influence each other.*

In the example, the network topology is shared over four departments, each with a separate OpenFlow controller. To provide the slicing, switches are assigned to a FlowVisor instance, where S_1 to S_3 are assigned to FlowVisor A and S_4 to S_5 to FlowVisor B. On A, two slices (P_1 and P_2) are configured. Slice P_1 is controlled by Guest Controller C_1, which indicates that C_1 has control over S_1 to S_3. The second slice, P_2 is assigned to FlowVisor C. A similar configuration is applied for S_4 and S_5, resulting in slice P_5 and P_6 . Both P_5 and P_6 have are configured with a separate controller and control switches S_1 to S_5 , but due to the dimensioning mechanisms of FlowVisor, the different departments do not influence each other and have (indirect) control over their own slice of the virtual topology.

The FlowVisor virtualization layer is implemented and multiple experiments have been performed by Sherwood et al. [34] to determine the costs and benefits. Adding an additional layer between the switch and control layer creates unwanted overhead. Experiments show that response times for the processing of Flow Requests increased from 12 ms to 16 ms. This means that FlowVisor accounts for an additional delay of 4 ms, in comparison to non-virtualized

OpenFlow topologies [34]. Besides the delay measurements, also experiments are performed to test bandwidth sharing between slices and CPU utilization on the assigned OpenFlow switches. To provide fair bandwidth sharing, network policies must include minimal bandwidth guarantees (QoS parameters) for each user, to prevent unfair use of the network by one of the users on the network. Besides sharing the traffic links, also the switches CPU time must be shared over multiple slices and controller. To prevent 100% utilization of the switch CPU, FlowVisor can limit the control traffic from the switches. Without this mechanism, the switches CPU's can be saturated and Flow Requests are dropped.

Network virtualization enables the possibility to share the topology with multiple users, each with their own network policies and network controller, resulting in a lower number of hardware switches and better network utilization. FlowVisor provides a transparent layer between switches and controllers to enable virtualization. Experiments show that virtualization is possible, but comes with additional overhead and the need for mechanisms to share switch resources. An aspect not covered in FlowVisor is security. Additional mechanisms for classification of traffic and checking of Flow Rules may be required to ensure full separation and isolation between network traffic on the slices.

3-2-6 Conclusion on scalability

Five different concepts on scalability have been reviewed in relation to application in large data center networks. All solutions propose a solution where workload is divided over multiple instances. More in depth, the solution space is roughly divided into two solution concepts. The first concept is provided by HyperFlow and ONIX, which distribute essential status information of switches to multiple controllers over a distributed control layer. This layer provides network applications at the controllers with a global view of the network and enable them to calculate optimal routing solutions. A drawback of a distributed layer with status information is the introduced complexity for synchronization between multiple instances. The solution of HyperFlow with a distributed file-system looks promising on paper, but Tootoonchian et al. in [31] showed that the applied synchronization method did not have the capabilities to cope with the high amount of synchronization information. ONIX solved this problem by utilizing two data stores, a DHT-table for high rate synchronization and a SQL-database for consistent status replication. For future solutions, where synchronization of information is needed over multiple network instances, the two data stores are a proven concept. Both data stores are available as open-source software.

The second concept embraces the philosophy to limit the number of events between the data and control plane and that it is unnecessary to have global control over all data flows in the network. Therefore in DevoFlow schemes are developed to classify flows (micro and elephant flows) and push processing to the data plane. Micro flows are classified as frequent local events and do not need processing by a centralized controller. Therefore DevoFlow and Kandoo developed frameworks to shift the processing of micro flows as close to the switch as possible and return to current state networking paradigms. DevoFlow introduces major modifications to the switch software modules to enable traffic statistics and processing of micro flows, where Kandoo introduces these modifications to a local controller, close to the switch. Elephant flows are processed by the single central controller to meet QoS requirements for the flows or better network utilization. The modifications made to the switch software

architecture by DevoFlow are perpendicular to the vision of SDN and OpenFlow networking, where a standard protocol between the data plane and control logic is advertised. Kandoo therefore offers a better concept, by allocating local controllers nearby the switches. We propose to combine the implementation of DevoFlow and Kandoo in a standard OpenFlow controller embedded in the switch hardware, utilizing the hardware resources of the switch optimally by performing frequent events. Control of the embedded OpenFlow control logic is provided by (multiple) root controllers.

FlowVisor does not belong to both solution concepts, as no specific applications and modules are added to solve scalability issues. By smartly sharing hardware resources over multiple controllers via the FlowVisor virtualization layer, these issues can be solved. Because FlowVisor is compliant with the OpenFlow protocol, the proposed solutions from HyperFlow and Kandoo can be applied.

It is difficult to come with one optimal scalability solution for SDN networks. Trade-offs have to be made by network designers and managers. Table 3-2 gives an overview of the proposed frameworks and their components. Besides observations from the review, also a column is defined for standardization. This indicates availability of used components in the framework. Unfortunately, only FlowVisor is available as open-source software, but on conceptual level standardized components can be used to reproduce the proposed frameworks. A high standardization in the table is indicating that the solution is build up from standard available OpenFlow components. A part of the table is dedicated to data storage solutions used in ONIX and HyperFlow and is useful for future distributed layer developments and comparisons.

Solution	*Standarized*	*Complexity*	*Decision*	*Flow Classification*
HyperFlow	$\rightarrow$	$\uparrow$	Global	X
ONIX	$\downarrow$	$\uparrow$	Global	X
DevoFlow	$\rightarrow$	$\rightarrow$	Semi-Global	✓
Kandoo	$\uparrow$	$\downarrow$	Semi-Global	✓
FlowVisor	$+$	$+/-$	Semi-Global	X
Solution	*Availability*	*Performance*	*Reliability*	
WheelFS	$\rightarrow$	$\downarrow$	$\uparrow$	
DHT	$\uparrow$	$\uparrow$	$\downarrow$	
SQL-database	$\uparrow$	$\downarrow$	$\uparrow$	

Table 3-2: Comparison of scalability solutions and components

3-3 Resiliency in SDN

In current state networks, when the control logic of a switch fails, only network traffic over that particular switch is affected. When failover paths are pre-programmed into neighboring switches, backup paths are available and on failure detection, backup paths can be activated. If the control logic in a SDN enabled network fails, the forwarding and routing capabilities of the network are down, resulting in drop of Flow Requests, undelivered data packets and an unreliable network. In an early stage of the development of the OpenFlow protocol, this problem has been identified and from protocol version 1.2, a *master-slave* configuration at the control layer can be applied to increase the network resiliency and robustness on failing OpenFlow controllers.

This section cannot continue without giving a definition of resiliency and robustness in respect to SDN networking. In here a distinction can be made between the controller or network topology point of view. We define robustness of a topology as the measure of connectivity in a network after removing links and switches, in other words: *How strong is the network connected?* Its resiliency indicates its ability to re-allocate redundant paths within a specified time window. On the controller side, the robustness of a single or group of OpenFlow controller(s) is defined as the resistance of controller(s) before entering failure states. The resilience is defined as the ability to recover control logic after a failure. As described, the definitions for robustness and resiliency can have different meanings, depending on the viewpoint of the designer. In this section, examples and proposals of both viewpoints are discussed.

Before proposed solutions on resiliency and robustness will be reviewed, a small retrospect to the scalability section is made, as all these solutions house the ability to increase the robustness of a SDN network. By partitioning the workload of a single controller over multiple instances, the robustness of the network is increased. Failure of a single controller will only result in an uncontrollable partition of the network. To recover from a failure and increase the resiliency, additional logic is required. For both viewpoints, timely detection of a failure and a fast failover process are basic requirements. To create a robust and resilient network, the network topology must include redundant paths [37]. For a resilient control layer, the network state must be synchronized and identical between master and slave controllers, to support seamless overtaking without the need for network initialization and discovery processes. Additional modules and synchronization schemes must meet these requirements without compromising the performance and adding unwanted latencies.

This section will review four specific solutions on resiliency to failures, where paragraph 3-3-1 is focused on the resiliency at the control layer, paragraph 3-3-2 to 3-3-5 give more insight in failure recovery[2].Section 3-3-6 summarizes the solutions and gives an overview of used components on strengths, frailties and availability.

3-3-1 Replication component

In [38] Fonseca et al. showed how the master-slave capabilities of the OpenFlow protocol can be utilized. This indicates that a Primary Controller (master) has control over all switches

[2]Section 3-3-2 to 3-3-5 discuss succeeding research performed by Ghent University, Department of Information

and on the master failure, a Backup Controller (slave) can take over control of the switches. The developers of the replication component [38] came up with a solution, indicated in this review as CPR, which integrates a replication component into a general OpenFlow controller. As replication component a *Primary-Backup* protocol is applied, to offer resilience against failures and a consistent view of the latest fail free state of the network, without emphasis on the performance requirements. The Primary-Backup protocol synchronizes the state of the Primary Controller with the Backup Controllers. In CPR two phases are distinguished, being the replication and recovery phase. During the replication phase, computed flows by the Primary Controller are synchronized over the Backup Controllers. After failure detection of the Primary Controller, the recovery process is initiated to reallocate a Primary Controller and restore flow computations. With the replication component integrated to an general OpenFlow controller, the solution can be classified as a $T(N/-/C^{+R}/0)$ topology. Hereby we denote that always a single Primary Controller is present, with $C-1$ Backup Controllers. The current implementation of the OpenFlow protocol allows a total of $C = 4$ controllers. In figure 3-10 the synchronization process of the CPR module is shown in the graphical framework.

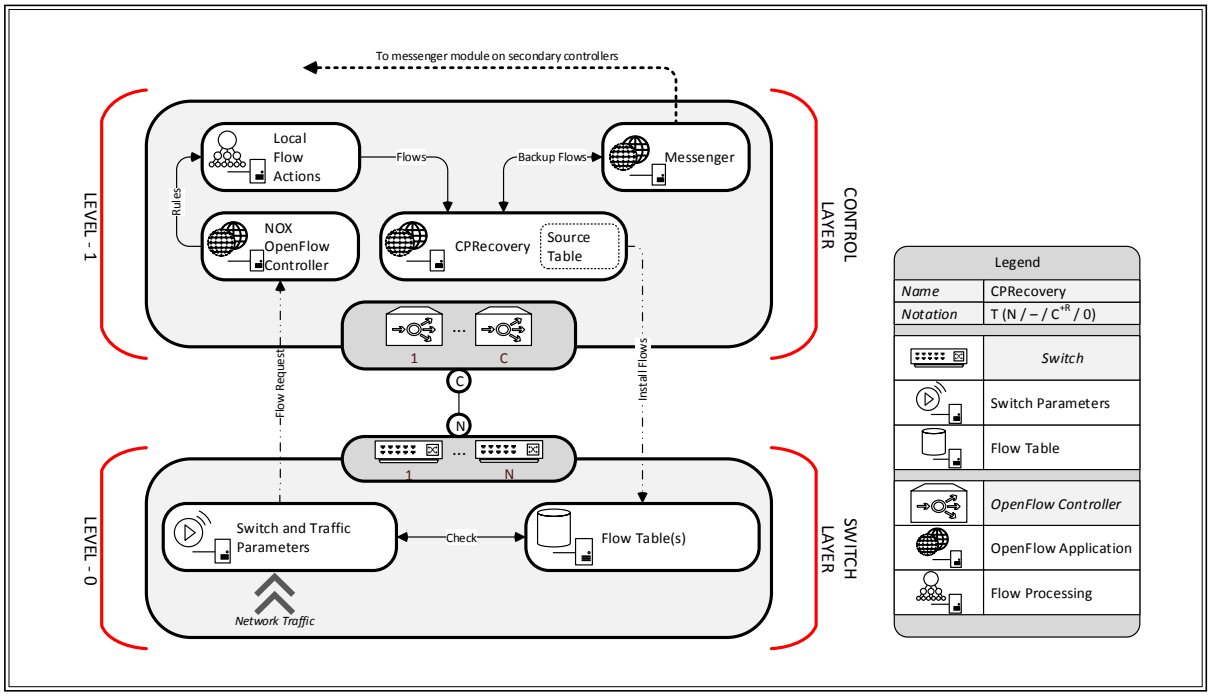


Figure 3-10: Replication component in a standard NOX controller - *A standard NOX-controller is enhanced to replicate flow installs over multiple slave controllers using OpenFlow protocol 1.2 (and higher) and the CPR module.*

The *CPRecovery* module connects to OpenFlow enabled switches and is built upon the NOX OpenFlow controller. Additional components, to enable replication, are integrated into the NOX controller as applications. The switches are configured in the master-slave setting, allowing multiple controllers in a predefined listing. During the replication phase, Flow Requests are sent from the switch to the primary controller. At the controller, the ordinary processes are executed for routing and forwarding purposes. After the flow is computed in the area flow scheduler, it is intercepted by the CPRecovery module. This module determines whether

the controller is assigned as primary and on positive identification the Flow Rule is added to the Source Table of the controller. Via the *Messenger* module, the source table of the backup controllers are synchronized using the Primary-Backup protocol. After all controllers are updated and synchronized, the Flow Rule is installed into the switches. The replication procedure enables a fully synchronized backup, before Flow Rules are installed to the switches. So when the Primary Controller fails, the second assigned Backup Controller can seamlessly take over network control. A drawback of this replication scheme is the additional latency introduced to the complete flow install process.

All OpenFlow switches can be configured to perform activity probing on the Primary Controller. If the primary controller fails to reply within configurable time window (τ), the network switch initiates the recovery phase and assigns the first Backup Controller from the list as Primary Controller. On the controller side, when a Join Request from the OpenFlow switches are received by a Backup Controller, this controller will set itself as Primary Controller and the replication phase is started. Update messages from the Primary Controller are also sent to the original Primary Controller and on its recovery, it is assigned as one of the secondary controllers.

The replication and recovery process seem to solve the resiliency problem with OpenFlow controllers, but we believe that the Primary-Backup protocol and recovery phase will fail in case of a temporary network partitioning and geographically separated controllers. To explain the potential flaw, the example topology $T(6/-/2^{+R}/0)$ of figure 3-11 is used.

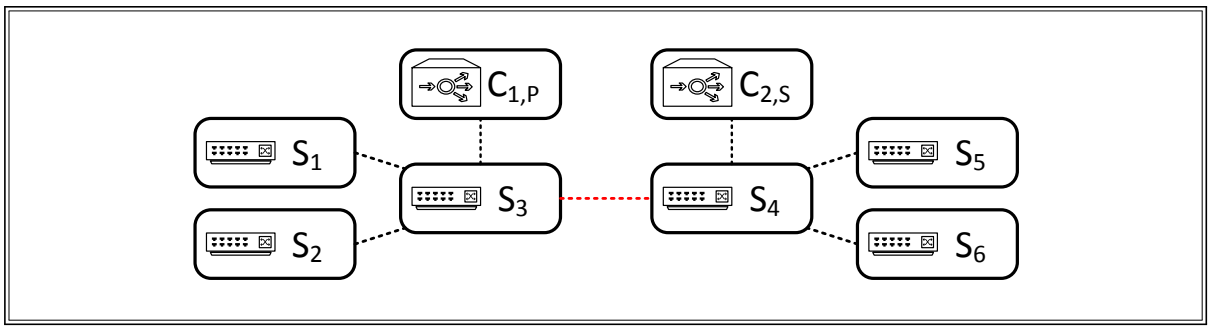


Figure 3-11: Example topology for network partitioning - *On a link failure between S_3 and S_4 the topology is partitioned and the Backup Controller $C_{1,S}$ will be assigned as Primary Controller by switch S_4 to S_6 using the Primary-Backup protocol in the recovery phase.*

On normal operation, controller $C_{1,P}$ is assigned as Primary Controller and controller $C_{2,S}$ as secondary (backup) controller. At time t the link $S_3 - S_4$ becomes unavailable and the network is partitioned into two components by the following procedure. Switches S_1 to S_3 are under control of the original primary controller, where the remaining switches (S_4 to S_6) will select the secondary controller as new controller, as the time window on activity probing expires on $t + \tau$. We question the behavior of the replication and recovery phase of the replication component (and the Primary-Backup protocol) in case link $S_3 - S_4$ becomes operational. Switches S_4 to S_6 will not re-assign to the original Primary Controller, so the network topology is partitioned until failure of the controller C_2. In [38] by Fonseca et al. no specific measurements are performed on influence on geographical positioning of OpenFlow controllers and its secondary problems, like Flow Rule synchronizing and Primary Controller selection. To solve this problem, a more advanced synchronization scheme is required, with

Primary Controller propagation and election schemes.

To test the functionality and the performance of the replication component, two simulations have been performed in [38]. In the first simulation the packet delay between two hosts in a tree topology with the Primary and Backup Controller connected to the top switch are measured. At specific times the Primary Controller is forced into a failure state. Where the packet delay on average equals 20 ms, the delay rises to approximately 900 ms during the recovery phase. After the rise, the packet delay normalizes to average and the network functions normally. Although the replication phase is successful, the delay during the recovery phase is too large for carrier grade networks providing voice services, which require delays with a maximum of 50 ms.

The second experiment measured the response time of a Flow Rule install. Therefore multiple measurements have been performed using the number of secondary controllers as variable. As described earlier, the CPRecovery module first synchronizes the secondary controllers, before installing a computed Flow Rule into the switch Flow Table. The measurements show that the response time increases linearly with the number of secondary controllers, with a minimum response time of 8 ms when no Backup Controller is configured and a maximum of 62 ms with 3 secondary controllers to synchronize. The linear expansion of the response times is unacceptable for data networks. We propose two modifications to increase the performance of the CPRecovery module and to lower the response times. The modifications holds the parallel execution of Flow Rules installation and synchronization to secondary controllers, or perform Flow Rule installation first and perform synchronization afterward.

3-3-2 Link failure recovery

Research performed in [39] by Sharma et al. has the aim to deploy the SDN philosophy to carrier grade networks. These network have high demands, looking to reliability and availability, which can be translated into network requirements, such as fast switchover times and minimal packet loss on network failures. Carrier grade networks must recover from a network failure within 50 ms, without impacting provided services. These are high demands, looking to the process-flow of the OpenFlow protocol and its current limitations. To guarantee performance of the network, two resilience mechanisms can be applied to network, being protection and recovery mechanisms. Protection mechanisms configure pre-programmed and reserved redundant paths at the forwarding plane in case of failures. On failure detection, no additional signaling is required with the control logic and the reserved path is directly available. This makes protection a proactive strategy to increase resiliency of networks. The recovery mechanism can either be a proactive or reactive strategy, where the proactive recovery strategy is similar to the protection mechanism, but requires no reserved resources. The reactive recovery strategy constructs paths on demand and dynamically based on current topology status. More details on protection and recovery are given in chapter 4.

In [39] three existing switching and routing modules (L2-Learning, PySwitch, Routing) from the NOX-controller are compared as recovery mechanisms. Additionally, a predetermined recovery mechanism is added. In the following list the modules are discussed shortly on the ability to recover links.

- *L2-learning* - The standard switching module of the NOX controller, which has similar functionality in comparison with a regular Layer-2 switch (Section 2-1). In the used

NOX-controller, the L2-Learning module lacks the Spanning Tree Protocol (STP), so in a network topology where switching loops exist, the module is not usable, as the flooding process will run for infinitely;

- *L2-learning PySwtich* - The functioning of this module is very similar to the standard L2-learing module. It is extended with two mechanisms to improve its performance. The first implemented extension adds Aging Timers (Section 2-4) from the OpenFlow protocol [1] to the installed Flow Rules, so that the switch autonomously can remove and update its Flow Table. Every time a switch processes a packet, the time-stamp of the flow rule is update. To protect the Flow Table, the hard-time must be larger than the idle-time. The second mechanism is the application of the STP [11], to remove networking loops in the topology;

- *Routing* - The routing module uses three mechanisms to compute Flow Rules. To enable routing functionality, the control logic must maintain network topology for path computations. To detect connected switches and link failures, on regular basis, an OpenFlow switch can sent Link Layer Discovery Protocol (LLDP) packets, containing information about the switch MAC-address, port number and VLAN-indicator [40]. A receiving OpenFlow switch replies with a LLDP-packet with its own parameters. When the reply packet is received by the corresponding switch, the assigned controller is informed about the detected link and the topology is updated. From the topology, the routing mechanism can compute and install paths between all switches and end hosts. The recovery capabilities of the routing module depends on the discovery mechanism and the set timeout interval. If a LLDP-packet is not received within the set interval, the switch declares the link lost and informs the controller of the status change;

- *Pre-determined* - The pre-determined module does not rely on learning and discovery mechanisms, but it implements path protection at the control layer. A network manager provides the controller with multiple redundant static paths. Based on priority and available paths, the controller chooses a path and installs it to the OpenFlow switches. On network failure detection, the controller can choose a redundant path from the provided paths, reducing the need for link discovery mechanisms and additional path calculations. As the network manager provides paths, no spanning tree protocol is needed to protect the network (assuming that the paths are checked on loops).

The first three mechanisms work dynamically and provide path restoration, where as the fourth mechanism is especially designed for path protection. When this module is used, the topology can be classified as $T(N/-/C^{+R}/0)$, because additional logic is added to the controller to improve its performance on link resiliency. Figure 3-12 projects the four recovery mechanisms to the graphical framework.

From the OpenFlow module at the switch, Flow Requests and link status information is exchanged with the controller. In figure 3-12 the four identified modules are drawn, but only one module will be active. Furthermore, the routing and PySwitch module are marked, to indicate the availability of the STP protocol. Each of the modules can determine Flow Rules, based on the available information. The L2-learning and Pre-determined modules install Flow Rules without aging timers, where the routing and PySwitch modules set timers to protect Flow Tables at the switch. In [39] simulations have been performed on a $T(6/-/1^{+R}/0)$

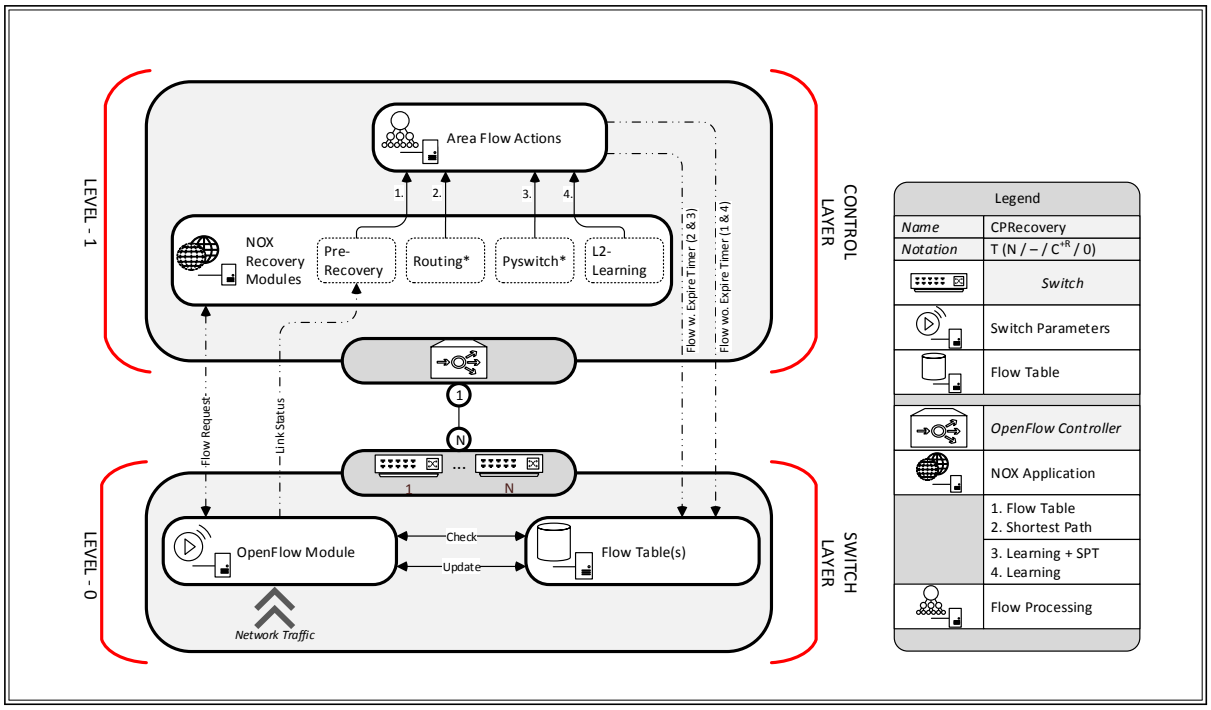


Figure 3-12: Recovery mechanisms projected to the graphical framework - *In total four mechanisms are available to recover from a link failure, where the Pre-Recovery module is especially designed for recovery. All mechanism have their own link failure detection methods. On link failure, the enabled mechanism will construct a new path and install these into the switch flow table*

topology to show the behavior of the different modules on network failures and measurements have been taken to identify if link restoration requirements for carrier grade networks are achievable. The implemented topology contained multiple loops and Sharma et al. [39] found that much traffic is traversing between OpenFlow switches and the controller to maintain link status information. Only the Pre-determined module uses less control traffic, which is expected from its static and fixed design. To simulate network traffic, ping packets are sent with an interval of 10 ms between two end-hosts. On specified times, a link failure was initiated and the recovery time, as well as the number of dropped packets are measured. Experimental results show that it takes 108 ms for a link failure detection to be indicated on the controller. This value is already above the required 50 ms, so any recovery mechanism discussed in this research will fail. The Pre-determined module acts immediately on a link failure, which resulted in recovery times of approximately 12 ms, resulting in a total recovery delay of 120 ms. Results for the routing and PySwitch modules show that recovery is dependent on the set aging timers. The L2-Learning mechanism fails recovery of a path without the application of Address Recovery Protocol (ARP) [41]. ARP is a Layer-3 protocol, which keeps a table with MAC-address to IP-address translations. To protect ARP-table, idle timeouts are configured and on expiring an end-host will re-initiate an ARP request. Until the corresponding destination is found, ARP-requests are flooded over the network. The flooding process triggers the L2-Learning module to update the forwarding table and a link failure can be recovered. A similar process is embedded using LLDP packets, where a link

is removed from the topology after a timer expires [3]. Table 3-3 summarizes properties and results of the four identified recovery mechanisms, where a distinction is made on how actual topology information is maintained, how the mechanism recovers link failures and on what time scale paths are recovered.

Name	*Update forwarding table*	*Recovery scheme*	*Recovery Time*
L2-Learning	Traffic / ARP	- / ARP	Seconds - Minute
L2-Learning PySwitch	Traffic / ARP	Aging timers / ARP	Seconds - Minute
Routing	LLDP	Aging timers / LLDP	Seconds - Minute
Pre-determined	Manually	Configured	MilliSeconds

Table 3-3: Comparison of properties and results on link recovery mechanisms

Unfortunately, no experiments have been performed by Sharma et al. on varying the idle and hard times of the routing and PySwitch modules. Reducing these times, we believe, will influence the recovery process positively. Also changing the default timeout timers of ARP and LLDP can improve the performance. The current implementation of the Pre-determined module acts fast on a link failure detection, but lacks the ability of constructing paths on demand by using dynamically received topology information. In [39] the scale of the topology is limited and it is easy for a network manager to create forwarding tables by hand. To enhance the recovery capabilities of the standard OpenFlow modules, an additional mechanism for fast link failure detection and propagation is required. For example, the routing mechanism is fed with actual link failure information and on failure, new shortest paths are constructed and pushed to the switches. This modification will probably lower the recovery time from *Seconds-Minute* to *Millisecond* order.

3-3-3 Recovery requirements on carrier grade networks

The research discussed in [39] was based on reactive recovery, where alternative paths are constructed and provided after link failure detection. As showed, the reactive and pre-determined recovery scheme implemented at the control layer does not meet the 50 ms recovery time requirement for carrier grade networks. In [4] Sharma et al. come with a similar proposal, but now applied on the switch layer. This reduces recovery time, as no communication with the control layer is required. Before more information is given about [4], more insight is given on recovery schemes of (carrier grade) networks. In total two schemes are distinguished [37], being protection and restoration schemes. The differentiation between the schemes is made on the time-frame on which a backup path is provided to the data plane. If a backup path is provided pro-actively, and before a node or link failure occurs, a scheme is defined as a protection technique. A reactive scheme provides backup paths after failure detection and is defined as a restoration technique. In the following enumeration multiple protection schemes are discussed. The notation for the schemes is in the form of $P - B$, where P is the number of primary paths and B the number of backup paths.

- *1 + 1* - A protection scheme where the primary and backup path are both active. This means that data is transmitted over the primary, as well as over the backup path. On

[3] The default timeout interval for ARP and LLDP equals 60 seconds.

failure of the primary path, the receiving host need to switch on incoming path and the path is recovered. Drawbacks of protection schemes are the increased bandwidth usage and additional logic at the receiving node to distinguish data packets from the primary and backup path;

- *1 : N* - A protection or restoration scheme where the primary path is recovered with a single path from N possible backup paths, where $N \in (1, 2, .., \infty)$. The backup paths are pre-configured in the data-plane for a protection scheme, where in the case of a restoration scheme the paths are provided by the control logic after detection of a failure. If $N > 1$, there are more backup paths available and load balancing is possible, but additional logic is needed to select a backup path;

- *M : N* - A protection or restoration scheme where M primary paths have N backup paths. This scheme is useful for traffic load balancing between two hosts. Traffic which normally travels over a single path is divided over M multiple primary paths to provide higher throughput. If one of the primary paths fails, a backup path must be selected. A major advantage of this scheme is the chance that there is no connection between two hosts is small, as $M + N$ paths must fail. On the other side, $M + N$ (disjoint) paths must be available between the hosts to make this scheme possible. This means that high demands are set to the network connectivity.

Sharma et al. [4] implemented the $1 : 1$ protection scheme as a protection mechanism at the switch layer. To enable protection, the Group Table concept of the OpenFlow protocol is utilized. In normal operation, the header from a packet is matched in the Flow Table and the packet is forwarded to the correct outgoing port or the packet is dropped. By applying Group Tables, a Flow Rule can also contain a link to a unique group. In the Group Table, one or more Action Buckets define actions based on status parameters. On change of these parameters, the Action Bucket(s) execute a predefined action. In case of protection schemes, when a failure is detected in the primary path, the backup path is enabled for the flow. For path failure detection the Bidirectional Forwarding Detection (BFD) [42] protocol is implemented in [4]. To monitor the complete path between multiple OpenFlow switches, a BFD session is configured between the entry and exit switch. If periodical messaging over the session fails, BFD assumes the path lost, updates the Action Bucket status in the OpenFlow switches and the protected path is activated[4]. The $1 : 1$ protection scheme implemented on a topology leads to a $T(N^{+R}/-/1/0)$ system.

The second implementation by Sharma et al [4] is the $1 : 1$ restoration scheme at the switch layer. An extension to the standard routing module of the NOX controller is made to increase the performance on resiliency. In [39] the failure detection capabilities of the routing module depends on the OpenFlow aging timers and the implementation of a topology module incorporating LLDP packets. The extended routing module uses the *Loss of Signal* (LoS) failure detection mechanism available in the OpenFlow protocol. LoS detect port changes in the switch from *Up* to *Down* and reports these to the controller. Other than BFD, LoS does not monitor complete paths, but only local links at the switch. On link failure detection, a notification is sent to the routing module and a new shortest path is constructed, without incorporating the failed link. The new shortest path with its corresponding Flow Rules is

[4]More details of the BFD protocol is given in section 4-3.

installed in the switches, after which the path is recovered. The proposed solution for path restoration is classified as $(T(N/-/1^{+R}/0)$.

Before looking to results we can state that the protection scheme will probably restore a paths faster, as no communication is required with the controller, as well as path computations. The restoration scheme is more adaptive and flexible, as paths are computed with status information of the current topology. In a large networks, both schemes can be applied, depending on network services provided. A combined scheme can be classified as a $T(N^{+R}/-/1^{+R}/0)$ topology, offering a flexible recovery mechanisms high and low priority services, where the graphical representation is found in figure 3-13.

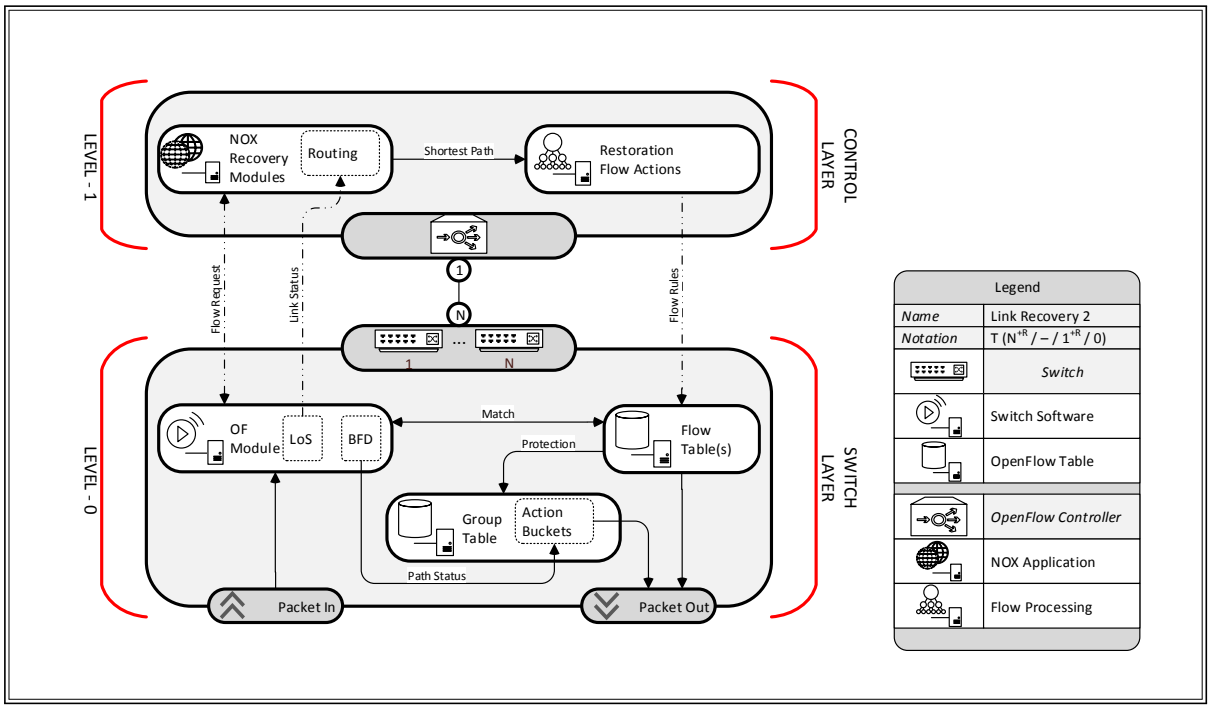


Figure 3-13: $1:1$ Recovery scheme projected to the graphical framework - *Two schemes are visible to recover from a failure. The protection scheme utilizes the BFD path failure detection protocol in cooperation with Group Tables and Action Buckets to enable* $1:1$ *path protection. Restoration of failed links is executed by the controller and a modified routing module which uses LoS for link failure detection. New constructed paths are installed to the switches, without incorporating the failed link*

In normal operation, the process of packet forwarding is similar to the general OpenFlow operation. On protected paths, the BFD protocol monitors path status and on failure the Action Buckets in the Group Table are updated. Actions defined in the Action Buckets, activate the protected path. In case of restoration with LoS, the OpenFlow module monitors a link failure, after which the routing module in the controller computes a new shortest path. An important aspect of the recovery process is the latency between time of link failure detection and the recovery of all affected flows. In [4] an analytical model is given for the restoration process. It gives a good indication where latency is introduced in the recovery process. The model has been extended with the protection scheme, to indicate the differences between both schemes.

$$T_R = T_{LoS} + \sum_{i=1}^{F}(T_{LU,i} + T_{C,i} + T_{I,i}) \tag{3-1}$$

$$T_P = max(T_{BFD,1}, ..., T_{BFD,N}) + \sum_{i=1}^{P} max(T_{AB,1,i}, ..., T_{AB,N,i}) \tag{3-2}$$

The total restoration time (T_R) is determined by the Loss-of-Signal failure detection time (T_{LoS}), the total time spend at the controller to look up the failed link (T_{LU}), the path calculation time (T_{CALC}), the flow install / modification time (T_I) and the number of flows (F) to restore. In here the propagation delay is removed from the original model and the propagation delay is joined and integrated with the failure detection and flow installation time. Hereby we assumed that the propagation delay is small ($<$ 1ms). The protection model is dependent to the BFD failure detection time (T_{BFD}), the time to process the action bucket (T_{AB}) and the number of flows affected by the link failure (P). Because a broken flow is only restored after the processing of the *slowest* of N switches in the path, the max operator is applied. To give more insight in the time flow of both recovery schemes, the analytical model is graphically represented in figure 3-14.

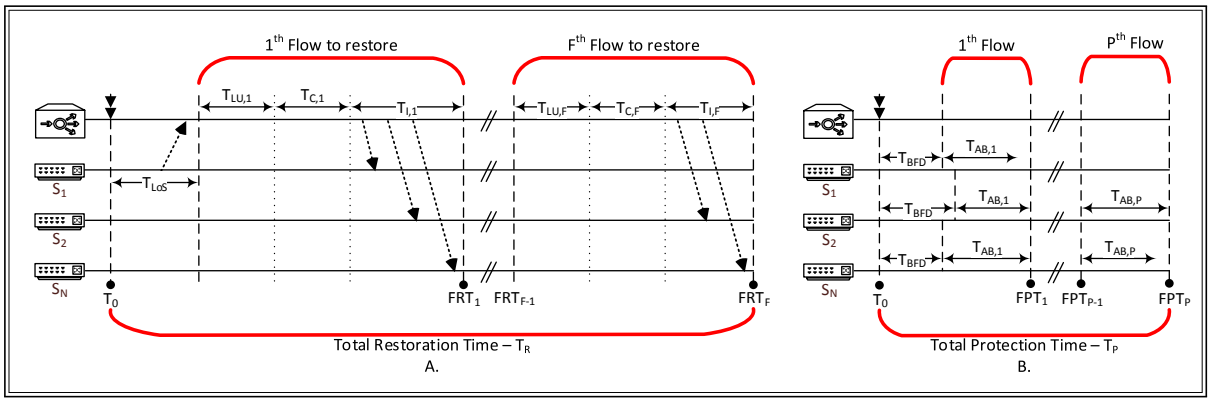


Figure 3-14: Analytic model for recovery process - *Figure 3-14A is adapted from [4] and represents the analytical model for the restoration process, where T_0 indicates the time of link failure, and the dotted arrows indicate the propagation time between switch and controller and FRT_x indicates the flow restore time. Figure 3-14B shows the process in time for the protection scheme, where each switch in the primary path receives a link failure indication and the Action Buckets in the Group Table are updated. The flow protection time (FPT_x) indicates the time path recovery.*

The switches in the protection scheme (figure 3-14B) can update simultaneously on failure detection and the link is recovered if the slowest switch completed its task. If the link failure influences more paths on a switch, it must perform the actions in the Group Table bucket multiple times in serial order for total recovery. Paths without protection are restored by the controller. In figure 3-14A the influence of the look-up process, calculation and flow installation is clearly visible. For each flow the controller must perform the actions, so when a large number of paths and flows are affected by a link failure, total restoration time will increase. We think this flow restoration process can be optimized by performing restorations in parallel. Also the behavior of the routing module is not clear. For example, when a link

between two switches fails, both switches will inform the controller about the failure. How the routing module behaves on this aspect is unclear from [4].

To give an indication of latency differences, multiple simulations and measurements have been performed on different topologies in [4]. Results are given in table 3-4, where the emphasis lays on the magnitude on the measurements and not the exact value. Delay times for to process Action Buckets are unknown and assumed in the order of several milliseconds at maximum.

Time	*Symbol*	*Delay (ms)*	*Relation*	*Comment*
Failure detection time (P)	T_{BFD}	40 - 44	Fixed	
Failure detection time (R)	T_{LoS}	100 - 200	Fixed	
Controller look-up time (R)	T_{LU}	1 - 10	Linear	Delay with 250 - 3000 flows
Path calculation time (R)	T_{CALC}	10 - 200	Linear	Delay with 25 - 300 paths
Flow installation time (R)	T_I	1 - 5	Linear	Delay with 1000 - 10000 flows

Table 3-4: Comparison of time delays in link restoration and protection

Both recovery schemes where able to recover paths on link failure detection. The main difference in performance is found in the failure detection mechanism. Where BFD only need 40 ms to detect a path failure, the LoS mechanism requires more than 100 ms to report a broken link. The main disadvantage of active path monitoring with BFD in the protection scheme is the introduced overhead for monitoring all protected paths. Also, fixed pre-planned configurations are inflexible and the during the performed experiments link failures did not influence the backup path for the protected paths. Restoration is more flexible by allocating restoration paths dynamically with up-to-date network topology information. Recovery times for both schemes are mainly dependent on the number of flows to recover paths in the network. Restoration times can exceed to 1000 ms, if a large number of flows need recovering. So, it is difficult to point the optimal recovery scheme for OpenFlow and SDN networks. If high requirements are set on recovery times, a $1:1$ protection scheme with active link monitoring is recommended. Otherwise, the $1:1$ restoration scheme has proven to restore network traffic within reasonable boundaries.

We think there are more recovery schemes worth investigating. A large performance increase can be achieved in the restoration scheme with the application of faster link monitoring, like BFD. Also a pro-active scheme, where the controller regularly updates the Flow Tables at the switch with actual primary and one or more backup paths. In this way, the advantages of both schemes are applied and a fast dynamic $1:N$ recovery scheme is available. A major downside of this approach is the increase of control and monitoring data between the switches and the controller, as well as increased OpenFlow table sizes at the switches.

3-3-4 OpenFlow-based segment protection in Ethernet networks

In [4] protection paths are pre-configured and triggered by monitoring protocol, when the path between source and destination is monitored down. Research performed in [43] by Sgambelluri et al. followed a similar approach, where not complete paths are protected, but only individual segments of a path. For failure detection a new mechanism was developed

and implemented to the OpenFlow protocol. The main idea of the research is to provide primary paths, as well as backup paths for each switch invoked in the primary path. Both paths are installed in the Flow Tables with different priorities and after failure detection, Flow Rules for the primary path are removed from the Flow Tables by additional mechanism in the OpenFlow protocol, after which the backup path becomes the primary path. In figure 3-15 the projection to the graphical framework is given (note the overlap with figure 3-13).

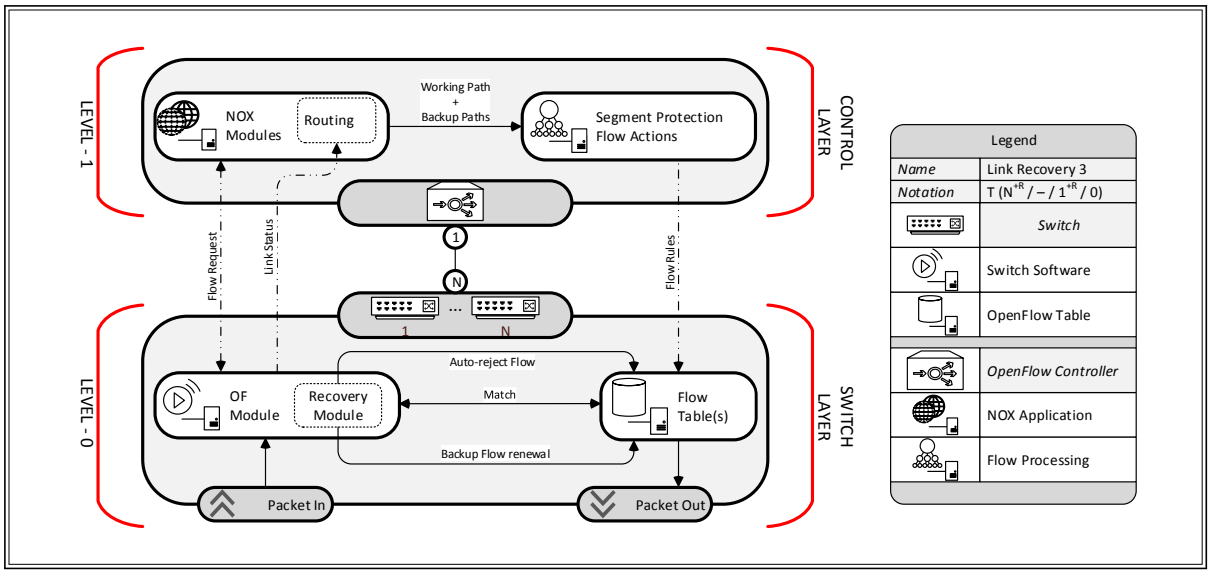


Figure 3-15: OpenFlow segment protection scheme projected to graphical framework - *Along with the primary path, backup paths are provided to the OpenFlow switch to protect segments. A extended Recovery Module in the OpenFlow module reject flows from the Flow Table after failure detection and activates backup paths for the segments. To prevent Flows Rules for the backup paths from removal by the idle timers, the recovery module transmit renewal messages over the backup segments.*

The failure detection mechanism in [43] is unknown and to trigger backup paths, two additional modules are added to OpenFlow protocol version 1.0. For removing the Flow Rules for the primary path from the Flow Tables, an *Auto-Reject* mechanism is developed. This mechanism deletes entries when the port status from the OpenFlow switches changes. The second developed mechanism is *Flow Renewal,* which is used to update flow entries for the backup segments. Backup segments are installed using idle timers and as long the primary path is active, update messages are sent over the backup segments to update the idle timers, preventing automatic flow removal.

Multiple experiments have been performed by Sgambelluri et al. [43] with the adapted version of OpenFlow, utilizing segment protection with the Auto-Reject and Flow Renewal mechanisms. Results show average recovery times around 30 ms with a maximum of 65 ms with a variable number of flow entries per switch. Because i) the applied failure detection mechanism, ii) the failure initiation method is unknown, iii) the measurements are performed on a simulated network and iv) the resolution of the measurements is low (2 ms), no firm conclusions can be drawn from the research. Besides that, an old version of OpenFlow is applied and the two additional mechanisms implemented have similar operations as the Fast Failover groups from OpenFlow protocol version 1.2. The fact that modifications and extensions must

be made to the OpenFlow protocol, leading to a non-standard implementation, makes that we do not recommend the found solution in [43] for large SDN implementations. The idea of segment protection is an improvement over path protection. A solution where the recovery mechanism is further localized to link protection is worth further investigation. Here the mechanisms for fast link failure detection at the switch and routing algorithms for link protection are required. In [43] examples of the routing module are given, but no information on the used algorithms is provided.

3-3-5 Failure recovery for in-band OpenFlow networks

The last proposal discussed in the SDN resiliency section is an extension to [4] (discussed in section 3-3-3) and distinguishes on how the controller is connected to the OpenFlow switches. In [4] the controller was connected in an *out-of-band* configuration, which indicates that separate connections from the switch to the controller are available. Only control traffic transverses over these connections, ensuring no delays or traffic congestion between controller and switches. In an *in-band* configuration, control traffic transverses over the same connections as data traffic. No additional network interfaces are needed, resulting in less hardware complexity. With the application of an in-band configuration to a network, Sharma et al. [44] discovered a problem. When a link failure occurs and the communication between a switch and controller is lost, the basic operations for an OpenFlow switch are to restore the connection by requesting a new connection after waiting for an echo request timeout [1]. The minimum value for this timeout is limited to 1 second, which is a magnitude of 20 too long for proper path recovery on carrier grade networks. Therefore in [44] the restoration and protection schemes from [4] are reapplied to solve the identified problem.

As discussed before, data traffic paths can be restored or protected from a link failure. In case of restoration, the failed primary paths cannot be restored without communication between switches and the controller. Therefore, first the control path must be restored, after which the primary paths can be reinstalled on the switches by the controller. In order to implement this priority of processing the link failure, the *Barrier Request and Reply Messages* concept from the OpenFlow protocol is utilized. In normal operation, OpenFlow messages can be reordered by switches for performance gains. To stop reordering, Barrier Request are sent by the controller and the switches must upon receiving a Barrier Request, process all leading messages before processing the next Flow Request. After processing of the Barrier Request, a Barrier Request Reply Message reply is sent to the controller. To clarify the restoration process in an in-band configuration, an example topology with process description is given in figure 3-16.

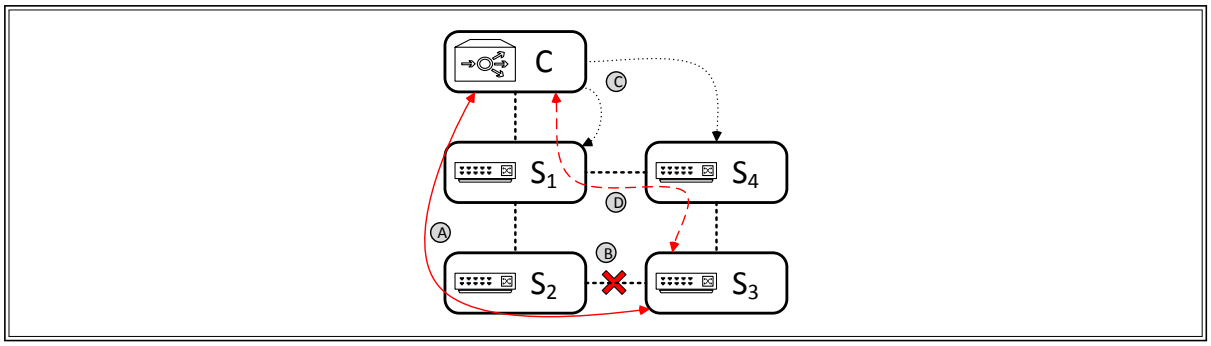


Figure 3-16: Example configuration for control traffic restoration - *A. Normal situation. B. Link failure and controller notification. C. Update intermediate switches to restore control path. D. Communication between $S3$ and controller is restored.*

In total four phases are distinguished from normal operation to link failure detection and restoration of the control channel for switch S_3 (example of figure 3-16):

- *Phase A* - Initial phase where the control traffic for switch S_3 is routed over switch S_1 and S_2 to the controller. In a normal out-of-band configuration, S_3 would have a separate network connection with the controller;
- *Phase B* - The link between switch S_2 and S_3 fails and the communication between S_3 and the controller stops. Switch S_2 monitors the link failure with the LoS mechanism and sends an error report to the controller via S_1;
- *Phase C* - The controller computes a new control path over S_1 and S_4 to S_3 with highest priority, where after new primary paths are recomputed. These paths cannot yet be installed into switch S_3, as the broken control path is still present in the Flow Tables. Therefore the controller first updates S_1 and S_4 with the addition of Barrier Requests;
- *Phase D* - The flow modification messages are processed and the Reply Message is send to the controller by S_1 and S_4. After both Barrier Reply Messages are received by the controlled, the new control path to switch S_3 is configured in the intermediate switches. The connection to the controller is restored and the recomputed primary paths can be installed to all switches.

As seen in the description of the phases, the use of Barrier Requests prevents processing of primary path restoration at the switch and restoration of paths between controller and switches has highest priority. Besides restoration, the control traffic path can also be recovered by a $1:1$ protection scheme. Protection of the control and primary paths is provided by BFD link failure detection and the Group Tables with Action Buckets. Main advantages of protection is that no communication is required with the controller and switches can autonomously update their Flow Tables. The protection scheme is equal to the scheme explained in section 3-3-3, so control and primary paths are protected by the pre-configured Group Tables.

Using the restoration and protection schemes, a total of four recovery schemes are possible. As with the out-of-band configuration, analytical models can be used to predict the behavior during the recovery process. The restoration model for the control traffic path is

a modification to the earlier restoration model. We developed model 3-3 and 3-4 to show the restoration times for an OpenFlow in-band configuration, where T_{RC} is the control traffic restoration time, T_B is the additional time delay introduced by the Barrier Message replay mechanism and $T_{IS,i}$ is the time to install and modify Flow Tables of intermediate switches. The restoration time for data traffic paths (T_{RD}) is a simplified from model equation 3-1, without LoS failure detection delay as the controller already is informed about the network failure.

$$T_{RC} = T_{LoS} + T_B + \sum_{i=1}^{S}(T_{LU,i} + T_{C,i} + T_{IS,i} + T_{I,i}), \tag{3-3}$$

$$T_{RD} = \sum_{i=1}^{F}(T_{LU,i} + T_{C,i} + T_{I,i}), \tag{3-4}$$

In table 3-5 the four identified recovery schemes are given, together with our analytical delay models.

Recovery Scheme (Control - Data)	*Symbol*	*Analytical Relationship*
Restoration - Restoration	$T_{R,R}$	$T_{RC} + T_{RD}$
Restoration - Protection	$T_{R,R}$	$max(T_{RC}, T_P)$
Protection - Protection	$T_{P,P}$	T_P
Protection - Restoration	$T_{P,R}$	$max(T_P, T_R)$

Table 3-5: Comparison of recovery schemes in in-band configurations

The analytical relationships in table 3-5 are assuming that the recovery and protection process do not influence each other at the switch. To clarify the models and relationships, the Restoration-Restoration scheme is applied to the example of figure 3-16 in a time flow model of figure 3-17.

It is clearly visible that primary path recovery must wait on restoration of the control paths to the switches. With a large number of control paths to restore, the complete restoration time will increase rapidly. The application of protection to recover the control or data path, removes the serial behavior. In [44] Sharma et al. performed multiple experiments on all four in-band recovery schemes. Results show that with the application of restoration schemes to recover primary paths, delays exceed the 50 ms requirement. Only the full protection recovery scheme meets the requirements, but in practice this scheme will not be applied due to large Flow Tables and the large number of configuration which have to be made by the network manager at the switches. The full restoration process achieves to recover the control and primary paths in the used topography within 200 ms, which is well within the 1 second time window of the regular OpenFlow restoration time. Applying LoS failure detection and Barrier Messages is thus an improvement over normal restoration. Looking to the differences in the results between $T_{R,R}$ and $T_{P,R}$, we can conclude that the performance difference is small[5]. This is expected, as only a few control paths need recovery. With this conclusions, we can

[5]Results show a T_{LoS} of approximately 50 ms in comparison with 100 ms measured in [4]

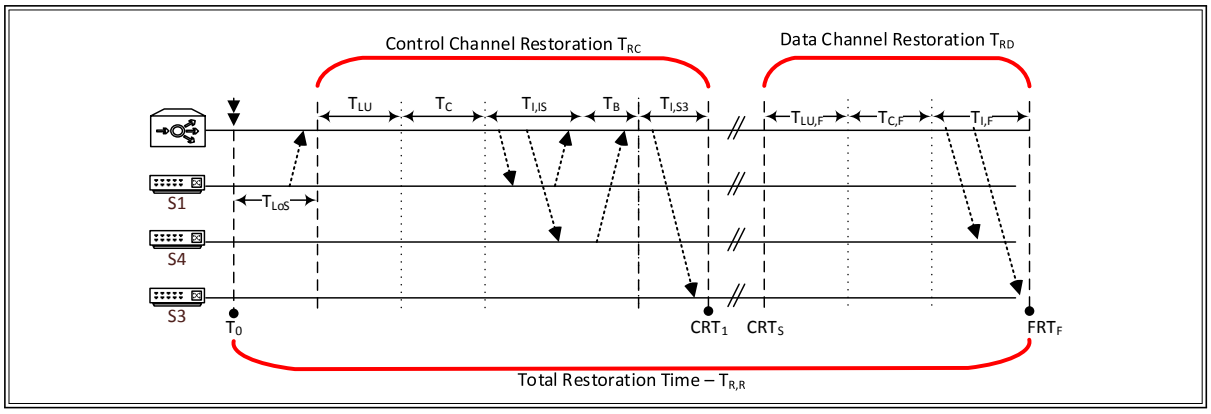


Figure 3-17: Analytical model for a total restoration process in an in-band configuration - *The communication and process order to restore the control and data traffic path. At the control restoration time (CRT_S) the connection between controller and switches is restored, where after the flows are restored.*

state that for recovery requirements there is no noticeable performance difference between in- and out-of-band configurations, when a full protection scheme is implemented. A comparison between restoration in both configurations is not possible due to the differentiation in delay in the measurements of [4] and [44]. It is possible to predict the behavior for both configurations with the derived analytical models, when reliable measurements are present for the defined parameters.

3-3-6 Conclusion on resiliency

In total four proposals are reviewed and a separation is this research field is identified. Where Fonseca et al. [38] is focused on the robustness on the controller side and developed a mechanism to utilize the master-slave concept for OpenFlow controllers, [39, 4, 44, 43] researched the ability to recover failed links. On the controller side we can state that "One controller is no controller" in terms of redundancy and robustness. With the failure of a controller the OpenFlow switches will loose the control layer functionality, resulting in an uncontrolled network topology. The found solution in [38] was functional, but came with a costly price of additional latency. On link recovery, four papers are discussed, where recovery concepts start with traditional mechanisms using aging timers, like the ARP protocol and LLDP packets, to recover failed links. These methods have proven to have a large latency and therefore active link failure detection mechanisms are introduced. With these mechanisms, switches and controllers receive link failure notifications with a small latency. Group Tables with Action Buckets and fast restoration of Flow Rules by the controller are two proven techniques, which have shown to recover paths within the order of $50 - 200$ ms. The problem remains that only one controller is available. An integral solution, where OpenFlow controllers are redundantly applied, together with the ability to recover paths in millisecond order is worth further research. Techniques used in the scalability research field can be joined with the OpenFlow protocol to apply master-slave configurations, where the path recovery schemes can be reused in a modified way for faster path recovery.

3-4 Security in SDN

The third research field within the SDN community is security. Network security is applied to control networks access, provide separation between users and protect the network against malicious and unwanted intruders. It remains a hot topic under SDN researchers, because a basic security level is expected from a new network technology, as well as the fact that network security applications can easily be applied to the network control logic.

3-4-1 General OpenFlow security

Before going into details, two levels of security are defined. The first level invokes logical connections between end hosts inside the network. Protocols like Secure Socket Layer (SSL) or packet encrypting techniques must ensure connection security. Within SDN, this level of security plays an important role, as the control connection between switches and the centralized controller must be ensured. The OpenFlow protocol [1] provides a mechanism to secure this connection, but is not required as start condition. It is up to the controller to secure the connection with OpenFlow switches and a number of controller implementations have not implemented link security mechanisms [45]. When no link security is applied, a malicious node can impersonate the controller and take over control of the switches. Network traffic can be re-routed for analysis and information extraction. Applying link security between the control and data layer is thus the first prerequisite which must be fulfilled to ensure integrity on the network.

The second level of security is centered to protect switches, servers and end hosts in the network. Numerous examples are present to indicate the threats to the network as a whole. Malicious software can intrude the network, infect hosts and gather information, but also flooding attacks can disable network servers or overload OpenFlow switches and controllers. Security mechanisms must be implemented on the network to detect malicious traffic and take necessary actions to block and reroute this traffic. In current networking state, network security is applied at higher networking layers. Routers and firewalls perform security tasks at layer 3, where end hosts and servers host security applications at layer 7. With SDN, there is a central authority which routes traffic through the network and enables the possibility to apply security policies to all layers in networking. Much research has been performed and the results of [46, 47, 48, 49] are used to determine security properties within SDN. All researchers follow roughly the same procedure to apply security to the network. The procedure consists on three steps where a short description of the process is given, as well as a reference to the performed research.

- *Classification* - Data flows through the network must be classified in order to determine malicious behavior and network attacks. Without classification it is impossible to protect the network and take according actions. The main source for traffic classification is found in traffic statistics [47];
- *Action* - Once a traffic flow is marked as malicious, the control layer must modify Flow Tables to protect the network and prevent propagation of the malicious traffic through the network. For each threat, different actions are needed, so the control layer must be flexible for quick adaption of new protection schemes [46];

- *Check* - The last process in the security process is the checking of computed flow rules with the applied security policy from the network manager. Flow rules may (unintentionally) disrupt the security policy and therefore an extra control process is needed. Preventing network security violations by checking Flow Rules before installation on the switches, completes the overall security process [48, 49].

3-4-2 Graphical framework

The three processes combined form the protection layer for the network and all can be implemented at the control layer, which results in a $T(N/-/1^{+S}/0)$ configuration. To give the most general view of this configuration, we assume no modifications to the switch layer. Figure 3-18 gives the general OpenFlow security configuration projected on the graphical framework.

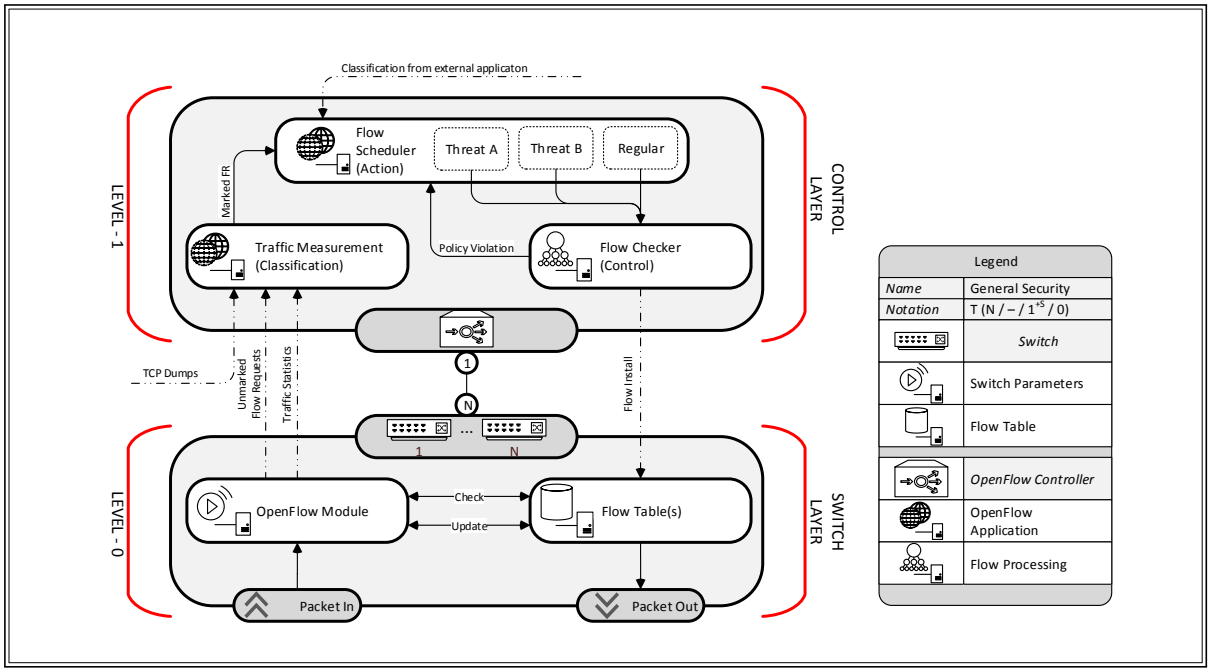


Figure 3-18: General security configuration projected on graphical framework - *The first step is to classify and mark incoming traffic with statistics from the OpenFlow switch or external measurements (Classification). Marked Flow Requests go to the Flow Scheduler where the requests are processed and according actions are assigned to flows (Action). The last step is to check the flows to the network security policy (Control) and install the flows at the switches. On policy violation the Flow Scheduler must be informed.*

As seen in figure 3-18, normal Flow Requests and traffic statistics enter the classification module in the OpenFlow controller. Traffic statistics can originate from the OpenFlow module at the switch and result from processed TCP and UDP traffic dumps. The classification module identifies malicious traffic flows and has two actions to perform. First, it must inform the Flow Scheduler with the presence of malicious flows in the network. Existing flows in the switch Flow Tables must be modified by the Flow Scheduler. Second, incoming Flow Requests must be marked, so that the Flow Scheduler can process the Flow Requests according the set

security policy. At the scheduler, multiple security modules are present, to install and modify Flow Tables with rules based on the security policy for regular traffic and counter measures for the different threats to the network. So, for each traffic flow there exists a unique set of rules, in order to protect the individual traffic flows within the network, as well as the network itself. The last step before a Flow Rule can be installed or modified to check against is the overall security policy of the network. A computed flow rule by a module in the Flow Scheduler can confirm the rule of that module, but may violate the security policies of other modules and the overall network security policy. After a Flow Rule is approved by the Flow Checker, it can be installed into the switch Flow Tables.

In theory, the classification process looks easy to execute, but Braga et al. [47] and Shin et al. [46] have proven otherwise. In [47] an effective solution is found to identify abnormal traffic and flooding attacks. The most obvious mechanism to classify traffic flows are continuous TCP and UDP traffic dumps. With these dumps, all information is present to identify malicious traffic is present, but it takes much computing resources to process all the dumps contentiously. Therefore an intelligent mechanism is employed to map traffic flows based on samples traffic statistics. Using the maps, all flows are characterized and abnormal traffic flows can be identified and removed from the network. This method is only to detect flooding attacks, so to detect other threats, more classification mechanisms are needed. In [46] an application layer is presented to apply classification modules, as well as security modules at the Flow Scheduler. A general application layer eases up implementation of modules for newly identified threats. Multiple examples in [46] show that with the application of classification and security modules, counter measures for network threats can be implemented to an OpenFlow environment.

3-4-3 Conclusion on security

As seen in section 3-2 on scalability, traffic classification is also needed for security applications. Without classification, by traffic measurement modules or external dumps, it is difficult to process Flow Requests for traffic from unknown nodes in the network. With marked Flow Requests it is possible to separate threats from regular traffic and therewith protect the network. With [46] Shin et al. has given a general framework to develop security applications, which can be quickly adapted to provide counter measures against newly identified threats. The framework also enables regular OpenFlow switches to replace security middle-boxes, like firewalls, as discussed in section 2-3. Checking computed Flow Rules before installation seem not obvious, but is needed to ensure the security on the network, as shown by Son et. al [48] and Porros et al. [49]. Rules to protect against one threat may provide opportunities for other threats and malicious traffic to attack the network. The three security steps can improve the security on SDN and OpenFlow networking and can, if configured and designed well, replace current security middle-boxes. This all comes with a price, as much overhead is introduced. Collecting and transmitting traffic statistics can lead to high loads on OpenFlow switches, where controllers must process statistics, compute Flow Rules and must check these before installing to switches. All these tasks introduce unwanted latency and for larger networks can not be performed by a single controller. Separating tasks over multiple controllers over the network, where switches with external connections are controlled by advanced security controllers. A layered structure with security applied where needed, is the ultimate goal for security applications in SDN.

3-5 Conclusion on classification

In this chapter a number of proposals have been surveyed on how Software Defined Networking is applied to networks and which problems are counted for. The implementation of the SDN philosophy and OpenFlow protocol resulted in better manageable networks, but an optimal network solution is not yet found. Three problem areas are identified, being scalability, robustness and security, which are by definition not specific SDN problem areas. The problem areas are inherited from regular IP networks, where SDN has to cope and has the ability to enhance the network to cope with network specific problems. In order to compare the solutions found, a generic graphical framework was developed. Proposed solutions are projected to the framework to point out strengths, frailties and possible enhancements.

SDN networks are more susceptible to scalability problems, than current state IP networks. The main issue is to divide the workload to control and manage the network over SDN switches, multiple controllers or both. This task was normally performed at switch level. Robustness of networks can be increased with the application of proactive path protection schemes, where controllers provide backup paths, beside the regular primary paths. On the other side, robustness of SDN networks is reduced with the introduction of a single central point of authority. Application of redundant controllers in the network is therefore a must. On security level SDN offers the ability to transform a switch to a functional firewall, where network security policies are translated to switch configurations for network protection. To do so, traffic statistics from the switch layer must be available to the central authority (leading to increased control traffic overhead) and computed configurations must be checked on possible security holes.

3-5-1 Optimal Solution

We think an optimal solution for SDN networking does not exist yet, as trade-offs must be made for each implementation. Full control over the network results in large quantities of control traffic and additional latencies. Depending on the size and traffic load on the network, and the requested performance of the network topology we propose a hybrid solution, where regular tasks for trusted traffic are performed by the switches themselves and tasks for more advanced and unknown traffic requires additional configuration from control layer. We believe that for each requested task one must survey on which layer the task must be located and what impact it has in relation with scalability and resiliency issues. Developed modules and applications must be compatible with existing SDN protocols and OpenFlow controllers to avoid unwanted complexity and incompatibility.

3-5-2 Problem Focus

The focus for the remaining part of this thesis is on robustness in SDN networks. This because the solutions found for scalability problems give enough constancy for future implementations, numerous proposals and a general framework are given to enhance security and limited research has been performed on robustness in SDN networks. We believe there is room for improvement on path and segment protection, so that Ethernet switches can be utilized for high availability networks providing failover times achieving industrial standards. In the next chapter will discuss path protection in SDN networks in more detail.

Chapter 4

Path protection in SDN

In chapter 3 we surveyed a number of topics in SDN networking. We found that more attention is needed for increasing the robustness and resiliency of SDN networks. This chapter will give more background information on protection and recovery schemes in section 4-1 , where section 4-2 discusses failure detection mechanisms for SDN networks. Section 4-3 discusses the Bidirectional Failure Detection protocol (BFD) in relation to minimize recovery times, where 4-4 discusses how OpenFlow provides path failover functionality to improve robustness and resiliency against failures in SDN networks. This chapter concludes with thoughts and remarks on network survivability in section 4-6.

4-1 Network survivability

A basic requirement for SDN networks, where high availability is needed for the provided services, is resilience to failures in the network of any kind. A limited number of switch or link failures, may not lead to overall failure of the network. Therefore three basic requirements for a SDN network must be taken into account, with the main goal to increase the survivability of a network [37].

1. *Network connectivity* - The network is (well) connected, meaning that each node in the network has at least one link to another node in the network. In this assumption no differentiation is made between switches and hosts in SDN networks. Network connectivity is based on two parameters, being link and node connectivity. Link connectivity is defined as the number of links which can be removed to disconnect the network. Similarly, node connectivity is defined as the number of nodes to remove from the network to disconnect it;

2. *Network augmentation* - The process of defining new links in the network, with the main purpose to increase the link and node connectivity in the network. Augmentation is based on the requested level of survivability, failure rate of nodes and additional costs of extra nodes and links in the network;

3. *Path protection* - The procedure of finding (multiple) disjoint paths between two hosts in the network.

Network connectivity and augmentation are mostly needed during the design and implementation phase of a network. The network connectivity must provide the ability to find multiple disjoint paths in the network. If the connectivity is too low, disjoint paths may not be found and network augmentation is needed. In the remaining part of this thesis, we assume that the network connectivity for suggested topologies is fixed and no augmentation is needed. We will now focus more on the network protection procedures.

4-1-1 Path protection schemes

To protect a network, a backup path must be computed by the control logic of the switches and provided to the data plane. When a failure occurs, the main purpose of the path protection schemes is to recover paths as soon as possible. In section 3-3-3 a description is given on restoration and protection schemes. A protection scheme does not guarantee a backup path, assuming it exists in the network after failure, as multiple link or switch failures can invoke the protection plan. Re-active pushing of backup paths utilizes the current state of the network and is therefore more flexible. From Sharma et al. [39] and [4] we can conclude that both a protection and restoration scheme can work in OpenFlow environments, but recovery of paths with reactive pushing is significantly slower. Both schemes require active status monitoring, which will be discussed in section 4-2. All schemes can all be implemented in an OpenFlow environment, where a $1:1$ protection scheme ($N = 1$) is favorable due to faster failover times in a case of failures.

4-1-2 Survivability techniques

Besides recovery schemes, the number of backup paths, as well as the time-frame of the recovery is discussed, also a differentiation can be made on how paths are recovered. These survivability techniques can be split into two categories, being routing recovery procedures and sharing resources. The first category determines the way of rerouting traffic over the backup path, where the second category defines the allowance of sharing resources between primary and backup paths. In total three routing procedures can be defined [37].

- *Path-based recovery* - A backup path between two hosts requires the computation of a disjoint path. The computation of disjoint paths require actual topology information and is ideally performed by the centralized control logic. Path-based recovery can enable optimal paths through the network, depending on used disjoint-path algorithms. When it is not possible to discover a disjoint path, no full backup path can be provided to the data plane;
- *Link-based recovery* - A backup sub-path is computed for each link in the primary path. The sub-path is defined as the path between two ends of the link. In case of a link failure, traffic is rerouted over the backup sub-path. This recovery technique is more suited for hop-by-hop routing, as the decision to reroute the traffic is made locally at the link, without concerning host to host path optimizations;

- *Segment-based recovery* - A backup sub-path is computed for each segment in the primary path. The segment consists of a number of successive links of a path. A segment starts and terminates at nodes that have backup paths disjoint to the rest of the segment.

Studying the routing recovery procedures, path-based recovery will have the most optimal resource utilization as the computation of backup paths involves the complete network, where disjoint path algorithms can minimize path costs. This implies signaling of the failure to the central logic and back to the switching nodes to failover to backup paths. Link based recovery will require no signaling on the failure, as nodes itself can detect the failure and handle accordingly. With segment-based recovery, signaling is reduced, but still necessary to trigger the first node in the segment to failover to the backup path. The best routing procedure thus depends on the availability and performance of failure signaling, as well as the routing techniques applied for traversing traffic.

As mentioned earlier, the second category defines the sharing of resources. In total two sharing policies are defined in [37]:

- *Dedicated resources* - In the design of the network, resources are separately reserved for primary and backup paths. Resource reservations can be made on links by reserving bandwidth or links can assigned entirely. A major advantage of dedicated resources is that there are always backup path possible, assuming that no error in the reserved resources occurs;
- *Shared resources* - Primary and backup paths share the same resources and no reservations for backup paths are made. With a disjoint backup path, the requirement holds that primary and backup paths not share the same link. On link failure, links in the backup path will have higher utilization and my not have enough resources available.

4-1-3 Survivability of SDN networks

With the protection schemes and survivability techniques discussed, it is possible to project these on SDN networks and the OpenFlow protocol. The first requirement is set by the recovery time. In appendix A the requirements for different network services and applications is given, from where we can learn that the time window for recovery is very small. The primary path must be recovered from link or switch failures within millisecond order. The second requirement states that the resource allocation must be optimized in the network. This means that no extra links or switches can be added to network topologies. With these two requirements, possibilities for the protection schemes is reduced to $1 : N$ path protection or restoration, where protection schemes with $N = 1$ have been proven faster in OpenFlow networks by Sharma et al. [4]. The choice for resource allocation is made based on optimized resource allocation. Dedicated resources for backup paths, would lead to inefficient resource allocation, therefore protection paths share resources with other primary paths. The choice for the most optimal survivability techniques is unanswered, therefore we look into more detail to these techniques with the aim to discover the behavior of traffic after failure detection in a protected network. At first glance, the centralized characteristics of SDN with actual topology information via the OpenFlow protocol, path-based recovery seems the optimal survivability

technique. But as discussed, this requires propagation of the failure (via the central control logic) to source node, where a backup path can be selected. In a non-protected network, during the time between failure detection and the installation of a new primary path (T_F), no traffic can reach the destination. To solve the disconnectedness between source and destination in time window T_F, a protection scheme can be applied to the switch layer and for path-based protection crankback routing [50] needs to be applied. Crankback routing implies the recursive returning of traffic back to the first node from which the traffic can be forwarded to the destination again. The process of crankback routing is given in figure 4-1.

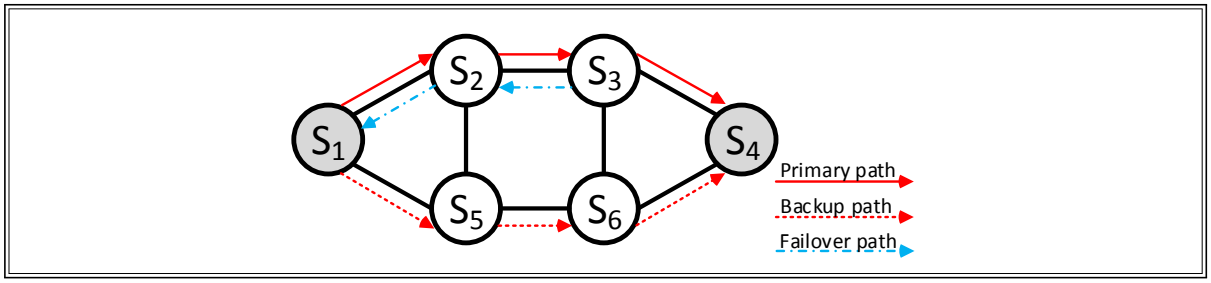


Figure 4-1: Routing behavior with path-based recovery during T_F - *A link failure on $S_1 \leftrightarrow S_2$ will be noticed by S_1 and all traffic will be forwarded over the backup path. Failures on the links $S_2 \leftrightarrow S_3$ and $S_3 \leftrightarrow S_4$ monitored by S_2 and S_3 require crankback routing during T_F, leading to doubled traffic load on link $S_1 \leftrightarrow S_2$ and $S_2 \leftrightarrow S_3$.*

In case of path protection, during a failure on link $S_2 \leftrightarrow S_3$ or $S_3 \leftrightarrow S_4$ the destination is not reachable without application of crankback routing. The first node where the failover path is joining the backup path, is source node (S_1), where for link- and segment-based protection this node can be located closer to the destination. Crankback routing results in double link utilization on link $S_1 \leftrightarrow S_2$ and $S_2 \leftrightarrow S_3$ during the failure, which can lead to congestion and packet loss. Link-based protection can minimize crankback routing during T_F for the same network, as shown in figure 4-2.

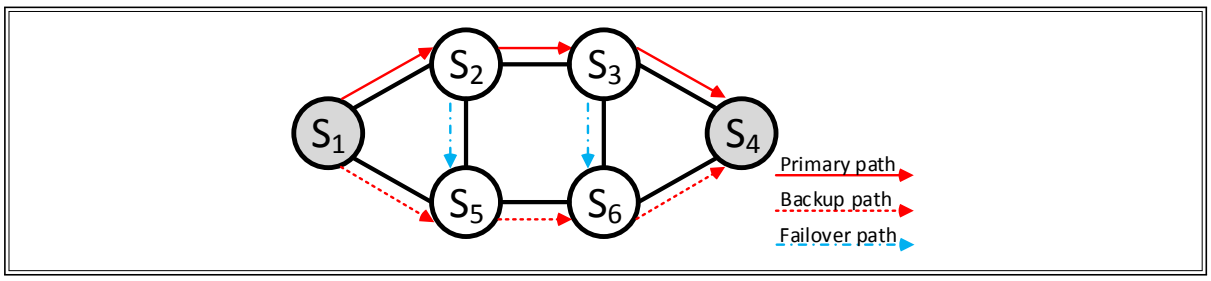


Figure 4-2: Routing behavior with link-based recovery during T_F - *Link failures on $S_2 \leftrightarrow S_3$ and $S_4 \leftrightarrow S_4$ are not cranked back to S_1, but forwarded to S_5 and S_6, from where the backup path continuous to the destination node S_4.*

A link failure on $S_2 \leftrightarrow S_3$ or $S_3 \leftrightarrow S_4$ in a link-based protection scheme for the given example network minimizes the need for crankback routing. From node S_2 and S_3 there exist sub-paths to the destination utilizing shortcut paths to the backup path. So, link-based protection minimizes the need for crankback routing, but network topologies exist that require crankback routing in order to maintain connectivity between source and destination during T_F , as seen in figure 4-3.

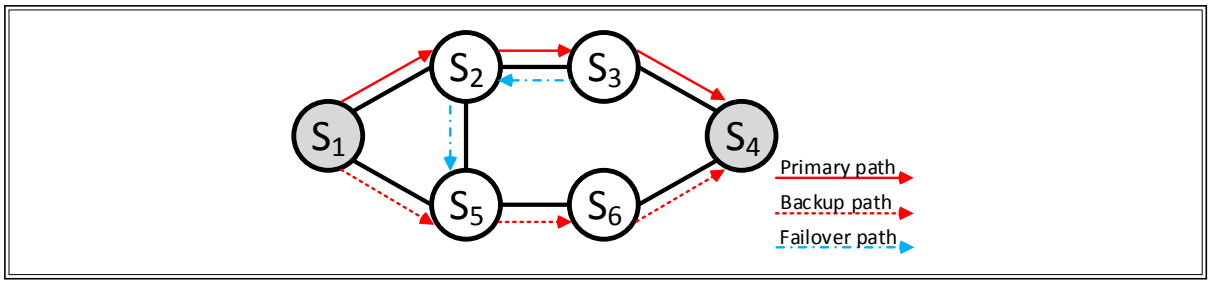


Figure 4-3: Routing behavior with best effort link-based recovery during T_F - *The network is modified by removing link $S_3 \leftrightarrow S_6$ from the topology. Link failures on $S_3 \leftrightarrow S_4$ will result in crankback routing over $S_2 \leftrightarrow S_3$. From node S_2 there exists a failover path to S_5 from where the backup path continuous to the destination node S_4.*

The network topology is unoptimized for link-based recovery, so that the failover path between node S_3 and S_6 does not exist. At node S_3 the only option in case of a failure is to return incoming packets. Looking to the characteristics of path- and link-based protection in a SDN network with no standardized fast failure signaling mechanism enabled, link-based protection offers best protection. A drawback of link-based protection is the number of paths to compute. For $1:1$ path-based protection, two paths must be calculated, where for $1:1$ link-based protection also paths must be computed for the intermediate nodes between the source and destination. In total $2+(P-2) = P$ path computations are required, where P is the length of the primary path. After failure detection by a node, the backup path can locally be assigned without communication with the control logic. Link-based protection minimizes crankback routing and therewith congestion and high link utilization are prevented during the failure window. To provide a full link-based protection scheme, the network topology may need optimizations, where existing topologies may need network augmentation. When the control logic has received the failure report, new primary, failover and backup paths are computed and configured at the switching nodes.

4-2 Failure detection mechanisms

In this section the mechanisms for failure detection are discussed in relation to SDN networking and the high requirements on the detection window. Current SDN implementations, like OpenFlow, are mostly based on Ethernet networks. Ethernet was not designed with failure detection and high availability requirements in mind, like optical and industrial networks do. Therefore more insight is given on failure detection mechanisms in Ethernet networks and if the high requirement standards are reachable with application of currently available Ethernet techniques and protocols. In figure 4-4 a typical Ethernet connection is given between nodes or a network switch.

The three lower layers of the OSI reference system all have the capabilities the detect link failures. On the physical and datalink layer, the Ethernet standard is responsible for the signaling over the medium between two nodes and ideally link failures are detected on this layer. When the link failure is detected, the status is propagated to higher layers, where protocols and control software can action accordingly. On higher levels there exists no physical, but a logical connection. Monitoring logical connections for link failure detection is possible, if

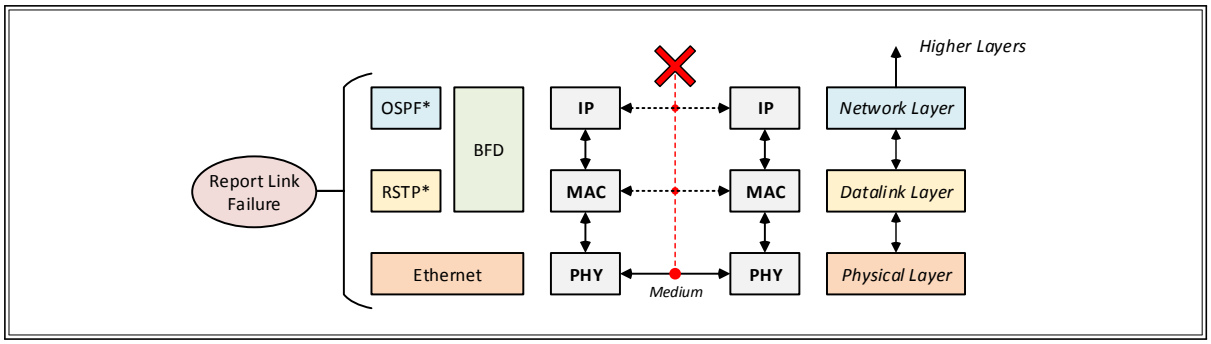


Figure 4-4: Failure detection mechanisms in lower three layers of IP Ethernet network - *A failure at the medium is detected by three lower levels of the OSI-layer system. The physical layer uses mechanisms from the Ethernet standard, where on the datalink layer for example the RSTP and BFD protocols and on the network layer OSPF and BFD are available for fault detection. Any detected failure will be reported to higher layers.*

we assume that lower situated layers function correctly. Using this assumption, link failure detection can also be performed by higher layers. Therefore protocols present on the datalink and network layer can be used for link status monitoring by a node. The network standards and protocols[1] from figure 4-4 are shortly discussed in relation the failure detection window:

- PHY - *Physical Layer* - The Ethernet standard has a failure detection mechanism build in. With a period of 16 ± 8 ms heartbeats are sent over the link. When the interface does not receive a response on sent heartbeats within a set interval of $50 - 150$ ms, the link is presumed disconnected [51]. From this we can conclude that the Ethernet standard is not capable to detect failures within the set time interval of 50 ms for carrier grade networks and link failure detection must be performed by the higher layers;
- MAC - *Datalink Layer* - There exist multiple protocols for failure detection on the datalink-layer, such as the Rapid Spanning Tree Protocol (RSTP)[52], with the aim to maintain a distribution tree of the network and update forwarding tables in Ethernet switches. All protocols found with failure detection mechanisms in Ethernet networks on the datalink-layer can be classified as slow. Update periods are in the order of seconds and are therefore not suitable for fast failure detection;
- IP - *Network Layer* - Normally Ethernet switches are designed to operate at the physical and datalink layer. The MAC-address of incoming traffic is matched to the forwarding table and the traffic is forwarded on the corresponding output port. This would indicate that fault monitoring on the third layer is not applicable. With the increased complexity in networks and the implementation of OpenFlow, network switches operate more and more on the network layer, the domain of routing protocols, like Open Shortest Path First (OSPF) [53]. Routing protocols keep status of the shortest paths between nodes in the network and update these in case of failures. The update windows of these protocols are not on millisecond order and are therefore not suitable.

The BFD protocol drawn in figure 4-4 is not discussed and will be discussed in section 4-3. In [4] the BFD protocol managed to detected path failures within 50 ms windows. Because

[1]Not all protocols with failure detection mechanisms are discussed, as this is out of scope of this thesis.

the protocol is designed to detect failures within a small time window and is implemented in numerous switches (MAC-layer) and routers (IP-layer), BFD is discussed in more detail in the next section.

4-3 Bidirectional Forwarding Detection protocol

The BFD protocol [42] implements a control mechanism to monitor the liveliness of pre-configured links or paths with potentially very low latencies. BFD is generally designed to operate on all datalink and network layers, but also on tunneling protocols and virtual circuits (figure 4-4). Specifically for IP-networks, an additional standardization is developed, RFC5881 [54], allowing IPv4 and IPv6 path monitoring. Both standardizations function using the same basic principles. Between two nodes a bidirectional connection is setup, where each node transmits control messages with the current state of the link or path and requested transmit (T_{tx}) and receive intervals (T_{rx}). A node receiving a control message, adjusts the receive interval to the requested transmit interval if $T_{tx} \geq T_{rx}$. When a series of packets is not received, the link or path is declared down. In theory an asynchronous configuration is possible, where the requested transmit interval is smaller than the offered receive interval ($T_{tx} < T_{rx}$). In that case, a node adjusts the transmit interval according $T_{tx} = T_{rx}$. For SDN implementations we advise a synchronous configuration, where $T_{rx} \leq T_{tx}$. Figure 4-5 gives the state diagram for the BFD protocol for a single host.

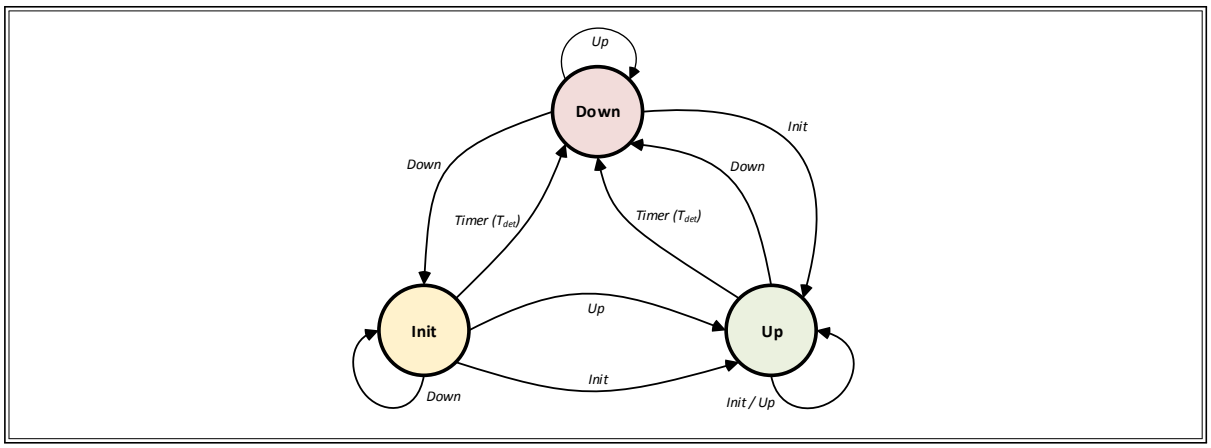


Figure 4-5: BFD-protocol state diagram for single node [42] - *The three states from the BFD protocol with transitions between the states given. The arrows between the states represent the received status from other nodes.*

The link or path status is monitored by a BFD session and can be in one of the three states DOWN, INIT and UP. On receiving a control message from the other node, the status can change. For example, the status transition from DOWN to INIT is initiated by receiving a DOWN status from the other session endpoint. To clarify the state diagram more, the three way handshake to built up a session for failure detection is illustrated in figure 4-6.

A session is built up after a three way handshake. Assume the link is functioning and on both nodes the BFD protocol is enabled on $t = 0$. Node A and B initialize with status DOWN and start transmitting control messages on $t = 0$ (Node A) and $t = a$ (Node B) according to the

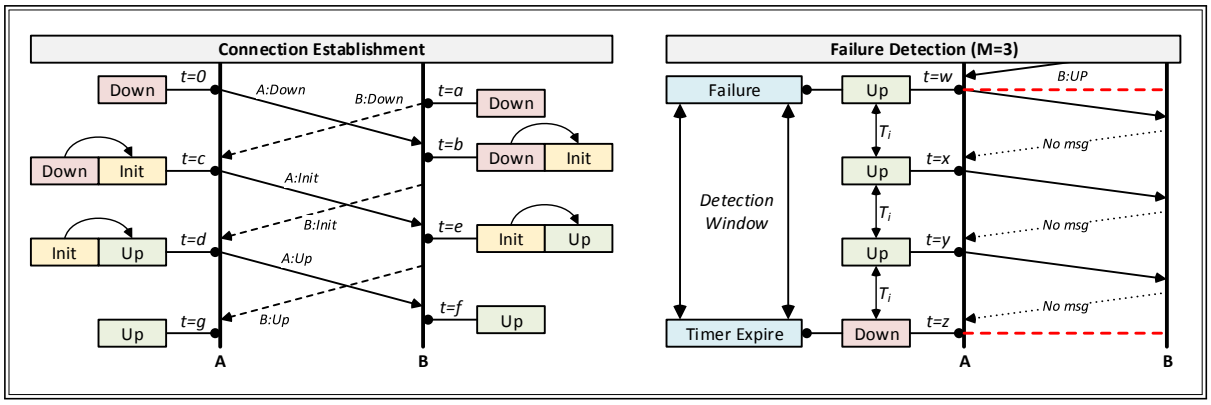


Figure 4-6: BFD-protocol three-way handshake and failure detection mechanism [55] - *At $t = 0$ Node A starts transmitting control messages. On arrival at $t = b$ at Node B, the status is changed from DOWN to INIT conform the state diagram. Node B also transmits control messages over the link. On $t = c$, Node A receives B:DOWN and changes its status. In the next transmit interval Node A transmits A:INIT, where after Node B can change its status to UP ($t = f$). On $t = g$ both nodes monitor the link up until $t = w$, when a link failure occurs. Node A expects control messages on $t = x$ and $t = y$ and on $t = z$ the failure detection time expires and the link is monitored down.*

configured transmit interval. Node B receives the control message containing A:DOWN on $t = b$ and will change its status conform the state diagram to B:INIT. On arrival ($t = c$) of the message B:DOWN at Node A, the status is changed to A:INIT and transmitted during next transmit interval. The link is monitored UP by Node A on $t = d$, as its own status was INIT before $t = d$ and the received message contained B:INIT. At time $t = e$ Node B receives A:INIT and changes its status to UP. As long control messages containing INIT or UP are received by both nodes (see state diagram in figure 4-5), the monitored status remains UP. A link or path is monitored down after receiving a DOWN message or the detection timer T_{det} expires. The timer expires when M control messages are not received (as illustrated in figure 4-6), leading to equation 4-1 for the detection time window (T_{det}).

$$T_{det} = M \cdot T_i \tag{4-1}$$

A default value for M is not given in the protocol description [42]. The Open vSwitch implementation defaults (and is hard coded) to $M = 3$. This value is not uncommon, as the widely used Border Gateway Protocol (BGP) [56] also implements a similar detection window with a default value $M = 3$. For further analysis and experiments during this thesis, we will embrace this value.

The BFD protocol is implemented in Open vSwitch, which gives us the ability to determine the detection windows using equation 4-1, but also take time measurements on a hardware testbed. To determine a lower bound for T_{det}, more investigation to the BFD protocol is needed. In theory the lower bound is determined by the trip-time of the control message. Now two problems arise, i) BFD introduces a $0 - 25\%$ time jitter to prevent time synchronization with other systems in the network and ii) the trip-time is not a constant time factor, as traffic load on links and length of the paths will influence the trip time negatively. The introduced time jitter leads to no problems for the detection window, as BFD reduces the transmission

interval for the next control message to equation 4-2, where $T_{i,set}$ is the set and negotiated transmit window and j is the random time jitter. Note that equation 4-2 is only valid for $M > 1$ [42].

$$T_i = j \cdot T_{i,set}, 0.75 \leq j \leq 1, M > 1 \tag{4-2}$$

The transmission interval is lower bounded by $j = 0.75$ and forms the upper bound for the trip time. This leads to equation 4-3, where T_{TT} is the maximum trip-time for the BFD control message. Choosing $T_{i,set}$ too small, leads to false negative detections, as the time windows is to small for BFD packets to transverse the link or path.

$$T_{TT} < 0.75 \cdot T_{i,set} \tag{4-3}$$

The minimum transmission window is found, when equation 4-3 is rewritten to equation 4-4, where $T_{i,min}$ is the lower bound for the transmission interval.

$$T_{i,min} > 1.33 \cdot T_{TT} \tag{4-4}$$

With equation 4-2 to 4-4 we found a lower boundary for the transmission window and with this boundary we can look further into the reachable detection window. If we expand T_{TT} from equation 4-4, the influence for longer paths becomes more clear. Equation 4-5 gives T_{TT}, where L is the number of links in the path, T_{trans} is the transmission delay introduced by the physical layer pushing packets onto the medium, T_{prop} is the propagation delay over the medium to the next node and T_{proc} is the processing time needed by end-nodes to process the control message.

$$T_{TT} = L \cdot (T_{trans} + T_{prop} + T_{proc}) \tag{4-5}$$

Minimizing equation 4-5 is only possible by minimizing L, as the transmission and processing delay are constant system variables. The minimum detection window is thus achieved with $L = 1$, which means that each switching node in the network monitors its outgoing links. An upper bound for T_{TT} can be determined by packet analysis, where we again assume that the system variables on each node are equal. Choosing the value for T_{TT} too small leads to a incorrect transmission window size and can cause BFD to produce false negatives. In order to determine a safe window size for $T_{i,min}$, the techniques used in the Transmission Control Protocol (TCP) [57] are used. During the development of TCP a similar problem was identified by Van Jacobson in [58], for optimizing transmission window sizes based on round-trip-times (RTT, T_{RTT}). The re-transmit timeout (RTO) intervals in TCP are computed using $\beta \cdot T_{RTT}$, where β account for the transmission variations. Choosing small transmission windows (β too small) causes packet corruption and large windows sizes (β too large) lead to inefficient use of the link or path. In [58] multiple algorithms are given, which dynamically adjust β according measured round-trip-times. For minimizing the detection windows by BFD, introducing a dynamic algorithm causes unwanted computational overhead and disturbs the simplistic design of the BFD protocol. Therefore we advise a fixed and conservative value $\beta = 2$ based on [59] for implementation in SDN networks. To account for the variations in the trip-time, the new lower bound for the transmission interval is given in equation 4-6.

$$T_{i,min} > 1.33 \cdot \beta \cdot T_{TT} \tag{4-6}$$

With the new lower bound for the transmission interval equation 4-1 is not valid anymore, because it assumes $T_{TT} \approx T_i$. With equation 4-6 the relation between the trip-time and the transmit interval becomes $T_{TT} < T_i$. In the extreme case, where the transmission interval is chosen large and the actual trip-time is very small, a failure initiated just after the control message is received, would lead to the relation $T_{TT} \ll T_i$ and the detection time given in equation 4-7.

$$T_{det} = (M + 1) \cdot T_i \tag{4-7}$$

The overall worst case detection window size T_{det} reachable, with aggregation of equation 4-5 and 4-6 is given in equation 4-8.

$$T_{det} = 1.33 \cdot \beta \cdot (M + 1) \cdot L \cdot (T_{trans} + T_{prop} + T_{proc}) \tag{4-8}$$

To link network survivability (section 4-1), the BFD protocol and equation 4-8 together, we discuss shortly to consequences of choosing path-, segment- or link-based protection. BFD delivers the needed failure detection mechanism and signaling for path- and segment based protection. A link failure between two end-nodes in the path or segment is noticed by BFD as the session is disrupted. After detection no signaling is required to source (segment) node, because this node performs the monitoring and can itself activate the backup path. Path-based protection is used in [4] and comes with a price. Longer paths ($L > 1$) will suffer from longer detection windows, according to equation 4-8. For link-based protection, the failure detection window can be minimized ($L = 1$). So when designing and configuring a robust network, a trade-off has to be made on the required failure detection window and the ability to provide path or segment protection. The argumentation is summarized in table 4-1, where $M = 3$ and $\beta = 2$ are processed in equation 4-8 and L_P and L_S are the path and segment length.

Protection	*Path length*	*Detection timer (T_{det})*	*Comment*
Path	$L_P > 1$	$10.66 \cdot L_P \cdot T_{TT}$	Signalling to end-nodes
Segment	$1 < L_S \leq L_P$	$10.66 \cdot L_S \cdot T_{TT}$	Signalling to segment source node
Link	$L = 1$	$10.66 \cdot T_{TT}$	Minimized detection windows

Table 4-1: Summary of failure detection windows with different protection schemes

4-4 Path failover in OpenFlow networks

In this section the OpenFlow implementation of liveliness monitoring and path failover functionality are discussed. For path failover, the Group Tables (available from OpenFlow protocol version 1.1, section 2-4) can be utilized. An interesting function that performs link status

monitoring and allows backup path triggering is the Fast Failover Group table. It can be configured to monitor the status of ports and interfaces and automatically, without contacting the control logic, switch over to pre-configured backup ports representing backup paths. The overall idea is given in figure 4-7.

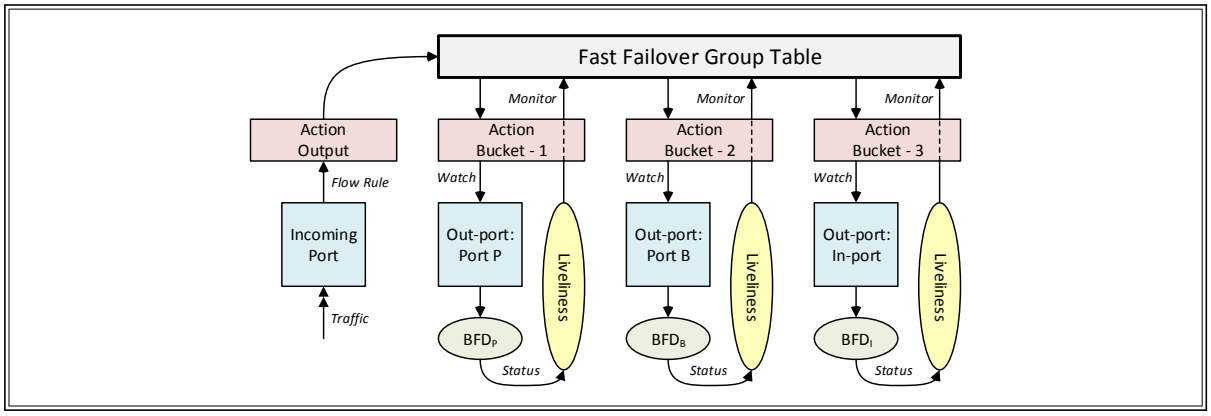


Figure 4-7: Liveliness monitoring and path failover on OpenFlow networks - *Incoming traffic is directed to the Fast Failover Group Table via a Flow Rule. The Group Table is configured with three Action Buckets, which are continuously monitored on liveliness. On each output port, liveliness is monitored and the status is updated in the Action Bucket. Depending on liveliness and priority of the Action Bucket, incoming traffic is forwarded.*

Incoming traffic is matched to a Flow Rule and the action output forwards traffic to the Fast Failover Group Table. The Group Table is configured with multiple Action Buckets, each containing a switch port or other Group Table to monitor. Each Action Bucket is assigned a priority, wherewith the primary, backup and protection paths can be distinguished. If no priority is given, the order of input determines the priority. In figure 4-7, an example is given where three Action Buckets are configured, each with an output port to monitor. Action Bucket 3 is configured as crankback path, as the incoming port is equal to the outgoing. On each of the output ports the liveliness of the link or path is monitored. Monitoring can be passive via OpenFlow's Loss-of-Signal detection or active with application of the BFD protocol. If the status is monitored down, the liveliness of the Action Bucket is also considered down [1] and the Fast Failover Group tables switches to the next *live* Action Bucket containing forwarding rules to recover the primary path. This failover is not negotiated with the control logic, so the OpenFlow switches in the network can independently operate after is has been configured with Fast Failover Group Tables containing protection paths.

4-5 Proposal for link-based protection

In the previous sections we have seen the possibilities for failure detection in SDN networks and specifically OpenFlow networks. This section will discuss our proposal for optimal protection against single link and switch failures. The main goal is to restore network connectivity within carrier grade and industrial network requirements (appendix A), as well as minimize the need for crankback routing. To minimize recovery times, active link monitoring is required, so for our proposal we will introduce BFD as liveliness monitoring mechanism. Figure 4-8 gives a flow diagram of our proposal, where P_{prim} and P_{backup} are the primary and backup path

between source and destination, P_{prot} are the protection paths for intermediate switches of P_{prim}, T_{BFD} is the failure detection time for the BFD protocol, T_{FF} is the time after which the primary path is restored, T_{IC} is the time required to inform the controller about the monitored failure, T_{RP} is the time required to remove the primary path and T_{NP} is the time required to compute new paths after failure detection.

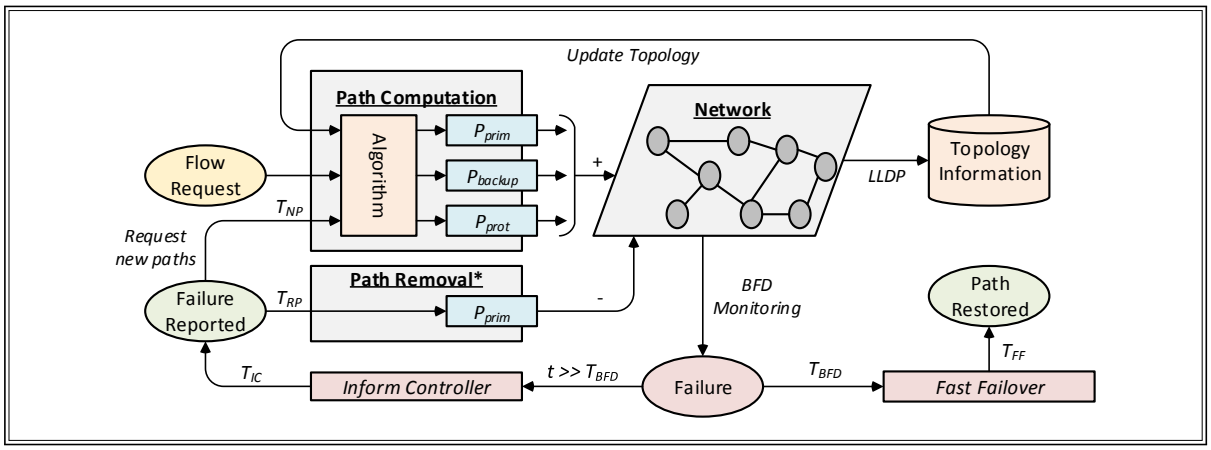


Figure 4-8: Proposal for optimal network protection - *A Flow Request arrives at the Path Computation module of a OpenFlow controller. With up to date topology information an algorithms must compute three path solutions for P_{prim}, P_{backup} and P_{prot}. After computations the paths are installed (+) as Flows in the OpenFlow switches. In the network, all switches are configured to perform link monitoring and on failure the Fast Failover Group Table mechanism is activated and the controller is informed. The fast failover mechanism restores P_{prim} via P_{backup} or P_{prot} at T_{FF} . On T_{IC} the controller is informed and can remove P_{prim} to activate P_{backup} as new primary path. Also a new path configuration can be requested, to re-ensure protection to the network.*

For our proposal, we utilize known and proven solutions. The OpenFlow network is configured for active link monitoring via the BFD protocol and the topology is maintained by a topology module at the OpenFlow controller with the application of LLDP-packets, as discussed in section 3-3-2. A Flow Request is processed by the path computing module of the OpenFlow controller. With actual topology information, P_{prim}, P_{backup} and P_{prot} are computed, where for delay optimizations, the primary path is installed first at the network, where after the other paths are computed and installed. On failure detection by the BFD protocol, the Fast Failover Group Table mechanism is initiated locally at the OpenFlow switches and triggers an error report to the OpenFlow controller. Here the relationship $T_{IC} \gg T_{FF}$ holds. At T_{FF} the primary path is recovered via P_{backup}, if the failure is related to the first link or switch in P_{prim}, where recovery is provided by P_{prot} otherwise. The controller has two options at arrival of the failure report; i) Remove P_{prim} from the network to activate P_{backup} as new primary path and after removal compute new path solutions, or ii) keep P_{prim} as primary path and directly compute new solutions. If crankback routing is required to provide protection for a certain primary path through the network, the removal of P_{prim} and activation of P_{backup} as primary path, removes crankback situation from the network. Looking to the time relations, we can state that the time required to remove the primary path is smaller than to compute new solutions ($T_{RP} < T_{NP}$).

4-6 Conclusion on network survivability in SDN

In this chapter the theory on network survivability has been briefly discussed in relation with SDN. For efficient network utilization and fast response times on link or path failures, the best performance is found in a 1 : 1 protection scheme. Reactive restoration scheme require communication with the network control logic, which consumes unwanted time. In order for switches and nodes in a network to react quickly on path and segment failures, detection mechanisms, backup paths and signaling schemes to propagate the failure are needed. In the BFD protocol, the failure detection and signaling scheme is found, so path and segment protection can be performed in SDN and OpenFlow networks. Two drawbacks adhere, as crankback routing is required and failure detection windows increase due to the longer path lengths. For link-based protection, crankback routing is not always required and the failure detection window is minimized using BFD. Choosing the best solution depends on failover requirements and network layout. In the next chapter we must investigate possible algorithms for path-, segment- and link-based protection schemes for implementation at the routing module for the centralized control logic.

Chapter 5

Path protection algorithms

In chapter 3 we reviewed a number of topics in SDN networking. We found that more attention is needed for increasing the robustness of SDN. Robustness issues are identified at the transport layer, where protection and recovery schemes are present to protect data paths through the network. Chapter 4 showed that BFD provides the necessary mechanisms to detect link-, segment and path failures, where link-based protection offers shortest failover times. In [4] path-based protection is applied, where [43] applied segment-based protection. In this chapter we search for an algorithm that provides link-based protection in SDN networks. Therefore, first the algorithmic problem for this algorithm is formulated in section 5-1, where section 5-2 discusses the solution space where within algorithms are found to solve the formulated problem. Section 5-3 discusses disjoint path algorithms, which can guarantee disjoint paths and offer direct solutions for path-based protection. In section 5-4 we propose a protection algorithm based on a disjoint path algorithm and a new developed extension to Dijkstra's shortest path algorithm to discover protection paths, where section 5-5 shows the results of the protection algorithm applied to an example topology. Section 5-6 shows possible enhancements and benefits of SDN in path discovery algorithm, where the complexity of the complete link-based protection algorithm is discussed in section 5-7. Section 5-8 concludes this chapter.

5-1 Algorithmic problem for link-based protection

The functional requirements for the link-based protection algorithm can be derived from section 4-1-3, where the path solutions provided as example in figures 4-1 and 4-3 are requested. Before going into more detail on specific protection algorithms, the functional description will be translated to an algorithmic problem. Consider a bidirectional and connected network $\mathcal{N}(N, L)$ with N switches and L links. Each link $l(S_i \rightarrow S_j)$ between a switch S_i and S_j is characterized by a link weight $w(S_i \rightarrow S_j)$, where the weight is non-negative ($w_i > 0$) and we assume the weight is equal in both directions $w(S_i \rightarrow S_j) = w(S_j \rightarrow S_i)$. For SDN and OpenFlow networks, this weight is a singular and measurable link parameter, such as

round trip time, packet loss or link utilization. The link weight is assumed additive, which means that some parameters must be transformed before summations can be applied. Path $P(S_A \rightarrow S_B)$ or $P_{S_A \rightarrow S_B}$ contains k switches and $k-1$ links (hops) to travel between switch S_A and S_B, where the path cost $C(P_{S_A \rightarrow S_B})$ equals the sum of weights as given in equation 5-1 [60].

$$C(P_{S_A \rightarrow S_B}) = \sum_{j=1}^{k-1} w(S_j \rightarrow S_{j+1}) \tag{5-1}$$

$\mathcal{P}_{S_A \rightarrow S_B}$ is the set of all paths between switch S_A and $S_B$in $\mathcal{N}$, where $P^*_{S_A \rightarrow S_B}$ is the shortest path for which $C(P_{S_A \rightarrow S_B})$ is minimized. We define S^*_i as the set of (protected) switches transversed in $P^*_{S_A \rightarrow S_B}$, where $i \neq B$ and the protection path from S^*_i to S_B as $PP(S^*_i)$. Furthermore, we define S^*_f and $l^*(S_i \rightarrow S_f)$ as respectively the switch and link in $P^*_{S_A \rightarrow S_B}$ in failure, where $f = i + 1$. Given a single link or switch failure, the link-based protection problem is to discover path $P^*_{S_A \rightarrow S_B}$ in network $\mathcal{N}$ and for the set of switches S^*_i compute and guarantee a protection paths $PP(S^*_i)$ towards the destination node S_B, assuming S^*_f or $l^*(S_i \rightarrow S_f)$ in failure, such that crankback routing or overall path costs are minimized.

With the link-based protection problem statement we want to ensure optimized use of network resources with the application of the shortest path $P^*_{S_A \rightarrow S_B}$ between source switch S_A and destination switch S_B. We also require protection paths for S^*_i to guarantee network connectivity for the provided service in case of a single link or switch failure. The problem can be solved with two different minimization settings. Unwanted crankback routing is minimized without bounding path cost or the path costs are optimized, where crankback routing is a feasible solution.

5-2 Solution space for protection algorithms

At first sight, the problem seems easy to solve. Execution of a shortest path algorithm in network $\mathcal{N}$ will give shortest path $P^*_{S_A \rightarrow S_B}$. From the set of switches S^*_i again apply the shortest path algorithm to the destination, while assuming switch S^*_f or link $l^*(S_i \rightarrow S_f)$ in failure. This seems a valid solution, but problems may arise with possible routing loops in the protection paths and guaranteed protection of the shortest path. In worst case, one may need a large number of shortest path computations to discover that protection is not possible on $P^*_{S_A \rightarrow S_B}$ on network $\mathcal{N}$. Therefore, we performed a literature research to other protection algorithms, which we can utilize as basis to solve our algorithmic problem for a link-based protection algorithm.

We identified two kind of algorithms during our research, being the concept of redundant trees and disjoint path algorithms. The concept of redundant trees [61] computes a pair of directed spanning trees, where each node is connected to both trees, leading to a *virtual* ring topology. After a link or switch failure is monitored in one tree, traffic can be forwarded over the redundant tree. Application of the redundant tree concept in IP networks is found in [62], where the BFD protocol is utilized for active link monitoring and triggering of the redundant tree. The problem with the application of redundant trees is that minimization of path cost is equal to the Steiner tree problem and proven NP-hard in [63]. Some variants of

the Steiner tree problem are solvable in polynomial time using heuristic algorithms. Although the redundant tree concept can provide protection against a single link or switch failure, no guarantee can be given on application of the shortest paths as primary path to optimize network allocation.

Disjoint path algorithms are well studied by J.W. Suurballe [64] and R. Bhandari [65]. The algorithms compute two disjoint paths between source and destination with different levels of disjointness. To compute disjoint paths, shortest path algorithms are applied together with temporary modifications to links and switches the network. The disjoint path algorithm can compute primary and backup paths with different objectives. For some objectives the algorithm can guarantee two disjoint paths, if they exist in the network, by performing two shortest path computations. This makes disjoint path algorithms a suitable basis for the link-based protection scheme, as in an early stadium of the algorithm, protection can be guaranteed. In the next section we will discuss the basics of disjoint path algorithms in relation to the defined algorithmic problem.

5-3 Disjoint path algorithms

Computing disjoint paths in a network is a requirement for path-based protection in network survivability. There are multiple algorithms that can discover shortest paths, like the Dijkstra [66] and Bellman-Ford algorithm [67]. Integrating these shortest path algorithms in SDN enables the possibility to compute optimal paths between two hosts in the network. Discovery multiple paths is a more challenging task. An obvious, but far from optimal method, is to compute all possible paths between two hosts and select the two most appropriate ones as disjoint path pair. This *brute force* method may bring solutions, but introduces computational overhead and unwanted latencies in path computations. In this section levels of disjointness, path objectives and the process to discover disjoint paths are further discussed and related to the link-based protection problem.

5-3-1 Requirements on disjoint paths

Path failures in networks can have two possible origins, being link or switch failures. If one of both fails, the path between two hosts will fail and the offered network service becomes unavailable. To discover disjoint paths, shared resources between primary and backup paths must be prevented. Using this basic statement, four levels of disjoint paths can be distinguished.

- *Regional disjoint paths* - The highest level of separation between primary and backup paths. Disjoint paths are discovered assuming a circular region of switches in failure state. Survivability is provided against failures in different geographic regions of a network [68];

- *Switch disjoint paths* - Discovered disjoint paths may not contain the same switches. This level of disjoint paths offers survivability on single switch and link failures;

- *Link disjoint paths* - Primary and backup paths may contain the same intermediate switch(es), but can not share link resources. This level offers only survivability on single link failures;
- *Partial disjoint paths* - Primary and backup paths may contain the same intermediate switches and can share link resources. Remaining parts of the paths can either be switch or link disjoint. Partial disjoint paths in a network must be minimized to offer the highest survivability probability.

For this thesis we will not set regional disjoint paths as requirement for survivability. To clarify (partial) link and switch disjointness levels, figure 5-1 gives an illustration.

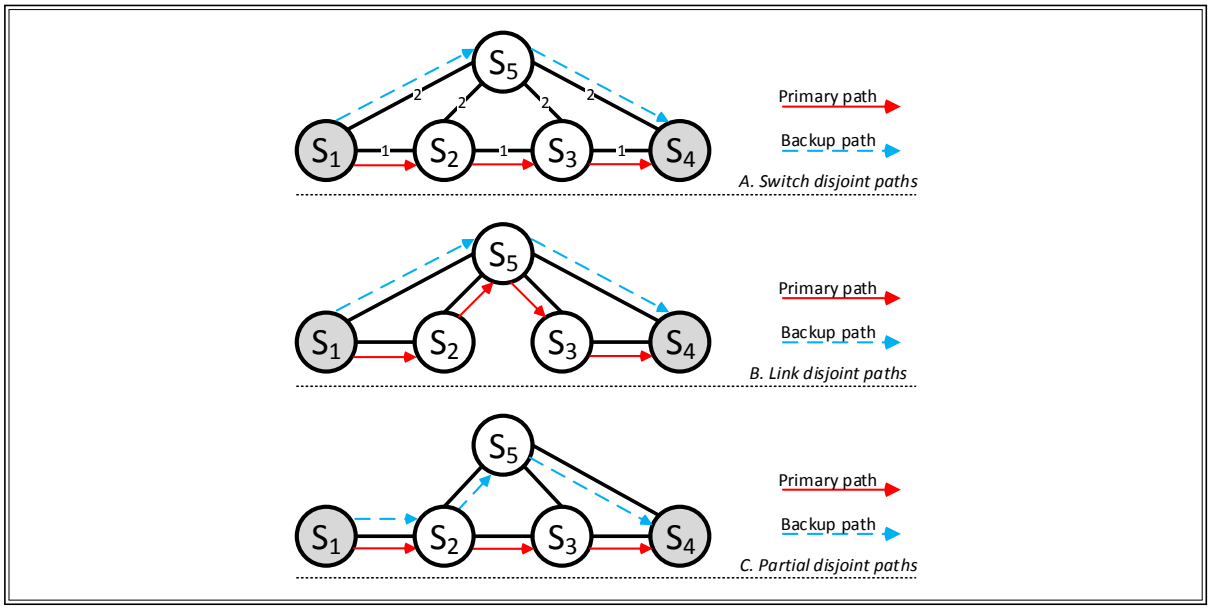


Figure 5-1: Three levels of disjoint paths - *In figure A to C the primary and backup paths are drawn from switch S_1 to S_4. In figure A a switch disjoint path is illustrated. The primary and backup path do not share links or switches. Figure B shows a link disjoint path, where both paths share switch S_5, but no links. In figure C the paths share $l(S_1 \rightarrow S_2)$ and the paths are therefore partial disjoint*

The paths drawn in figure 5-1 show the difference between three survivability levels. In figure 5-1A the primary and backup path from S_1 to S_4 share no switches and links, and can therefore be classified as switch disjoint paths. To show the characteristics of a link-disjoint path (figure 5-1B), link $S_2 \leftrightarrow S_3$ is removed from the network, forcing the primary path over S_5. The backup path travels also over S_5, but shares no links with the primary path. In figure 5-1C link $S_1 \leftrightarrow S_5$ is removed. The backup path must travel over link $S_1 \rightarrow S_2$ as there is no other link available. At switch S_2, the backup path separates from the primary path and travels switch disjoint to the destination. In general, the best level of protection is achieved with switch disjoint paths, as these paths are resilient against switch failures and targeted attacks to particular switches. Partial disjoint paths are least favorable and are therefore not further applied in this thesis. Depending on the network topology and connectivity, the highest level of disjointness must be pursued. The disjointness levels as discussed are directly applicable to path-based protection schemes, as complete paths are region, switch or link disjoint. For

our link-based protection problem the primary path does not have to be completely disjoint, as per hop forwarding decisions are made. Therefore we can change the level of disjointness for our link-based protection problem to *next-hop link disjoint* and *next-hop switch disjoint* paths.

5-3-2 Disjoint path objectives

The disjointness levels for protection against failures on the primary path are set, but on cost optimization the disjoint path algorithm must solve some problems. In [37] and [69] five objectives are defined for cost optimizations during discovery of disjoint paths.

- *MIN-SUM* - The sum of the primary and backup paths is minimized. This objective is set to minimize the overall cost of both paths. Algorithms exist that discover MIN-SUM disjoint paths in the network, if they exist;
- *MIN-MAX* - The maximum cost of both primary and backup paths are minimized. This objective is desirable for load-balancing applications, as the maximum cost for all requested paths is minimized. The MIN-MAX objective has no exact algorithms that produce solutions, as finding paths for this objective is proven NP-hard [70]. The MIN-MAX objective is in [69] used as secondary objective to the MIN-SUM objective. If there exist multiple disjoint path pairs that satisfy the MIN-SUM requirements, the primary and backup path pair is chosen such that the cost of the most expensive path is minimized;
- *MIN-MIN* - The minimum cost of the disjoint paths are minimized. This objective is desirable in $1:1$ or $1:N$ path recovery, because the cost of the primary path is minimized. Backup paths are only needed on link or switch failure conditions, which, we assume, occur with a low probability. The MIN-MIN object will thus minimize the load on the network under normal conditions. This is the ideal objective to increase the survivability on a SDN network without increasing overall load on the network. Unfortunately, discovering MIN-MIN disjoint paths is an APX-hard problem [71]. As with MIN-MAX, the MIN-MIN objective can be used as secondary objective for the MIN-SUM requirement. Hereby the cost for the primary path are minimized, when more MIN-SUM disjoint path pairs exist;
- *Bounded* - The cost for a path must be less or equal than a bound set for each of the paths. Bounds on path costs are determined by the provided network service traversing over the path. Within boundaries for the primary and backup paths, multiple disjoint paths are possible. The bounded objective is a less strict objective compared to the MIN-MAX and MIN-MIN objectives. Finding bounded disjoint paths, is also proven APX-hard [71];
- *Widest* - The smallest link capacity on multiple paths is minimized. Services that require a minimum link capacity are best served by this objective, as disjoint paths are found, keeping the minimum link capacity as boundary. When demands on capacity between the primary and backup path differ, the problem to find widest disjoint path becomes NP-complete [72].

As seen, each of the objectives have specific applications. The best solution to increase the survivability of SDN networks and to minimize path cost for the primary path, is the MIN-MIN objective. Because there exists no exact algorithm that will produce MIN-MIN disjoint paths, an alternative solution is needed to discover the most optimal disjoint paths in SDN networks. The MIN-SUM objective will discover disjoint paths in the network and therewith guarantee protection for the primary path. Protection for the intermediate switches and path cost minimization is not provided by the MIN-SUM objective. Further research to apply MIN-SUM algorithms as basis for our link-based protection problem is thus required.

5-3-3 Discovery of disjoint paths

Before going into detail how to discover MIN-SUM disjoint paths, we first want to show the need for MIN-SUM algorithms and why the MIN-MIN objective seems obvious to implement, but is not guaranteeing disjoint path pairs. In the introduction of this section, the brute force method of discovering disjoint paths was briefly introduced. With the brute force method, all possible paths $\mathcal{P}_{S_A \to S_B}$ in network $\mathcal{N}(N, L)$, must be computed. In worst case scenario, where network $\mathcal{N}$ is fully connected, the number of paths in the network is equal to $e(N-2)!$ [73]. Depending on the number of switches and links in the network and the objective of the algorithms, discovering disjoint paths can be time consuming. To reduce computational overhead, the brute force method to discover next-hop link or switch disjoint paths is a far from optimal solution to our problem.

A second and a more directed method is the *Remove-Find* (RF) method from [74]. For completeness, the Remove-Find algorithm is given in appendix B. The RF-algorithm complies to the MIN-MIN objective and would partly solve the link-based protection problem, because the primary path between switch S_A and S_B is by definition minimized with application of the first discovered shortest path ($P^*_{S_A \to S_B}$). In [74] two major disadvantages of the RF algorithm are given, being i) the cost of the backup paths can be significantly larger than the primary (shortest) path and ii) the RF-algorithm will not guarantee disjoint paths in a network where at least two disjoint paths exist. To demonstrate the *greedy* behavior of the RF-algorithm, an example is given in figure 5-2.

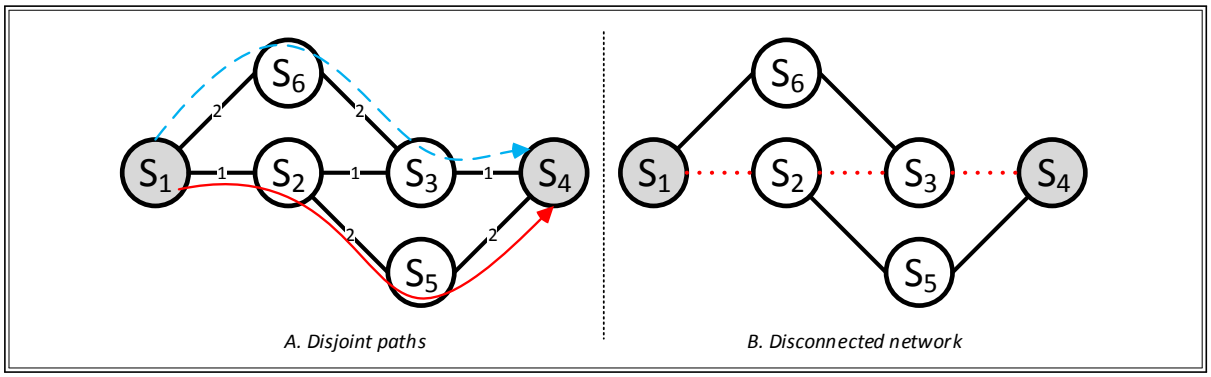


Figure 5-2: Disjoint paths with Remove-Find algorithm - *In figure 5-2A the possible disjoint paths are illustrated between S_1 and S_4. After application of the RF algorithm, shortest path $S_1 \to S_2 \to S_3 \to S_4$ is removed from the network and the topology of figure 5-2B remains, with no connectivity between S_1 and S_4.*

In figure 5-2A the possible disjoint paths are illustrated in the network. The shortest path between S_1 and S_4 in this network is found as $S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4$. When the RF-algorithm is applied in figure 5-2B and the shortest path is removed from the network, there exists no remaining connectivity between S_1 and S_4. We can conclude that the RF-algorithm complies with the MIN-MIN objective, but this achievement comes with the price that disjoint paths may not be discovered in the network, even though they exist. The directive to remove the shortest path from the network is used in the development of the link-based protection algorithm. For us, the best method to guarantee disjoint paths and therewith protection for the primary path is found in the application of MIN-SUM algorithms.

5-3-4 Computing MIN-SUM disjoint paths

The process to discover MIN-SUM disjoint paths in a network, originates from the minimum cost flow in flow networks [37]. In flow networks, the amount of flow between a source and destination can be divided over multiple links, if the amount of flow is greater than the capacity of one of the connecting links to the source or destination. This means that links which are utilized completely and therefore have no remaining capacity, can not be used for a second time and flow must be divided over the remaining links, which have a spare capacity. A path found, using remaining capacity on intermediate links, is called an augmenting path. So, the disjoint behavior which is required to guarantee primary and backup paths, is coming from maximizing flows through a network using augmenting paths. By introducing cost as additional parameter on the links and the objective to minimize paths costs between the source and destination, the minimum cost flow problem is equal to the MIN-SUM algorithm.

All MIN-SUM disjoint path algorithms are based on an algorithm found by J.W. Suurballe [64], also known as Suurballe's algorithm. With the algorithm it is possible to compute K disjoint paths between a source and destination switch, using the $k-$shortest path algorithm [75]. Suurballe's algorithm discovers $K-1$ augmenting paths, which are optimized to minimize the sum of all discovered paths and can be adjusted to discover link and switch disjoint paths in $O(N^2 \log N)$ time. An efficiency improvement with more efficient heap structures was made to Suurballe's algorithm in the Suurballe-Tarjan algorithm [76]. With the improvement it is possible to compute two ($K = 2$) disjoint paths in a network with the computation of only two shortest paths with Dijkstra's algorithm [66]. The Suurballe-Tarjan algorithm has a running time of $O(L \log_{(1+L/N)} N)$. R. Bhandari made a simplification to Suurballe's algorithm by discovering augmenting paths with the application of temporary link and switch modifications to the network. Bhandari's algorithm [65] also utilizes the $k-$shortest path algorithm and has equal running time in comparison with Suurballe's algorithm. For this thesis we used Bhandari's algorithm to discover disjoint paths. A description on how (partial) link and switch disjoint paths are computed using Bhandari's algorithm is given in Appendix C. From appendix C we will utilize the following parameters for the remaining part of this thesis:

- *Cost* - $C(X)$ or C_X - The cost for path X.
- *Shortest path* - Q_1 - The shortest path in network $\mathcal{N}$;
- *Simple paths* - Q_k - Higher order shortest paths found in the transformed network $\mathcal{N}_T$, where $k > 1$;

- *Disjoint paths* - $\mathcal{D}$ - Set of disjoint paths constructed from shortest path Q_1 and discovered simple paths Q_k, where $k > 1$;
- *Primary path* - P_1 - Disjoint path from $\mathcal{D}$ for which $C(P_1)$ is minimized;
- *Backup paths* - P_k - The backup paths for primary path P_1, where $k > 1$.

With the discussion of Bhandari's algorithm a method is found to compute disjoint paths. When Bhandari's algorithm discovers a feasible solution, there exist at least two paths between the source and destination switch. This implies that the primary path can be protected with in worst case application of crankback routing. The remaining part of the problem is to solve path optimizations, where we demand the primary path cost C_{P_1} is equal to the cost of the shortest path C_{Q_1}. Per definition $C_{P_1} \geq C_{Q_1}$, which indicates that solutions computed by Bhandari's algorithm do not meet the MIN-SUM + MIN-MIN objective and therewith no solves our link-based protection problem.

5-4 Protection algorithm for link-based protection paths

In this section we will propose a scheme to solve the link-based protection problem, where we choose $K = 2$. With $K = 2$ a $1 : 1$ protection scheme is suggested to provide survivability for the protected switches S_i^* against a single link or switch failure. Our proposal consists of two phases:

- Phase A - *Disjoint path discovery* - Computation of shortest path and disjoint path pair as described in the previous section. If feasible paths are discovered, protection can be guaranteed and protection paths can be discovered in phase B;
- Phase B - *Protection path discovery* - For each protected switch in the shortest path a protection path towards the destination must be discovered. To discover the requested protection paths, an extension is made on the modified Dijkstra algorithm from appendix D. The aim for the extended Dijkstra algorithm is to minimize path cost or the application of crankback routing.

The proposal solves the link-based protection problem and includes the computation for P_{prim}, P_{backup} and P_{prot} from figure 4-8, which are discussed in our proposal for optimal network protection in section 4-5. For the description of the two phases we assume that the connectivity of network is on such level that all required paths can be discovered.

5-4-1 Phase A - Disjoint path discovery

Phase A is applied to discover paths Q_1, Q_2, P_1 and P_2 with Bhandari's and Dijkstra's modified[1] algorithms. If simple path Q_2 exists, given a link or switch failure, a protection for the shortest path Q_1 can be guaranteed. The process for phase A is given in pseudo code in algorithm 1.

[1]The modified Dijkstra algorithm can discover shortest paths where negative link weights are present in the network and provide protection against routing loops during path discovery.

Algorithm 1 Phase A - Discover shortest path between S_A and S_B in $\mathcal{N}$

STEP 1 - DISCOVER SHORTEST PATH

1. EXECUTE Modified Dijkstra's algorithm ($\mathcal{N}$, S_A, S_B) $\Longrightarrow Q_1$
2. IF Q_1 exists:
3. Go to Step 2
4. ELSE:
5. Stop algorithm

Dijkstra's algorithm is executed to compute the shortest path Q_1 between source S_A and destination switch S_B. When no shortest path is discovered, there exists no path between S_A and S_B and the protection algorithm stops. In the next step the disjoint path pair is discovered with Bhandari's algorithm. The pseudo code for Step 2 is given in algorithm 2, where L_{Q_1} represents the number of hops in path Q_1 and s is the requested survivability.

Algorithm 2 Phase A - Discover disjoint path pair between S_A and S_B in $\mathcal{N}$

STEP 2 - DISCOVER DISJOINT PATH PAIR

1. EXECUTE Bhandari's algorithm ($\mathcal{N}$, S_A, S_B, Q_1, s) $\Longrightarrow Q_2$, P_1, P_2
2. IF P_2 exists:
3. SET $PP(S_A) = P_2$
4. ELSE:
5. Stop algorithm
6. IF $L_{Q_1} > 1$:
7. Go to Phase B
8. ELSE:
9. RETURN $\{Q_1, PP(S_A)\}$

With Q_1 and the required survivability s (link or switch failure protection), simple path Q_2 is discovered and disjoint paths P_1 and P_2 are constructed using Bhandari's algorithm. When no disjoint path pair is discovered, the algorithm terminates. The outcome for P_1 can be discarded, as we set the shortest path as primary path for cost minimization. Backup path P_2 is set as protection path for S_A ($PP(S_A)$), as P_2 is per definition next-hop disjoint from Q_1. If the shortest path between S_A and S_B is a direct link ($L_{Q_1} = 1$), the protection algorithm returns the results as for no other switches protection is required. Otherwise ($L_{Q_1} > 1$), the algorithm will continue with phase B.

Before the discussion of phase B, the cost relationship between Q_1 and P_1, as well as Q_2 and P_2, need explanation. Depending on the network topology and link weights, Bhandari's algorithm provides two possible outcomes. In table 5-1 the relationship between disjoint paths P_k and shortest paths Q_k are given with respect to the appearance of joint links during the discovery process of Bhandari's algorithm. Joint links are present during computations if one of the links in shortest path Q_1 is traversed in reverse by simple path Q_2. This will result in the fact that disjoint paths P_1 and P_2 are constructed from segments from both Q_1 and Q_2 (Appendix C).

Path Q_k	*Path D_k*	*Joint Link found*	*Relation*
Q_1	P_1	yes	$Q_1 \neq P_1$, $C_{Q_1} \leq C_{P_1}$
Q_1	P_1	no	$Q_1 = P_1$
Q_2	P_2	yes	$Q_2 \neq P_2$, $C_{Q_2} \geq C_{P_2}$
Q_2	P_2	no	$Q_2 = P_2$

Table 5-1: Relationship between P_k and Q_k in different network topologies

The relationships in table 5-1 show that the cost for the primary path P_1 is equal or greater than the cost for shortest path Q_1, if joint links are present during computations. To prove these statement, the example topologies from figure 5-1A and 5-2A can be used. If Bhandari's algorithm is applied to figure 5-1A, no joint link exists between Q_1 ($S_1 \rightarrow S_5 \rightarrow S_4$) and Q_2 ($S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4$), so $P_1 = Q_1$ and $C_{P_1} = C_{Q_1}$. This equality implies that the found solution solves our problem on cost minimization for the primary path. Application of Bhandari's algorithm to figure 5-2A leads to joint link $S_2 \rightarrow S_3$ between $Q_1(S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4)$ and $Q_2(S_1 \rightarrow S_6 \rightarrow S_3 \rightarrow S_2 \rightarrow S_5 \rightarrow S_4)$. After construction of $P_1(S_1 \rightarrow S_2 \rightarrow S_5 \rightarrow S_4)$ and $P_2(S_1 \rightarrow S_6 \rightarrow S_3 \rightarrow S_4)$, we find $C_{Q_1} < C_{P_1}$ ($C_{Q_1} = 3, C_{P_1} = 5$). To guarantee cost minimization, we must set the primary path as the shortest path. Herewith the disjointness between Q_1 and P_2 (backup path) is compromised, but our requirement on *next-hop* link or switch disjointness is still in place, as P_2 ($PP(S_A)$) provides protection for link failure on link $S_1 \rightarrow S_2$ or failure at switch S_2. Due to the next-hop behavior, full disjointness between the shortest path and protection paths is thus not required.

5-4-2 Phase B - Protection path discovery

In phase B the protection paths for protected switches S_i in the shortest path Q_1 are computed. In algorithm 3 a global overview of phase B is given in pseudo code, where m is the requested minimization setting (crankback reduction or cost minimization).

The first action in phase B is to remove the links from the shortest path from the network. Herewith potential routing loops are avoided, but also potential protection paths are removed. In the protection algorithm the problem with removal of potential solutions is accounted for. For each protected switch S_i , the protection algorithm will be executed to discover protection paths. For switch $S_i = S_A$ the protection algorithm needs no execution, as the protection path was set in phase A. The discovered shortest path and protection paths are returned and can be translated to Flow Rules and configured to OpenFlow switches.

Algorithm 3 Phase B - Discover protection paths for S_i^* in $\mathcal{N}$

1. REMOVE Q_1 from $\mathcal{N}$
2. FOR S_i in Q_1 , where $i \neq A, B$:
3. EXECUTE Extended Dijkstra algorithm ($\mathcal{N}$,Q_1,Q_2,m,s) $\Longrightarrow PP(S_i)$
4. RETURN $\{Q_1, PP(S_i)\}$

5-4-3 Extended Dijkstra's algorithm

To discover protection paths, we extended the modified version of Dijkstra's algorithm (appendix D) to utilize intermediate products from Bhandari's algorithm, reduce computational overhead and restore solutions with the removal from the shortest path from the network. Dijkstra's algorithm normally consists of three steps, being initialization, switch selection and relaxation. On all three steps extensions are made to discover protection paths with the requested survivability and minimization setting. Extensions are made in such a way that the original behavior of Dijkstra's algorithm remains and computational overhead is reduced. Important steps in the extended algorithm will be discussed in detail with support of pseudo code.

For the algorithm we define that the protection path starts at switch S_i, but not per definition terminates at S_B. To reduce computational overhead, we will utilize earlier discovered paths. When a shortcut is found to an earlier discovered path, the remaining path will not be added to the protection path. This is decision is made to simplify translation from (protection) paths to OpenFlow Flow Rules. The cost for the protection path will be calculated for the complete path, starting at S_A and terminating at S_B. To do so, we can easily compare the cost for link-based protection with path-based protection schemes.

Initialization

During the initialization, initial cost and path parameters are set, and remaining paths are computed. Remaining paths are used in the switch selection phase, where a further explanation on remaining paths is given later on this section. In algorithm 4 the pseudo code for the initialization phase is given, where m is the minimization setting, $\mathcal{C}(S_x)$ is the cost array to travel to switch S_x from S_i, U is the set of all switches in the network, C_P is the cost for the discovered path P, $C(S_A \rightarrow S_i)$ is the path cost to travel from S_A to S_i over the shortest path and $C(CR_{S_i})$ is the path cost for applied crankback routing for switch S_i.

In the modified Dijkstra algorithm, the initial path cost $C_P(S_i)$ is set to zero. Because the protection algorithms starts at switch S_i and we set the initial path cost to $C_P(S_i) = C(S_A \rightarrow S_i)$. Also, the initial cost array $\mathcal{C}(S_x)$ to travel to undiscovered switches in the network, differs from the modified Dijkstra algorithm. Regularly, all values in the cost array, except the source switch ($C(S_A) = C(S_A \rightarrow S_i)$), are set to infinity. For the protection algorithm,

Algorithm 4 Cost initialization for extended Dijkstra algorithm

STEP 1A - COST INITIALIZATION

1. CASE m is crankback reduction:
2. SET $\mathcal{C}(S_x) = \infty$, $S_x \in U$
3. CASE m is cost minimization:
4. SET $\mathcal{C}(S_x) = C(CR_{S_i})$, $S_x \in U$
5. SET $C_P(S_i), \mathcal{C}(S_i) = C(S_A \rightarrow S_i)$

two configuration options are possible. The first option is to set $\mathcal{C}(S_x)$ to infinity, as with the Dijkstra algorithm, where with crankback routing will be minimized. Because path Q_1 is removed from the network, the shortest path algorithm is forced to discover other paths. Discovered protection paths may have greater costs than the crankback cost $C(CR_{S_i})$ for switch S_i. The crankback cost $C(CR_{S_i})$ can be calculated with equation 5-2, where $C(PP_{S_A})$ is the protection path for switch S_A.

$$C(CR_{S_i}) = 2 \cdot C(Q_1(S_A \rightarrow S_i)) + C(PP_{S_A}) \tag{5-2}$$

Equation 5-2 can be explained as follows. First a packet must travel from S_A to S_i over the shortest path. At S_i a failure is detected and the packet must transverse the same path back to S_A, as crankback routing is applied. From switch S_A the protection path is chosen towards destination switch S_B.

The second configuration option sets cost array $\mathcal{C}(S_x)$ to undiscovered switches equal to $C(CR_{S_i})$. Here with the cost for protection paths is upper bounded by equation 5-3.

$$\mathcal{C}(S_x) < C(CR_{S_i}) \tag{5-3}$$

The second option guarantees that the cost of discovered protection paths never exceed the cost of crankback routing. Choosing one of both options is dependent on the requirements for the protection paths. One must choose which requirement is needed to protect the network. When optimized resource allocation and minimal path costs has high priority, the second configuration option to upper bound path costs is preferable. If minimizing crankback routing has high priority, the first configuration option is favorable. In chapter 6 simulations have been performed for both options to determine performance differences between both options.

With the cost parameters initialized, path initialization can start. The path initialization actions are given in algorithm 5 , where S_f is the switch in failure, Q_{1R} is the remaining shortest path, Q_{2R} is the remaining simple path, P is the discovered path and H is the priority queue.

The first step in path initialization is to set switch S_f, which is used in the switch selection process. In case of switch protection, S_f is the next hop in the shortest path to guarantee next hop switch disjointness. Next-hop link disjointness is guaranteed with the removal of the shortest path from the network. Therefore S_f is set as an not existing switch for link failure

Algorithm 5 Path initialization for extended Dijkstra algorithm

STEP 1B - PATH INITIALIZATION

1. CASE s is switch protection:
2. SET $S_f = S_{i+1}$, where $i + 1 \neq B$
3. SET $Q_{1R} = Q_1(S_{i+2} \rightarrow S_B)$, where $i + 2 < B$
4. CASE s is link protection:
5. SET $S_f = S_{-1}$
6. SET $Q_{1R} = Q_1(S_{i+1} \rightarrow S_B)$, where $i + 1 < B$
7. If S_i in Q_2:
8. SET $Q_{2R} = Q_2(S_i \rightarrow S_B)$
9. Else:
10. SET $Q_{2R} = Q_2$
11. SET $P = S_i$
12. INSERT $\{C_P(S_A), S_i, P\}$ in H
13. Go to Step 2 (SWITCH SELECTION)

protection. The next step is to compute the remaining shortest path Q_{1R} and remaining simple path Q_{2R} to destination switch S_B. These remaining paths are computed to classify switch neighbors during switch selection (Step 2) with the goal:

- *Provide valid solutions* - Q_{1R} - Because we removed the shortest path from the network, also possible solutions for protection paths are removed. Next-hop disjointness provides the required protection for our problem and a protection path that avoids switch S_f and returns to the *remaining* shortest path to the destination, is therefore a valid solution for link-based protection;
- *Reduce routing loops* - Q_{2R} - To reduce computational overhead, path Q_2 from Bhandari's algorithm, can be re-used. If no joint links where present during Bhandari's discovery process, conform table 5-1, $P_2 = Q_2$. If during the switch selection process a neighbor switch S_n is selected which is present in path Q_2 ($S_n \in Q_2$), that segment towards the destination does not need discovery, as it is already discovered. Reusing this segment for the protection path reduces computational overhead, but can cause routing loops when $P_2 \neq Q_2$ (joint links present) and switch $S_i \in Q_2$. If during the switch selection process a neighbor S_n is selected which is present in path Q_2 ($S_n \in Q_2$) and is closer to the source switch, re-application of the segment $Q_2(S_n \rightarrow S_B)$ causes routing loops in the form of $P(S_A \rightarrow S_i \rightarrow S_n \rightarrow S_i \rightarrow \cdots)$. Therefore the remaining simple path Q_{2R} is computed from S_i.

If link disjointness is set as required protection, the remaining shortest path Q_{1R} is computed as follows. Link $S_i \rightarrow S_{i+1}$ is assumed in failure state and Q_{1R} is defined as the remaining shortest path $Q_1(S_{i+1} \rightarrow S_B)$. For switch disjointness, switch S_{i+1} is assumed in failure and Q_{1R} becomes the path $Q_1(S_{i+2} \rightarrow S_B)$. The intermediate switch indicator (i) is limited by the destination switch, leading to path boundaries $i+1 < B$ (link disjoint) and $x+2 < B$ (switch disjoint). The remaining shortest path Q_{1R} is always computed, where as the computation of the remaining simple path Q_{2R} is not always possible. When paths Q_1 and Q_2 are completely disjoint and share no joint links, Q_{2R} does not have to be computed, as intermediate switch S_i is not present in Q_2. In the case of joint links, for at least two intermediate switches in the shortest path, Q_{2R} must be computed. The last step in the initialization phase is to add switch S_i to the discovered path P and insert C_{PP}, S_i and P into priority queue H.

Switch Selection

The second step in the protection algorithm is switch selection, where conform Dijkstra's algorithm the switch with minimal path costs is extracted from the priority queue and its neighbors are selected from the network $\mathcal{N}$. For the protection algorithm the selection process is extended to classify neighbor switches to provide protection and reduce computational overhead. The switch selection process is given in algorithm 6, where S_j is the extracted switch from the priority queue, V is the set of visited switches and S_n is the selected neighbor for switch S_j in network $\mathcal{N}$.

From priority queue H, the switch is selected with the lowest path cost C_P. If selected switch S_j is not visited yet, it is added to V. In the regular Dijkstra algorithm, the neighbors S_n of S_j are directly forwarded to the relaxation process (Step 3), where in our protection algorithm the neighbors are first classified. Classification of neighbor switches can have four different outcomes:

1. *Failure* - S_n^F - Switch S_n is identified as the switch in failure S_f ($S_n = S_f$). This classification is only possible when switch failure protection is chosen;
2. *Short* - S_n^S - Switch S_n is classified as *Short* if S_n is present in the remaining shortest path Q_{1R} ($S_n \in Q_{1R}$). During the selection process, selected switch S_j can have multiple neighbors which are present in the remaining shortest path. In the relaxation process the most optimal path is identified and chosen;
3. *Backup* - S_n^B - Switch S_n is classified as *Backup* if S_n is present in the remaining simple path Q_{2R} ($S_n \in Q_{2R}$). For this classification also multiple neighbors can be classified with S_n^B;
4. *Unvisited* - S_n^U - Switch S_n is not classified as one of the above and is therefore classified as regular unvisited switch.

After the classification the selected neighbor is forwarded to the relaxation process to determine if the path costs for the protection paths can be relaxed. The algorithm stops when the extracted switch S_j is the destination switch, where after the discovered protection path is returned. When no feasible protection path is discovered and the priority queue is empty, crankback routing must be applied. The protection path is cranks back to switch S_{i-1}, where

Algorithm 6 Switch selection for extended Dijkstra algorithm

STEP 2 - SWITCH SELECTION

1. WHILE H is not empty:
2. EXTRACT-MIN $\{C_P(S_j), S_j, P\}$ such that $C_P(S_j) = min_{S_x \in H} \mathcal{C}(S_x)$
3. IF S_j is S_B:
4. Stop algorithm and return $\{PP(S_i), C(PP(S_i))\}$
5. IF S_j not in V:
6. ADD S_j to V
7. FOR all neighbors S_n of S_j in $\mathcal{N}$:
8. CLASSIFY S_n
9. RELAXATION $\Longrightarrow PP(S_i), C(PP(S_i))$
10. IF $PP(S_i)$ not discovered:
11. SET $PP(S_i) = S_i + S_{i-1} + PP(S_{i-1})$
12. SET $C(PP(S_i)) = C(S_A \rightarrow S_i) + w(l_{i-1 \rightarrow i}) + C(PP(S_{i-1})) - C(S_A \rightarrow S_{i-1})$
13. Stop algorithm and return $\{PP(S_i), C(PP(S_i))\}$

$i - 1 \geq A$. This means that the protection path not fully cranks back to source switch S_A, but only one hop downstream of the shortest path, resulting in the protection path given on line 11 in algorithm 6. The corresponding path cost equation for $C(PP(S_i))$ is given on line 12 of the algorithm. With forced crankback routing two situations can occur. The first situation is that for all downstream switches no protection path is discovered and for all these switches forced crankback routing is applied. Protection is provided by crankback routing to the source switch. The second situation is that for one of the downstream switches a protection path is discovered other that crankback routing, indicating that no full crankback to the source switch is required.

Relaxation

During the relaxation process, the path costs to travel to the selected neighbor S_n are compared with the cost array $\mathcal{C}(S_x)$. The relaxation process for the protection algorithm is given in algorithm 7.

When a neighbor switch is classified as S_n^F, no relaxation actions are performed. Traveling over failure switch S_f must be rejected to ensure next-hop switch disjointness in case of switch failure protection. The relax condition for unvisited neighbor S_n^U are equal to the regular Dijkstra algorithm. On relaxation, the cost array $\mathcal{C}(S_n)$ and path cost $C_P(S_n)$ for

Algorithm 7 Relaxation for extended Dijkstra algorithm

STEP 3 - RELAXATION

1. CASE S_n is classified as S_n^F:
2. PASS
3. CASE S_n is classified as S_n^S:
4. If $C_P(S_j) + w(l_{j\to n}) + C(Q_{1R}(S_n \to S_B)) < \mathcal{C}(S_D)$:
5. SET $C_P(S_B), \mathcal{C}(S_B) = C_P(S_j) + w(l_{j\to n}) + C(Q_{1R}(S_n \to S_B))$
6. SET $PP_{S_i} = P + S_n$
7. INSERT $\{C_{PP}(S_B), S_B, PP_{S_i}\}$ in H
8. CASE S_n is classified as S_n^B:
9. If $C_P(S_j) + w(l_{j\to n}) + C(Q_{2R}(S_n \to S_B)) < \mathcal{C}(S_D)$:
10. SET $C_P(S_B), \mathcal{C}(S_B) = C_P(S_j) + w(l_{j\to n}) + C(Q_{2R}(S_n \to S_B))$
11. SET $PP_{S_i} = P + S_n$
12. INSERT $\{C_{PP}(S_B), S_B, PP_{S_i}\}$ in H
13. CASE S_n is classified as S_n^U:
14. IF $\mathcal{C}(S_j) + w(l_{j\to n}) < \mathcal{C}(S_n)$:
15. SET $C_P(S_n), \mathcal{C}(S_n) = \mathcal{C}(S_j) + w(l_{j\to n})$
16. SET $P = P + S_n$
17. INSERT $\{C_P(S_n), S_n, P\}$ in H

the neighbor switch is updated and the discovered path P is extended with S_n. The path cost, neighbor switch and the discovered path are inserted to the priority queue H for further processing. For the two other classifications, being S_n^S and S_n^B , the relaxation is performed in relation with the computed remaining paths to the destination. Instead of inserting neighbor switch S_n into the queue, destination switch S_B is inserted together with the discovered protection path $PP(S_i)$ and its path cost. In this manner, at switch selection, the destination switch is extracted from the queue and the algorithm will return the discovered protection path. A big benefit of this method is that the algorithm can terminate computations for the resulting path from S_n to destination switch S_D, as these are already computed. The classification and relaxation process will thus require less computations in comparison with regular Dijkstra iterations. Note that the discovered protection path $PP(S_i)$ only travels to neighbor switch S_n. From switch S_n the protection path follows one of both remaining paths. Flow Rules for the remaining paths and the discovered protection path would be

equal, so extending $PP(S_i)$ with segments $Q_{1R}(S_n \rightarrow S_B)$ or $Q_{2R}(S_n \rightarrow S_B)$ provides no extra information. The total path cost is calculated for the complete paths. So, from source switch S_A over the shortest path to switch S_i, from where the protection path is followed to destination switch S_B. As mentioned at the beginning of this section, this is done to compare path cost between link- and path-based protection schemes.

5-4-4 Complexity for extended Dijkstra algorithm

In appendix D, the complexity for the modified Dijkstra algorithm is derived at $O((L + N) \cdot \log N)$ with application of binary heaps, where N is the number of switches and L is the number of links. Binary heaps are chosen, as these are utilized in the actual software implementation. The difference between the modified Dijkstra and the extended algorithm is mainly found in the application of the remaining paths and neighbor switch classification. The computations to determine the remaining paths require $O(N)$ complexity, as the maximal path length is $N - 1$. Classification of the neighbor switches require no extra complexity in comparison with the modified Dijkstra algorithm. With the extensions, the overall complexity for the extended Dijkstra algorithm is determined at $O((N + L) \cdot \log N + N)$

5-5 Example for link-based protection algorithm

To clarify the choices made during the development of the protection algorithm and the overall process of the algorithm with the two minimization configuration options for cost initialization, protection paths will be discovered in an example topology. In figure 5-3 the results of phase A of the protection algorithm is illustrated, where between switch S_1 and S_4, shortest path Q_1 is discovered with Dijkstra's algorithm, after which Bhandari's algorithm is executed to discover Q_2 and construct P_2.

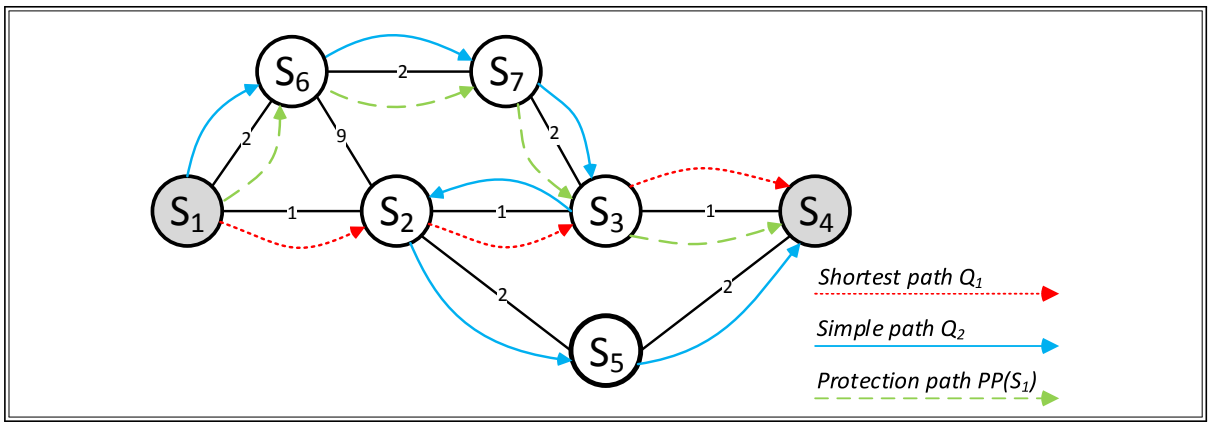


Figure 5-3: Example topology with phase A of protection algorithm executed - *The shortest path is discovered as $Q_1(S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4)$. With Bhandari's algorithm the simple path is discovered as $Q_2(S_1 \rightarrow S_6 \rightarrow S_7 \rightarrow S_3 \rightarrow S_2 \rightarrow S_5 \rightarrow S_4)$. Between Q_1 and Q_2 a joint link is identified as $l(S_2 \rightarrow S_3)$. Path P_2 is constructed as $P_2(S_1 \rightarrow S_6 \rightarrow S_7 \rightarrow S_3 \rightarrow S_4)$ and is set as protection path $PP(S_1)$ for switch S_1.*

In the example topology of figure 5-3 Dijkstra's algorithm discovered the shortest path as $Q_1(S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4)$. With Bhandari's algorithm network modifications are performed to discover the second shortest path $Q_2(S_1 \rightarrow S_6 \rightarrow S_7 \rightarrow S_3 \rightarrow S_2 \rightarrow S_5 \rightarrow S_4)$. Between Q_1 and Q_2 a joint is identified as $l(S_2 \rightarrow S_3)$. This means that $C(P_1) > C(Q_1)$ and in phase B the remaining simple path Q_{2R} must be computed. From Q_1 and Q_2 the disjoint paths P_1 and P_2 are constructed. We set P_2 as protection path $PP(S_1)$ for source switch S_1. Protection path $PP(S_1)$ is not complete disjoint from the shortest path, as link $l(S_3 \rightarrow S_4)$ is joint. This is not a problem, as we only require next-hop disjointness. For switch S_2 and S_3 protection paths are discovered in phase B with the execution of our extended Dijkstra algorithm. In figure 5-4 the results for phase B are given.

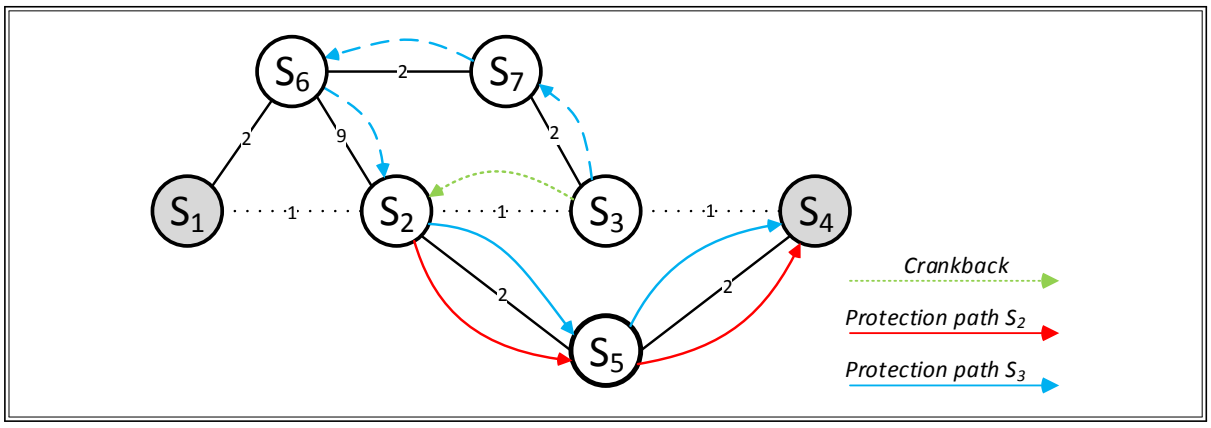


Figure 5-4: Example topology with phase A of protection algorithm executed - *Path Q_1 is removed from the network and for intermediate switches S_2 and S_3 the protection paths are discovered. The first neighbor for S_2 is found in S_5, where S_5 is present in the remaining simple path Q_{2R}. Therefore neighbor switch S_5 is classified as S_n^B . In the relaxation process the protection path is set as $PP_{S_2}(S_2 \rightarrow S_5)$* with cost *$C(PP_{S_2}) = 5$. From switch S_5 the path continuous to destination switch S_4. For S_3 the outcome is dependent on the initial cost configuration setting. The discovered path $PP^1_{S_3}(S_3 \rightarrow S_7 \rightarrow S_6 \rightarrow S_2 \rightarrow S_5 \rightarrow S_4)$, avoids crankback routing, where path $PP^2_{S_3}(S_3 \rightarrow S_2 \rightarrow S_5 \rightarrow S_4)$ minimizes path costs.*

During phase B of the computing scheme Q_1 is removed from the network. The next step is to compute and set protection paths for switches S_2 and S_3. We start with the initialization process for intermediate switch S_2. The initial cost array to reach unvisited switches ($\mathcal{C}(S_x)$) can be set to the crankback cost of switch S_2 to minimize cost for protection paths or $\mathcal{C}(S_x)$ is set to infinity to minimize crankback routing. With equation 5-2 the crankback cost is calculated as follows. From S_2 the path cranks back to S_1, from where PP_{S_1} is followed, resulting in $C(CR_{S_2}) = 9$. The remaining shortest path for S_2 is computed as $Q_{1R} = Q_1(S_3 \rightarrow S_4)$. Because switch S_2 is present in Q_2, the remaining simple path is computed as $Q_{2R} = Q_2(S_5 \rightarrow S_4)$. After initialization, the switch selection process starts.

Switch S_5 is selected first as neighbor and is classified as S_n^B. The cost for the discovered protection path $PP_{S_2}(S_1 \rightarrow S_2 \rightarrow S_5 \rightarrow S_4)$ is determined at $C(PP_{S_2}) = 5$. Because the cost is lower than the initial cost for the crankback path, this solution is relaxed and destination switch S_4 is inserted into the priority queue. The second neighbor S_6 does not lead to relaxation and is therefore discarded as solution. From the queue switch S_4 is extracted and $PP(S_2)$ is returned.

For switch S_3 the extended Dijkstra algorithm is executed again, where the crankback cost is calculated at $C(CR_{S_3}) = 11$. The only neighbor for S_3 is found in S_7, which leads to protection path $PP^1_{S_3}(S_3 \rightarrow S_7 \rightarrow S_6 \rightarrow S_2 \rightarrow S_5 \rightarrow S_4)$ with cost $C(PP_{S_3}) = 19$. From here two options are possible. If cost minimization has highest priority, the discovered path will be discarded, as $C(PP^1_{S_3}) > C(CR_{S_1})$. Cranking back to switch S_2 and utilize PP_{S_2} leads to relaxation, as the $C(PP^2_{S_3}) = 7$. If crankback minimization has highest priority, the discovered path $PP^1_{S_3}$ is valid and will be returned as solution. As seen in the provided example, the protection algorithm can discover protection paths for different requirements. Either the protection path cost is minimized or crankback routing is minimized. Also the number of computations to discover the protection path is reduced with the application of remaining paths.

5-6 Enhancing path computations with SDN

Although not the scope of this thesis, information available at the OpenFlow controller can enhance shortest path computations. The protection algorithm discussed in the previous sections, uses integer link weights for path computations. In real-world networks, integer link weights are not allocated to links, but non-fixed parameters like available bandwidth or link utilization are present. In [7] OpenFlow is utilized to determine QoS metrics, based on real-time measurements on available bandwidth, delay and packet loss. Such QoS metrics would be ideal cost indicators for network links. We will show that the protection algorithm can easily be modified and enhanced to incorporate QoS metrics from other OpenFlow modules. In [77] a proposal is given on implementing bandwidth parameters for shortest path computations. Two schemes are defined, being *short-wide* and *wide-short* path computing. In here, two link parameters, bandwidth and hop count, are combined to discover the shortest paths. The short-wide scheme computes a shortest path based on hop count and on equal costs, the bandwidth cost is used as secondary selection mechanism. Wide-short is exactly the opposite, where the widest path is computed using bandwidth parameters and on equal bandwidth, the hop-count selects the shortest path. Bandwidth information on switch links are directly available from the OpenFlow protocol. Though, optimal and exact multi-parameter shortest path computations can be executed with algorithms, like the Self Adapting Multi Constraint Routing Algorithm (SAMCRA) [78] for QoS routing purposes. SAMCRA performs multiple (number of cost parameters) Dijkstra iterations to compute intermediate shortest paths for each parameter. Afterwards, a full iteration over the discovered solutions is performed to compute the optimal multi-constraint shortest path. In [77] the goal is to retrieve the short-wide or wide-short path with a single Dijkstra iteration.

The problems with the wide-short and short-wide schemes is the translation of bandwidth to a cost parameter. In [77], no solution is given, but the Open Shortest Path First (OSPF) [53] provides a guideline. OSPF computes shortest paths based on bandwidth and applies equation 5-4 for cost translation, where $c_{l,B}$ is the link weight, B_{ref} is the reference bandwidth set by the network manager and B_l is the actual bandwidth of the link.

$$c_{l,B} = \frac{B_{ref}}{B_l} \tag{5-4}$$

A default reference bandwidth is set as $100Mbps$, a standard bandwidth for home Ethernet networks. For larger networks, $1Gbps$ links are set a minimum standard. In chapter 6 simulations have been performed to show how information available at SDN networks can enhance shortest path computations. Ideally, QoS cost parameters are combined to one cost parameter with some additional logic, where after shortest paths algorithms can discover the most optimal path to avoid the more complex relaxation and path selection processes. Although this seems an easy process, the general QoS path selection is an NP-hard problem [79]. In [79] an exact polynomial time algorithm is given that selects multi-constrained paths using link-state information with bounded error margins. Integration of QoS path selection with protection algorithms to increase robustness will remain a hot topic in SDN network research.

5-7 Complexity of computing scheme for link-based protection paths

Algorithms and computing schemes are commonly judged on computing complexity. With the computing scheme discussed in detail, the total complexity for the optimal solution can be derived in worst case situations and the actual software implementations kept in mind. To give a clear overview of the complexity of the protection algorithm, the actions performed are given in table 5-2. Here the actions from the Bhandari's algorithm, discussed in appendix C, are also mentioned. The complexity for the implemented modified Dijkstra algorithms is determined at $O((N+L)\log N)$ in appendix D, where the complexity for the extended algorithm is determined at $O((N+L)\log N+N)$ in subsection 5-4-4.

Operation	*Complexity*	*Comment*
1,4. Dijkstra discover $Q_1 + Q_2$	$O((N+L)\log N)$	Binary heaps
2. Reverse Links	$O(N)$	Related to path length
(3.) Split swithes + rewire links	$O(N+L)$	Only for switch protection
(5.) Translate Q_2	$O(N)$	Only for switch protection
6. Find joint linkes	$O(N)$	Related to path length
7. Construct P_1 and P_2	$O(N)$	Related to path length
8. Remove Q_1 from network	$O(N)$	Phase B algorithm
9. Discover protection paths	$O(N((N+L)\log N+N))$	Related to path length
Total complexity	$O(N^2(\log N+1)+NL\log N)$	Derived from operation 9.

Table 5-2: Overall complexity of protection algorithm

As seen in table 5-2, Bhandari's disjoint path algorithm (Operation 1 to 7) is dominated by the complexity of the modified version of Dijkstra's algorithm. Operation 2, 5, 6, 7 and 8 are all related to the path length (due to the nature of the shortest path algorithm), which is in worst case equal to N. The switch splitting operation (3.) is related to the path length and the rewire actions can include all links in the network, which why the complexity is set as $O(N+L)$. Operation 9 dominates the overall complexity, as the extended Dijkstra algorithm must be performed in worst case $N-2$ times. The total complexity for the protection algorithm is therefore derived from this operation and the result is an algorithm that runs in

polynomial time. This makes that the scheme is unsuited for very large networks. Looking to the application area of the scheme, very large networks are not expected, as the number of switches assigned to a SDN controller is limited by scalability issues. Also the worst case scenario is applied, where the path length is equal to the number of switches in the network. In reality, paths will be shorter and therewith the computing time is reduced.

5-8 Conclusion on link-based protection algorithm for SDN networks

In this chapter we defined the algorithmic problem to increase survivability of the network. Our problem was to discover a cost minimized path between two switches in the network and for all switches in the discovered path, guarantee protection to a single link or switch failure. To solve our problem, we investigated two existing solutions, being redundant path trees and disjoint path algorithms.

Redundant trees do not offer path cost minimization, which is one of our requirements. Suurballe's and Bhandari's disjoint path algorithms can guarantee protection in the network with the execution of only two Dijkstra shortest path computations. Therefore Bhandari's disjoint path algorithm is chosen as basis for our protection algorithm.An investigation into disjoint path algorithm showed that four disjoint levels can be distinguished, where in this chapter only link and switch disjoint paths are discussed. The downside of Bhandari's disjoint path algorithm is that only MIN-SUM path pairs can be computed, where the MIN-MIN cost objective is required to solve our problem. Another problem is that Bhandari's algorithm does not provide solutions for intermediate switches and is therefore only usable for path-based protection schemes. Because we do not require full link or switch disjoint paths, but only next-hop disjointness, we can use use the provided solutions from Bhandari's algorithm and also comply with the MIN-MIN cost objective to solve our algorithmic problem.

With Bhandari's algorithm used as basis, we developed a protection algorithm that minimizes cost on the primary path, guaranteeing protection for intermediate switches and reducing computational overhead. To discover protection paths for the intermediate switches, an extension is made to Dijkstra's shortest path algorithm, such that requested survivability is guaranteed and crankback routing or protection path cost is minimized. The result is an algorithm that runs in polynomial time and solves the algorithmic problem. As the algorithm is designed for SDN networks, link parameters available at the network controller can be integrated to the protection algorithm, resulting in a more optimized, but more important, more robust networks. We showed an example on how our protection algorithm discovers the requested paths with different minimization settings applied. The next step is to measure the benefits of our protection algorithm by performing a large number of simulations to show the reduction in forced crankback routing and reduction in path cost in comparison with path-based protection schemes.

Chapter 6

Simulation results link-based protection algorithm

In this chapter we will perform simulations on our developed protection algorithm for link-based protection paths. The main goal is to show that link-based protection is a valid protection scheme for networks, where paths costs or crankback routing can be minimized. To perform simulations, we implemented the protection algorithm into a software module and a large number of simulations have been executed on random and real-life networks. In section 6-1 a description of the simulation is given, where used software tools and applied networks are shortly discussed. Section 6-2 shows graphical results of intermediate products of our protection algorithm, where disjoint path discovery is combined with *shortest-widest* and *widest-shortest* routing requirements. To show characteristics of computed protection paths, multiple graphical outputs are given in section 6-3. Section 6-4 discusses the simulation results for the protection algorithm with the minimized cost and crankback configuration settings. This chapter is concluded in section 6-5.

6-1 Description of simulation

The goal of the simulations is to show the performance gain from link-based over path-based protection schemes. For each simulation a network topology is generated and between two random chosen switches, a shortest path with protection paths is requested from our protection algorithm. We set crankback routing and path-based protection as reference solution. Each of the discovered protection paths during the simulations is compared with the reference solutions on two parameters. The first performance parameter is path cost. For path-based protection, for each intermediate switch in the shortest path, the cost for protection is equal to the crankback cost given in equation 5-2. Equation 6-1 gives the path cost ratio α between link- and path based protection, where $C(PP_{S_i})$ is the cost for the protection path discovered with our protection algorithm (link-based protection) and $C(CR_{S_i})$ is the reference path cost (path-based protection) at intermediate switch S_i and N is the total number of requested protection paths during the simulations.

$$\alpha = \frac{\sum_{j=1}^{N} \frac{C(PP_{S_i})}{C(CR_{S_i})}}{N} \tag{6-1}$$

In equation 6-1 the path cost ratio provides a normalized indicator between link- and path based protection averaged over a large number of simulations. For path cost minimization settings α is bounded by $0 < \alpha \leq 1$, where α is only lower bounded with minimized crankback settings by $\alpha > 0$. If $\alpha < 1$, the link-based protection algorithm offers path solutions with overall lower path costs. With $\alpha > 1$, protection paths are discovered, but with overall higher path costs in comparison with path-based protection. The second performance indicator is the ratio of *forced* crankback paths. When no protection path is discovered by the protection algorithm, crankback routing is necessary. Equation 6-2 gives the crankback reduction ratio β between link- and path based protection, where N_{CR} represents the number of forced crankback solutions found during the simulations.

$$\beta = 1 - \frac{N_{CR}}{N},\ N_{CR} \leq N \tag{6-2}$$

In equation 6-2 the crankback reduction ratio β is bounded by $0 \leq \beta \leq 1$. When no forced crankback routing was required during simulations ($N_{CR} = 0$), the need for crankback routing is completely reduced ($\beta = 1$). If $\frac{N_{CR}}{N} = 0.2$, meaning that for 20% of the protection paths forced crankback routing was necessary, the ratio for forced crankback routing is reduced with 0.8. This indicates that for 80% of the intermediate switches for which a protection path is requested, a better solution is discovered than crankback routing. With the two defined indicators it is possible to show the performance of our proposed link-based protection algorithm.

6-1-1 Available tools

To implement our proposed protection algorithm to a software module, we researched various software solutions which provide network simulations, path discovery algorithms and processing tools. The Python NetworkX [80] module (version 1.8) is identified as most suitable for our needs, as it can generate a variety of random networks. Switches and links can easily be modified and multiple well known routing algorithms are implemented as standard. Because NetworkX is a Python-module, additional programming is performed using standardized software modules from the Python programming language version 2.7.6 [81] and the Spyder development environment [82]. From NetworkX the graph generators are used to create random networks for simulations. The network generators create a Python dictionary with switches and links, where switch-names and link parameters must be added by the user after generation.

The Dijkstra shortest path algorithm is available as module within NetworkX, but lacks clarity to implement the required modifications for our needs. Therefore we implemented the modified Dijkstra algorithm (Appendix D-0-7), to a NetworkX compatible module. The required modifications for our extended version of Dijkstra's algorithm and Bhandari's disjoint path algorithm are implemented upon our developed module and used for extensive testing and simulations on a variety of random networks. Overall we advise the NetworkX module for network prototyping and algorithm testing.

The Spyder interpreter is developed for scientific and prototyping purposes. From scratch, statistical, processing and plotting (SciPy, NumPy, MatPlotLib) modules are loaded, which simplified the development and implementation of the protection algorithm, as well as processing of the obtained simulation results to retrieve the performance indicators α and β.

6-1-2 Random real world networks

For simulations, we utilized Erdös-Rényi (ER) and Barabási-Albert (BA) random networks available within NetworkX, as well as real-world topologies. The generation of ER and BA random networks and the example real-world networks are shortly discussed.

Erdös-Rényi networks

ER networks [83] are random networks, where the degree distribution of the switches is normal distributed (Gaussian). The degree of a switch is the number of connections with other switches. An ER-random graph is generated with two parameters, being the number of switches in the network (N) and the probability (p_{attach}) that a switch connects to an another switch in the network. The generation of the network is as follows. An initial network of size N is defined and for each switch to any other switch in the network a random number is generated between $0 \leq p_{random} \leq 1$. If $p_{random,x} \leq p_{attach}$, a connection between the switches is defined. Otherwise, no connection is established. A higher value for p_{attach} will result in more links and higher connectivity in the network. The network generation process does not guarantee a connected network. ER-networks can be used for network simulations where each switch has approximately the same degree.

Barabási-Albert networks

Barabasi-Albert random networks [84] have a power-law degree distribution, which indicates that a large number of switches in the network have a low degree and a few have a very high degree. The high degree switches can be visioned as central nodes, where connections are centralized and connectivity to other parts of the network are provided. These aspects are also visible in data-networks, where a centralized switch is intersecting and connecting multiple parts of the network. BA-random networks are often used for simulating large data networks, as data networks often have a power-law distribution.

BA-random networks are generated with two parameters, being the number of switches and the number of links to attach (M) from a new switch to existing switches. The generation of a BA-network is based on preferential attachment and starts with an initial network of size M. Each new switch has M new links, which connect to the existing network via a probability distribution. The higher the degree of a switch, the higher the probability that a new link will attach to that switch. So, higher degree switches will establish more connections with newly attached switches. The attachment process continues until a network with size N is generated.

Real-world topologies

Two common used reference networks for real-world applications and algorithm performance testing are i) the sample US network (USNET), which is constructed from $N = 24$ switches and $L = 43$ links and ii) the Pan-European COST239 (Pan-EUR) test network, constructed from $N = 11$ switches and $L = 26$ links [85, 86]. Both topologies are given in figure 6-1.

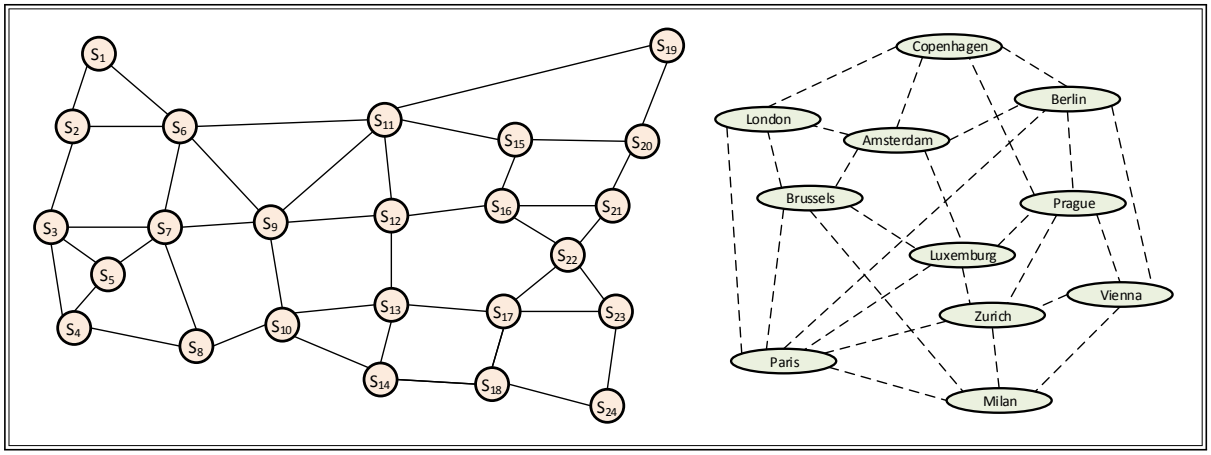


Figure 6-1: Real-world topologies used for simulations - *Left: USNET - Right: Pan-European COST239*

6-1-3 Network parameters and number of simulations

To perform simulations on the defined random networks, basic network parameters must be set. From the complex network point of view, the number of switches must be high to fully express the characteristics and degree distributions of random networks. A high number of switches and links will result in long simulation times. In chapter 3 scalability issues for SDN are discussed, from which we learned that the number of SDN switches assigned to a single centralize controller must be limited. Therefore we have chosen to simulate BA and ER networks with $N = 100$ switches. The BA-networks are taken as reference, where $M \in (2, 6, 10)$, to vary the number of links and connectivity of the network. To equalize the conditions between ER- and BA-networks, a translation is made from the number of links in BA-network to the same order of links in ER-networks. NetworkX provides a random network generator that uses the number of links as input for ER-random network generation, instead of attachment probability. For our simulations, this module is utilized, with the order of links in BA-networks as reference.

Link cost can be chosen unity over all links or can be randomly assigned within a defined interval. Random assignment of the link cost will provide more variety in paths and simulates integration of QoS parameters shortest path discovery. For our simulations, link cost (l_c) is assigned randomly from $l_c \in (1, 2, ..., 100)$. As secondary link parameter, link bandwidth (l_B) can be assigned randomly from $l_B \in (10, 100, 1000, 10000)$. The number of simulations (V) determines the resolution of the results. Because we also want to analyze the results at per hop basis and longer paths occur with a low rate, a high number of simulations is required for more detailed analysis. Multiple test runs have been performed to determine

the optimal number of simulations, where $V = 10^5$ provides a good compromise between simulation running time and resolution of the results.

6-2 Graphical results disjoint path algorithm

Before the results for the performance indicators are given, intermediate results from the protection algorithm (phase A) are shown in figures 6-2 to 6-5. These intermediate results are clarification examples and many more simulations have been executed in section 6-4. For the illustrations, disjoint paths are computed with Bhandari's algorithm in an ER-random network with $N = 15$, $p_{attach} = 0.7$ and $k = 2$. In figure 6-2 only the link parameter l_c is set. For clarity sake, the link costs are not plotted.

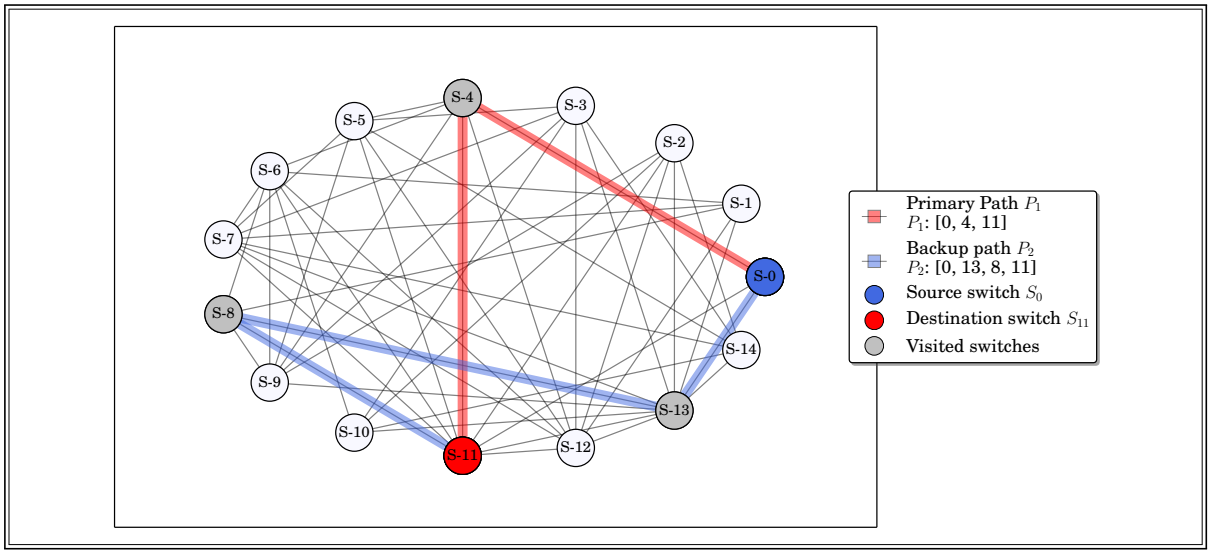


Figure 6-2: Disjoint path pair between S_0 and S_{11} using switch failure protection - *The primary path P_1 and backup path P_2 are completely disjoint. A failure on switch S_4 does not lead to disconnection between the source and destination.*

The primary path is computed as $P_1(S_0 \rightarrow S_4 \rightarrow S_{11})$, where the disjoint backup path is discovered as $P_2(S_0 \rightarrow S_{13} \rightarrow S_8 \rightarrow S_{11})$. Both paths do not share intermediate switches and are therefore switch disjoint. For figure 6-3 and 6-4 modifications at the relaxation process are added to our implementation of Dijkstra's algorithm with the aim to show the effects of *shortest-widest* paths. The link cost is set to unity ($l_c = 1$) for hop count computations and the second link parameter l_B is added to the network dictionary.

The shortest-widest algorithm first selects the path based on the number of hops and as secondary parameter the bandwidth is used. Figure 6-4 shows an example, where the hop count for the discovered paths is equal, but the overall bandwidth for the primary path is higher.

In the random network of figure 6-4, P_1 and P_2 are discovered with equal hop length. P_1 is chosen as primary path, as it has the lowest cost on the second parameter (higher bandwidth). For shortest widest paths (hop count is used as primary cost parameter), the higher order

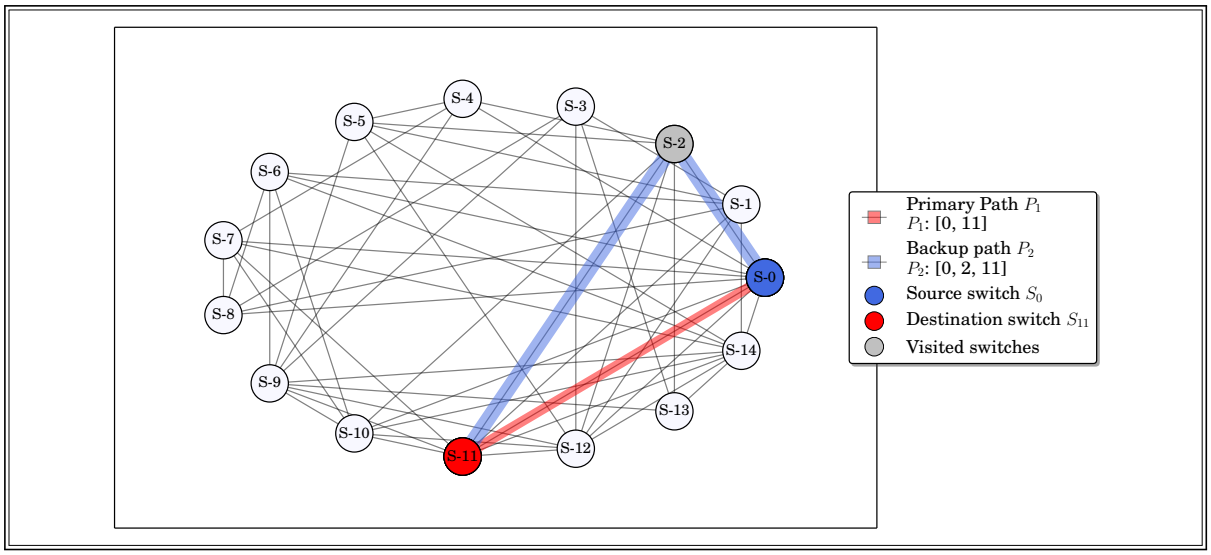


Figure 6-3: Disjoint path pair between S_0 and S_{11} using switch failure protection and shortest-widest path discovery - *The discovered path P_1 is the shortest path between S_0 and S_{11}, as the path only needs one hop (direct link). Backup path P_2 has more overall bandwidth, as visible to the width of the links in the paths, but takes more hops.*

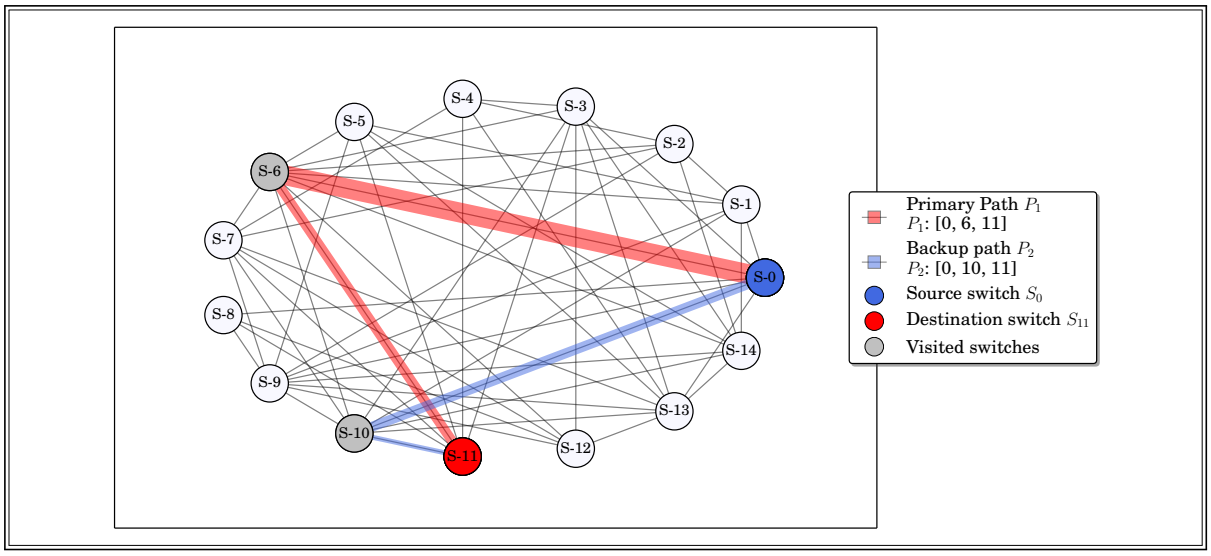


Figure 6-4: Disjoint path pair between S_0 and S_{11} with equal hop count discovered with shortest-widest settings - *The discovered paths have equal hop count, but P_1 is chosen as primary as the overall bandwidth is greater.*

parameters will have more influence, as more paths exist between source and destination with equal hop length. In figure 6-5 an example is given of a widest-shortest disjoint path.

The example of figure 6-5 shows that small modifications to the cost parameters of the shortest path algorithm can result in major differences in discovered disjoint paths. When multiple cost parameters are available, a choice has to be made which parameter is leading during the path computations. If an overall optimized path is required, SAMCRA [87] like algorithms

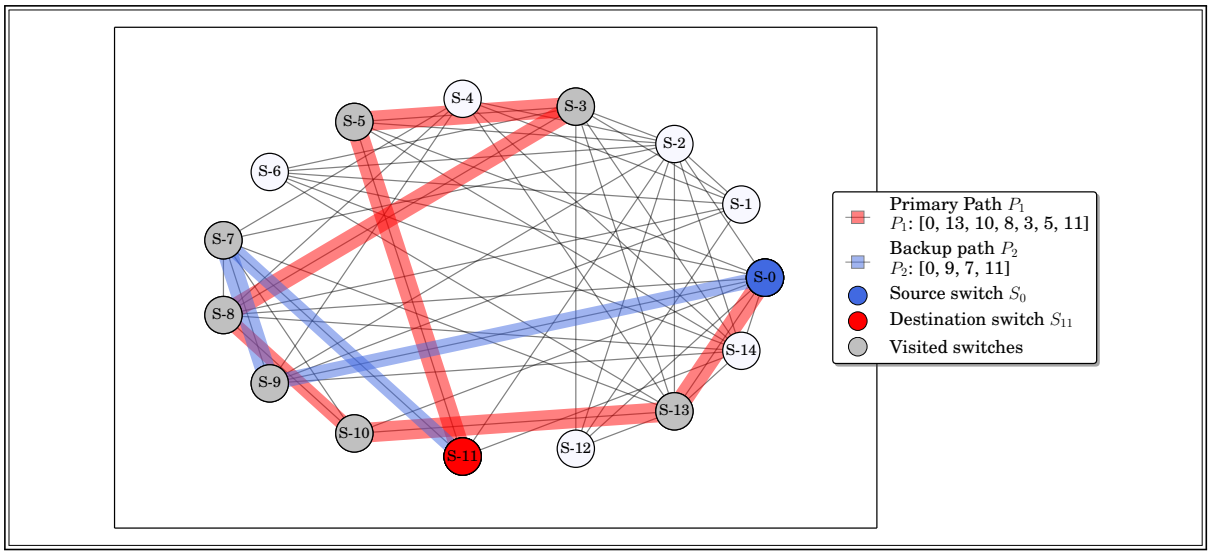


Figure 6-5: Disjoint path pair between S_0 and S_{11} with equal hop count discovered with widest-shortest settings - *The discovered primary path has lowest bandwidth cost and is therefore chosen over the backup path, which has lower hop count.*

must be applied. In case one link parameter has preference over the other, algorithms that compute shortest-widest or widest-shortest paths can be applied. These algorithms obtain a multi-constraint path within a single Dijkstra run, without compromising performance of the algorithm.

6-3 Graphical results link-based protection algorithm

In this section multiple scenarios are shown from simulation output of the protection algorithms, where variations are made in i) switch or link protection and ii) cost or crankback minimization. We generated BA-random networks with $N = 15$, $M = 2$ or $M = 6$ and the shortest path is requested between switch S_0 and S_{11}. In figure 6-6 the protection paths are given for each intermediate switch in the shortest path, providing switch failure protection with minimized cost settings.

For each intermediate switch in the shortest path a protection path is discovered, which provides protection against switch failures. Due to the cost minimization setting, all discovered protection paths have lower path costs in comparison with applied crankback routing. At switch S_2, a protection path is discovered to the remaining shortest path, where for switch S_6 and S_3 protection is provided by simple path Q_2. The example from figure 6-6 also shows that the simple path Q_2 plays an important role in the execution of our protection algorithm. It not only provides protection for S_0, but it also provides the basis for protection paths PP_{S_6} and PP_{S_3}. No joint links were present in the disjoint path computation process (phase A), so the simple path is equal to the protection path for S_0 ($PP_{S_0} = Q_2$). Figure 6-7 shows an example topology where a joint link was present in the execution of Bhandari's algorithm. The paths are discovered in a random BA-network with switch failure protection and minimized crankback settings, where $N = 15$ and $M = 2$.

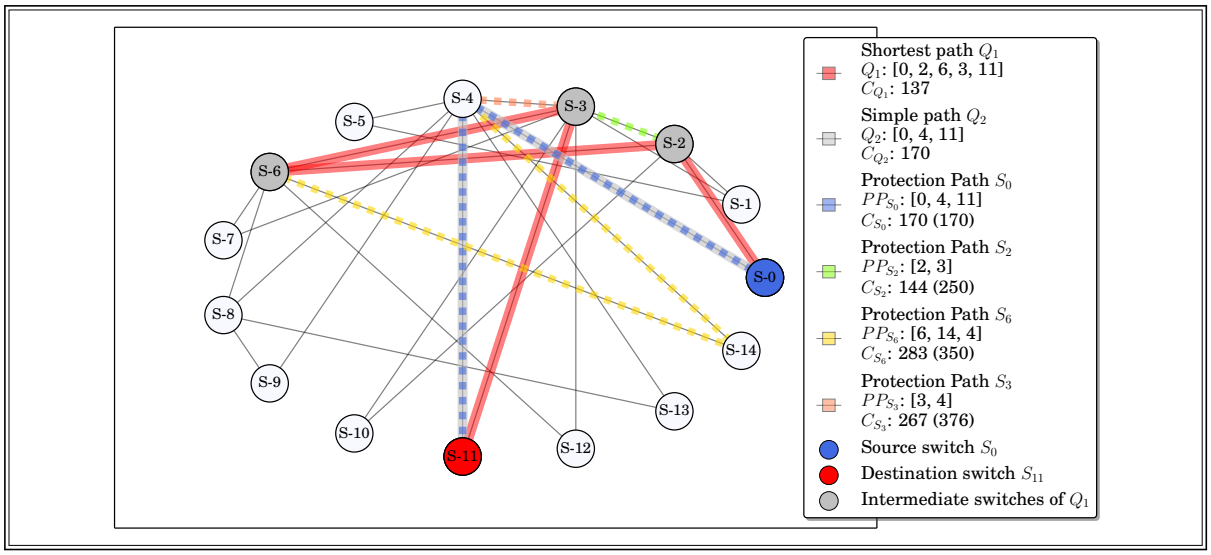


Figure 6-6: Protection paths on shortest path between S_0 and S_{11} with minimized cost settings ($M = 2$) - *For each intermediate switch in the shortest path Q_1 protection paths are computed. Simple path Q_2 (PP_{S_0}) provides protection against failure of switch S_2. From S_2 the path $S_2 \rightarrow S_3$ provides protection against failures of S_6 and joins with the remaining shortest path. Protection for S_6 is discovered via S_{14} to S_4, where PP_{S_6} merges with Q_2. At S_3, again a protection path is discovered to Q_2.*

Due to the presence of the joint link in the computing process, the shortest path Q_1 and the simple path Q_2 are not switch disjoint and share links $S_2 \rightarrow S_9$ and $S_9 \rightarrow S_3$. For path-based protection, this would not be a valid option, but for link-based protection, protection path $PP_{S_0}(P_2)$ provides protection against failure of switch S_2. Protection path PP_{S_9} has higher cost $C(PP_{S_9}) = 299$ compared with the cost for crankback routing $C(CR_{S_9}) = 233$. With minimized cost settings this path would not have been discovered and crankback routing would have been applied. In figure 6-8 the connectivity of the network is increased with $M = 6$ and the behavior of cost minimized protection paths is illustrated.

The application of cost minimization leads to unnecessary crankback routing, as valid solutions are not discovered. Instead of completely cranking back to the source switch, path $PP_{S_{14}}$ only cranks back to switch S_6. This choice leads to lower path cost $C(PP_{S_{14}}) = 75$ in comparison to full crankback routing $C(CR_{S_{14}}) = 117$. At last, we show the behavior of the protection algorithm for link failure protection in figure 6-9.

Protection path $PP_{S_{10}}$ is discovered with the assumption that only the link $S_{10} \rightarrow S_4$ is in failure and switch S_4 is in functional state. This path provides a detour to the remaining shortest path via switch S_1 and S_7. From switch S_4 a path is discovered directly to the destination switch S_{11}. With only link monitoring applied, one can not determine if the link $S_{10} \rightarrow S_4$ is in failure or switch S_4 is in total failure. Therefore link failure protection schemes can be considered less robust against failures.

In this section we have shown graphical results for the link-based protection algorithm. If we compare the outcome with path-based protection and disjoint path pairs, link-based protection invokes more switches and links in the network. This indicates that more Flow Rules have to be configured to provide the necessary protection in the network. As a direct result,

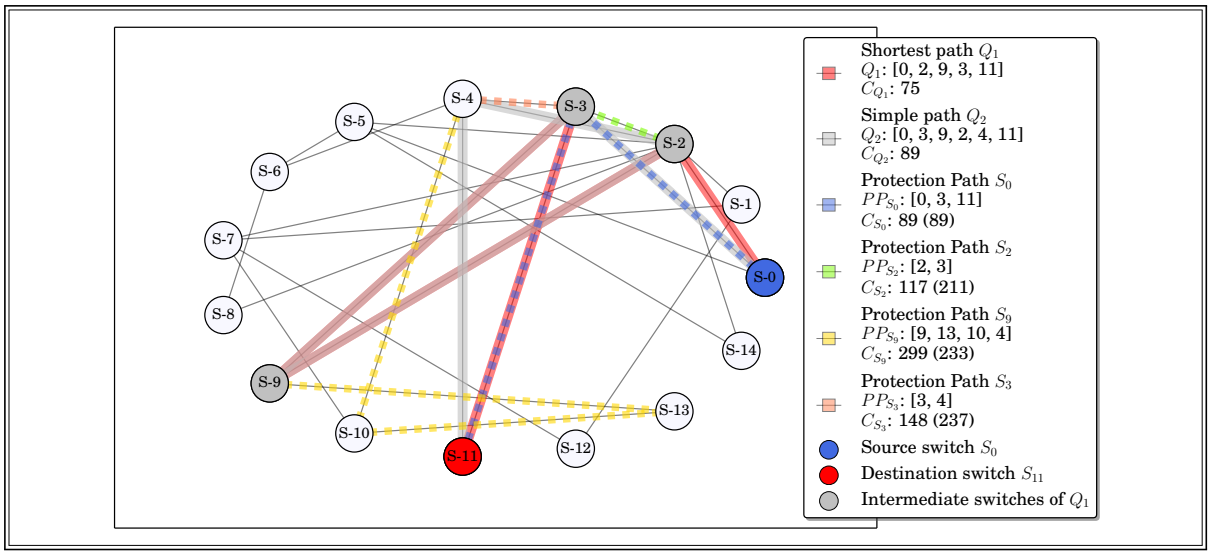


Figure 6-7: Protection paths on shortest path between S_0 and S_{11} with minimized crankback settings ($M = 2$) - *Backup path P_2 provides protection against failure of switch S_2. From S_2 a path is discovered to the remaining shortest path, where for switch S_9 and S_3 a path is discovered to the simple path Q_2. Due to crankback minimization PP_{S_9} is discovered with greater cost in comparison with crankback routing $C(PP_{S_9}) > C(CR_{S_9})$.*

Flow Tables will have more entries, which can have a negative influence on the switching performance. On the other hand, our algorithm enables the shortest path, which in normal operation, leads to optimal usage of the network. Example networks showed that the protection algorithm provides valid solutions to protect the shortest path with link-based protection paths. These benefits overrule the disadvantages of more Flow Rules and entries at the switch forwarding table. In the next section we will give quantitative measurements to support this conclusion.

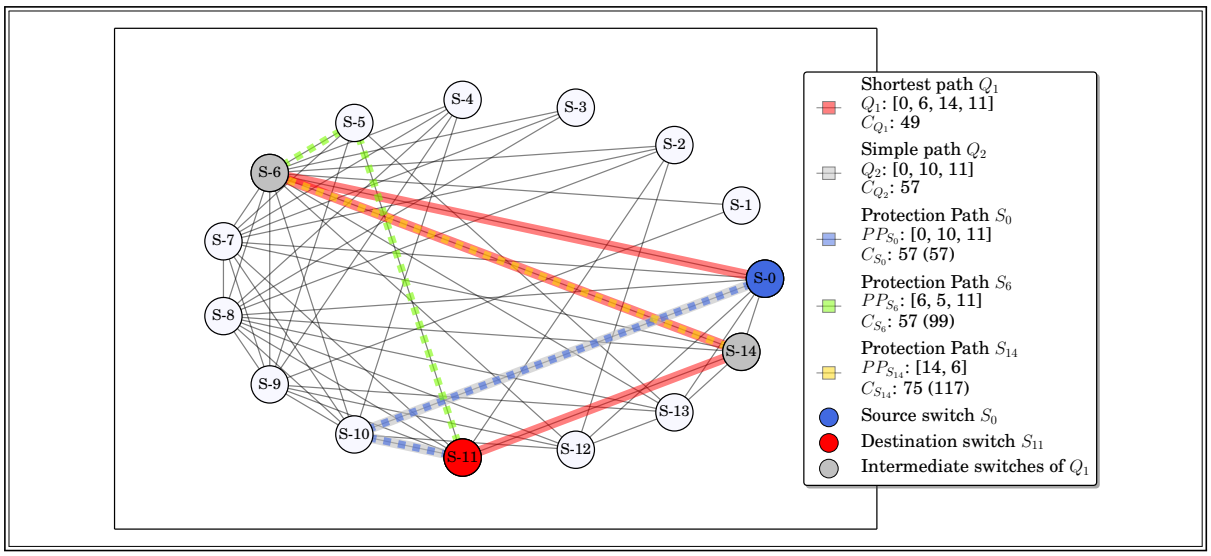


Figure 6-8: Protection paths on shortest path between S_0 and S_{11} with minimized cost settings and crankback routing ($M = 6$) - *At switch S_6 multiple protection paths exist to the simple path Q_2, for example path $S_{14} \rightarrow S_8 \rightarrow S_{10}$. These paths are not discovered as valid solutions, as the cost for these paths is higher than the cost for crankback routing. Therefore crankback routing is applied at S_{14}, from where the protection path of switch S_6 (PP_{S_6}) is chosen.*

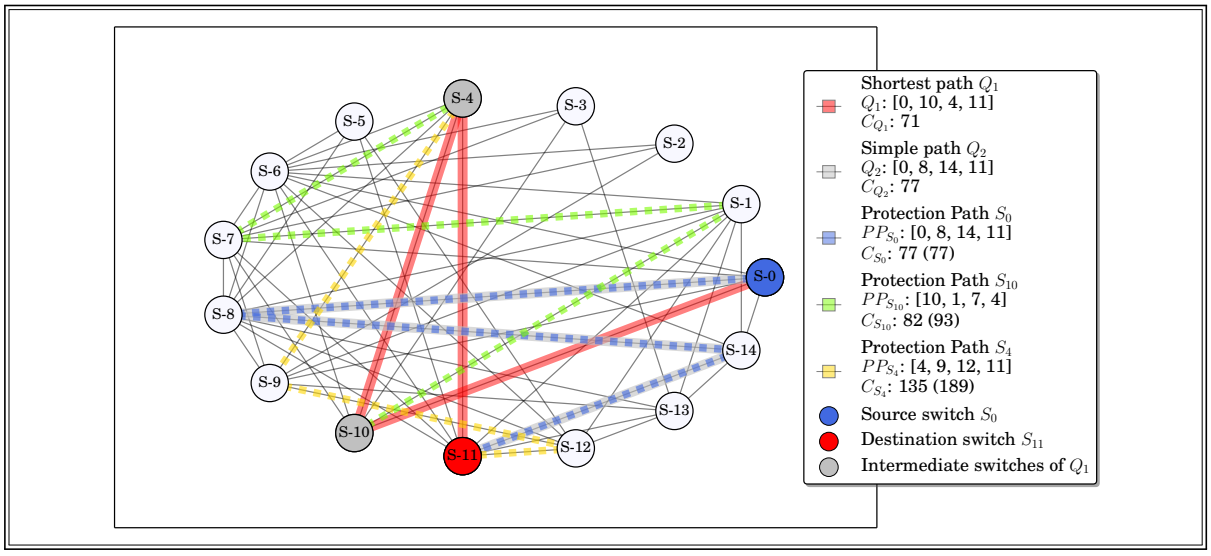


Figure 6-9: Protection paths on shortest path between S_0 and S_{11} with minimized cost settings and link failure protection ($M = 6$) - *At switch S_{10} a protection path is discovered for a link failure on $S_{10} \rightarrow S_4$. Where switch failure protection would discard S_4 as possible outcome, link failure protection assumes switch S_4 in functional state and the discovered path $S_{10} \rightarrow S_1 \rightarrow S_7 \rightarrow S_4$ is a valid protection path.*

6-4 Performance of link-based protection algorithm

Simulations for the link-based protection algorithm are performed on the described random and real-world networks with the parameters of section 6-1-3. During the simulations, all requested and discovered protection paths are stored for processing. A detailed breakdown of the simulation results is given in Appendix E. From the 100.000 simulations we performed for each network and setting, between 67.000 and 96.000 simulations delivered usable outputs for comparisons. The other simulations did not provide protection paths due to i) the absence of a shortest path between the source and destination switch, ii) the shortest path was a direct link between source and destination and iii) the simple path could not be discovered by Bhandari's algorithm in the topology (no guaranteed protection of the shortest path).

Figure 6-10 shows the results for the defined performance indicators on BA and ER random networks with $M = 2$. Results for cost gain α are related to the y-axis on the left, where the scale for crankback reduction ratio β is given on the right y-axis. On the x-axis a differentiation is made between link and switch failure protection, as well as the minimization settings, cost versus crankback.

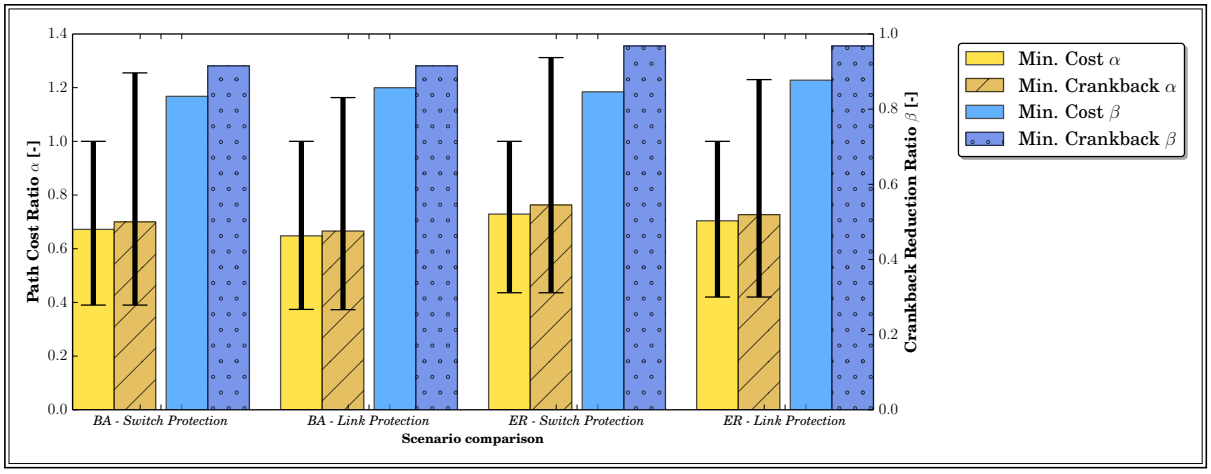


Figure 6-10: Simulation results on BA and ER random networks with $M = 2$ - *The path cost ratio α with 95% confidence interval is given on the left axis, where crankback reduction β is given on the right axis. An overall cost reduction of 25% and crankback reduction of 80% is achieved.*

In figure 6-10 the averages for α and β are computed, where also the 95% confidence interval for α is shown. We will first look into the cost parameter of the discovered protection paths. The average path cost ratio α (equation 6-1) is calculated over all solutions, including forced crankback. Overall we can state that the path ratio is at least 0.76, which indicates that cost for protection paths reduced with more than 24%, where highest cost reduction is found in BA-networks ($> 30\%$). This phenomenon can be explained with the use of difference the degree distributions between BA- and ER-networks. The probability that a shortest path will traverse over high degree switches in a BA-network is higher than over lower degree switches. High degree switches have more connections and therefore higher probability of an low cost link. During protection path discovery for BA networks, intermediate switches will have more direct neighbors and less probability of forced crankback routing in comparison to

ER networks. As expected, the cost minimized setting resulted in lower cost for protection paths in comparison with minimized crankback settings. The difference is only a few percent, which is explained by the fact that the protection path cost is not upper-bounded. The 95% confidence interval for the cost ratio supports this observation, as the upper boundary of the interval of the minimized crankback setting varies between 1.16 in BA-networks with link failure protection and 1.31 in ER-networks with switch failure protection.

If we look to the crankback reduction ratio β (equation 6-2), we can state a minimum of 85% reduction of forced crankback routing can be achieved using our link-based protection algorithm. The difference between BA- and ER-networks is very small with minimized cost settings, but a small difference is noticed between switch and link failure protection. Switch failure protection achieves lowest crankback reduction, which is expected. During computations a switch is assumed in failure, together with its attached links. The number of links in the network is therewith reduced, as well as the number of possible path solutions. In the case of minimized crankback settings, forced crankback routing is further reduced to at least 92%. ER-networks seem to benefit the most from the minimized crankback option.

For the second simulation the network connectivity is increased and random networks are generated with $M = 6$, where the results are presented in figure 6-11.

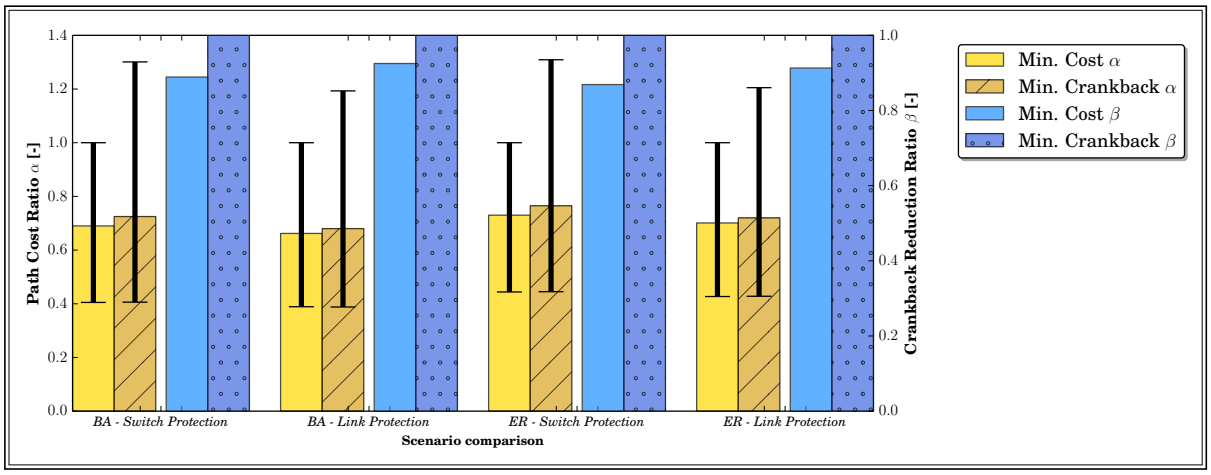


Figure 6-11: Simulation results on BA and ER random networks with $M = 6$ - *An overall cost saving of 25% and crankback reduction of 85% is achieved. With minimized crankback setting the need for crankback routing is removed.*

If we compare α between the scenario's $M = 2$ and $M = 6$, only a small advantage is noticeable in favor of the more connected network. The additional saving holds only a couple of percents, which indicates that to reduce the path cost ratio, no higher connectivity is required in the network. The percentage of crankback reduction only increased with a few percent in case of cost minimized settings. On minimized crankback reduction, the benefits of higher connectivity are very noticeable, as a crankback reduction of $\beta = 1$ is achieved. So, for higher connected networks, the minimized crankback setting leads to the absence of forced crankback routing as possible solution for link-based protection. The simulations are repeated with $M = 10$. Between $M = 6$ and $M = 10$ no noteworthy differences are noticed and therefore the simulation results are not shown. In figure 6-12 the simulation results are shown for the real-world networks.

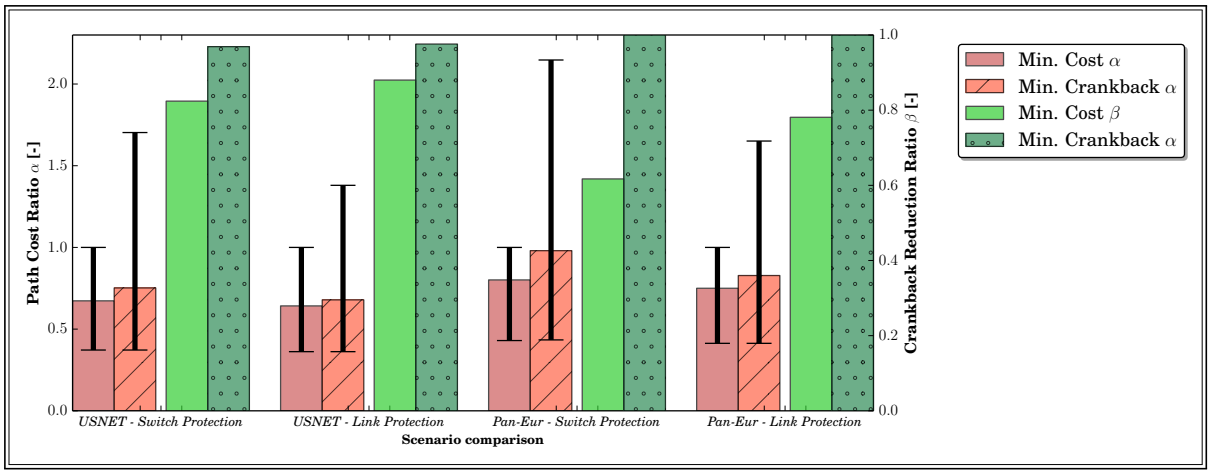


Figure 6-12: Simulation results on real-world USNET and Pan-EUR networks - *A minimum cost saving of* 25% *and at least crankback reduction of* 82% *is achieved on the USNET. On Pan-EUR networks less path cost and crankback reduction is achievable, as with minimized crankback settings applied, the path cost can exceed the crankback cost with a factor* 2.

The USNET achieved the best path cost ratio of all simulated networks, with a lower bound for the confidence interval of at least 37% of the path cost for crankback routing. An increase of the average path cost of respectively 8% and 4% is noticed between cost and crankback minimization on USNET, where on Pan-EUR networks an increase of 18% and 8% is noticed. Looking to the confidence intervals for path cost, we can state that with switch failure protection the path cost can exceed the cost for crankback routing on USNET networks with a factor 1.7 and even a factor 2.1 on Pan-Eur networks. Due to the high degree the switches in the Pan-EUR network, the need for forced crankback routing is completely removed.

6-5 Conclusion on protection algorithm simulations

The protection algorithm is implemented together with the Python NetworkX module and a large number of simulations have been performed to show intermediate results and performance of our protection algorithm. In total, more than a million networks are simulated, on which the protection algorithm with the extended Dijkstra algorithm is executed. A repetition of simulations always provided equal (and desired) outcomes. In the protection algorithm a large number of variables and parameters are present. Each parameter influences the outcome of the simulations, therefore the choice is made to limit the simulation parameters to link or switch failure protection, minimized cost or crankback settings and network connectivity. The protection algorithm provides link-based protection, as in theory would be expected, but also offers solutions for path-based protection as intermediate result. Modifications to the Dijkstra shortest path algorithm showed that additional parameters can be used for multi-constraint paths, where one parameter is leading. The main objective of the protection algorithm to reduce the need for forced crankback routing is achieved. In low connected networks ($M = 2$), application of crankback routing is reduced with at least 82%, while keeping average path cost well below the cost of crankback routing. If a choice has to be made which settings to choose for the protection algorithm, we advise switch failure protection with minimized crankback

routing. Switch failure protection offers better protection for a marginal increase in average path cost, while in well connected networks ($M = 6$ and Pan-EUR), the need for crankback routing is completely removed. In table 6-1 we summarized the results for switch failure protection with minimized crankback settings for all simulated networks, where $CI_{\alpha_{min}}$ and $CI_{\alpha_{max}}$ are the lower and upper bound for the 95% confidence interval.

Network	*Solutions*	α_{cost}	$CI_{\alpha_{min}}$	$CI_{\alpha_{max}}$	β
BA-2	96395	0.7	0.39	1.255	0.915
BA-6	95023	0.725	0.406	1.301	1
ER-2	78161	0.763	0.436	1.312	0.968
ER-6	94389	0.765	0.445	1.309	1
USNET	86834	0.752	0.372	1.703	0.969
Pan-EUR	67482	0.98	0.434	2.147	1

Table 6-1: Summarized results for simulated networks

From table 6-1 we can conclude that the deviation in the confidence interval for the real-world networks for the average path cost is greater in comparison with BA and ER random networks. For Pan-EUR networks, the average cost for link-based path protection almost equals the cost for crankback routing, but no forced crankback routing need to be applied. If our protection algorithm is applied to an existing network, we advise to perform simulations with different settings to find the most optimal setting for the required protection for that particular network.

While the defined performance indicators do not differ much with different network connectivity or network type, the classification and how the protection algorithm discovers protection paths, can show much variation. In Appendix E these variations are visible, where it is noteworthy that during the discovery process, the joining of the protection path with the simple and source switch protection path is preferred. This observation supports the choice to utilize intermediate products from Bhandari's algorithm to reduce the number of computations and improve the overall performance of the protection algorithm.

Chapter 7

Experimental results fast failover

In this chapter recovery experiments are performed on our proposal from section 4-5, with the main goal to determine recovery times on different testbeds and topologies configured with multiple settings to enable link-based path protection on SDN networks. Section 7-1 describes the experiments, where we give an insight on how the experiments are executed and which problems crossed our path during scenario testing and development of required tools. Results of baseline measurements are given in section 7-2, where section 7-3 gives measurement results of failover recovery with application of active link monitoring by BFD. Section 7-4 compares and analyzes measurements, where section 7-5 concludes this chapter.

7-1 Description of experiment

To determine the performance of the proposed solution in section 4-5, five steps must be accomplished.

1. *Topology with paths* - Multiple topologies must be defined to show failover behavior in regular and crankback conditions;

2. *Liveliness monitoring* - Failures must be monitored by OpenFlow switches to enable link-based path protection by triggering Fast Failover Group Tables;

3. *Generate traffic* - Traffic must be generated in such a way that precise failover measurements can be performed;

4. *Failure initiation* - Network failures must be initiated to the defined network topologies, such that shortest paths are disturbed;

5. *Measure failover times* - A measurement tool is needed to determine failover times, utilizing the generated traffic.

To execute the defined steps from above, two testbeds were available for experiments, being the TU Delft NAS software switch based testbed and the SURFnet physical switch testbed. The software switch testbed is built up from multiple general purpose servers enhanced with multiple network interfaces. Each server contains a 64-bit Quad-Core Intel Xeon CPU running at 3.0 GHz with 4 GB of memory and 6 independent Ethernet interfaces at 1 Gbps. Links in topologies are based on physical Ethernet connections between the servers. In this way, no additional delay is introduced during failover measurements, as no virtualization overlays are applied (VLAN) and links are not shared with other users and applications. To realize OpenFlow switch capabilities at the testbed, the Open vSwitch software implementation is configured on the Linux Ubuntu 13.10 operating system. The physical testbed consists of Pica8 P3920 switches on firmware release 2.0.4 interconnected via 10Gbps optical connections. OpenFlow capabilities on the Pica8 switches are enabled via a modified version of Open vSwitch.

7-1-1 Network topologies

For the experiments we defined three topologies, on which the primary and protection paths are configured with OpenFlow version 1.3. The first topology is a basic three switch network to measure failover times without applied crankback routing and protection paths. Figure 7-1 shows the basic topology in a functional state, where a traffic generator (TG) transmits data packets to the traffic capturer (TC). In the figure OpenFlow port x is indicated by $OF-x$, liveliness monitoring with LM, normal Flow Tables are annotated with N, Group Tables with GT and Fast Failover Group Tables with FF.

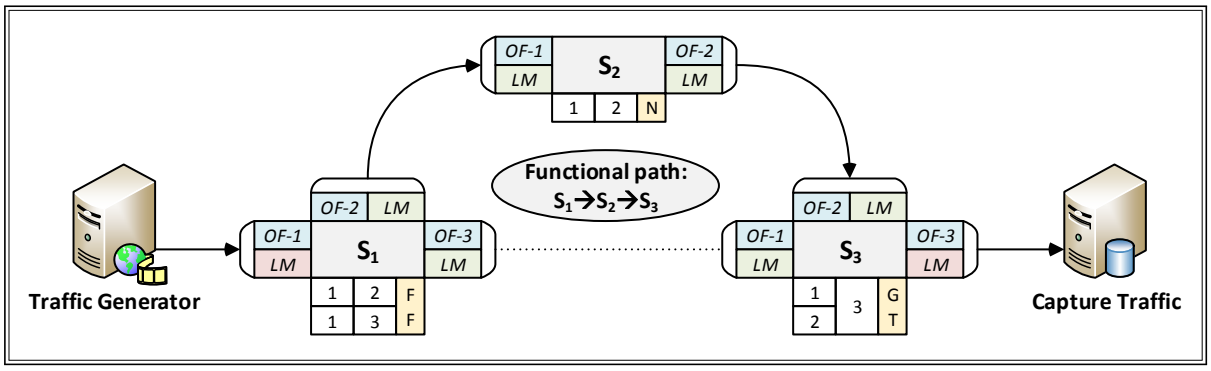


Figure 7-1: Basic topology in functional state - *The primary path between the generator and capturer is configured over path $S_1 \rightarrow S_2 \rightarrow S_3$, via the Flow Rules in (Group) Tables. There is chosen for a Group Table at switch S_3, as both the traffic from switch S_1 and S_2 must be forwarded to the traffic capturer.*

The Flow Tables show the forwarding rules from TG to TC. In functional operation, the shortest path is configured as $S_1 \rightarrow S_2 \rightarrow S_3$. Data packets arrive at switch S_1 on port $OF-1$. Switch S_1 is configured with a Fast Failover Group Table. The high priority Flow Rule of the Group Table forwards the packets to outgoing port $OF-2$, thus enabling the shortest path over $S_1 \rightarrow S_2 \rightarrow S_3$. After failure detection the packets must be forwarded using the low priority Flow Rule to outgoing port $OF-3$ of switch S_1. At switch S_2 the packets are forwarded to switch S_3, via a normal Flow Rule. Switch S_3 has a regular Group

Table configured, which forwards traffic from incoming ports $OF-1$ and $OF-2$ to outgoing port $OF-3$. The failure scenario holds a failure on link $S_1 \leftrightarrow S_2$, leading to the network state given in figure 7-2.

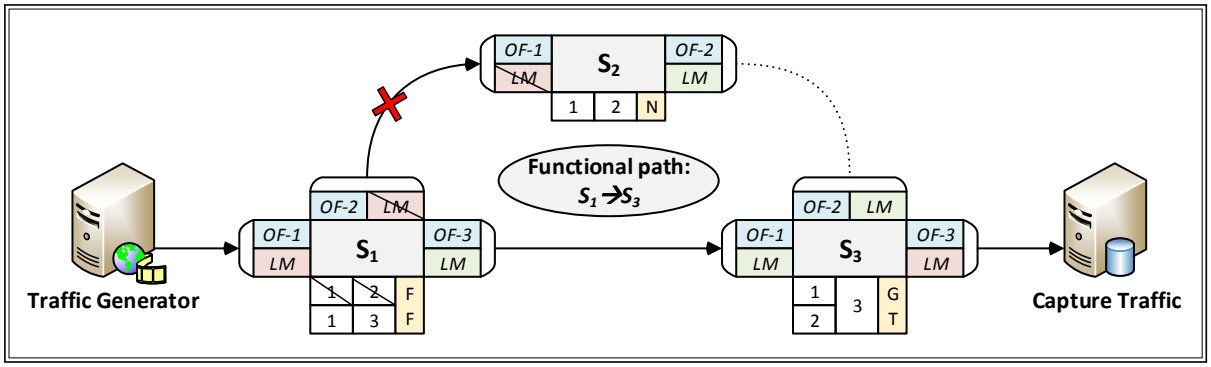


Figure 7-2: Basic topology in failure state - *A link failure is initiated between switch S_1 and S_2. The switches monitor the failure via liveliness monitoring, where after the Fast Failover Group Table disables the high priority Flow Rule. Traffic now redirected over path $S_1 \rightarrow S_3$.*

The failure is monitored by the liveliness monitoring mechanism on one of both testbeds. After detection, the Fast Failover Group Table on switch S_1 is triggered and the high priority Flow Rule is temporary disabled. Incoming packets are now forwarded to outgoing port $OF-3$ and the packet stream is recovered via path $S_1 \rightarrow S_3$.

With the basic topology from figure $7-1$ and 7-2 the behavior during crankback routing can not be monitored. Also the length for the shortest path is too limited to draw conclusions on applicability on larger real-world topologies. Therefore we defined a ring topology, as given in figure 7-3. Flow Rules at the Flow Tables needed for forced crankback routing are marked with C.

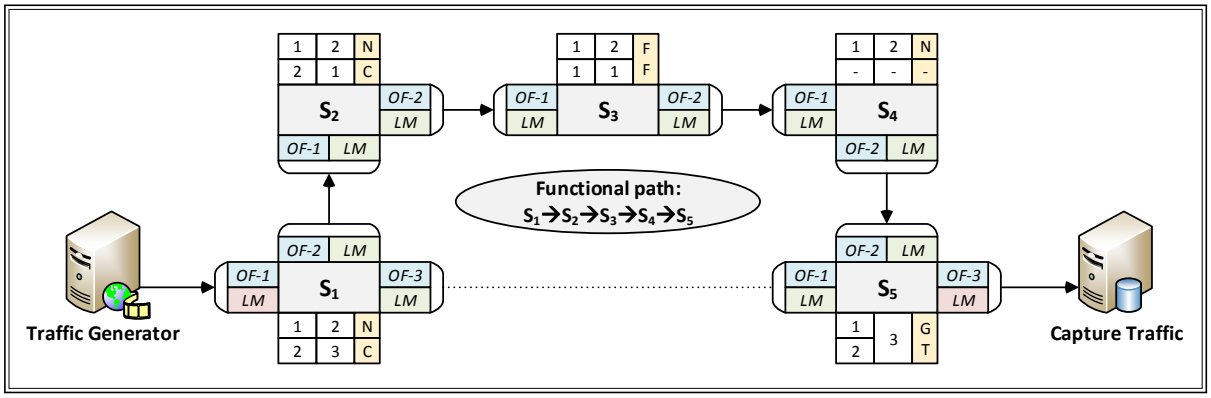


Figure 7-3: Ring topology in functional state - *The shortest path is configured as $S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow S_5$, where switch S_3 is configured with a Fast Failover Group Table for path failover functionality.*

To enable crankback routing, the incoming port must be set equal to the outgoing port. During experiments we discovered that Flow Rules to enable crankback routing required additional configuration, as Flow Rules containing the same in- and outgoing port are rejected by default. In the OpenFlow protocol a virtual port is defined, allowing crankback routing.

The OpenFlow implementation on the Pica8 switches did not support the virtual port setting on the Fast Failover Group Table. Therefore failover experiments on the hardware switch testbed were only possible with the basic topology configured. To measure recovery times in the ring topology and show the effect of crankback routing after failure initiation, a link failure is initiated between switch S_3 and S_4. The failure state of the ring topology is given in figure 7-4.

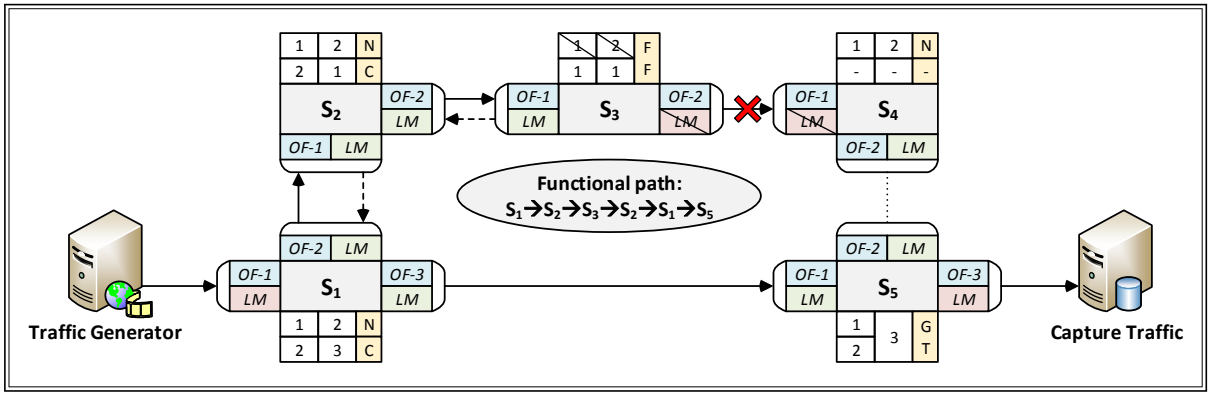


Figure 7-4: Ring topology in failure state - *A link failure is initiated between switch S_3 and S_4, which triggers the Fast Failover Group Table at switch S_3. Incoming packets at port $OF-1$ of switch S_3, must be cranked back via the virtual OpenFlow port to switch S_1, via switch S_2. At switch S_1, a crankback Flow Rule is configured to forward returning packets to switch S_5 via outgoing port $OF-3$.*

The initiated link failure leads to crankback routing, where traffic must transverse path $S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_2 \rightarrow S_1 \rightarrow S_5$ to reach the capture server. As a result of crankback routing, traffic utilization on links $S_1 \leftrightarrow S_2$ and $S_2 \leftrightarrow S_3$ has doubled. During simulations, we ensured that traffic load does not exceed 50% of the link capacity, preventing packet loss by *over* utilization.

To test whether link-based protection scales with network size, the failover experiments are also executed on the real-world topology USNET, described in section 6-1-2. On this topology, our routing scheme was also simulated and if the failover experiments succeed on this network, by providing protection on carrier and industrial network grade, we can conclude that our proposed solution scales with network size and is applicable to other real-world networks. Figure 7-5 shows the configuration of the USNET topology, which is implemented on the software switch testbed for failover experiments.

The paths over the defined topology are discovered with our protection algorithm, where link cost is chosen unity, and link failure protection and crankback reduction settings are applied. All paths are configured to the network, using OpenFlow configuration rules similar to the basic and regular topologies. The link cost unity made that no forced crankback routing was needed.

For all defined topologies we utilized the Open vSwitch command line interface to configure the switches and install Flow Rules on the switches. On the NAS testbed, the servers are configured with a software switch with corresponding interface bridge. Required Ethernet interfaces are added as OpenFlow ports to the switch via Open vSwitch tool *ovs-vsctl* [88] to the interface bridge. For the hardware testbed port configuration was provided by SURFnet.

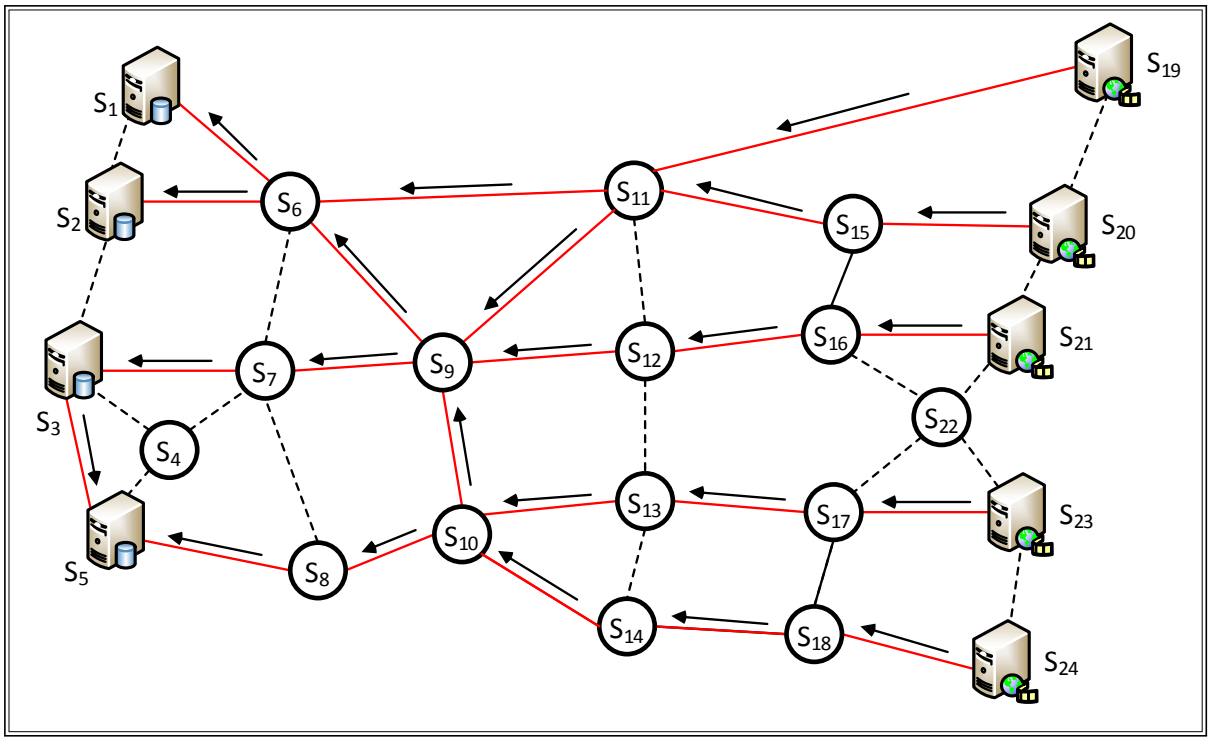


Figure 7-5: USNET topology for failover experiments - *All switches in the topology are configured as OpenFlow switch, where switches S_{19},S_{20},S_{21}, S_{23} and S_{24} are assigned as traffic generator and S_1, S_2, S_3 and S_5 as traffic capturer. Each traffic generator transmits traffic to all capturers, leading to a total of 20 shortest paths. The arrow mark the direction of traffic over the shortest paths. Dashed links are used for protection and backup paths.*

On all defined switches no OpenFlow controllers were assigned and installation of Flow Rules was done via the Open vSwitch tool *ovs-ofctl* [89] and the interface bridge. All switches were configured for OpenFlow protocol version 1.3 to enable Fast Failover Group Tables. For ease of configuration, testing and switching between scenarios, the required configurations and computed Flow Rules were stored in Linux script files.

7-1-2 Liveliness monitoring

The main goal of liveliness monitoring is trigger Action Buckets in the Fast Failover Group Tables to recover primary paths in the network. For liveliness monitoring we utilize the standard OpenFlow protocol and network modules available in Open vSwitch. Our approach during the experiments is to keep the OpenFlow protocol and Open vSwitch implementation standardized. Required modifications and improvements are made directly to standard modules and no extended modules are designed. From section 4-4 and figure 4-7 we learned that liveliness monitoring can be provided by passive Loss-of-Signal detection (OpenFlow protocol) and active link monitoring with the BFD protocol. Open vSwitch version 2.09 HEAD 5865a8a, installed on the software testbed, implemented BFD conform RFC5880 [42]. The Pica8 switches lacked active link monitoring capabilities, so liveliness monitoring for the Action Buckets depends on Loss-of-Signal detection. During initial testing the Loss-of-

Signal detection and failover functionality on both the software and hardware testbed worked flawlessly. Initial experiments on the software testbed with BFD monitoring showed two problems:

- Problem 1 - *BFD transmission interval* - The first problem was identified in the BFD implementation of Open vSwitch. BFD sessions between two OpenFlow enabled switches could be configured and link status was monitored as expected, but the transmission interval refused settings lower than 100 ms. To reach carrier and industrial grade recovery requirements, lower transmission intervals are required. Open vSwitch documented 100 ms as minimum transmit interval [23] and hard coded this value into the C/C++ source code of *bfd.c*. With some simple modifications to the source code, the minimum transmission and receive interval was set to 1 ms. Lower values are not possible due to the definition of intervals in the source code as C/C++ integers. Any decimal number is effectively rounded to its lower integer by this definition. Because the transmit interval is integrated in the control message, converting the value to a float, is not possible without major modifications to the Open vSwitch source code;
- Problem 2 - *Open vSwitch missing status link* - The second problem came up when link failures were initialized. BFD detects the failure within millisecond order and sets the port status to down. The Action Buckets of the Fast Failover Group Tables are not triggered until the failure was detected by the operating system and Ethernet device via Loss-of-Signal detection. It seemed that the monitored BFD status does not influence the Action Bucket liveliness. A modification was made to the source code of the module responsible for the Fast Failover Group Table (*ofproto-dpif-xlate.c*), so that the BFD liveliness is also checked as part of the interface and Action Bucket liveliness.

After our solutions where implemented to the source code of Open vSwitch, the network modules where re-compiled and installed to the software testbed. The primary paths on the topologies could now be protected via Fast Failover Group Tables and (active) liveliness monitoring. To show that active link monitoring is required on the defined networks, we perform baseline failover measurements on both testbeds in the simple configuration. As we pursue the requirements for carrier and industrial grade networks, three BFD detection windows are defined for the experiments. The first transmission interval for the BFD protocol is set to $T_{i,set} = 15$ ms, which leads to a predicted and theoretical failure detection time of 45 ms (Section 4-3) and wherewith a total recovery time of 50 ms (carrier grade) is achievable . To further reduce the recovery time, the experiment is also performed with $T_{i,set} = 5$ ms. Both previous defined experiments were successful. Using equation 4-8 from section 4-3 and the actual round-trip-times (ping) between the switches, we determined that the minimal configurable value for $T_{i,set}$ of 1 ms will not produce false negatives, so the BFD transmission interval is also set to it's minimum value ($T_{i,set} = 1$ ms) to show the maximum failover performance capable on our software switch testbed. In figure 7-6 we show a Wireshark capture from switch S_2 were monitor the BFD status between switch S_2 (10.0.0.2) and S_3 (10.0.0.3) on the simple topology with $T_{i,set} = 1$ ms.

From figure 7-6 it is visible that our modification to reduce the transmission interval is successfully implemented. Both switch S_2 and S_3 actively monitor the link, where BFD control messages are transmitted by both switches with a delay of approximately 1 ms. Both monitor the link up, as control messages are received from one another. The implementation of the

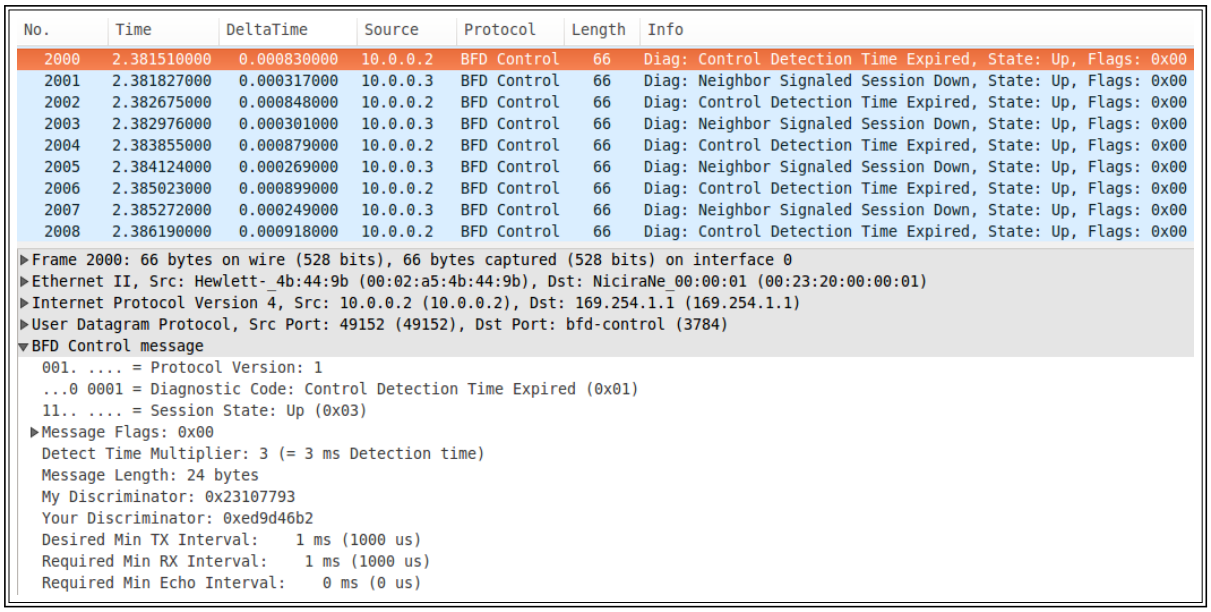


Figure 7-6: Capture of BFD status monitoring in Wireshark - *In the capture the flow of control messages in time are visible. The transmit and receive interval for BFD are set to* 1 *ms and the detection multiplier is set to* $M = 3$.

BFD protocol in Open vSwitch functions as expected from the theory described in section 4-3. Both switches transmit additional diagnostic information with the control messages. The diagnostic messages show the last reason for change in session state. For switch S_2 the detection windows expired, where for switch S_3 the status was changed with reception of a session down control message of S_2.

7-1-3 Traffic generation

In order to simulate network traffic over the testbeds and perform measurements to determine the recovery time, we use the *pktgen* [90] packet generator available in the Linux Ubuntu operating system. The packet generator creates packets with fixed delay, pre-determined size and sequence numbers, resulting in an UDP data stream. *Pktgen* operates in kernel space and is assigned to one of the available processing threads in the system. In the following list the most important settings are described with emphasis on the experiments and different topologies.

- *Interface Name* - On the traffic generators of the defined topologies *pktgen* is enabled in kernel space and a transmitting interface is coupled to a processor thread. For the simple and ring topology, the Ethernet interface ($EthX$) connected to the network is coupled with *pktgen*. For the USnet topology (figure 7-5), switches S_{19}, S_{20}, S_{21}, S_{23} and S_{24} are assigned with two duties, being an OpenFlow switching device and traffic generator. Directly coupling *pktgen* to an Ethernet interface would disable failover capabilities as the control logic of the switch is bypassed and generated packets are directly transmitted over the outgoing port. Therefore we coupled the bridging interface

to *pktgen*, wherewith generated packets are transmitted into the OpenFlow switch and the control logic provides forwarding conform configuration and installed Flow Rules;

- *Packet Size* - For the experiment packet size is not an important setting, but we made sure that link utilization not exceeded 50% during experiments on the ring topology. We configured the standard packet size of *pktgen*. With Wireshark we determined that the actual frame size of a single packet is equal to 58 bytes;

- *Source address* - The MAC and IP address from where the traffic is generated. For analysis and packet tracing purposes we set the source address equal to IP address of the traffic generating server. For the experiments we configured the testbed with an 10.0.0.0 subnet. The *pktgen* module automatically assigns the MAC-address of the coupled interface as source MAC address;

- *Destination IP address* - We used IP addresses as matching criteria in the installed Flow Tables. In *pktgen* a *min* and *max* IP address can be configured, allowing traffic generation to multiple receivers. On the simple and ring topology, the traffic generator transmits data packets to a single destination, so we configured the *min* and *max* address identical. For the USnet topology the assignment of multiple addresses was very useful, as a single traffic generator could produce traffic for multiple receivers. The *min* and *max* addresses were respectively set to 10.0.0.1 (S_1) and 10.0.0.5 (S_5). As switch S_4 is no traffic capture server, generated traffic is dropped by Flow Rules at the traffic generators;

- *Destination MAC address* - The MAC address was not used for packet forwarding, but are important for our failover measurements. Our measurement tool requires a MAC address to successfully capture traffic. For the simple and ring topology the destination was a single traffic capturer, so the MAC address could be configured. For the USnet topology multiple receivers were assigned, where *pktgen* has no capability of allowing multiple destination MAC addresses. To solve this problem we utilize a Flow Rule to change the destination MAC address in the packet header. With the IP address in the packet header the destination is known. At the last forwarding switch in the path towards the destination, with the OpenFlow option *mod_dl_dst,* the packet header is changed to the actual MAC address of the traffic capture server;

- *Count* - Defines the number of packets to transmit. During experiments we ensured that the number of packets to transmit a big enough for the length of the experiment;

- *Delay* - In *pktgen* the delay between packet transmissions can be configured in nanoseconds. Because we measure failover times in order of millisecond, a measurement accuracy of 0.1 ms is required. To account for rounding errors, the measurement tool must receive packets with a delay of 0.05 ms (T_{delay}). For the simple and ring topology *pktgen* is configured with this value. The USnet topology needed a different configuration, as multiple destinations are configured. It seemed that *pktgen* transmits packets with the configured delay sequentially to the destinations. In order to reach the requirement to receive packets at all receivers with a delay of 0.05 ms, *pktgen* must transmit packets with a delay of 0.01 ms. To ensure that our measurements are valid, we determined the timing accuracy of transmitted packets with Wireshark at 0.005 ms. This value is sufficient for the requested transmit delay of 0.01 ms;

- *Sequence Number* - Although the sequence number is not a setting which is adjustable, it is the parameter of *pktgen* packet which we require to determine failover times. Every time a packet is generated, an incremental and unique sequence number is assigned to that packet.

With the information from the list we can calculate the network load generated by *pktgen*. For the simple and ring topology with $T_{delay} = 0.05$ ms in total 20.000 packets per second are transmitted. With 58 bytes (464 bits) per packet, the network load equals 8.9 Mbit per second. Herewith we showed that on the ring topology the network the traffic does not exceed the 50% capacity (500 Mbps) and that over-utilization is not in case during crankback routing. Overall we can state that *pktgen* is a powerful tool, but we experienced that the timing accuracy dropped when transmission rate is increased and that managing *pktgen* is not always an easy task.

7-1-4 Failure initiation

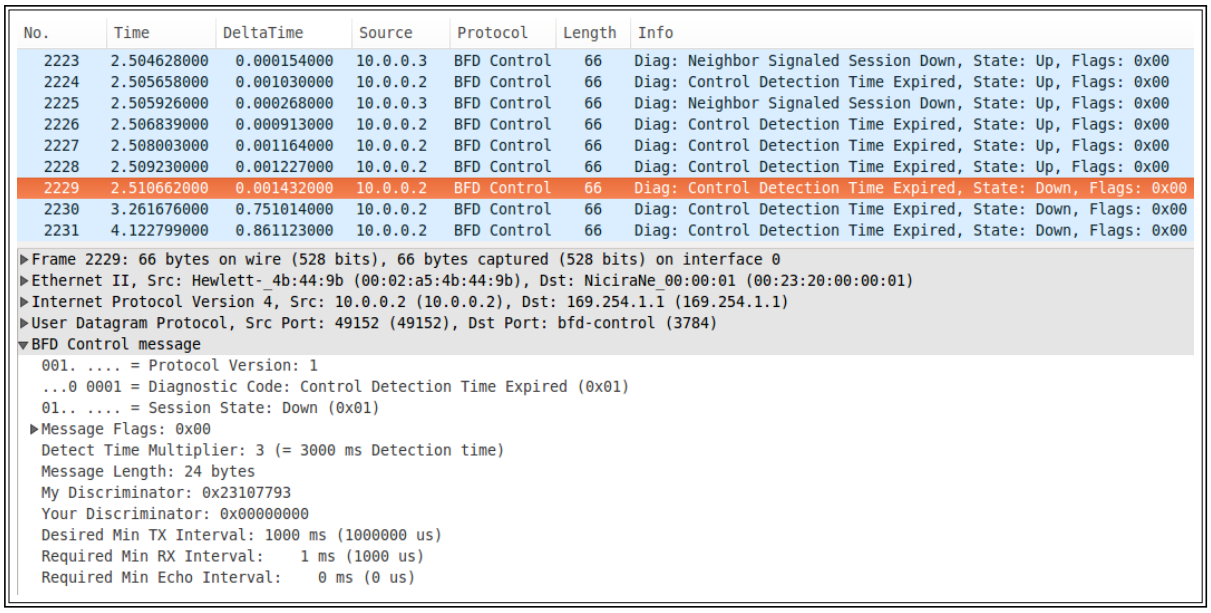
To initiate a failure between the traffic generator and capture server, a link failure on the primary path must be initiated. During the experiment, the *pktgen* UDP stream transverses over the network in one direction (from traffic generator to capture server). By bringing the interface administratively down via the Linux command *ifdown* on one of the upstream switches, the downstream switch must detect the failure using liveliness monitoring. To ensure the failure initiation method is valid, we performed a small experiment on the simple topology on the software switch testbed to show that no triggers were sent from the upstream switch via the BFD protocol to the downstream switch. Between switch S_2 and S_3 a BFD link monitoring session is configured with $T_{i,set} = 1$ ms. After the session is stable and monitored up, the interface attached to port OF-2 (figure 7-1) at switch S_3 is brought administratively down. Figure 7-7 shows the Wireshark capture taken on switch S_2 were the failure is monitored by BFD.

Between packet number 2225 and 2226 (figure 7-7) a failure is initiated via the *ifdown* command on switch S_3. Switch S_2 does not receive control messages from S_3 and transmits three control messages, being packets 2226, 2227 and 2228. After the detection multiplier expires, the monitored status changed from UP to DOWN. From the time column in figure 7-7 it is visible that the failure was monitored down after approximately 3 ms and furthermore we monitored no trigger down from switch S_3. Herewith we can conclude that active liveliness monitoring with BFD and the failure initiation method via *ifdown* is valid.

7-1-5 Failover measurements

To determine the recovery time, sequence numbers and the transmit delay from *pktgen* are utilized. Missing sequence numbers of captured packets determine the start and recovery time of the failure by equation 7-1, where $T_{recovery}$ is the recovery time, S_{Cap} is the last captured sequence number, S_{Store} is the second last sequence number stored for comparison and T_{delay} is the configured packet delay between *pktgen* packets.

$$T_{recovery} = (S_{Cap} - S_{Store} - 1) \cdot T_{delay} \tag{7-1}$$



No.	Time	DeltaTime	Source	Protocol	Length	Info
2223	2.504628000	0.000154000	10.0.0.3	BFD Control	66	Diag: Neighbor Signaled Session Down, State: Up, Flags: 0x00
2224	2.505658000	0.001030000	10.0.0.2	BFD Control	66	Diag: Control Detection Time Expired, State: Up, Flags: 0x00
2225	2.505926000	0.000268000	10.0.0.3	BFD Control	66	Diag: Neighbor Signaled Session Down, State: Up, Flags: 0x00
2226	2.506839000	0.000913000	10.0.0.2	BFD Control	66	Diag: Control Detection Time Expired, State: Up, Flags: 0x00
2227	2.508003000	0.001164000	10.0.0.2	BFD Control	66	Diag: Control Detection Time Expired, State: Up, Flags: 0x00
2228	2.509230000	0.001227000	10.0.0.2	BFD Control	66	Diag: Control Detection Time Expired, State: Up, Flags: 0x00
2229	2.510662000	0.001432000	10.0.0.2	BFD Control	66	Diag: Control Detection Time Expired, State: Down, Flags: 0x00
2230	3.261676000	0.751014000	10.0.0.2	BFD Control	66	Diag: Control Detection Time Expired, State: Down, Flags: 0x00
2231	4.122799000	0.861123000	10.0.0.2	BFD Control	66	Diag: Control Detection Time Expired, State: Down, Flags: 0x00

Figure 7-7: Capture of BFD monitoring link down in Wireshark - *The failure is monitored after the timing window expired at packet number 2229. After the link is monitored down the transmit interval is increased to a default value of* 1000 *ms.*

In equation 7-1 the subtraction of 1 is inserted to correct the recovery time window, as the window is constructed by $n+1$ received packets. The network in functional state will not drop packets and the recovery time equals zero. When a failure is initiated, the difference between S_{Cap} and S_{Store} equals the number of packets lost and the recovery time can be calculated. To capture the data stream, we developed a real-time measurement tool. The Wireshark capture application introduced too much latency in the capture and required results could not be produced without additional programming. Equation 7-1 is implemented in our real-time measurement tool. The tool is programmed with Python, where a UDP socket is defined to capture *pktgen* packets with the desired IP address and UDP port number. In the early stages we discovered that the destination MAC address is required for the socket to capture packets on the incoming interface. The problem was solved as described in section 7-1-3. On arrival of a UDP packet, the UDP and Ethernet encapsulation must be removed to retrieve the *pktgen* sequence number. In Wireshark standardized display filters exist to retrieve bits from received packets and show protocol information stored in the packet, as shown in figures 7-6 and 7-7. The protocol display filters do not exist in standard Python programming language, so retrieving the sequence number was a challenging task. With information from the Wireshark display filter for *pktgen* [91] and the Python module *unpack*, we managed to retrieve the desired sequence numbers.

On the simple and ring topology only a single destination IP address is assigned, so the sequence numbers at the packets to the capture server are on sequential order. Via equation 7-1 $T_{recovery}$ is calculated for each received packet. When $T_{recovery} > 0$ the observation is stored to a log file for further analysis. Validation on the USnet topology showed that equation 7-1 needed modifications. Received packets at the traffic capture servers contained gaps of four missing sequence numbers. This is was expected, as a single *pktgen* instance generates packets to five destinations, where each generated packet contains an unique sequence number. So

the factor $S_{Cap} - S_{Store}$ of equation 7-1 equals to 5. To correct this artifact, the recovery time for the USnet topology is modified to equation 7-2.

$$T_{recovery} = (\frac{(S_{Cap} - S_{Store})}{5} - 1) \cdot T_{delay} \tag{7-2}$$

7-1-6 Scenario control

All five required steps are accomplished, so failover measurements could be performed. To execute failover measurements multiple times, a scenario server is added to the topologies. Via Secure Shell (SSH) [92] connections the following tasks are executed.

1. *Configure switches* - Assigned OpenFlow switches are configured with required forwarding tables and liveliness monitoring via locally stored Linux scripts;
2. *Start traffic generation* - The *pktgen* module is configured and started with the required configuration via locally stored Linux scripts;
3. *Start measurement tools* - The Python real-time capture tool on the Traffic Capture server(s) is initiated;
4. *Initiate failures* - Failures are initiated by the scenario server via *ifdown*. For the simple and ring topology a single link failure is initiated on a predefined link (figures 7-2 and 7-4). For the USnet topology randomly a link is chosen from a list of links of all configured primary paths and brought administratively down. Failures are initiated when the configured network is in stable condition and the liveliness mechanism is monitoring the link up. Per topology we defined a number of repetitions;
5. *Restore network* - The initiated link failure is restored by bringing the corresponding interface up via the Linux command *ifup* after the network is in stable condition with activated backup and protection paths. Primary paths are restored via the priority selection process of the Fast Failover Group Tables;
6. *Store log files* - Traffic generation and measurement tools are stopped after the number of defined failover repetitions is executed. Monitored failover times are stored and retrieved from the Traffic Capture servers for further analysis.

We performed multiple test runs to determine the optimal time window and steady period for failure initiation and network restoration. The time window is chosen such that the packet stream between the servers is steady for at least 10 seconds. Smaller steady periods proved unstable with Loss-of-Signal liveliness detection. We also performed tests with longer steady periods (30 and 60 seconds), but no differences in measurements where noticed.

7-2 Experiment - Baseline measurements

To show that active BFD link monitoring is required for path recovery at carrier grade level, we performed five baseline measurements on the software and hardware testbeds for the simple topology. An overview of the baseline measurements on the simple topology is given in table 7-1.

	Testbed	*Liveliness*	*Repetitions*	*Goal*
A.	TU Delft NAS	None	10	Determine delay Fast Failover Group Table
B.	TU Delft NAS	LoS	100	Determine baseline recovery performance
B.	SURFnet	LoS	100	Determine baseline recovery performance
C.	TU Delft NAS	None	100	Determine failover delay $ifdown$
C.	SURFnet	None	100	Determine failover delay $ifdown$

Table 7-1: Performed scenarios on TU Delft NAS and SURFnet testbeds

The measurement for scenario A is to determine the delay of the switch failover time of the Fast Failover Group Table. Therefore we initiated a removal of the high priority Flow Rule via the command-line interface on the testbeds. Multiple runs (> 10) showed that the change in the Fast Failover Group Table resulted in no loss of *pktgen* packets. This indicated that a trigger to the Group Table introduces no additional processing delay. In scenario B baseline measurements are performed with Loss-of-Signal detection by the physical layer of the Ethernet devices to determine the baseline performance of the testbeds. Failures are initiated as explained and repeated 100 times for analysis. Before failure initialization, the network is stable and the packet stream is in steady state. In figure 7-8 the baseline measurements for the TU Delft NAS testbed is given, where $\bar{x}$ gives the average measured recovery time.

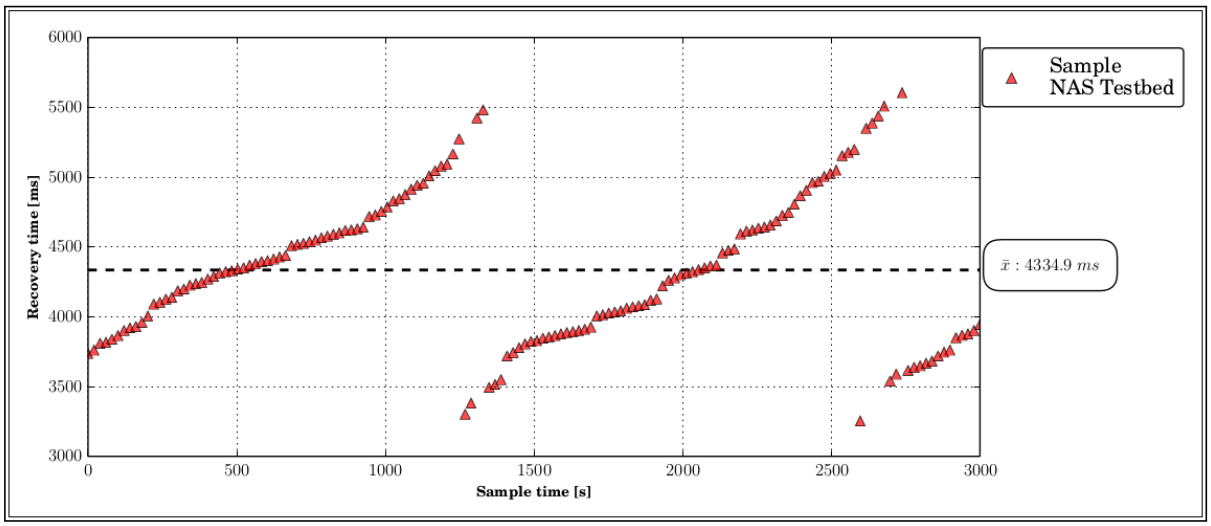


Figure 7-8: Baseline measurements at Open vSwitch testbed - *The recovery time is measured* 100 *times in a period of* 3000 *seconds, where measured recovery time is in the order of seconds, with an average of* 4334 *ms. In the samples a pattern is noticed, whose origin can not be explained.*

Failure detection by the physical layer via Loss-of-Signal detection and the propagation of the failure through to the operating system and Open vSwitch results in a recovery time in the order of seconds. An average recovery time of 4335 ms was determined after analysis. In the captured results some minor packet loss was noticed, where a small number (< 5) of *pktgen* packets was not received by the capture server. We performed extra simulations with higher packet rates to ensure the correct functioning of the capture script. The script worked flawlessly on higher packet rates, so the packet loss was probably introduced by processing at the operating system or the Open vSwitch module. Another point we noticed was the switching behavior of the OpenFlow Fast Failover Group Table on restoring the network topology to its original state with the *ifup* command. The Fast Failover mechanism in the Group Table automatically selects the outgoing port with the highest priority. On restoration to the primary path, the high priority Flow Rule is selected as primary and we would expect a functional state of the primary path after the switch over. In the measurements, however, we noticed that the switch-over causes packet loss up to 2 seconds. The last observation point is the recovery time pattern in figure 7-8. It seems that a (scheduler) process in the Ubuntu operating system, Open vSwitch or in one of the network layers influences the recovery time. Increasing the steady period between failure initializations and topology restoration does not influence the pattern. Operating manuals of Ubuntu and Open vSwitch do not provide a closing argument for this phenomenon. Identical measurements are performed on the SURFnet hardware testbed, where figure 7-9 shows measured recovery times.

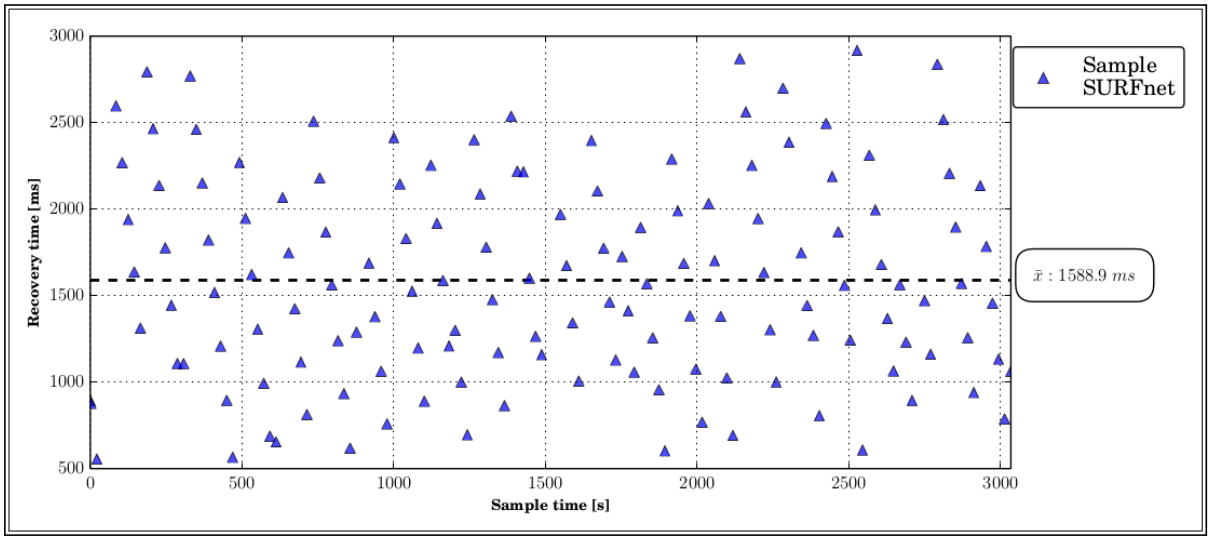


Figure 7-9: Baseline measurements at SurfNet testbed - *The overall recovery time is faster in comparison with the Open vSwitch testbed. An average recovery time of* 1589 *ms is determined, with a maximum of approximately* 2900 *ms. No pattern is visible in the recovery samples.*

The SURFnet testbed provides faster recovery times in comparison with the Open vSwitch testbed. An average of 1589 ms is required to recover the path, which is substantially faster than recovery with Open vSwitch. During packet capturing no packet loss is noticed, but the phenomenon of packet loss during restoration to the primary path remains. We suspect that the physical layer introduces this behavior, by reporting the link status *up* to upper-layers, while packet transfer over the link is not yet enabled. The fact remains that the recovery time is still in order of seconds, instead of the requested millisecond order. Results in figure 7-8 and

7-9 show that the Loss-of-Signal link monitoring is not suitable for fast recovery purposes.

For scenario C, we measured the difference between failure detection and actual recovery. On both testbeds we brought down the interface performing the active failure detection (Switch S_1 of figure 7-1). This measurement shows possible recovery time without our modifications to link the BFD status to the Fast Failover Group Table. The failover is initiated by the *ifdown* command on both testbeds, rather than the physical layer or BFD status. Figure 7-10 show the recovery times using the *ifdown* command as failover trigger for scenario C.

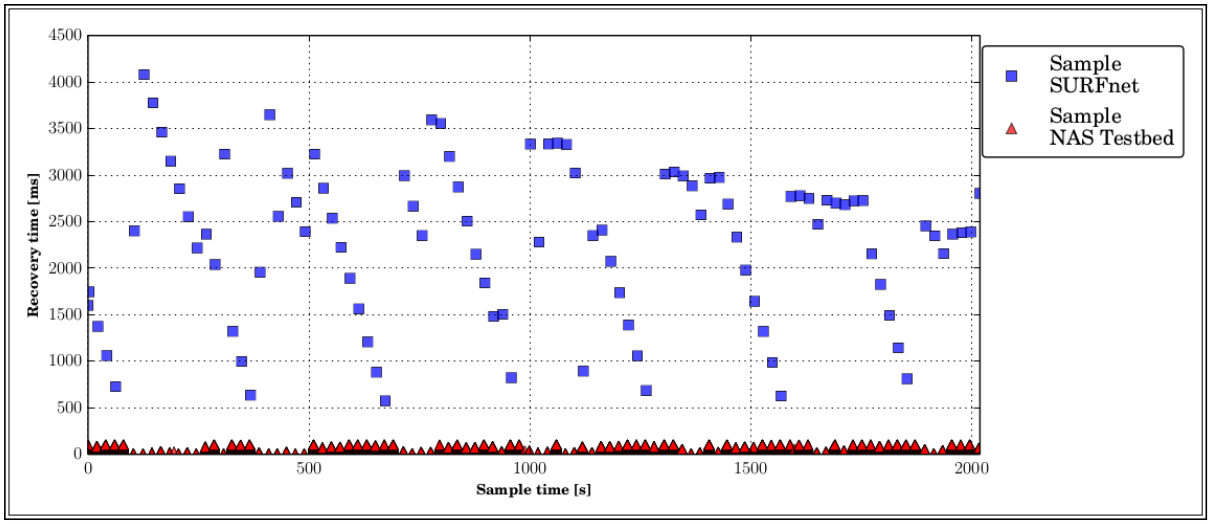


Figure 7-10: Baseline measurements by administratively bringing outgoing interface down - *The software testbed requires on average* 50.83 *ms to recover the path after failure detection, while the Pica8 switches need* 2274 *ms to recover the path after administratively bringing down the interface.*

The average recovery time for the SURFnet testbed is determined at 2274 ms after bringing fast failover interface administratively down, which is actually slower than by Loss-of-Signal detection. We suspect that the Pica8 switches disable the interface and afterward update the administrative status of the Fast Failover Group table. On the Open vSwitch testbed the average recovery time dropped to 50.8 ms, which is in the requested millisecond orders, but remains to slow to recover paths in carrier-grade networks. Herewith we have shown that administratively bringing down the interface after failure detection is not a possible solution for fast recovery and our proposed solution to trigger the Fast Failover Group Table with BFD status remains valid. In the next section we will show the recovery times with active BFD monitoring enabled.

7-3 Experiments - Active link monitoring

Performance measurements for active link monitoring are only performed on the Open vSwitch testbed, as the Pica8 switches from the SURFnet testbed does not support BFD monitoring. In appendix A the failover requirements for carrier grade and industrial networks are given. For carrier-grade networks a recovery time of 50 ms is required, where industrial networks demand even fast recovery of 20 ms and below. To reach these requirements, we configured

the BFD transmission intervals, as discussed in section 7-1-2, as $T_{i,Set} \in (1, 5, 15)$ ms. In table 7-2 an overview is given of the performed scenarios with active link monitoring on the TU Delft NAS testbed.

	Topology	*Liveliness*	*Repetitions*	*Goal*
D.	Simple	BFD	100	Determine recovery performance simple topology
E.	Ring	BFD	100	Determine recovery performance ring topology
F.	USnet	BFD	3400	Determine recovery performance real-life network

Table 7-2: Performed scenarios to determine recovery performance on TU Delft NAS with active link monitoring

To fully test our proposal, recovery measurements are performed at the three defined topologies. Figure 7-11 shows the recovery times for scenario D with BFD enabled and configured to the discussed and defined transmission interval $T_{i,Set}$.

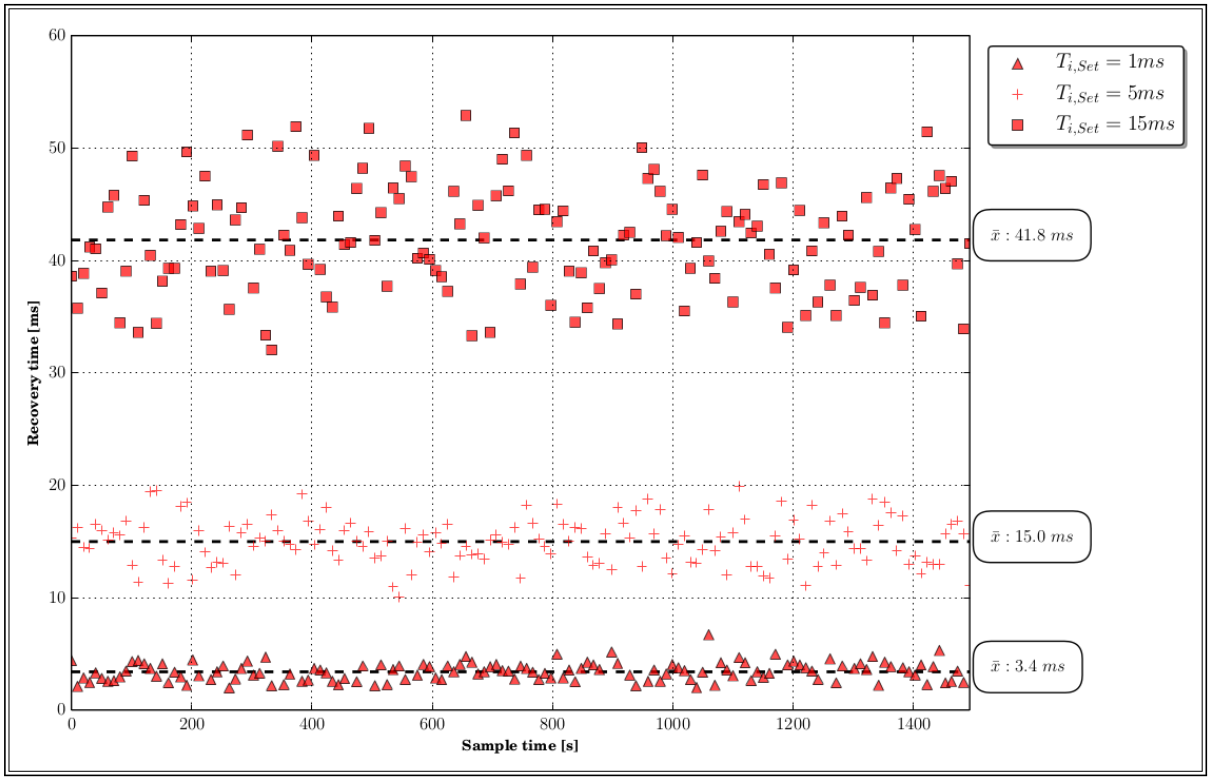


Figure 7-11: Recovery measurements on simple topology at TU Delft NAS testbed - *The average recovery time for* $T_{i,Set} \in (1, 5, 15)$ *ms is respectively determined at* 3.4, 15.0 *and* 41.8 *ms.*

The results are close to theoretical values with $M = 3$. For $T_{i,Set} = 15$ ms the average recovery time ($\bar{x} = 41.8$ ms) is lower than expected. This phenomenon can be explained by the introduced time jitter (equation 4-2), which lowers the transmission interval with a maximum of 25%. For $T_{i,Set} = 5$ ms the average recovery time equals the theoretical recovery time of 15 ms, while the average recovery time of $T_{i,Set} = 1$ ms is longer as expected. We suspect that the large number of packets generated by *pktgen*, the transmission and processing

delay, and the trigger to the Fast Failover Group Table, introduce minor delays. Overall we can state that active link monitoring by BFD reduced recovery times to millisecond order. Due to limitations in the BFD module of Open vSwitch, we could not reduce the transmission window below 1 ms. Enabling BFD monitoring had another positive effect on the simulations, as during the restoration process to the primary path no packet loss was monitored. With BFD link monitoring enabled, switching back to the primary path is delayed until BFD confirms link status and no packets are dropped, which is an improvement over the initial observations with the baseline measurements.

The measurements were also performed on the ring topology (scenario E). Figure 7-12 shows the measured recovery times for the selected transmission intervals.

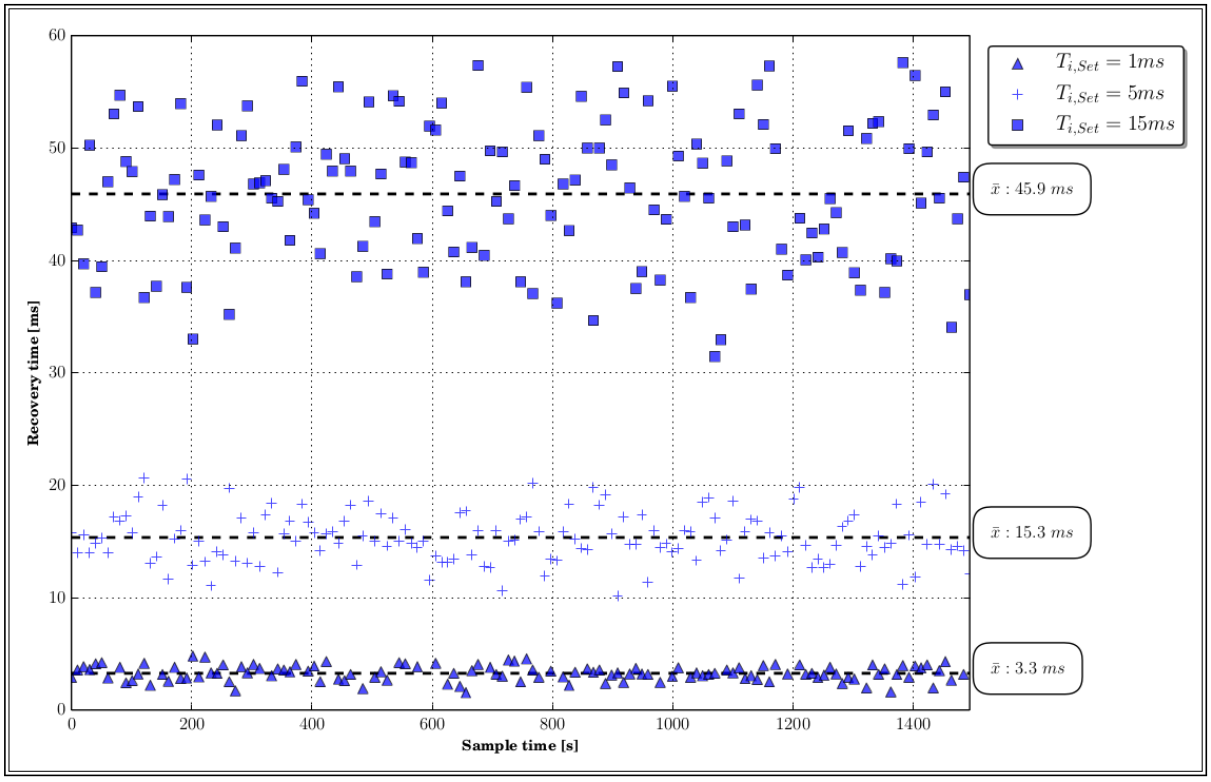


Figure 7-12: Recovery measurements on ring topology at TU Delft NAS testbed - *The average recovery time for* $T_{i,Set} \in (1, 5, 15)$ *ms is respectively determined at* 3.3, 15.4 *and* 45.9 *ms.*

The measured recovery times for $T_{i,Set} = 1$ ms and $T_{i,Set} = 5$ ms do not differ much from scenario D, despite longer primary and backup paths, and the need for crankback routing. For $T_{i,Set} = 15$ ms the average recovery increased with 4.1 ms to 45.9 ms. We cannot find valid reasoning why the recovery time increased, as the network showed similar recovery times with lower transmission intervals, which on itself require more processing at the switches.

The experiments on the real-world topology showed some problems with the settings and parameters defined. To keep the same delay between arriving packets at the destination, the transmit delay for *pktgen* must be reduced with a factor 5, resulting in 100.000 packet transmissions per second. In the case of switch S_{19}, all primary path transverse over the link to S_{11}. The large number of *pktgen* and BFD control messages lead to congestion at the

Open vSwitch implementation. This resulted in wrongly monitored link status, unwanted delay on arrival of *pktgen* packets and thus failed recovery measurements. BFD failing to monitor the status is unacceptable and to overcome this problem, BFD session monitoring must be integrated at switches hardware for high performance line-level monitoring. To reduce the traffic load at intermediate switches, the *pktgen* transmission delay is configured to 0.2 ms, leading to an arrival delay of 1 ms at the destinations. This arrival delay lowers the accuracy of the recovery measurements, therefore the number of repetitions is increased to 3400. The BFD transmission interval is set to $T_{i,Set} = 5$ ms. Figure 7-13 shows the recovery measurements for scenario F.

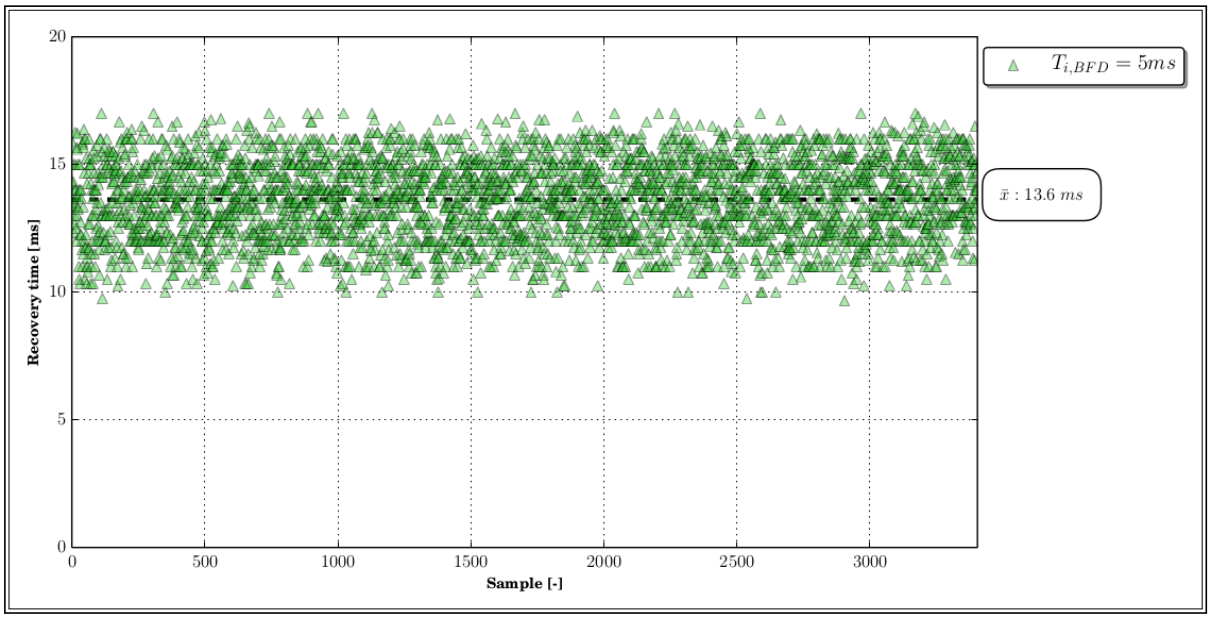


Figure 7-13: Recovery measurements on USnet topology at TU Delft NAS testbed - *The average recovery time with* $T_{i,Set} = 5$ *ms calculated from more than* 3400 *samples equals* 13.6 *ms.*

The experiment results for scenario F showed that an average recovery time of 13.6 ms is required on the USnet topology. This is well below the theoretical failover value of 15 ms. The most import result from this experiment is that our solution is independent from path length and scales with real-world network sizes. There exist some performance problems with the Open vSwitch implementation of BFD and the large number of packets to transport per second. We believe future Ethernet SDN switches with line-level enabled BFD monitoring can easily meet the failover requirements for carrier grade networks.

7-4 Experiment - Analysis

The results from scenario D, E and F are merged in the bar diagram of figure 7-14, where the average recovery time for the selected transmit windows is given with the 95% confidence level.

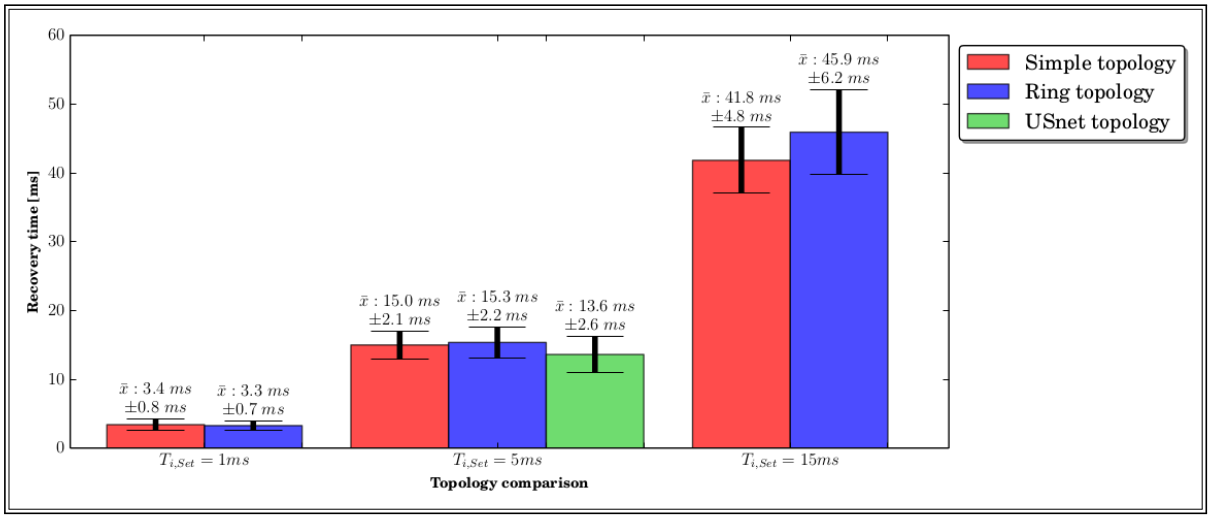


Figure 7-14: Comparison between recovery times on TU Delft NAS testbed using BFD monitoring - *A marginal performance difference is noticed between topologies when BFD is configured with $T_{i,Set} = 1$ ms. On $T_{i,Set} = 5$ ms the USnet topology shows best recovery times, but has the highest confidence interval. On the largest transmission window the average recovery time and the 95% confidence interval increased on the ring topology.*

The bar diagram with the confidence intervals show that there is only a substantial difference between the topologies with a larger transmission window for BFD. With the BFD transmission window set to $T_{i,Set} = 1$ ms the results are almost equal, which is remarkable as paths on the ring topology are almost twice as long as the simple topology and crankback routing is applied. This indicates that crankback routing does not influence recovery times negatively. The average recovery time on the USnet topology is lowest, which is remarkable because path lengths are longer on this topology. One would expect that if packets must transverse a longer path, recovery would take longer. Our experiments proved otherwise and showed that our proposal scales with path lengths and therefore with network size.

7-5 Conclusion on failover measurements

In this chapter we gave an description of the steps required to perform recovery experiments on our proposal to utilize Fast Failover Group Tables and active link monitoring. We defined three topologies on which liveliness monitoring is applied, measurement packets are generated by the Linux module *pktgen*, packet are captured by our real-time measurement tool and failures are initiated to the topology. With the topologies, multiple scenarios are configured at the TU Delft NAS testbed with Open vSwitch switches and the SURFnet testbed with Pica8 switches. Using the *pktgen* packet generator and a new developed packet capture script, recovery times are measured with high accuracy. Baseline measurements showed that active link monitoring by BFD is necessary for sub 50 ms recovery times, but also to prevent switch-over side effects during restoration to the primary path. On both testbeds Loss-of-Signal provides recovery times in order of seconds, where the hardware testbed is substantially faster than the Open vSwitch configuration. When the switch performing the recovery is administratively brought down, the Open vSwitch show a recovery time of 50.8 ms, which is insufficient to protect voice connections and meet requirements on carrier grade and industrial networks. With BFD monitoring enabled, recovery times below 50 ms are easily possible. With the lowest transmission interval possible at this moment, a recovery time of 3.4 ± 0.8 ms is possible. Our proposal does not only provide fast failover, it can handle crankback routing, scales with longer paths and larger networks and all without compromising performance. This makes our proposed solution not only usable for protection of carrier-grade networks, but also for industrial networks with time critical applications and high demands on robustness. At last, we saw during experiments on the USnet that the current implementation of Open vSwitch can not handle a large amount of packets and perform link monitoring simultaneously. This calls for purpose build Ethernet switches with active link monitoring at line level.

Chapter 8

Conclusion and future work

8-1 Conclusion

This thesis can globally be divided into three problem areas, being i.) a review into Software Defined Networking as new networking paradigm with its strengths and weaknesses in comparison with current network solutions (chapter 2 and 3), ii.) the research and experiments into path protection schemes and failure detection on common Ethernet networks in relation with SDN (chapter 4 and 7), and iii.) the investigation and development of routing algorithms that enable path protection schemes, while preserving optimal traffic flows on regular and non-purpose designed networks (chapter 5 and 6).

In chapter 2 we discussed the basic principles of SDN, were the control layer from switching and routing network devices is decoupled from the data plane and merged into a centralized control logic. The centralized logic, itself controlled by an application layer, has a global view from the network and has the capabilities to dynamically and ad-hoc control the hardware devices for optimal traffic flows through the network. For communication between the data plane and the control logic, the OpenFlow [1] protocol is commonly utilized. With OpenFlow, an inflexible switching network is transformed into an adaptable network capable of forwarding data on higher network layers. To gain insight of application areas of SDN, we have given an overview of existing SDN research work in chapter 3, where separation is made on scalability, robustness and security issues. To distinguish proposed solutions, we created a graphical framework in which solution are compared easily. Due to the centralization of the control logic, scalability issues exist on the number of hardware devices to control by a single entity. Solutions can be found in increasing performance of the central logic [31] or reduce the number of tasks to perform by the logic [30, 32]. To improve the security on SDN networks, malicious data flows must be marked and handled accordingly [47, 46]. Also generated network configurations must be checked on security flaws, before installation on the hardware devices [49]. On robustness two problem areas can be distinguished, being the resistance of the controller and the network itself against failures [39]. On both robustness areas we discovered opportunities for improvement, as we were curios which improvement SDN could bring on common IP Ethernet networks. Overall we found there is much research

needed on SDN, as no optimal and integrated solution is available at time of writing of this thesis.

To increase the robustness on networks in general, path recovery mechanisms exist. These mechanisms must detect a network failure and restore the path to retain connectivity. For carrier-grade networks, recovery is required within 50 ms [3], where industrial network demand recovery within 20 ms [5] and lower to not disrupt the provided network service. In chapter 4 we identified that pro-active protection schemes provides the fastest recovery, where a backup path is configured to the network together with the primary path. Path-based protection recovers the primary path with a complete disjoint backup path, where for link-based protection on each intermediate switch in the primary path a protection path must be configured [37]. Path-based protection, applied in [4], has two major drawbacks, as crankback routing is required and failure detection requires more time. A link-based protection scheme minimizes the need for crankback routing and failure detection times. Failure detection, can be provided by the physical layer or protocols at the datalink and network layer, such as Loss-of-Signal, RSTP [52] and OSPF [53]. We identified that none of these protocols can reach the required detection time in order of milliseconds. Therefore the active link monitoring protocol, available in Open vSwitch [23], BFD [42] is chosen to perform failure detection. For BFD we derived worst case theoretical failure detection windows based on packet trip-times and path length. In OpenFlow, one can configure Fast Failover Group Tables that monitor outgoing ports or groups, where based on priority and monitored link status, traffic is forwarded. We proposed to link the BFD link status to Fast Failover Group Tables for fast link-based path recovery. In chapter 7 we implemented our proposal to three OpenFlow enabled topologies to perform experiments on failure recovery times. The first topology was designed to measure the failover time in its most simplistic form, where the second utilized a ring topology to enable longer path lengths and the effect of crankback routing on recovery times. A real-world network, USNET [85, 86], is configured to ensure our proposal scales with path lengths and network size. Failure recovery, based on Loss-of-Signal detection, was in the order of seconds, where with active link monitoring by BFD we were able to reduce the recovery time to 3.4 ± 0.8 ms. On the ring and USNET topology, the recovery time remained equal, wherewith we can conclude that our proposal for link-based protection is valid, scales with network size and can cope with crankback routing. The experiments showed that with the application of active link monitoring, cheap common Ethernet networks can reach the failover requirements for expensive and purpose designed carrier-grade and industrial networks.

The last subject of this thesis was the investigation into algorithms that provide paths to enable the link-based protection scheme on network topologies in chapter 5. Therefore we defined the algorithmic problem for the link-based protection algorithm. For our problem, we found that disjoint path algorithms provides a solid basis, as no specific topology design is required and protection can be guaranteed with only two Dijkstra shortest path iterations. Suurballe's [64] and Bhandari's [65] algorithms provide disjoint path pairs to protect paths against link or switch failures with the MIN-SUM objective [69, 37]. The obtained path pairs are unsuitable for link-based protection without the application of crankback routing. As crankback routing is unwanted in a network topology and the MIN-SUM disjoint paths do not guarantee the usage of the optimal shortest path, we developed a protection algorithm that discovers protection paths for each intermediate switch in the shortest path. In our protection algorithm, we combined Bhandari's and an extended Dijkstra algorithm, such that protection against link or switch failures is provided by next-hop disjointness, while minimizing path costs

or crankback routing. The extended Dijkstra algorithm we developed, discovers protection paths, minimizes computational overhead and prevents routing loops. Overall, our protection algorithm solves the link-based protection algorithm problem and is applicable to regular networks. To prove the performance of our algorithm, a large number of simulations have been performed on random (ER-, and BA-networks [83, 84]) and real-world (USNET and Pan-European COST239) networks. The simulated topologies represent networks, on which our protection algorithm (and failover mechanism) can be applied. Results, in chapter 6, showed that our protection algorithm is capable of reducing the cost of protection paths with at least 25% in comparison with path-based protection schemes and reduces protection by forced crankback routing with at least 80% in sparse random networks. In well connected networks, the need for crankback routing is completely removed, while providing the required protection. Between link and switch failure protection only a small difference is noticed on path cost and crankback reduction. As the used link-based monitoring protocols can not guarantee correct functioning of the switch during a link failure, we suggest switch failure protection with minimized crankback routing as optimal setting for our protection algorithm.

Conclusive we have:

- shown the strengths and weaknesses that come with the application of SDN by providing a graphical framework to ease the classification of SDN solutions;
- investigated failure detection methods on common Ethernet networks and concluded that active link monitoring is required to reduce detection and failover times;
- proved that our proposal to link BFD status with the OpenFlow Fast Failover Group Table can reduce failover times to millisecond order;
- investigated existing routing and protection algorithms and concluded that these do not provide the capability to enable link-based protection without the application of crankback routing;
- developed and tested a link-based protection algorithm that fulfills our problem on failure protection and provides cost minimized protection paths and reduced the need for crankback routing;
- have shown that the robustness of common networks can be increased to the level of carrier-grade and industrial networks with the application of SDN.

8-2 Future Work

Although we believe SDN provides a solid foundation for future network solutions, especially in case of increasing robustness, there is room for future work on the following subjects.

Failover behavior of OpenFlow Fast Failover Group Table

In our current implementation the monitored link status (from BFD) is linked to the Fast Failover Group Table. Based on the priority and link status the forwarding direction is

determined. In functional state this means that the primary path is selected over the backup paths. When the outgoing link of the primary path shows jittering behavior, BFD will monitor this, and the Fast Failover Group Table will switch over between the primary and backup path on each monitored status change. This behavior is unwanted, as after each path failover, connectivity is lost for a short period of time. We believe the number of switch overs must be reduced to a minimum and an additional mechanism at the Fast Failover Group table is required to prevent this behavior.

Reduce BFD transmission interval

During the experiments the lowest BFD transmission interval possible to configure was 1 ms. This leads to a worse case failure detection window of 4 ms. From our latency measurements and the theoretical model we derived, it is possible to reduce the transmission interval below 1 ms. Due to the definition of the transmission interval in the BFD protocol as integer, further reduction on the transmission interval was not possible. Therefore we propose to redefine the parameters within the BFD protocol and integrate line-level detection mechanisms to reduce the transmission interval to the lowest interval possible. We expect that a failure detection window of 1 ms lays within the boundaries of the current Ethernet network technologies.

Integration with SDN and OpenFlow controllers

Our link-based protection algorithm is implementation to a standalone Python module compatible with the NetworkX. To fully benefit of the available features of the algorithm, it must be integrated to a SDN or OpenFlow controller. With actual network information and link status parameters, traffic flows over the network can be optimized and protected with the application of our protection algorithm.

Expand link-based protection algorithm

The current protection algorithm is dependent on a large number of parameters, which all influence the outcome. During this thesis we decided to limit the number of parameters to cost and crankback minimization, and link and switch failure protection. The parameter we abandoned is the possibility to compute paths that only provide partial protection against link and switch failures. This situation occurs when there exist no two disjoint paths between the source and destination switch during the discovery of Bhandari's algorithm. Although, Bhandari's algorithm is capable to produce partial disjoint paths, as we have shown in appendix C, we did not adapt the protection algorithm to it. To ensure correct functioning of the protection algorithm on all network topologies and offer the best protection possible, the algorithm must be expanded with the capabilities to cope with partial protection paths.

Appendix A

Failover requirements

The failover requirement of 50 ms in carrier-grade networks is used in this thesis as general requirements and is set as minimum requirement. In [93] an overview is given on applications on (industrial) networks and the desired failover times for the application to function correctly. The applications with desired failover time is given in the following summation:

- Web surfing / Mail [3] - $4sec$ - No high requirements set on recovery times. Longer recovery times are experienced as inconvenient;
- Gaming / Remote connections [3] - $< 200ms$ - Higher delays are inconvenient. Requirement for gaming depends on game type;
- Video streaming / Gaming [3] - $< 100ms$ - First Person Shooters and video-conversations with lip synchronization;
- Audio streaming / VOIP [3] - $< 150ms$ - Requires additional technology to smooth audio, otherwise the requirement changes to $< 50ms$ (carrier-grade network);
- Process Automation [5] - $200ms$ - Monitoring of automated processes in industrial networks;
- Time critical control processes [5] - $20ms$ - Industrial Ethernet for bump-less recovery;
- Machine control [6] - $5 - 10ms$ - Direct control of machinery in industrial networks.

Appendix B

Remove Find algorithm

The Remove Find algorithm [74] is given algorithm 8, where K is the number of requested disjoint paths, Q_k is the k-th shortest path, S_A is the source switch, S_B the destination switch and $\mathcal{N}$ the network where within the disjoint paths are discovered.

Algorithm 8 Remove Find algorithm for K disjoint paths

1. WHILE $k < K$:
2. EXECUTE Dijkstra's algorithm ($\mathcal{N}$, S_A, S_B) $\Longrightarrow Q_k$
3. IF Q_k exists:
4. REMOVE Q_k from $\mathcal{N}$
5. SET $k = k + 1$
6. ELSE:
7. Stop path discovery

The overall process of algorithm 8 is straight forward. At first, Q_1 is calculated between S_A and S_B. If a shortest path exists, this is removed from network $\mathcal{N}$, k is incremented and simple path Q_2 is calculated. The algorithm terminates when K paths are found or no simple path in found in $\mathcal{N}$.

Appendix C

Disjoint path network modifications

In this appendix Bhandari's algorithm to find disjoint paths is discussed with the three levels on disjoint path requirements as guideline. Depending on the requirement on disjoint level, modifications are made to the network, forcing shortest path algorithms to find disjoint paths. In stead of starting with the highest level, switch disjoint paths, first the easier understandable algorithm for link and partial link disjoint paths are discussed in section C-0-1 and C-0-2. The network modifications needed for switch and partial switch disjoint path calculation are discussed in section C-0-3 and C-0-4. Bhandari's algorithm uses a modified Dijkstra algorithm to find shortest paths in the network. A description of the (modified) Dijkstra algorithm in comparison to other shortest path algorithms are discussed in appendix D.

C-0-1 Link disjoint transformations

Bhandari's algorithm uses the link weight and link direction modifications to the network to find link disjoint paths. In algorithm 9, the process is given in three steps [94], where K is the number of disjoint paths, k is the disjoint path indicator with $k \geq 1$, Q_k is the k^{th} shortest path in network $\mathcal{N}$, J is the set of joint links between discovered shortest paths, $l(i, j)$ is a link from switch S_i to S_j, Q_I is the path wherewith Q_x intersects and P_k is the found and constructed disjoint path.

Algorithm 9 finds disjoint paths, because the direction of earlier found paths is reversed and links which are crossed multiple times, are stored in a temporary set and compared on path construction. When the shortest path algorithm arrives on a switch, the next switch is chosen based on the link weight of the outgoing links. In the second and higher iterations of the algorithm, the shortest path algorithm can be forced to travel backwards on earlier found shortest and simple paths, until a new link is found which leads to the destination switch. In figure C-1 an example of the the link modification process is given for $K = 2$.

The process in figure C-1 illustrates four steps of the link disjoint algorithm. In the Step 1 (figure C-1A) the shortest path Q_1 in $\mathcal{N}$ between S_1 and S_4 is calculated, where after the direction of Q_1 is reversed and the link weight is inverted in $\mathcal{N}$ (Step 2 - figure C-1B). Step

Algorithm 9 Link modifications for K disjoint paths

STEP 1 - NETWORK MODIFICATION $(S_A, S_B, \mathcal{N}, K)$

1. WHILE $k \leq K$:
2. EXECUTE Dijkstra's algorithm ($\mathcal{N}$, S_A, S_B) $\Longrightarrow Q_k$
3. IF Q_k exists and $k \neq K$:
4. a. REVERSE direction of links in Q_k
5. b. MULTIPLY link cost of each link in Q_k with -1
6. c. SET $k = k + 1$
7. ELSE:
8. Stop path discovery

STEP 2 - JOINT LINKS IDENTIFICATION

1. FIND joint links J in $\{Q_1, ..., Q_K\}$

STEP 3 - PATH CONSTRUCTION

1. IF J not empty:
2. WHILE $k \leq K$:
3. SET $Q_x = Q_k$
4. WHILE S_B not reached:
5. POP link $l(i, j)$ from Q_x
6. IF link l in J:
7. REPLACE $l(i, j)$ with $l(i, h)$ from Q_I
8. REPLACE Q_x with Q_I
9. ADD link l to P_k
10. SET $k = k + 1$

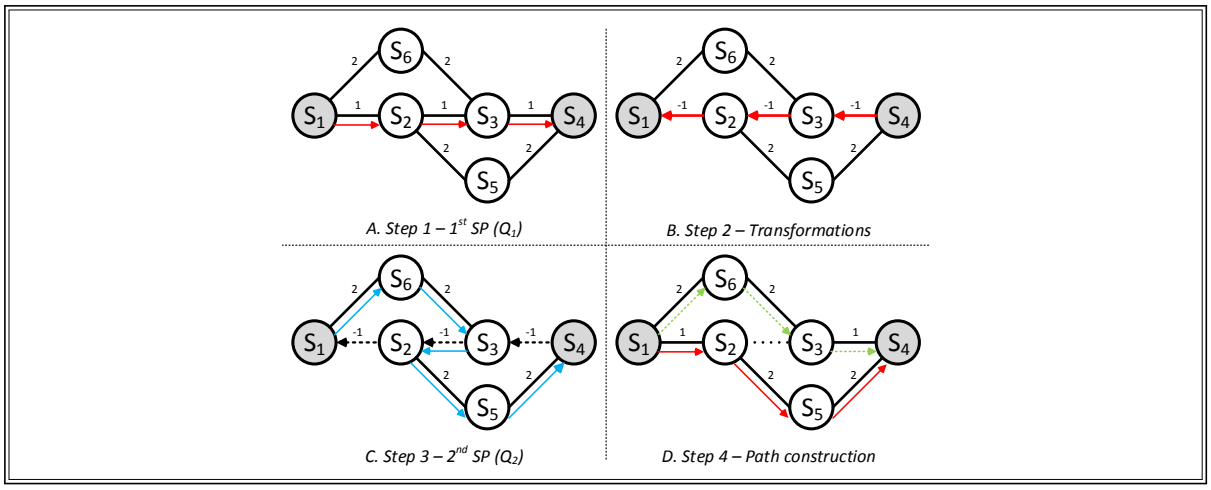


Figure C-1: Network transformation process for link disjoint paths - *In figure C-1A the shortest path (Q_1) between S_1 and S_4 is found. The second step is to reverse the direction of Q_1 and invert the link weight (figure C-1B). In figure C-1C the second shortest path (Q_2) is found using the modified Dijkstra algorithm. During the second step of the algorithm, link $S_2 \leftrightarrow S_3$ is identified as joint link. Disjoint path construction for P_1 starts with the link $S_1 \leftrightarrow S_2$, where the second link ($S_2 \leftrightarrow S_3$) is joint between Q_1 and Q_2. Therefore the next links to add for P_1 are taken from Q_2 until destination is found. A similar process counts for P_2, leading to disjoint paths of figure C-1D.*

3 (figure C-1C shows the characteristic behavior of the link disjoint algorithm. At switch S_1 the only option for the second shortest path Q_2 is to travel to switch S_3 via S_6, where at switch S_3 the path is forced to transverse link $S_2 \leftrightarrow S_3$ for the second time to reach switch S_2, S_5 and eventually the destination switch S_4. The decision made at S_2 it is not to travel to the starting switch S_1, because this switch is already in the path and a loop is prevented. During Step 4 (figure C-1D) link $S_2 \leftrightarrow S_3$ is identified as joint link and disjoint path P_1 can be constructed from link $S_1 \leftrightarrow S_2$. The second link from Q_1would be $S_2 \leftrightarrow S_3$, but this is identified as joint link with Q_2. The second link added to P_1 from Q_2 is $S_2 \leftrightarrow S_5$ and this process continues until path $S_1 \rightarrow S_2 \rightarrow S_5 \rightarrow S_4$ is constructed. For P_2 the first two links are added from Q_2. The third link $S_3 \leftrightarrow S_4$is originating from Q_1, as link $S_3 \leftrightarrow S_2$ is a joint link.

C-0-2 Partial link disjoint transformations

A network topology does not always provide multiple links between intermediate switches, as seen in the example topologies so far. Also, in the special case where a segment of the network contains a path graph, link disjoint paths cannot be found. The best solution is to find partial link disjoint paths and minimize traversed links. In [65] and [95] an addition is made to the link disjoint algorithm (algorithm 9). The additional rule is given in algorithm 10, where the identification of joint links and the disjoint path construction process of are not shown, because they are equal to algorithm 9.

In algorithm 10 α is defined as $\alpha = c_l + \epsilon$, where c_l is the cost of the link and ϵ an additional high cost. The addition of action $4a$ of the algorithm makes it possible to travel not only

Algorithm 10 Link modifications for K partial link disjoint paths

STEP 1 - NETWORK MODIFICATION

1. WHILE $k \leq K$:
2. EXECUTE Dijkstra's algorithm ($\mathcal{N}$, S_A, S_B) $\Longrightarrow Q_k$
3. IF Q_k exists and $k \neq K$:
4. a. SET each link in direction of Q_k with weight α
5. b. REVERSE direction of links in Q_k
6. c. MULTIPLY link cost of each reversed link in Q_k with -1
7. d. SET $k = k + 1$
8. ELSE:
9. Stop path discovery

backwards over earlier found shortest and simple paths, but also forwards over a traversed link at cost α. The additional cost ϵ must be chosen to discourage the shortest path algorithm to traverse this link. In [65] and [95] a suggestion for ϵ is given as equation C-1, where M is a constant.

$$\epsilon = M \cdot \sum c_l, c_l \in \mathcal{N} \tag{C-1}$$

Small values for M will encourage the use of partial link disjoint paths. In [65] a value of $M = 4$ as suggested as optimum. Figure C-2 gives an illustration of the process of the application of algorithm 10, where the topology of figure C-1 is extended with an extra switch (S_0) with a single connection to the existing network.

The process in figure C-2 differs from figure C-1 in transformation method and calculating the second shortest path *(Q_2)*. Because it possible to travel over an earlier found path (figure C-2B) , two partial disjoint paths can be found. Due to high cost α, the Q_2 travels from S_3 to S_2, instead of traveling directly to the destination switch S_4 (figure C-2C). The construction of P_1 starts with the first link of Q_1, being $S_0 \rightarrow S_1$. This link is transversed by both Q_1 and Q_2, but is not a joint link, as the travel direction is equal. Therefore link $S_0 \rightarrow S_1$ from Q_1 is added to P_1. The construction process continues until joint link $S_3 \rightarrow S_2$ is reached, where link $S_3 \rightarrow S_4$ is added to P_1 and the destination is found via path $S_0 \rightarrow S_1 \rightarrow S_6 \rightarrow S_3 \rightarrow S_4$. For P_2 a similar construction process leads to path $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_5 \rightarrow S_4$. The link modification process in Bhandari's algorithm enables the possibility to compute link disjoint paths and with the possibility to travel over earlier found path at a high cost, the algorithm can find partial link disjoint paths. This will increase the network survivability on link failures, but a switch failure can still lead failure of multiple disjoint paths.

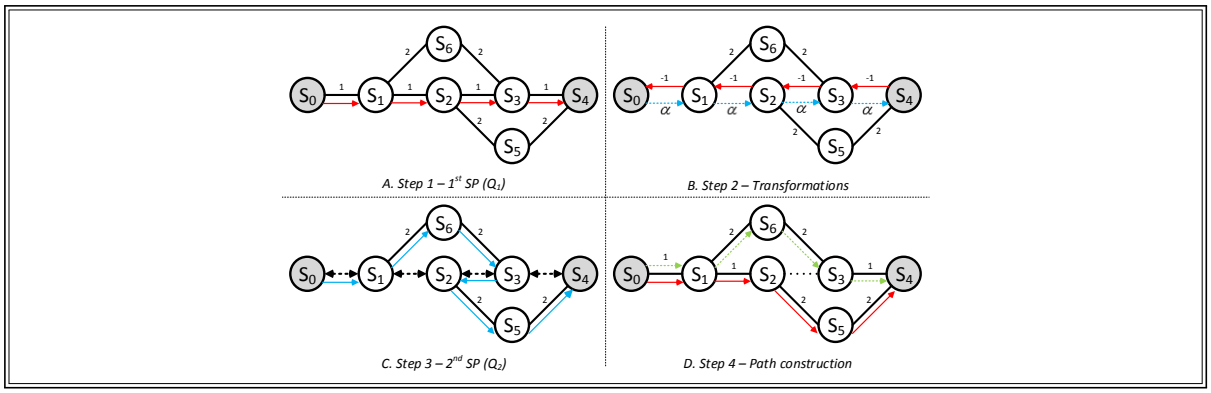


Figure C-2: Network transformation process for partial link disjoint paths - *In figure C-2A the shortest path (Q_1) between S_0 and S_4 is found with Dijkstra's algorithm. Step 2 (figure C-2B) illustrates the link transformations applied to $\mathcal{N}$ in both directions of Q_1. In C-2C the simple path Q_2 is discovered, where the link $S_0 \leftrightarrow S_1$ is used twice, despite the high cost α. The remaining path is equal to the path found in figure C-1C. Step 4 (figure C-2D) shows the constructed partial link disjoint paths P_1 and P_2 in $\mathcal{N}$.*

C-0-3 Switch disjoint transformations

R. Bhandari made also simplification to switch disjoint path calculation process of Suurballe's algorithm. The algorithm to compute switch disjoint paths is an extension to the link disjoint path algorithm, discussed in section C-0-1. Where algorithm 9 only blocks the possibility travel over a found path twice, it remains possible to visit the same switch for the second time, using different links (figure 5-1B gives an example of this situation). To retain the possibility of a second visit to a switch, R. Bhandari applies the switch splitting technique to network and rewire incoming and outgoing links over split switches. Step 1 of the splitting algorithm is given in algorithm 11 for K disjoint paths in network $\mathcal{N}$, where S_x^* is a cloned switch from switch S_x. Step 3 (finding joint links) and 4 (path construction) are equal to algorithm 9)

Actions 1 to 5 in algorithm 11 are equal to the link disjoint algorithm. Actions $6c$ to $9f$ describe the switch splitting process. At first, switch S_x is split into switches S_x and S_x^*, where after the link $S_x^* \rightarrow S_x$ is added with cost 0. The next step is to rewire in- and outgoing links on S_x and S_x^*, such that arrivals from a non visited random switch in $\mathcal{N}$, to a split switch S_x, always leads to an additional travel over an earlier calculated shortest path. From the split switch S_x^* it is possible to travel to non-visited switches in the network. If the outgoing links would be rewired to S_x, a path would arrive at S_x, but would also leave from S_x. The splitting and according rewiring makes that simple paths $Q_{k>1}$ contain links and switches which do not exist in $\mathcal{N}$. Therefore the paths Q_2 to Q_k must be translated to replace S_x^* with S_x. It is possible that the link $S_x^* \rightarrow S_x$ is present in Q_k, which leads to the translation $S_x \rightarrow S_x$. Double instances of S_x are removed to prevent calculation and construction errors in follow up steps. Figure C-3A to C-3D illustrate the splitting process given in algorithm 11.

As seen in figure C-3, the splitting process is more complex to execute. After link reversal and weight modification of Q_1, the intermediate switches S_2 and S_3 are split to S_2^* and $S_3^*$and connected via the zero cost link. The links non-directed links connected to S_2 are converted

Algorithm 11 switch modifications for K disjoint paths

STEP 1 - NETWORK MODIFICATION

1. WHILE $k \leq K$:
2. EXECUTE Dijkstra's algorithm ($\mathcal{N}$, S_A, S_B) $\Longrightarrow Q_k$
3. IF Q_k exists and $k \neq K$:
4. a. REVERSE direction of links in Q_k
5. b. MULTIPLY link cost of each link in Q_k with -1
6. c. SPLIT intermediate switches ($S_x \neq S_A, S_B$) in Q_k to $\{S_x, S_x^*\}$
7. d. CREATE unidirectional link from S_x^* to S_x with cost 0
8. e. REWIRE incoming links from Q_k on S_x to S_x^*
9. f. REWIRE outgoing links from S_x to S_x^*
10. g. SET $k = k + 1$
11. ELSE:
12. Stop path discovery

STEP 2 - TRANSLATE PATHS

1. WHILE $k \leq K$:
2. a. REPLACE S_x^* with S_x in Q_k
3. b. REMOVE double instances of S_x from Q_k
4. c. SET $k = k + 1$

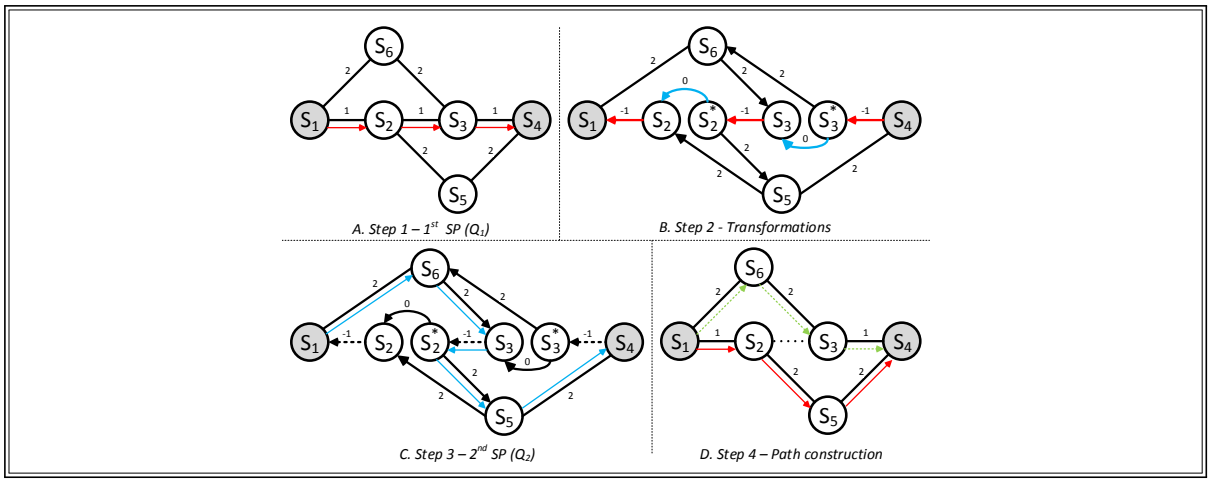


Figure C-3: Network transformation process for switch disjoint paths - *The first step is to calculate the shortest path (Q_1) between S_1 and S_4 (figure C-3A). Figure C-3B. show action 5a. to 10f. in the algorithm, where Q_1 is reversed, link weights inverted, switches are split and all incoming and outgoing links are rewired. Step 3 (figure C-3C) gives path Q_2 ($S_1 \rightarrow S_6 \rightarrow S_3 \rightarrow S_2^* \rightarrow S_5 \rightarrow S_4$) after the link and splitting modifications. Translation of Q_2 leads to $S_1 \rightarrow S_6 \rightarrow S_3 \rightarrow S_2 \rightarrow S_5 \rightarrow S_4$ and the disjoint path are constructed (figure C-3D).*

to directed links, where the reversed shortest path link $S_3 \rightarrow S_2$ is rewired to $S_3 \rightarrow S_2^*$, the non-shortest path incoming link $S_5 \rightarrow S_2$ remains untouched and the outgoing link $S_2 \rightarrow S_5$ is rewired to $S_2^* \rightarrow S_5$. A similar process is executed for S_3, leading to the second shortest path Q_2 of $S_1 \rightarrow S_6 \rightarrow S_3 \rightarrow S_2^* \rightarrow S_5 \rightarrow S_4$ (figure C-3C). After translation Q_2 becomes $S_1 \rightarrow S_6 \rightarrow S_3 \rightarrow S_2 \rightarrow S_5 \rightarrow S_4$ and the path construction process computes the two disjoint paths as shown in figure C-3D.

C-0-4 Partial switch disjoint transformations

The last algorithm discussed, is the partial switch disjoint algorithm by [65]and [95]. Again, a small addition is made to the original algorithm. As with partial link disjoint paths, the links in the already found paths may be traveled at cost α. With this modification alone, a path can travel to S_x, but from S_x there are no other outgoing links than already found paths. With analogy to α for traversed links, high cost β is defined to travel from S_x to S_x^*, which allows switches to be visited by multiple disjoint paths. The addition of high cost β leads to algorithm 12. As with algorithm 10, only the modification process is given.

The value for high cost β is not given in [65]. With the following reasoning, a indication for β can be determined. Because the resistance to travel to the split switch S_x is already embedded in α, there seems no reason to give β a high value and $\beta = 0$ is a valid option. To create a diversity in the partial disjoint paths, the cost to travel over switch S_x must increase for each path that traverses it. Thus, for diversity in the partial disjoint paths, β must be in the order size of α, which is equal to equation C-1. In figure C-4 the modification and path computing process for algorithm 12 are illustrated for the topology used in figure C-2.

In figure 12 the extensive modifications network are visible. The partial link disjoint modifications with high cost α allow links to be used for multiple paths and the partial switch

Algorithm 12 Switch modifications for K partial switch disjoint paths

STEP 1 - NETWORK MODIFICATION

1. WHILE $k \leq K$:
2. EXECUTE Dijkstra's algorithm ($\mathcal{N}$, S_A, S_B) $\Longrightarrow Q_k$
3. IF Q_k exists and $k \neq K$:
4. a. SET each link in direction of Q_k with high cost α
5. b. REVERSE direction of links in Q_k
6. c. MULTIPLY link cost of each link in Q_k with -1
7. d. SPLIT intermediate switches in Q_k to $\{S_x, S_x^*\}$
8. e. CREATE link from S_x^* to S_x with cost 0
9. f. CREATE link from S_x to S_x^* with high cost β
10. g. REWIRE incoming links from Q_k on S_x to S_x^*
11. h. REWIRE outgoing links from S_x to S_x^*
12. i. SET $k = k + 1$
13. ELSE:
14. Stop path calculation

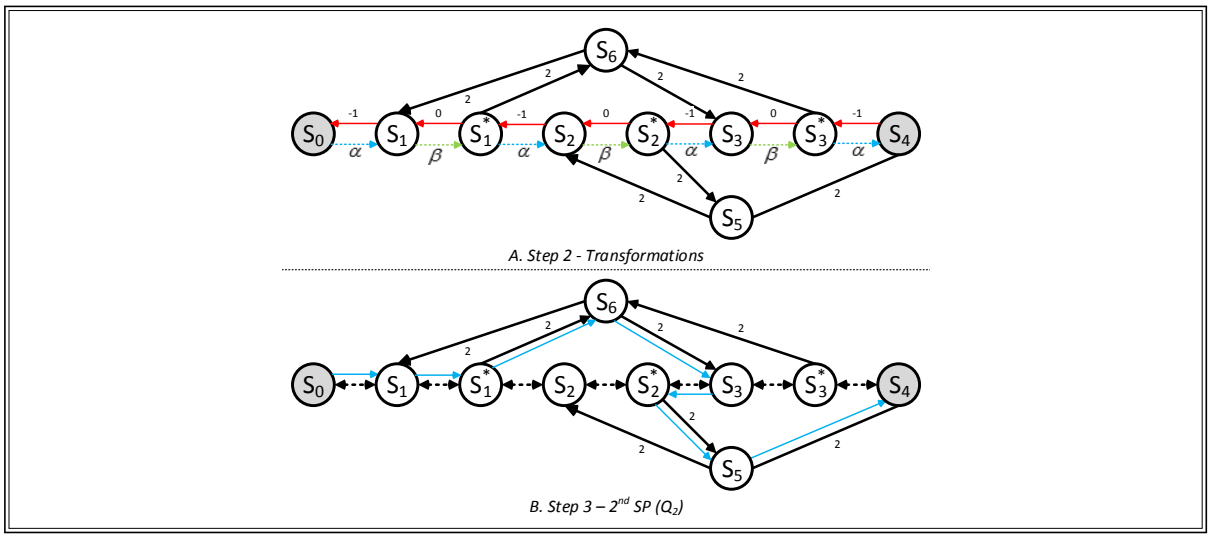


Figure C-4: Network transformation process for partial switch disjoint paths - *In figure 12A the link weight modifications for partial switch disjointness are drawn. Link from Q_1 are reversed, link cost multiplied by -1 and intermediate switches are split. Between S_2 and S_2^* an additional link is added with high cost β. The high cost links α and β make it possible to transverse switches S_1 and S_1^* and find the second shortest path Q_2 (figure 12B).*

modification (β) allows switches to be visited multiple times by multiple paths. The second shortest path $Q_2$is found as $S_0 \rightarrow S_1 \rightarrow S_1^* \rightarrow S_6 \rightarrow S_3 \rightarrow S_2^* \rightarrow S_5 \rightarrow S_4$, which translates to $S_0 \rightarrow S_1 \rightarrow S_6 \rightarrow S_3 \rightarrow S_2 \rightarrow S_5 \rightarrow S_4$ in $\mathcal{N}$. Disjoint paths can be constructed as described in subsection C-0-2. Because switches are split in in $\mathcal{N}$, the length of the computed shortest path for $k > 1$ increases. This indicates that a shortest path algorithm requires more time to compute a solution.

Appendix D

Shortest path algorithms

In this appendix the modified Dijkstra algorithm is discussed in relation to the original Dijkstra algorithm, in relation to other shortest path algorithms and in terms of computational complexity. Considerations for the shortest path algorithm are given in section D-0-5. In section D-0-6 and D-0-7 the original and modified algorithm for the Dijkstra algorithm are given.

D-0-5 Overview of shortest path algorithms

Algorithms for computing shortest paths in a network or graph are well studied. Three well known algorithms in chronological order of discovery are the Bellman-Ford [67], Dijkstra [66], and the Floyd-Warshall [96] algorithm, each with their strengths and weaknesses. From this three basic algorithms, multiple modified versions can be found with specific aims and purposes. For Bhandari's disjoint path algorithm the requirements for the shortest path algorithm are: i) a shortest path without loops is computed between the source and destination switch and ii) in the shortest amount of time possible. From Bhandari's algorithm (Appendix C) we learned that the algorithm must handle negative link costs as result of the required link modifications.

The Bellman-Ford algorithm discovers shortest paths from a source to every other switch in the network in $O(N \cdot L)$ time, where N is the number of switches and L is the number of links in the network. Advantages of the algorithm are found in the ability to process negative link weights and detect negative-loop cycles in a network. Dijkstra's original shortest path algorithm is classified as a greedy algorithm and has a reduced complexity $O(N^2)$ compared with the Bellman-Ford algorithm. Dijkstra's original algorithm can not handle negative link weights nor detect negative loop cycles. The last basic shortest path algorithm from Floyd-Warshall is categorized as an all-pairs shortest path algorithm, has a complexity of $O(N^3)$ and can handle negative link weights. All observations from above are summarized in table $D-1$.

If we make the assumption that $N < L$, the conclusion can be drawn that Dijkstra's algorithm will provide minimal computational delay. Because Bhandari's algorithm introduces negative

Algorithm	*Complexity*	*Negative edges*	*Comment*
Bellman-Ford	$O(N \cdot L)$	Yes	
Dijkstra	$O(N^2)$	No	Greedy algorithm
Floyd-Warshall	$O(N^3)$	Yes	All-pair algorithm

Table D-1: Comparison of basic shortest path algorithms

link weights, the Bellman-Ford algorithm seem a better choice to implement in the disjoint path algorithm, as the algorithm is also more efficient than the Floyd-Warshall algorithm. In [97] M. Fredman and R. Tarjan proposed a modification to Dijkstra's algorithm, so that it can handle negative link weights and the complexity reduced to $O(L + N \log N)$. The modified Dijkstra's algorithm is therefore the most optimal shortest path algorithm which complies to the set requirements.

D-0-6 Original Dijkstra shortest path algorithm

For completeness the original Dijkstra algorithm is briefly discussed. The basic functioning of the algorithm is based on minimizing the cost to the next switch (neighbor) locally, which determines the greedy behavior of the algorithm. Three basic steps are defined in Dijkstra's algorithm, being i) initialization, where the initial costs for the source switch and all destination switches are set, ii) the selection and processing of the neighbor switch and iii) the relaxation process, where path costs to the next hop switches are updated. For software implementations of the algorithm, queues are commonly used, to extract switches during the switch selection process. Algorithm 13 gives Dijkstra's algorithm in pseudo code, where S_A is the source switch, S_B is destination switch, $\mathcal{C}(S_x)$ is the cost array to travel to switch S_x from S_A, $w(l_{j\to n})$ is the weight of the link between two switches, S_j is the selected switch, and U is the set of all switches in the network.

Dijkstra's algorithm calculate the cost to travel to from source node S_A to destination node S_B. In the initialization step, the cost array $\mathcal{C}$ with costs to travel to all switches is defined and each value is set to infinity, except for S_A, which is set to $\mathcal{C}(S_A) = 0$. Also a list of predecessors π is defined and set to an not existing switch S_{-1}. During step 2 the switch with the lowest cost is selected from cost array $\mathcal{C}$. When the extracted switch is the destination switch, the algorithm is terminated and $\mathcal{C}$ and π are returned. Otherwise the selected switch is sent to the relaxation process. At the relaxation process the cost to all it's neighbors S_n in array $\mathcal{C}$ are relaxed and predecessor array π is updated, if they comply to the set relax condition. With the list of predecessors the path from source to destination can be constructed.

As mentioned earlier, this algorithm is not suitable with negative link costs and does not guarantee loop free paths. A negative link in the network cost will result in a continuous relaxation loop, which will result in routing loops and no termination of the algorithm. Another disadvantage of this algorithm is that the path must be constructed with the use of predecessors after the algorithm has terminated.

The running time for the algorithm can be derived as follows. In Step 2 the minimum cost must be extracted from the array, which consists of a linear search in the array with length

Algorithm 13 Dijkstra's original shortest path algorithm

STEP 1 - INITIALIZATION

1. a. SET $\mathcal{C}(S_x) = \infty$, where $S_x \in U$
2. b. SET $\mathcal{C}(S_A) = 0$
3. c. SET $\pi(S_x) = S_{-1}$, where $S_i \in U$

STEP 2 - SWITCH SELECTION

1. EXTRACT-MIN S_j such that $\mathcal{C}(S_j) = min_{S_x \in U}\mathcal{C}(S_x)$
2. IF $S_j = S_B$:
3. Stop algorithm and return $\{\mathcal{C}, \pi\}$
4. ELSE:
5. Go to step 3

STEP 3 - RELAXATION

1. For all neighbors S_n of S_j in $\mathcal{N}$:
2. IF $\mathcal{C}(S_j) + w(l_{j\to n}) < \mathcal{C}(S_n)$:
3. a. SET $\mathcal{C}(S_n) = \mathcal{C}(S_j) + w(l_{j\to n})$
4. b. SET $\pi(S_n) = S_j$
5. c. Go to Step 2

N. The extraction process must be performed in worst case N times to visit all switches. The switch selection requires thus $O(N^2)$ time. Relaxation of the cost is performed (step 3) with a maximum of L (number of links) times, where each relaxation computation takes $O(1)$ time. In total the given Dijkstra's algorithm runs in $O(N^2 + L)$, which is can be reduced to the overall complexity$O(N^2)$.

D-0-7 Modified Dijkstra shortest path algorithm

Three modifications are required in order to use Dijkstra's algorithm in the disjoint path algorithm. First, the algorithm be adapted such that negative links costs can be applied. Second, the algorithm must directly output the shortest path, without the need for path construction and at last the running time must be reduced. All modifications are assimilated in algorithm 14, where $\mathcal{C}$ is the cost array, P is the discovered path, C_P is the cost for path P, P^* is the shortest path between S_A and S_B, V is the set of visited nodes and Q is a priority queue.

The first modification includes the application of a priority queue to store intermediate results. More on the benefits of priority will be discussed later on this section. To enable negative

Algorithm 14 Dijkstra's modified shortest path algorithm

STEP 1 - INITIALIZATION

1. a. SET $\mathcal{C}(S_x) = \infty$, $S_x \in U$
2. b. SET $C_P(S_A) = 0$,$\mathcal{C}(S_A) = 0$
3. c. SET $P = S_A$
4. d. ADD S_A to V
5. e. INSERT $\{C_P(S_A), S_A, P\}$ in Q

STEP 2 - SWITCH SELECTION

1. WHILE Q is not empty:
2. EXTRACT-MIN $\{C_P(S_j), S_j, P\}$ such that $C_P(S_j) = min_{S_x \in Q}\mathcal{C}(S_x)$
3. IF S_j is S_B :
4. Stop algorithm and return $\{P^*, \mathcal{C}\}$
5. IF S_j not in V:
6. a. ADD S_j to V
7. b. Go to Step 3

STEP 3 - RELAXATION

1. FOR all neighbors S_n of S_j in $\mathcal{N}$:
2. IF $C_P(S_j) + w(l_{j\to n}) < \mathcal{C}(S_n)$:
3. a. SET $\mathcal{C}(S_n) = \mathcal{C}(S_j) + w(l_{j\to n})$
4. b. SET $C_P(S_n) = C_P(S_j) + w(l_{j\to n})$
5. c. SET $P = P + S_n$
6. d. INSERT $\{C_P(S_n), S_n, P\}$ in Q
7. e. Go to Step 2

link weights, the set of visited switches V is introduced. After a switch is extracted from the priority queue, it is relaxed and added to V. This mechanism prevents multiple iterations of the algorithm with the same switch and negative link weights will not result in continuous updates of the cost array. The priority queue is filled with a set containing the path discovered so far P, its corresponding cost C_P and the selected switch S_j. During the relaxation process, the cost array $\mathcal{C}(S_n)$, path cost $C_P(S_n)$ and path P are updated, if the relaxation condition is met. The algorithm is will terminate when the destination switch S_B is extracted from Q.

As discussed earlier, the running time of Dijkstra's algorithm can be reduced to to $O(L + N \log N)$. In [97] and [98] the switch selection process in the queue is optimized from a linear array search to a priority queue, which utilizes Fibonacci and binary heaps. Instead of storing the variables in an array, the variables are stored in a priority queue. A priority queue represents a smart tree, where computational tasks are performed in a more efficient way, reducing the running time and complexity. Two priority queues will be compared, being Fibonacci and binary heaps. Fibonacci heaps provide the best optimization, where binary heaps are easier for software implementations. In table D-2, the computation complexity for linear arrays, Fibonacci and binary heaps are given for the selection and relaxation steps of the modified Dijkstra algorithm.

Queue operation	*Operations*	*Array*	*Binary*	*Fibonacci*
Extract minimum	N	N	$\log N$	$\log N$
Insert	L	1	$\log N$	1
Total complexity		$O(L + N^2)$	$O((N + L) \log N)$	$O(L + N \log N)$

Table D-2: Complexity comparison for linear arrays and optimized heaps

The extraction complexity in binary and Fibonacci heaps equals $O(\log N)$, compared the $O(N)$ for linear arrays. At worst case, the algorithm is iterated N times, so the heaps complexity equals $O(N \cdot \log N)$. Insert variables to the priority queue requires $O(1)$ for linear arrays and Fibonacci heaps, where binary heaps require $O(\log N)$. The maximum number of insertions is in the order of the number of links L, which result in the total complexity for Dijkstra's algorithm as given in table D-2. Applying Fibonacci heaps for priority queues will result in the most optimal (software) implementation for the modified Dijkstra algorithm, but binomial heap optimization is mostly applied to heap queues in programming languages like Python.

Appendix E

Detailed simulation results link-based protection algorithm

In this Appendix we give the detailed breakdown of the simulation results of the link-based protection algorithm. The results are separated within different sections, where separation is made on network, protection and minimization setting. Parameters found in the results are summarized in table E-1.

Classification	*Number*	*Number Ratio* β	*Cost Ratio* α	*Cost Conf. Interval*
Via shortest path	Q_N	Q_R	Q_α	Q_{CI}
Via destination	D_N	D_R	D_α	D_{CI}
Via backup path	B_N	B_R	B_α	B_{CI}
Via crankback	C_N	C_R	C_α	C_{CI}

Table E-1: Simulation result parameter description

In table E-1 the discovered protection paths are classified on how the destination is found. For each classification the number of appearances x_N, the ratio between the number of appearances and the total number of discovered solutions x_R, the cost ratio (equation 6-1) $x_\alpha$and the cost ratio interval x_{CI} is given. The results are presented in graphical and tabular form, where the performance indicators are analyzed on a per hop basis. For example, Q_R gives the ratio of protection paths which join with remaining shortest path Q_{1R} and D_α gives the path cost ratio between the average cost of the protection path discovered directly to the destination and the cost for crankback routing $C(CR_{S_i})$.

E-1 BA networks switch failure protection with minimized cost setting

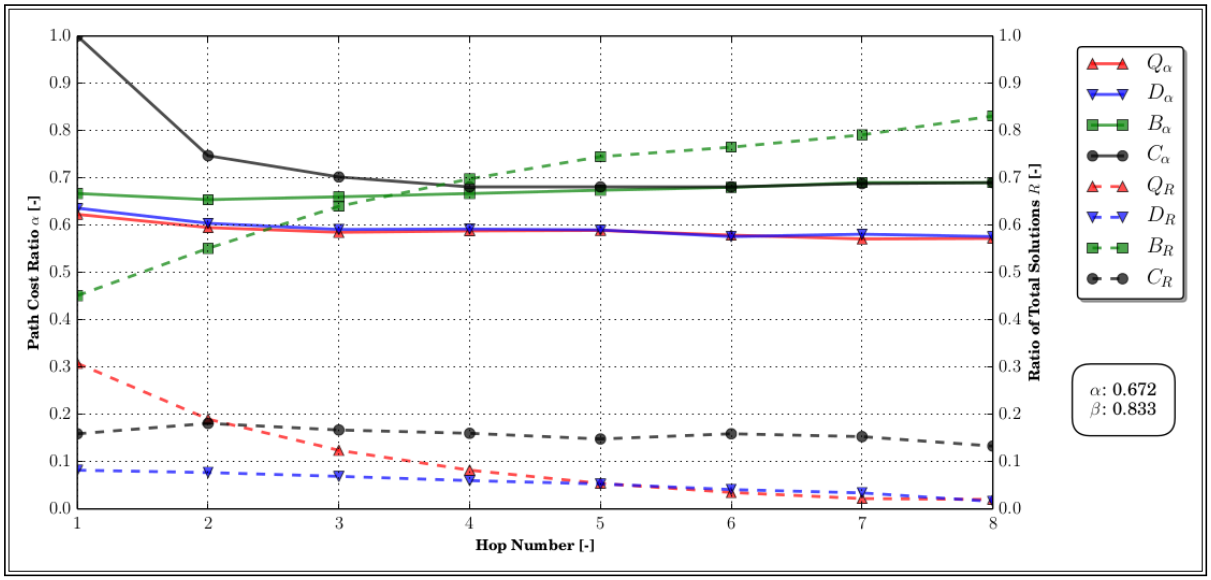


Figure E-1: Switch failure protection - BA-networks - minimized cost - $M = 2$

Hop	*Total*	Q_N	Q_R	Q_α	Q_{CI}	D_N	D_R	D_α	D_{CI}
1	96362	29711	0,308	0,623	[0.35 0.943]	7923	0,082	0,636	[0.372 0.956]
2	82748	15760	0,19	0,595	[0.348 0.933]	6393	0,077	0,604	[0.366 0.946]
3	59040	7341	0,124	0,585	[0.355 0.917]	4087	0,069	0,591	[0.364 0.928]
4	34244	2791	0,082	0,588	[0.347 0.92]	2064	0,06	0,592	[0.363 0.928]
5	16521	886	0,054	0,589	[0.346 0.923]	873	0,053	0,59	[0.355 0.918]
6	6647	230	0,035	0,579	[0.34 0.895]	272	0,041	0,576	[0.358 0.886]
7	2141	47	0,022	0,571	[0.335 0.832]	73	0,034	0,581	[0.347 0.883]
8	610	12	0,02	0,572	[0.305 0.856]	10	0,016	0,576	[0.349 0.778]
		B_N	B_R	B_α	B_{CI}	C_N	C_R	C_α	C_{CI}
		43433	0,451	0,667	[0.404 0.962]	15295	0,159	1	[1. 1.]
		45615	0,551	0,654	[0.401 0.955]	14980	0,181	0,747	[0.416 1.]
		37771	0,64	0,66	[0.409 0.952]	9841	0,167	0,702	[0.408 1.]
		23899	0,698	0,667	[0.414 0.952]	5490	0,16	0,681	[0.401 1.]
		12314	0,745	0,674	[0.425 0.952]	2448	0,148	0,681	[0.409 0.977]
		5086	0,765	0,68	[0.43 0.952]	1059	0,159	0,681	[0.42 0.972]
		1694	0,791	0,69	[0.442 0.958]	327	0,153	0,688	[0.391 0.979]
		507	0,831	0,69	[0.441 0.937]	81	0,133	0,69	[0.399 0.966]

Table E-2: Switch failure protection - BA-networks - minimized cost - $M = 2$

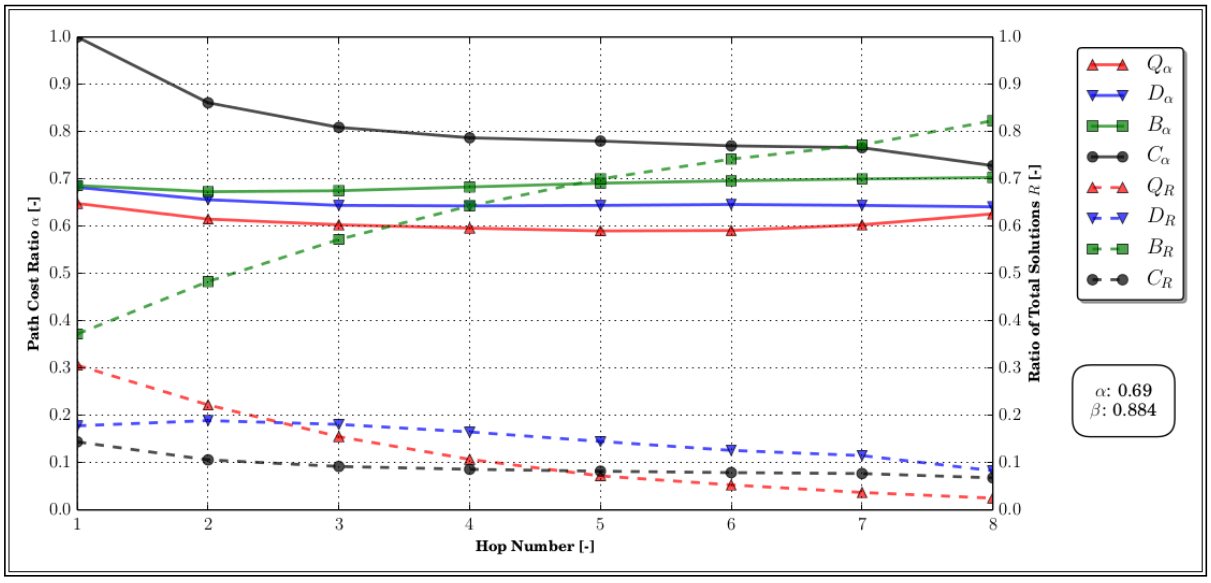


Figure E-2: Switch failure protection - BA-networks - minimized cost - $M = 6$

Hop	*Total*	Q_N	Q_R	Q_α	Q_{CI}	D_N	D_R	D_α	D_{CI}
1	94999	29071	0,306	0,648	[0.379 0.953]	16896	0,178	0,682	[0.41 0.967]
2	80976	17997	0,222	0,615	[0.374 0.932]	15290	0,189	0,656	[0.402 0.951]
3	58750	9127	0,155	0,603	[0.373 0.923]	10613	0,181	0,644	[0.398 0.951]
4	35416	3787	0,107	0,596	[0.37 0.91]	5830	0,165	0,643	[0.4 0.942]
5	17649	1272	0,072	0,59	[0.367 0.905]	2564	0,145	0,644	[0.393 0.935]
6	7260	385	0,053	0,591	[0.366 0.904]	915	0,126	0,646	[0.39 0.933]
7	2478	91	0,037	0,603	[0.368 0.922]	284	0,115	0,644	[0.403 0.939]
8	747	19	0,025	0,626	[0.357 0.959]	62	0,083	0,641	[0.402 0.901]
		B_N	B_R	B_α	B_{CI}	C_N	C_R	C_α	C_{CI}
		35363	0,372	0,686	[0.411 0.966]	13669	0,144	1	[1. 1.]
		39110	0,483	0,673	[0.409 0.96]	8579	0,106	0,861	[0.529 1.]
		33590	0,572	0,675	[0.414 0.959]	5420	0,092	0,809	[0.5 1.]
		22756	0,643	0,683	[0.422 0.96]	3043	0,086	0,787	[0.5 1.]
		12360	0,7	0,691	[0.43 0.959]	1453	0,082	0,78	[0.474 0.992]
		5389	0,742	0,696	[0.435 0.959]	571	0,079	0,77	[0.475 0.988]
		1913	0,772	0,7	[0.439 0.959]	190	0,077	0,766	[0.477 0.958]
		615	0,823	0,703	[0.45 0.959]	51	0,068	0,728	[0.435 0.983]

Table E-3: Switch failure protection - BA-networks - minimized cost - $M = 6$

E-2 ER networks switch failure protection with minimized cost setting

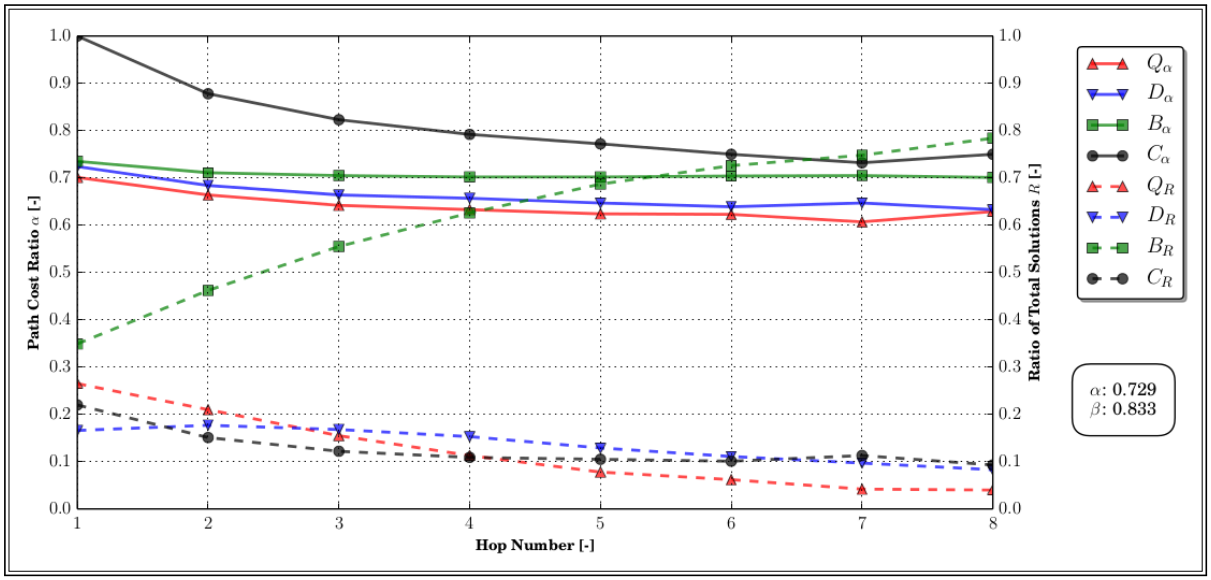


Figure E-3: Switch failure protection - ER-networks - minimized cost - $M = 2$

Hop	*Total*	Q_N	Q_R	Q_α	Q_{CI}	D_N	D_R	D_α	D_{CI}
1	78047	20682	0,265	0,701	[0.426 0.968]	12990	0,166	0,724	[0.446 0.975]
2	68868	14481	0,21	0,664	[0.413 0.954]	12169	0,177	0,684	[0.429 0.964]
3	53100	8217	0,155	0,642	[0.405 0.942]	8913	0,168	0,664	[0.406 0.955]
4	35206	3963	0,113	0,633	[0.395 0.933]	5395	0,153	0,657	[0.406 0.948]
5	19697	1545	0,078	0,624	[0.396 0.91]	2549	0,129	0,647	[0.399 0.937]
6	9355	579	0,062	0,623	[0.39 0.902]	1037	0,111	0,639	[0.389 0.93]
7	3800	159	0,042	0,607	[0.387 0.889]	369	0,097	0,647	[0.38 0.941]
8	1331	53	0,04	0,629	[0.295 0.918]	110	0,083	0,633	[0.405 0.869]
		B_N	B_R	B_α	B_{CI}	C_N	C_R	C_α	C_{CI}
		27219	0,349	0,735	[0.455 0.978]	17156	0,22	1	[1. 1.]
		31821	0,462	0,711	[0.442 0.972]	10397	0,151	0,878	[0.549 1.]
		29478	0,555	0,705	[0.44 0.968]	6492	0,122	0,823	[0.521 1.]
		22027	0,626	0,702	[0.438 0.965]	3821	0,109	0,792	[0.485 1.]
		13540	0,687	0,702	[0.444 0.965]	2063	0,105	0,772	[0.49 1.]
		6795	0,726	0,704	[0.442 0.96]	944	0,101	0,75	[0.475 0.987]
		2844	0,748	0,705	[0.439 0.963]	428	0,113	0,732	[0.449 0.965]
		1044	0,784	0,701	[0.434 0.962]	124	0,093	0,75	[0.453 0.97]

Table E-4: Switch failure protection - ER-networks - minimized cost - $M = 2$

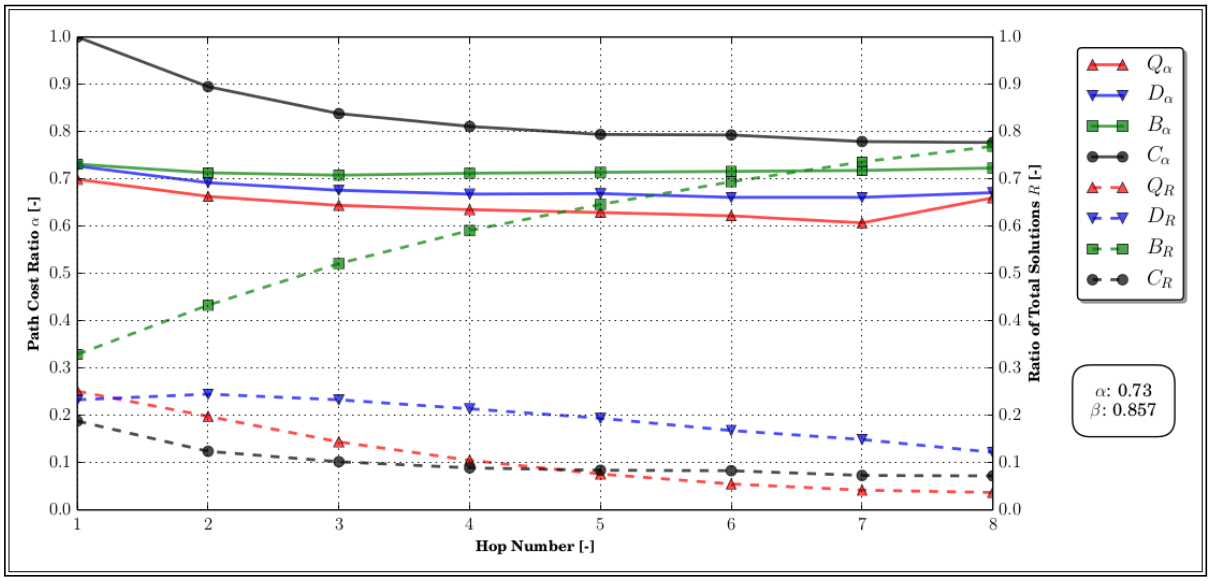


Figure E-4: Switch failure protection - ER-networks - minimized cost - $M = 6$

Hop	*Total*	Q_N	Q_R	Q_α	Q_{CI}	D_N	D_R	D_α	D_{CI}
1	94440	23681	0,251	0,699	[0.433 0.964]	22023	0,233	0,727	[0.458 0.972]
2	80666	15993	0,198	0,663	[0.415 0.95]	19756	0,245	0,692	[0.44 0.964]
3	59656	8586	0,144	0,644	[0.412 0.932]	13910	0,233	0,676	[0.429 0.953]
4	37374	3928	0,105	0,635	[0.41 0.923]	8011	0,214	0,668	[0.424 0.946]
5	19555	1493	0,076	0,629	[0.395 0.924]	3790	0,194	0,669	[0.423 0.946]
6	8784	481	0,055	0,622	[0.402 0.882]	1476	0,168	0,661	[0.418 0.919]
7	3218	134	0,042	0,607	[0.405 0.843]	479	0,149	0,661	[0.407 0.94]
8	1030	38	0,037	0,66	[0.441 0.898]	126	0,122	0,671	[0.409 0.938]
		B_N	B_R	B_α	B_{CI}	C_N	C_R	C_α	C_{CI}
		31028	0,329	0,731	[0.459 0.974]	17708	0,188	1	[1. 1.]
		34902	0,433	0,713	[0.448 0.97]	10015	0,124	0,895	[0.581 1.]
		31088	0,521	0,708	[0.441 0.968]	6072	0,102	0,838	[0.544 1.]
		22099	0,591	0,712	[0.449 0.967]	3336	0,089	0,811	[0.53 1.]
		12636	0,646	0,714	[0.447 0.967]	1636	0,084	0,794	[0.518 1.]
		6096	0,694	0,716	[0.453 0.966]	731	0,083	0,793	[0.519 0.988]
		2370	0,736	0,718	[0.444 0.967]	235	0,073	0,779	[0.514 0.981]
		792	0,769	0,723	[0.466 0.965]	74	0,072	0,777	[0.486 0.983]

Table E-5: Switch failure protection - ER-networks - minimized cost - $M = 6$

E-3 BA networks link failure protection with minimized cost setting

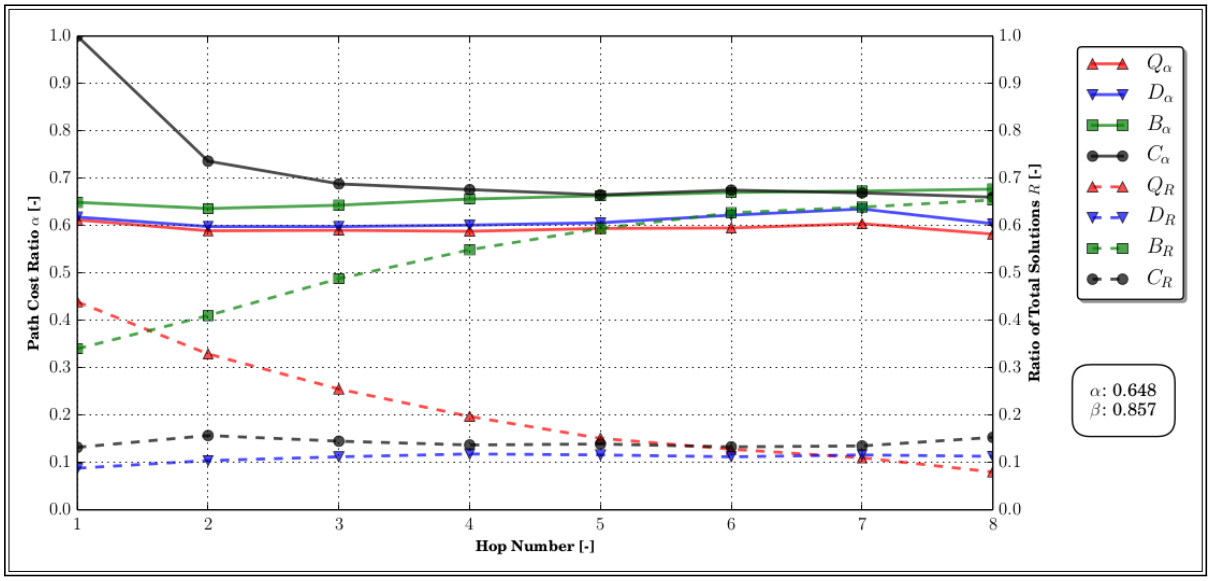


Figure E-5: Link failure protection - BA-networks - minimized cost - $M = 2$

Hop	*Total*	Q_N	Q_R	Q_α	Q_{CI}	D_N	D_R	D_α	D_{CI}
1	96387	42322	0,439	0,612	[0.336 0.94]	8518	0,088	0,618	[0.351 0.947]
2	82921	27268	0,329	0,589	[0.342 0.929]	8640	0,104	0,598	[0.358 0.93]
3	59456	15138	0,255	0,59	[0.35 0.927]	6655	0,112	0,598	[0.367 0.932]
4	34658	6836	0,197	0,588	[0.36 0.91]	4073	0,118	0,601	[0.374 0.917]
5	16609	2513	0,151	0,594	[0.366 0.908]	1934	0,116	0,606	[0.378 0.919]
6	6488	831	0,128	0,595	[0.366 0.904]	728	0,112	0,622	[0.366 0.928]
7	2175	239	0,11	0,604	[0.37 0.888]	253	0,116	0,635	[0.384 0.909]
8	627	50	0,08	0,582	[0.379 0.884]	71	0,113	0,604	[0.391 0.909]
		B_N	B_R	B_α	B_{CI}	C_N	C_R	C_α	C_{CI}
		32811	0,34	0,649	[0.391 0.953]	12736	0,132	1	[1. 1.]
		33987	0,41	0,636	[0.393 0.945]	13026	0,157	0,736	[0.422 1.]
		29034	0,488	0,643	[0.4 0.943]	8629	0,145	0,688	[0.399 1.]
		19011	0,549	0,656	[0.408 0.948]	4738	0,137	0,676	[0.402 0.989]
		9861	0,594	0,663	[0.415 0.952]	2301	0,139	0,665	[0.415 0.968]
		4068	0,627	0,67	[0.416 0.946]	861	0,133	0,675	[0.407 0.969]
		1390	0,639	0,673	[0.438 0.936]	293	0,135	0,669	[0.406 0.946]
		410	0,654	0,677	[0.447 0.926]	96	0,153	0,66	[0.407 0.956]

Table E-6: Link failure protection - BA-networks - minimized cost - $M = 2$

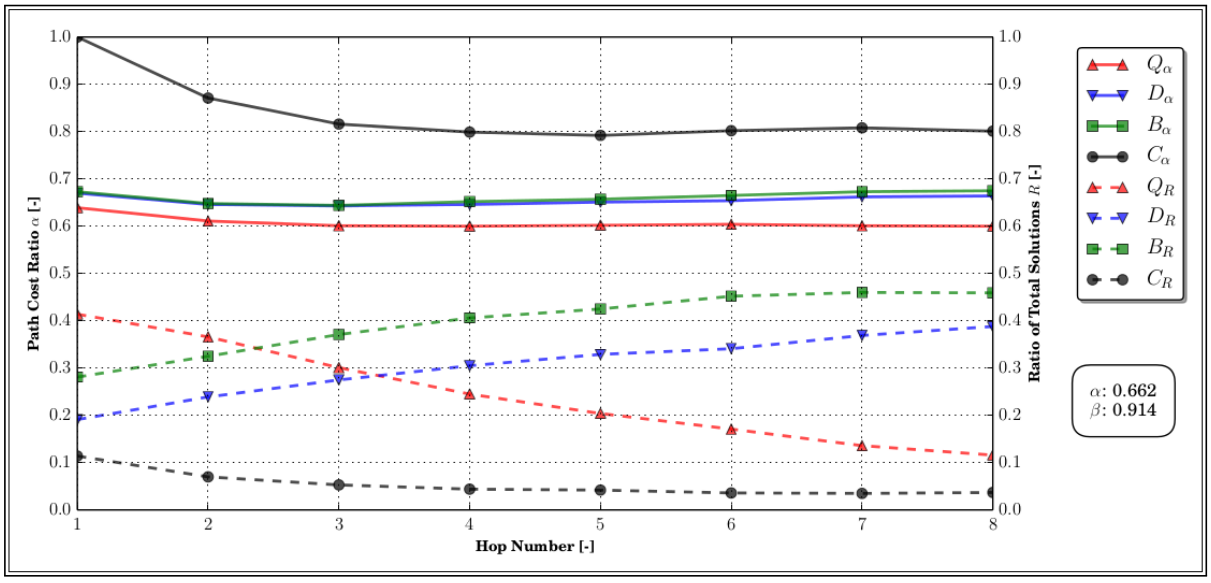


Figure E-6: Link failure protection - BA-networks - minimized cost - $M = 6$

Hop	*Total*	Q_N	Q_R	Q_α	Q_{CI}	D_N	D_R	D_α	D_{CI}
1	94993	39324	0,414	0,639	[0.367 0.95]	18099	0,191	0,67	[0.398 0.96]
2	81033	29638	0,366	0,611	[0.365 0.931]	19385	0,239	0,646	[0.393 0.951]
3	58848	17690	0,301	0,601	[0.367 0.921]	16175	0,275	0,643	[0.397 0.945]
4	35265	8642	0,245	0,6	[0.37 0.918]	10763	0,305	0,646	[0.403 0.941]
5	17531	3577	0,204	0,602	[0.376 0.918]	5768	0,329	0,651	[0.404 0.939]
6	7240	1241	0,171	0,604	[0.371 0.897]	2469	0,341	0,654	[0.409 0.934]
7	2550	346	0,136	0,601	[0.385 0.873]	942	0,369	0,662	[0.425 0.93]
8	748	87	0,116	0,6	[0.372 0.912]	290	0,388	0,664	[0.439 0.917]
		B_N	B_R	B_α	B_{CI}	C_N	C_R	C_α	C_{CI}
		26714	0,281	0,673	[0.403 0.962]	10856	0,114	1	[1. 1.]
		26298	0,325	0,648	[0.398 0.945]	5712	0,07	0,871	[0.556 1.]
		21841	0,371	0,644	[0.397 0.942]	3142	0,053	0,816	[0.526 1.]
		14317	0,406	0,652	[0.407 0.946]	1543	0,044	0,799	[0.517 1.]
		7452	0,425	0,657	[0.413 0.94]	734	0,042	0,792	[0.51 0.989]
		3269	0,452	0,665	[0.417 0.938]	261	0,036	0,802	[0.56 0.989]
		1173	0,46	0,673	[0.427 0.927]	89	0,035	0,808	[0.552 0.991]
		343	0,459	0,675	[0.419 0.908]	28	0,037	0,801	[0.457 0.942]

Table E-7: Link failure protection - BA-networks - minimized cost - $M = 6$

E-4 ER networks link failure protection with minimized cost setting

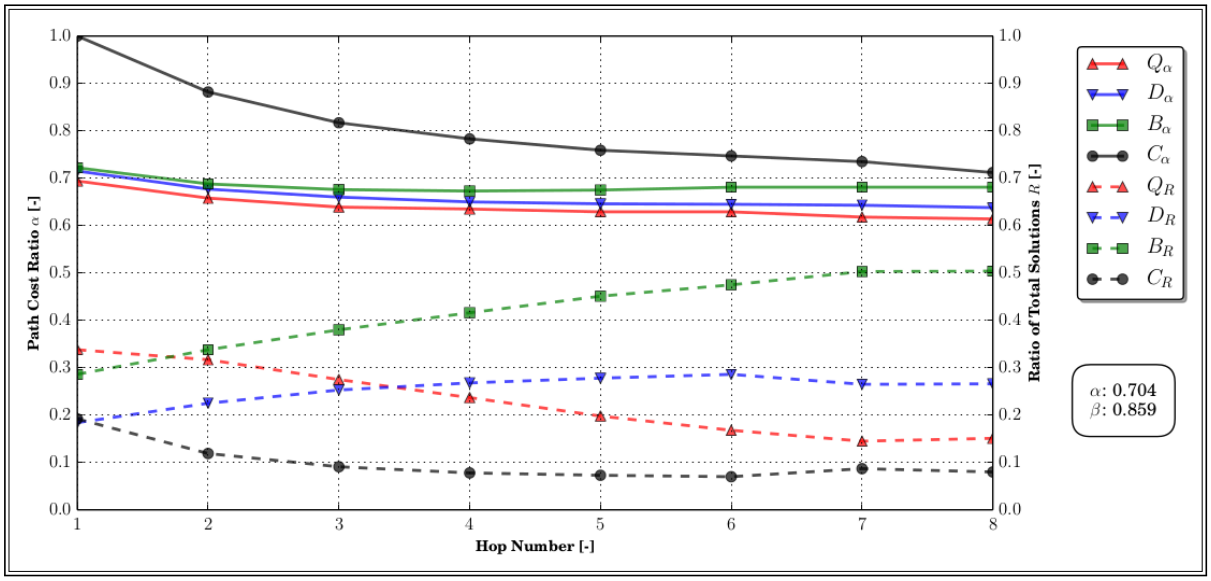


Figure E-7: Link failure protection - ER-networks - minimized cost - $M = 2$

Hop	*Total*	Q_N	Q_R	Q_α	Q_{CI}	D_N	D_R	D_α	D_{CI}
1	78181	26429	0,338	0,694	[0.415 0.968]	14364	0,184	0,715	[0.435 0.973]
2	68984	21897	0,317	0,658	[0.407 0.953]	15553	0,225	0,677	[0.421 0.962]
3	53518	14742	0,275	0,639	[0.394 0.94]	13533	0,253	0,66	[0.41 0.949]
4	35422	8403	0,237	0,635	[0.393 0.932]	9507	0,268	0,65	[0.403 0.942]
5	20005	3952	0,198	0,629	[0.402 0.918]	5560	0,278	0,646	[0.407 0.936]
6	9499	1596	0,168	0,629	[0.401 0.916]	2720	0,286	0,645	[0.393 0.939]
7	3988	579	0,145	0,618	[0.4 0.896]	1058	0,265	0,643	[0.38 0.944]
8	1435	216	0,151	0,614	[0.392 0.901]	381	0,266	0,638	[0.422 0.9]
		B_N	B_R	B_α	B_{CI}	C_N	C_R	C_α	C_{CI}
		22377	0,286	0,722	[0.451 0.974]	15011	0,192	1	[1. 1.]
		23327	0,338	0,688	[0.43 0.964]	8207	0,119	0,882	[0.558 1.]
		20360	0,38	0,676	[0.421 0.954]	4883	0,091	0,817	[0.519 1.]
		14747	0,416	0,673	[0.417 0.952]	2765	0,078	0,783	[0.494 1.]
		9028	0,451	0,675	[0.414 0.951]	1465	0,073	0,759	[0.473 0.994]
		4515	0,475	0,681	[0.428 0.958]	668	0,07	0,747	[0.478 0.981]
		2005	0,503	0,681	[0.43 0.946]	346	0,087	0,735	[0.476 0.979]
		723	0,504	0,681	[0.435 0.942]	115	0,08	0,712	[0.409 0.955]

Table E-8: Link failure protection - ER-networks - minimized cost - $M = 2$

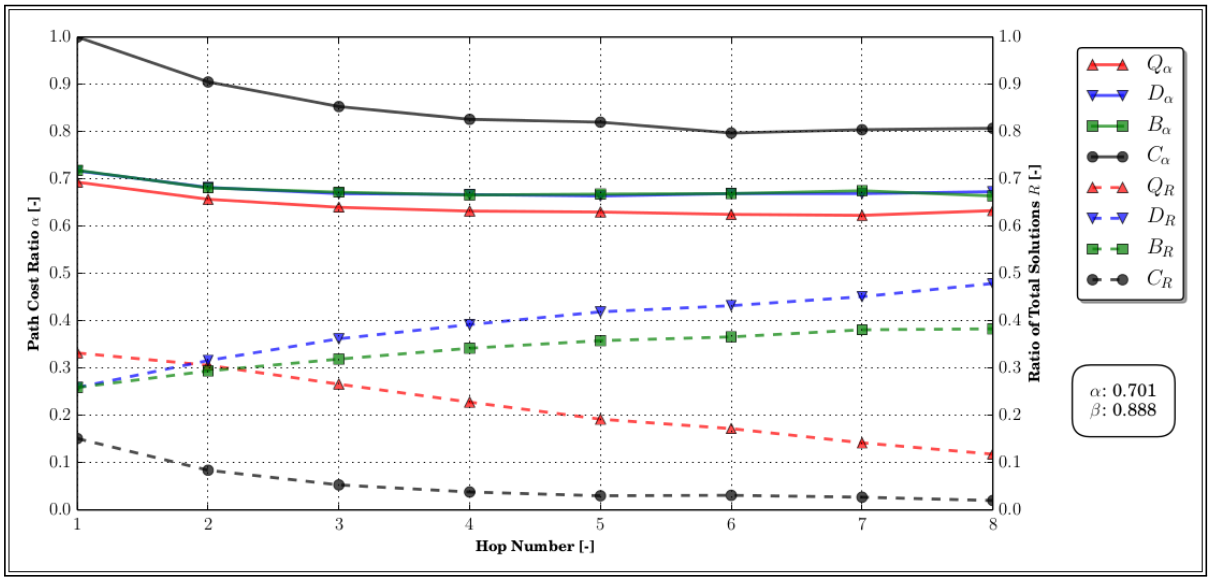


Figure E-8: Link failure protection - ER-networks - minimized cost - $M = 6$

Hop	*Total*	Q_N	Q_R	Q_α	Q_{CI}	D_N	D_R	D_α	D_{CI}
1	94393	31323	0,332	0,693	[0.421 0.966]	24420	0,259	0,717	[0.448 0.968]
2	80524	24667	0,306	0,657	[0.41 0.947]	25472	0,316	0,682	[0.43 0.957]
3	59606	15872	0,266	0,64	[0.403 0.935]	21570	0,362	0,669	[0.425 0.946]
4	37261	8493	0,228	0,632	[0.405 0.924]	14614	0,392	0,667	[0.425 0.944]
5	19773	3806	0,192	0,63	[0.402 0.911]	8292	0,419	0,664	[0.425 0.935]
6	8779	1507	0,172	0,625	[0.398 0.905]	3790	0,432	0,669	[0.421 0.937]
7	3376	479	0,142	0,623	[0.379 0.899]	1521	0,451	0,669	[0.428 0.912]
8	1042	123	0,118	0,633	[0.391 0.897]	499	0,479	0,673	[0.449 0.928]
		B_N	B_R	B_α	B_{CI}	C_N	C_R	C_α	C_{CI}
		24413	0,259	0,718	[0.446 0.969]	14237	0,151	1	[1. 1.]
		23640	0,294	0,681	[0.426 0.957]	6745	0,084	0,905	[0.597 1.]
		19001	0,319	0,672	[0.423 0.951]	3163	0,053	0,853	[0.547 1.]
		12737	0,342	0,666	[0.421 0.941]	1417	0,038	0,826	[0.539 1.]
		7080	0,358	0,668	[0.427 0.94]	595	0,03	0,82	[0.557 1.]
		3214	0,366	0,669	[0.426 0.934]	268	0,031	0,797	[0.545 0.981]
		1286	0,381	0,675	[0.424 0.936]	90	0,027	0,804	[0.497 0.983]
		399	0,383	0,664	[0.426 0.904]	21	0,02	0,807	[0.593 0.965]

Table E-9: Link failure protection - ER-networks - minimized cost - $M = 6$

E-5 BA networks switch failure protection with minimized crankback setting

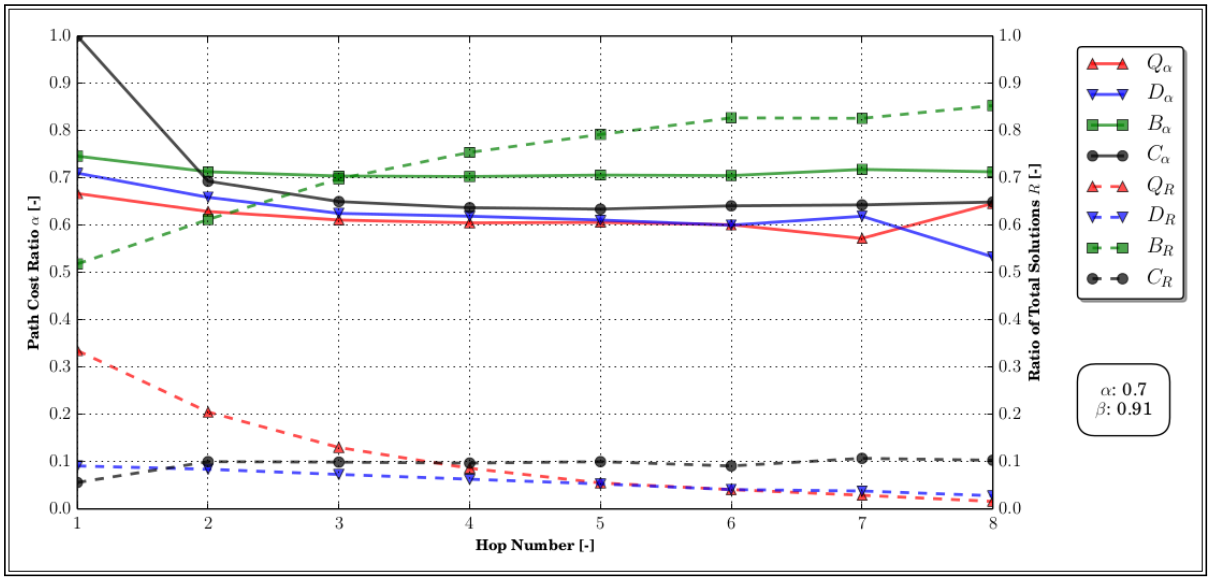


Figure E-9: Switch failure protection - BA-networks - minimized crankback - $M = 2$

Hop	*Total*	Q_N	Q_R	Q_α	Q_{CI}	D_N	D_R	D_α	D_{CI}
1	96395	32287	0,335	0,667	[0.355 1.228]	8800	0,091	0,71	[0.369 1.409]
2	82996	16988	0,205	0,629	[0.351 1.173]	6969	0,084	0,659	[0.367 1.285]
3	59236	7712	0,13	0,611	[0.363 1.095]	4312	0,073	0,625	[0.365 1.144]
4	34593	2963	0,086	0,605	[0.357 1.102]	2189	0,063	0,619	[0.365 1.129]
5	16284	899	0,055	0,606	[0.353 1.027]	865	0,053	0,611	[0.355 1.061]
6	6460	265	0,041	0,601	[0.354 1.061]	265	0,041	0,6	[0.354 1.035]
7	2097	61	0,029	0,572	[0.414 0.836]	79	0,038	0,619	[0.411 1.]
8	563	9	0,016	0,646	[0.373 1.145]	16	0,028	0,533	[0.353 0.85]
		B_N	B_R	B_α	B_{CI}	C_N	C_R	C_α	C_{CI}
		49917	0,518	0,746	[0.407 1.446]	5391	0,056	1	[1. 1.]
		50766	0,612	0,713	[0.408 1.325]	8273	0,1	0,693	[0.404 1.13]
		41376	0,698	0,704	[0.411 1.228]	5836	0,099	0,65	[0.388 1.057]
		26092	0,754	0,703	[0.42 1.173]	3349	0,097	0,637	[0.391 1.015]
		12890	0,792	0,706	[0.425 1.153]	1630	0,1	0,634	[0.384 1.033]
		5340	0,827	0,705	[0.429 1.116]	590	0,091	0,641	[0.408 0.948]
		1732	0,826	0,718	[0.444 1.11]	225	0,107	0,643	[0.398 0.951]
		480	0,853	0,713	[0.452 1.055]	58	0,103	0,649	[0.337 1.04]

Table E-10: Switch failure protection - BA-networks - minimized crankback - $M = 2$

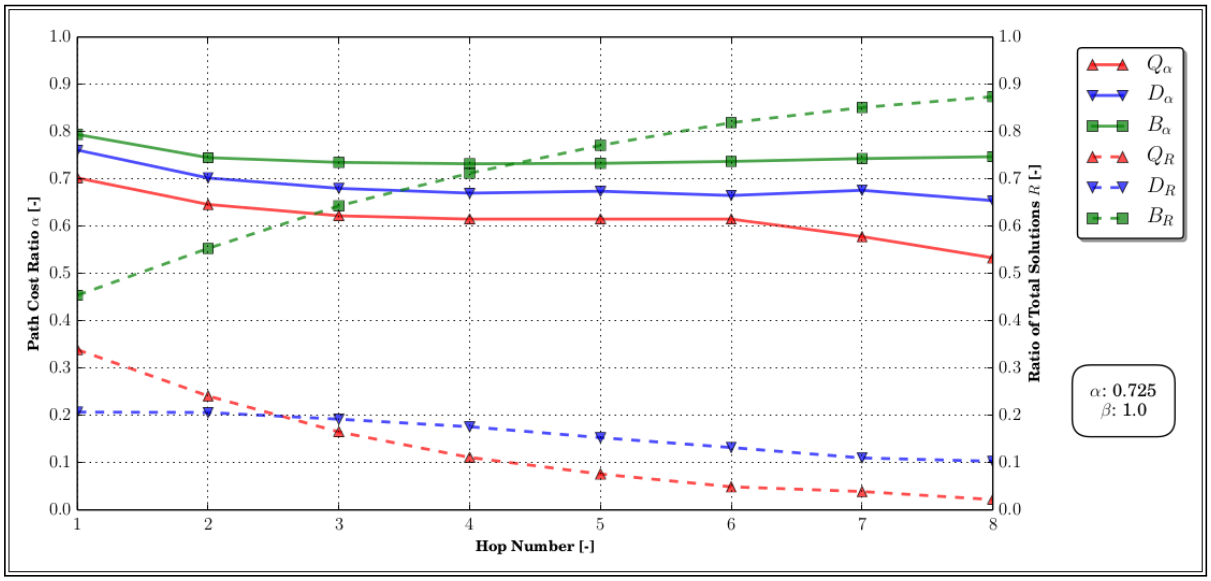


Figure E-10: Switch failure protection - BA-networks - minimized crankback - $M = 6$

Hop	*Total*	Q_N	Q_R	Q_α	Q_{CI}	D_N	D_R	D_α	D_{CI}
1	95023	32227	0,339	0,702	[0.384 1.267]	19684	0,207	0,761	[0.416 1.39]
2	80991	19500	0,241	0,646	[0.374 1.134]	16677	0,206	0,702	[0.404 1.245]
3	58751	9709	0,165	0,622	[0.376 1.059]	11291	0,192	0,68	[0.398 1.154]
4	35554	3957	0,111	0,615	[0.374 1.031]	6272	0,176	0,67	[0.396 1.112]
5	17769	1349	0,076	0,615	[0.375 1.02]	2725	0,153	0,674	[0.405 1.089]
6	7366	363	0,049	0,615	[0.376 1.061]	969	0,132	0,665	[0.397 1.085]
7	2512	97	0,039	0,578	[0.316 0.971]	277	0,11	0,676	[0.411 1.108]
8	716	16	0,022	0,533	[0.341 0.733]	74	0,103	0,654	[0.382 0.975]
		B_N	B_R	B_α	B_{CI}	C_N	C_R	C_α	C_{CI}
		43111	0,454	0,794	[0.423 1.542]	1	0	1	[1. 1.]
		44810	0,553	0,745	[0.419 1.348]	4	0	0,654	[0.494 0.791]
		37750	0,643	0,735	[0.419 1.286]	1	0	0,741	[0.741 0.741]
		25322	0,712	0,732	[0.426 1.229]	3	0	0,619	[0.426 0.76]
		13695	0,771	0,733	[0.433 1.192]	0	0	0	['-', '-']
		6034	0,819	0,737	[0.441 1.182]	0	0	0	['-', '-']
		2138	0,851	0,743	[0.454 1.202]	0	0	0	['-', '-']
		626	0,874	0,747	[0.452 1.129]	0	0	0	['-', '-']

Table E-11: Switch failure protection - BA-networks - minimized crankback - $M = 6$

E-6 ER networks switch failure protection with minimized crankback setting

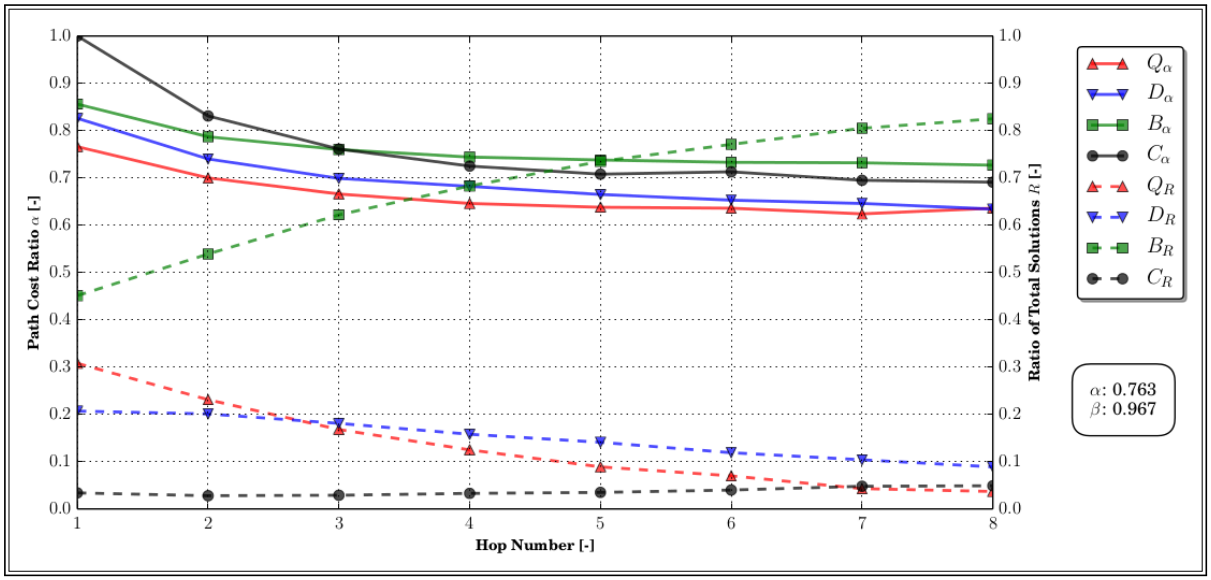


Figure E-11: Switch failure protection - ER-networks - minimized crankback - $M = 2$

Hop	*Total*	Q_N	Q_R	Q_α	Q_{CI}	D_N	D_R	D_α	D_{CI}
1	78161	24074	0,308	0,766	[0.431 1.321]	16199	0,207	0,826	[0.464 1.482]
2	68920	15923	0,231	0,7	[0.412 1.176]	13863	0,201	0,74	[0.43 1.288]
3	53485	9007	0,168	0,666	[0.403 1.106]	9663	0,181	0,699	[0.418 1.146]
4	35336	4426	0,125	0,646	[0.397 1.015]	5588	0,158	0,682	[0.409 1.092]
5	20018	1784	0,089	0,638	[0.391 1.007]	2826	0,141	0,665	[0.391 1.08]
6	9716	677	0,07	0,636	[0.385 0.982]	1154	0,119	0,653	[0.396 1.017]
7	3890	169	0,043	0,624	[0.386 1.031]	406	0,104	0,646	[0.387 0.927]
8	1451	54	0,037	0,636	[0.371 0.893]	129	0,089	0,634	[0.413 0.937]
		B_N	B_R	B_α	B_{CI}	C_N	C_R	C_α	C_{CI}
		35261	0,451	0,856	[0.467 1.587]	2627	0,034	1	[1. 1.]
		37178	0,539	0,787	[0.449 1.38]	1956	0,028	0,831	[0.513 1.28]
		33290	0,622	0,76	[0.444 1.269]	1525	0,029	0,761	[0.474 1.16]
		24139	0,683	0,744	[0.444 1.188]	1183	0,033	0,725	[0.453 1.089]
		14707	0,735	0,738	[0.445 1.155]	701	0,035	0,708	[0.449 1.018]
		7494	0,771	0,733	[0.452 1.12]	391	0,04	0,713	[0.445 1.039]
		3130	0,805	0,732	[0.45 1.114]	185	0,048	0,695	[0.454 0.964]
		1197	0,825	0,727	[0.441 1.081]	71	0,049	0,691	[0.456 0.985]

Table E-12: Switch failure protection - ER-networks - minimized crankback - $M = 2$

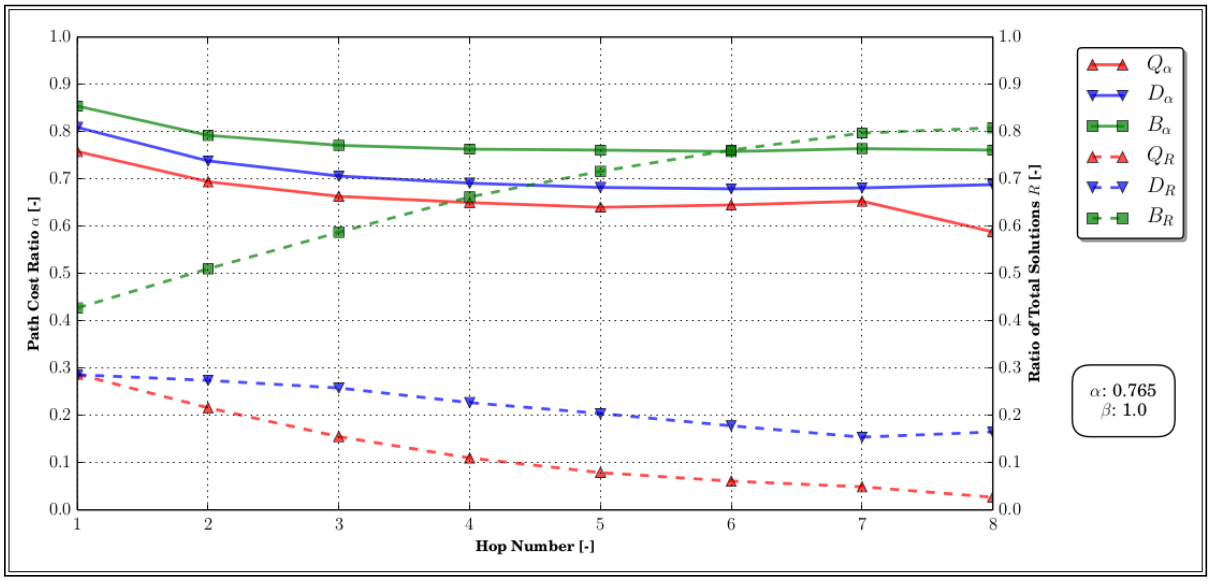


Figure E-12: Switch failure protection - ER-networks - minimized crankback - $M = 6$

Hop	*Total*	Q_N	Q_R	Q_α	Q_{CI}	D_N	D_R	D_α	D_{CI}
1	94389	27089	0,287	0,758	[0.438 1.288]	26997	0,286	0,809	[0.467 1.378]
2	80691	17467	0,216	0,694	[0.426 1.132]	22100	0,274	0,738	[0.444 1.227]
3	59608	9255	0,155	0,663	[0.411 1.067]	15367	0,258	0,706	[0.433 1.12]
4	37478	4132	0,11	0,65	[0.409 1.019]	8524	0,227	0,691	[0.426 1.071]
5	19773	1570	0,079	0,64	[0.396 0.978]	4040	0,204	0,682	[0.422 1.058]
6	8807	533	0,061	0,645	[0.405 0.971]	1571	0,178	0,679	[0.416 1.043]
7	3311	161	0,049	0,653	[0.417 0.984]	511	0,154	0,681	[0.428 1.031]
8	1100	30	0,027	0,588	[0.313 0.866]	181	0,165	0,688	[0.43 1.006]
		B_N	B_R	B_α	B_{CI}	C_N	C_R	C_α	C_{CI}
		40303	0,427	0,854	[0.472 1.58]	0	0	0	['-', '-']
		41123	0,51	0,792	[0.454 1.377]	1	0	0,634	[0.634 0.634]
		34986	0,587	0,771	[0.449 1.286]	0	0	0	['-', '-']
		24822	0,662	0,763	[0.453 1.23]	0	0	0	['-', '-']
		14163	0,716	0,761	[0.456 1.193]	0	0	0	['-', '-']
		6703	0,761	0,758	[0.463 1.159]	0	0	0	['-', '-']
		2639	0,797	0,764	[0.473 1.165]	0	0	0	['-', '-']
		889	0,808	0,761	[0.465 1.127]	0	0	0	['-', '-']

Table E-13: Switch failure protection - ER-networks - minimized crankback - $M = 6$

E-7 BA networks link failure protection with minimized crankback setting

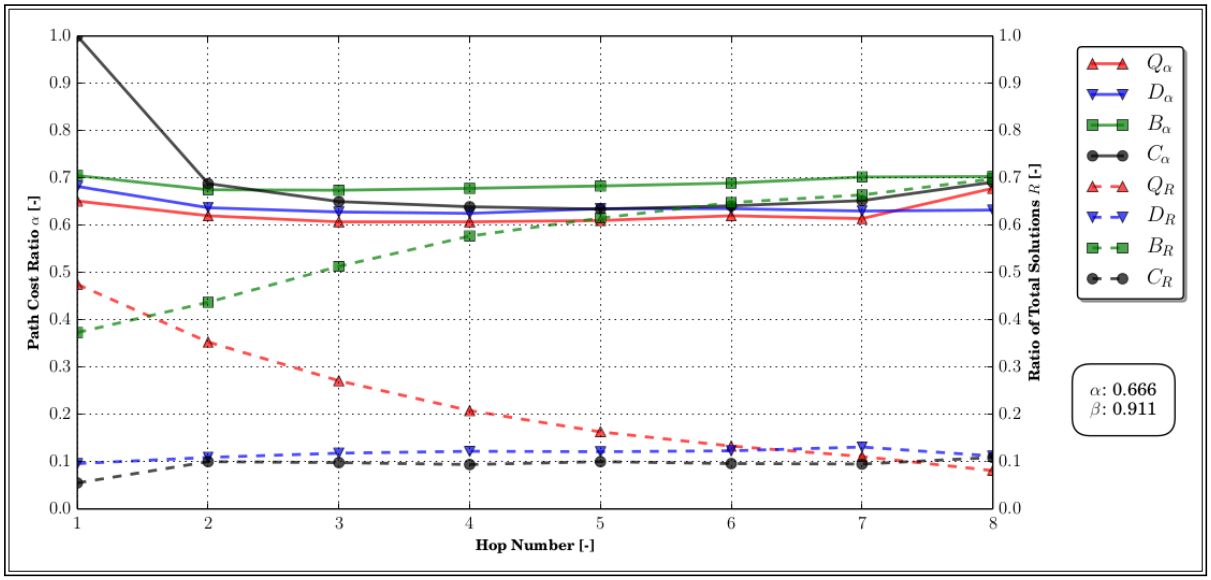


Figure E-13: Link failure protection - BA-networks - minimized crankback - $M = 2$

Hop	*Total*	Q_N	Q_R	Q_α	Q_{CI}	D_N	D_R	D_α	D_{CI}
1	96416	45839	0,475	0,651	[0.34 1.206]	9278	0,096	0,682	[0.363 1.344]
2	82803	29213	0,353	0,62	[0.344 1.143]	9065	0,109	0,637	[0.362 1.182]
3	59235	16054	0,271	0,607	[0.351 1.08]	6965	0,118	0,628	[0.368 1.121]
4	34459	7151	0,208	0,607	[0.362 1.045]	4187	0,122	0,625	[0.374 1.084]
5	16239	2655	0,163	0,61	[0.361 1.025]	1964	0,121	0,635	[0.384 1.051]
6	6420	856	0,133	0,62	[0.371 1.009]	789	0,123	0,635	[0.378 1.029]
7	2129	236	0,111	0,614	[0.392 0.972]	278	0,131	0,63	[0.395 0.948]
8	606	49	0,081	0,678	[0.365 1.11]	68	0,112	0,632	[0.404 1.]
		B_N	B_R	B_α	B_{CI}	C_N	C_R	C_α	C_{CI}
		35964	0,373	0,705	[0.396 1.294]	5335	0,055	1	[1. 1.]
		36212	0,437	0,675	[0.397 1.188]	8313	0,1	0,688	[0.396 1.112]
		30401	0,513	0,674	[0.399 1.143]	5815	0,098	0,65	[0.389 1.045]
		19886	0,577	0,678	[0.41 1.111]	3235	0,094	0,639	[0.397 1.014]
		9989	0,615	0,683	[0.419 1.079]	1631	0,1	0,634	[0.402 0.974]
		4160	0,648	0,689	[0.425 1.062]	615	0,096	0,641	[0.401 0.981]
		1413	0,664	0,702	[0.443 1.07]	202	0,095	0,652	[0.381 1.014]
		423	0,698	0,703	[0.412 1.054]	66	0,109	0,691	[0.491 0.949]

Table E-14: Link failure protection - BA-networks - minimized crankback - $M = 2$

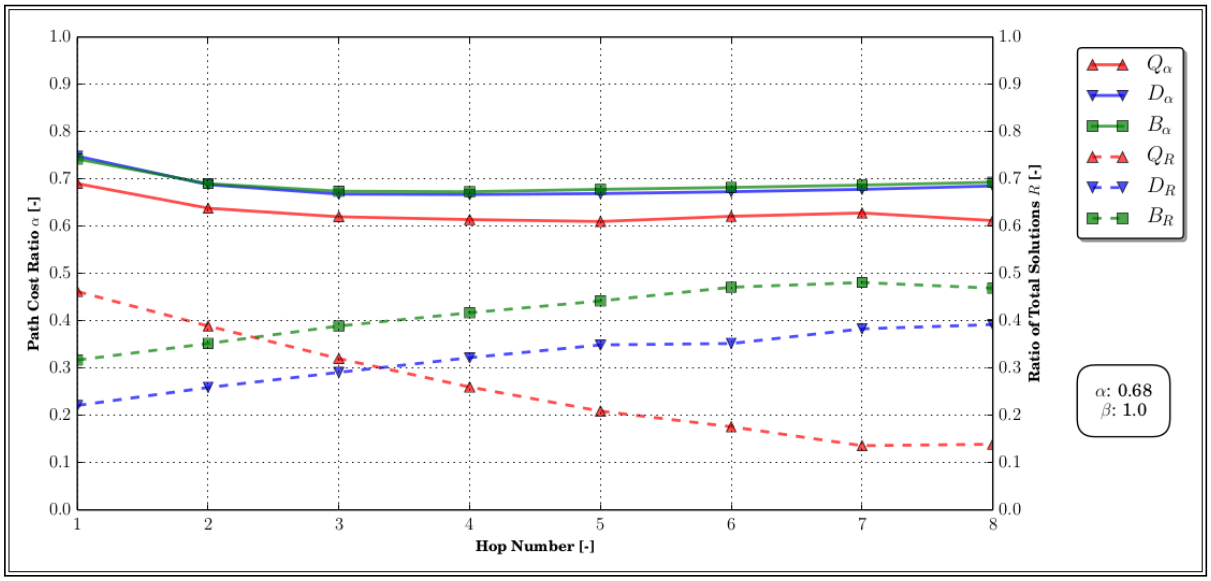


Figure E-14: Link failure protection - BA-networks - minimized crankback - $M = 6$

Hop	*Total*	Q_N	Q_R	$Q_{C_{avg}}$	P_{CI}	D_N	D_R	$D_{C_{avg}}$	D_{CI}
1	95002	43910	0,462	0,69	[0.371 1.25]	20966	0,221	0,748	[0.404 1.403]
2	81002	31505	0,389	0,638	[0.366 1.121]	20995	0,259	0,688	[0.396 1.205]
3	58904	18832	0,32	0,62	[0.369 1.064]	17151	0,291	0,668	[0.395 1.117]
4	35658	9282	0,26	0,614	[0.376 1.]	11495	0,322	0,667	[0.398 1.078]
5	17721	3708	0,209	0,61	[0.38 0.986]	6177	0,349	0,669	[0.403 1.043]
6	7300	1286	0,176	0,621	[0.384 0.986]	2573	0,352	0,673	[0.41 1.051]
7	2544	346	0,136	0,628	[0.385 1.]	975	0,383	0,678	[0.424 1.047]
8	734	102	0,139	0,612	[0.385 0.842]	288	0,392	0,685	[0.424 1.017]
		B_N	B_R	$B_{C_{avg}}$	B_{CI}	C_N	C_R	$C_{C_{avg}}$	C_{CI}
		30125	0,317	0,742	[0.408 1.358]	1	0	1	[1. 1.]
		28499	0,352	0,69	[0.4 1.212]	3	0	0,697	[0.644 0.8]
		22919	0,389	0,674	[0.398 1.123]	2	0	0,809	[0.635 0.984]
		14881	0,417	0,673	[0.405 1.096]	0	0	0	['-', '-']
		7835	0,442	0,678	[0.413 1.078]	1	0	0,532	[0.532 0.532]
		3440	0,471	0,682	[0.408 1.058]	1	0	0,702	[0.702 0.702]
		1223	0,481	0,687	[0.415 1.041]	0	0	0	['-', '-']
		344	0,469	0,693	[0.447 1.022]	0	0	0	['-', '-']

Table E-15: Link failure protection - BA-networks - minimized crankback - $M = 6$

E-8 ER networks link failure protection with minimized crankback setting

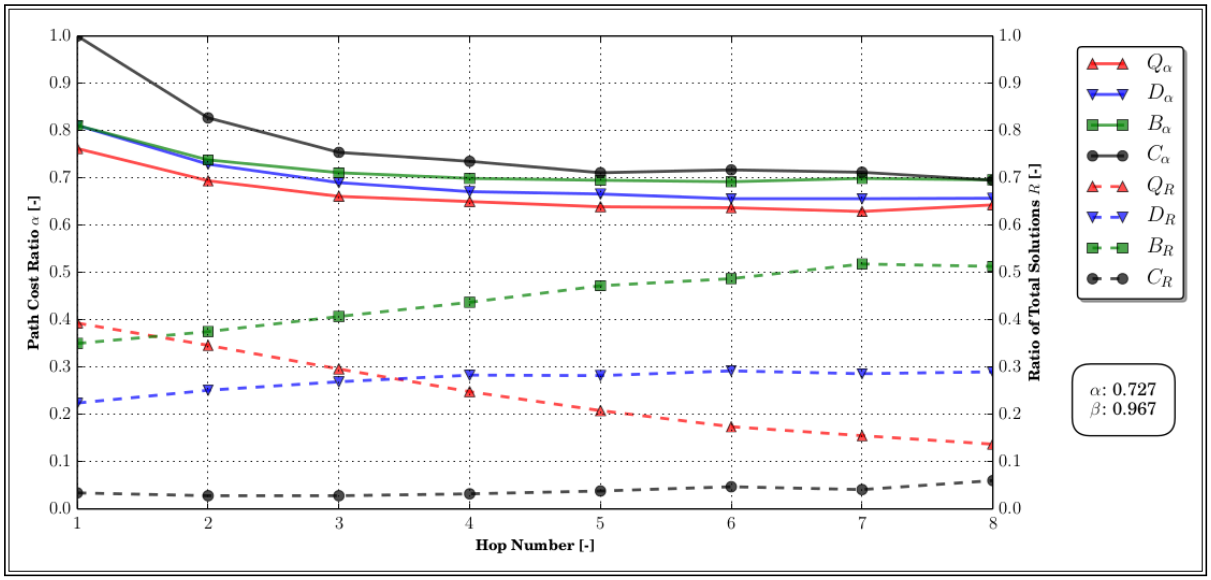


Figure E-15: Link failure protection - ER-networks - minimized crankback - $M = 2$

Hop	*Total*	Q_N	Q_R	Q_α	Q_{CI}	D_N	D_R	D_α	D_{CI}
1	77926	30598	0,393	0,762	[0.422 1.33]	17468	0,224	0,811	[0.441 1.452]
2	68676	23760	0,346	0,694	[0.407 1.165]	17242	0,251	0,729	[0.419 1.259]
3	52964	15682	0,296	0,661	[0.397 1.077]	14255	0,269	0,69	[0.412 1.135]
4	35082	8715	0,248	0,65	[0.399 1.033]	9914	0,283	0,671	[0.405 1.086]
5	19795	4119	0,208	0,639	[0.399 0.989]	5578	0,282	0,666	[0.404 1.044]
6	9458	1648	0,174	0,637	[0.392 0.993]	2764	0,292	0,656	[0.401 1.009]
7	3888	603	0,155	0,629	[0.387 0.951]	1113	0,286	0,656	[0.402 0.989]
8	1410	193	0,137	0,643	[0.392 0.991]	409	0,29	0,657	[0.403 1.019]
		B_N	B_R	B_α	B_{CI}	C_N	C_R	C_α	C_{CI}
		27237	0,35	0,811	[0.454 1.426]	2623	0,034	1	[1. 1.]
		25734	0,375	0,738	[0.43 1.237]	1940	0,028	0,827	[0.508 1.289]
		21554	0,407	0,711	[0.425 1.155]	1473	0,028	0,754	[0.478 1.151]
		15342	0,437	0,699	[0.425 1.104]	1111	0,032	0,735	[0.439 1.112]
		9352	0,472	0,695	[0.43 1.076]	746	0,038	0,711	[0.446 1.063]
		4605	0,487	0,692	[0.423 1.057]	441	0,047	0,717	[0.446 1.024]
		2014	0,518	0,699	[0.429 1.055]	158	0,041	0,712	[0.419 1.046]
		724	0,513	0,696	[0.437 1.033]	84	0,06	0,695	[0.393 1.003]

Table E-16: Link failure protection - ER-networks - minimized crankback - $M = 2$

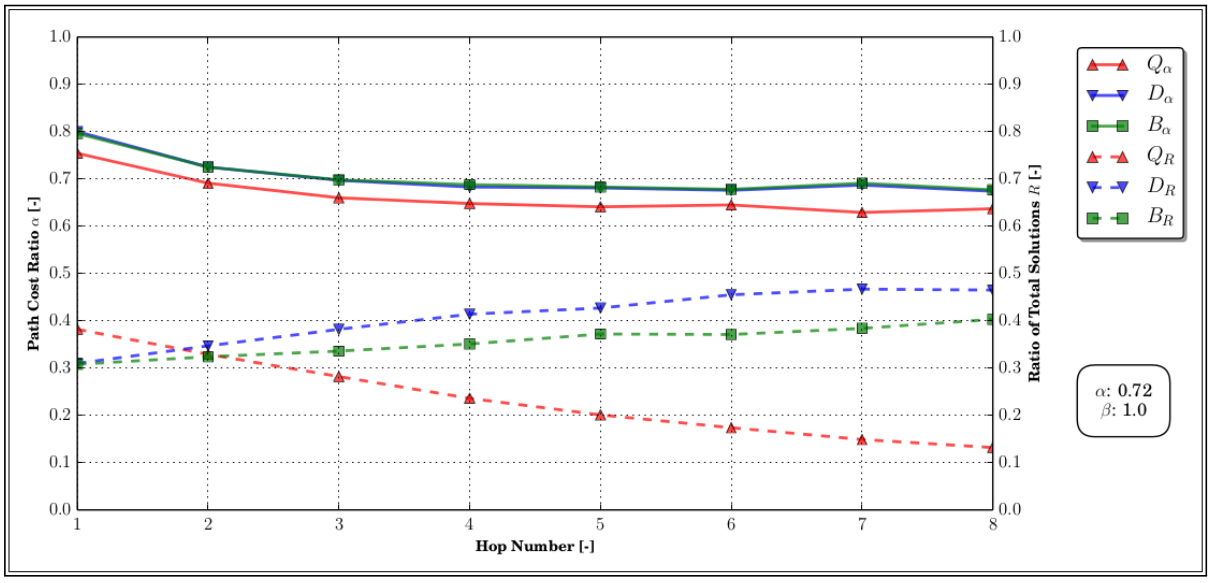


Figure E-16: Link failure protection - ER-networks - minimized crankback - $M = 6$

Hop	*Total*	Q_N	Q_R	Q_α	Q_{CI}	D_N	D_R	D_α	D_{CI}
1	94459	36084	0,382	0,754	[0.428 1.286]	29285	0,31	0,8	[0.458 1.382]
2	80680	26553	0,329	0,691	[0.415 1.139]	28014	0,347	0,725	[0.435 1.2]
3	59338	16754	0,282	0,66	[0.405 1.053]	22653	0,382	0,697	[0.426 1.111]
4	37161	8766	0,236	0,648	[0.406 1.011]	15367	0,414	0,683	[0.425 1.046]
5	19627	3939	0,201	0,641	[0.405 0.986]	8390	0,427	0,681	[0.428 1.024]
6	8746	1519	0,174	0,645	[0.406 0.964]	3978	0,455	0,676	[0.429 0.987]
7	3349	498	0,149	0,629	[0.416 0.907]	1565	0,467	0,687	[0.441 1.013]
8	1067	141	0,132	0,637	[0.439 0.882]	496	0,465	0,674	[0.429 0.944]
		B_N	B_R	B_α	B_{CI}	C_N	C_R	C_α	C_{CI}
		29090	0,308	0,796	[0.453 1.392]	0	0	0	['-', '-']
		26113	0,324	0,725	[0.433 1.202]	0	0	0	['-', '-']
		19931	0,336	0,698	[0.426 1.112]	0	0	0	['-', '-']
		13028	0,351	0,688	[0.426 1.064]	0	0	0	['-', '-']
		7297	0,372	0,683	[0.425 1.035]	1	0	0,807	[0.807 0.807]
		3249	0,371	0,678	[0.424 1.007]	0	0	0	['-', '-']
		1286	0,384	0,691	[0.436 1.03]	0	0	0	['-', '-']
		430	0,403	0,677	[0.439 0.975]	0	0	0	['-', '-']

Table E-17: Link failure protection - ER-networks - minimized crankback - $M = 6$

E-9 USNET switch failure protection with minimized cost setting

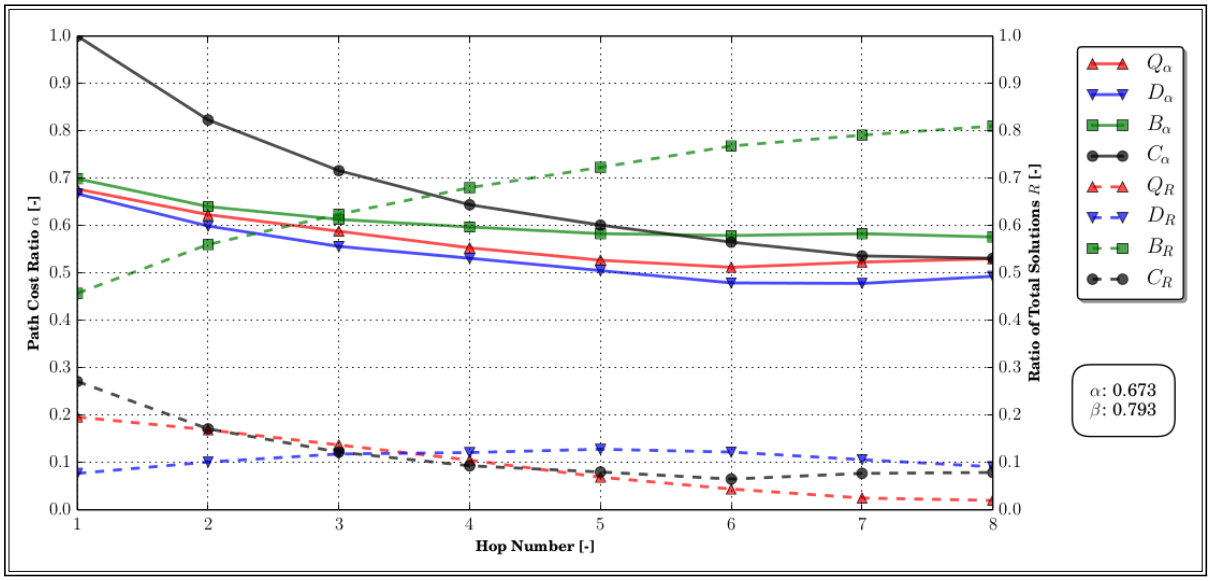


Figure E-17: Switch failure protection - USNET-network - minimized cost

Hop	*Total*	Q_N	Q_R	Q_α	Q_{CI}	D_N	D_R	D_α	D_{CI}
1	86939	17008	0.196	0.677	[0.395 0.965]	6652	0.077	0.667	[0.369 0.973]
2	68034	11467	0.169	0.623	[0.372 0.943]	6857	0.101	0.599	[0.341 0.942]
3	47972	6556	0.137	0.588	[0.366 0.912]	5649	0.118	0.556	[0.338 0.906]
4	30072	3163	0.105	0.553	[0.354 0.876]	3649	0.121	0.531	[0.328 0.894]
5	16156	1107	0.069	0.527	[0.35 0.83]	2076	0.128	0.505	[0.322 0.854]
6	7382	328	0.044	0.512	[0.337 0.811]	899	0.122	0.479	[0.325 0.793]
7	2680	68	0.025	0.523	[0.34 0.879]	285	0.106	0.478	[0.328 0.806]
8	769	15	0.02	0.53	[0.333 0.844]	70	0.091	0.493	[0.336 0.72]
		B_N	B_R	B_α	B_{CI}	C_N	C_R	C_α	C_{CI}
		39736	0.457	0.699	[0.416 0.971]	23543	0.271	1	[1. 1.]
		38075	0.56	0.64	[0.378 0.96]	11635	0.171	0.823	[0.47 1.]
		29921	0.624	0.613	[0.367 0.951]	5846	0.122	0.716	[0.405 1.]
		20455	0.68	0.597	[0.361 0.94]	2805	0.093	0.644	[0.378 1.]
		11676	0.723	0.583	[0.356 0.928]	1297	0.08	0.601	[0.368 0.949]
		5672	0.768	0.579	[0.356 0.919]	483	0.065	0.565	[0.344 0.939]
		2120	0.791	0.583	[0.36 0.914]	207	0.077	0.536	[0.35 0.829]
		623	0.81	0.576	[0.362 0.877]	61	0.079	0.531	[0.345 0.841]

Table E-18: Switch failure protection - USNET-network - minimized cost

E-10 Pan-EUR switch failure protection with minimized cost setting

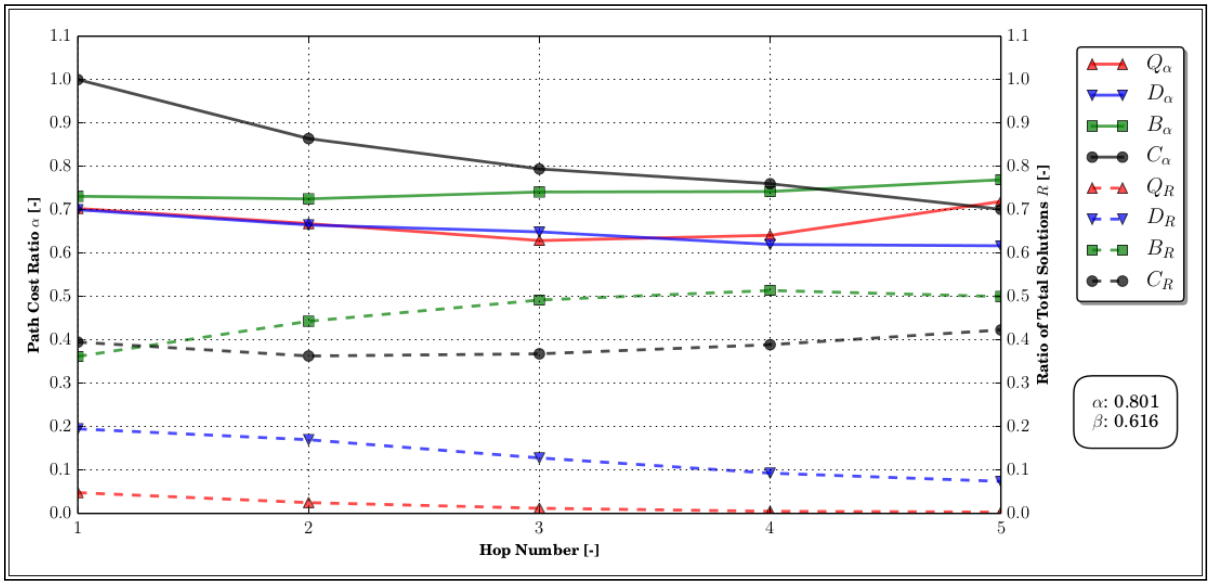


Figure E-18: Switch failure protection - Pan-EUR-network - minimized cost

Hop	*Total*	Q_N	Q_R	Q_α	Q_{CI}	D_N	D_R	D_α	D_{CI}
1	67453	3256	0.048	0.703	[0.404 0.972]	13144	0.195	0.7	[0.406 0.974]
2	31981	788	0.025	0.668	[0.388 0.963]	5435	0.17	0.665	[0.385 0.973]
3	10332	128	0.012	0.629	[0.411 0.915]	1322	0.128	0.649	[0.372 0.963]
4	2208	11	0.005	0.641	[0.406 0.923]	205	0.093	0.62	[0.364 0.912]
5	326	1	0.003	0.719	[0.719 0.719]	24	0.074	0.617	[0.341 0.963]
		B_N	B_R	B_α	B_{CI}	C_N	C_R	C_α	C_{CI}
		24396	0.362	0.731	[0.419 0.981]	26657	0.395	1	[1. 1.]
		14159	0.443	0.725	[0.41 0.981]	11599	0.363	0.864	[0.494 1.]
		5080	0.492	0.741	[0.423 0.982]	3802	0.368	0.794	[0.456 1.]
		1134	0.514	0.742	[0.433 0.977]	858	0.389	0.76	[0.426 1.]
		163	0.5	0.769	[0.393 0.986]	138	0.423	0.701	[0.386 0.962]

Table E-19: Switch failure protection - Pan-EUR-network - minimized cost

E-11 USNET link failure protection with minimized cost setting

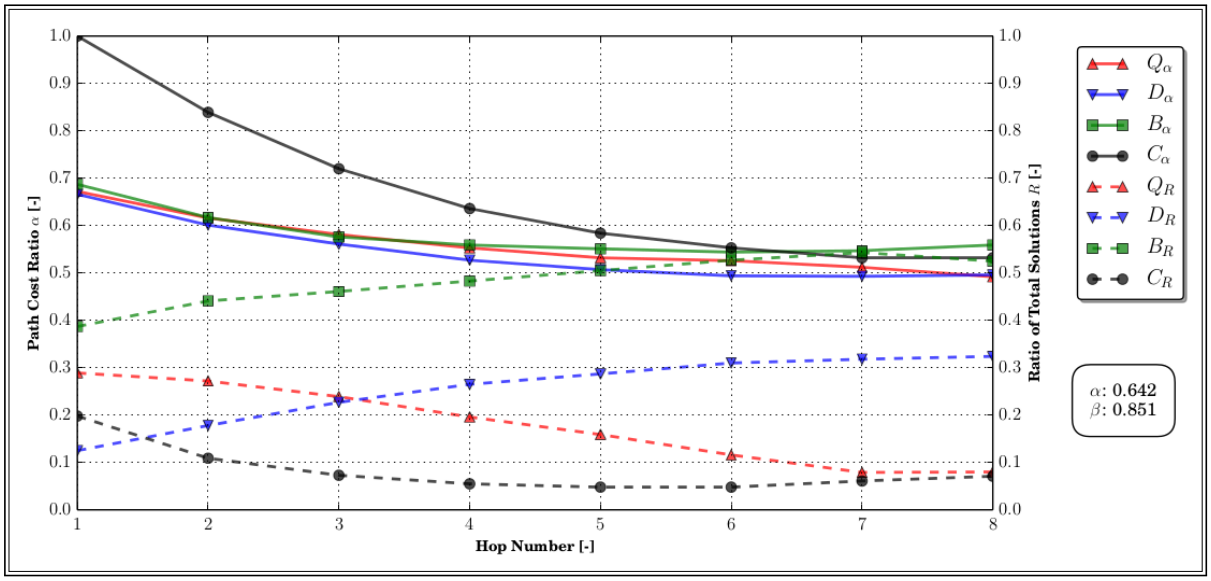


Figure E-19: Link failure protection - USNET-network - minimized cost

Hop	*Total*	Q_N	Q_R	Q_α	Q_{CI}	D_N	D_R	D_α	D_{CI}
1	86832	25125	0.289	0.672	[0.385 0.965]	10881	0.125	0.666	[0.369 0.964]
2	67798	18439	0.272	0.616	[0.36 0.938]	12073	0.178	0.601	[0.357 0.94]
3	47765	11435	0.239	0.581	[0.358 0.905]	10826	0.227	0.561	[0.347 0.897]
4	29805	5855	0.196	0.553	[0.352 0.864]	7898	0.265	0.527	[0.336 0.847]
5	16090	2565	0.159	0.532	[0.352 0.833]	4624	0.287	0.507	[0.331 0.815]
6	7401	855	0.116	0.526	[0.34 0.815]	2294	0.31	0.494	[0.339 0.775]
7	2723	214	0.079	0.512	[0.347 0.806]	865	0.318	0.493	[0.341 0.765]
8	775	62	0.08	0.492	[0.363 0.722]	251	0.324	0.496	[0.321 0.788]
		B_N	B_R	B_α	B_{CI}	C_N	C_R	C_α	C_{CI}
		33592	0.387	0.687	[0.408 0.967]	17234	0.198	1	[1. 1.]
		29903	0.441	0.617	[0.369 0.945]	7383	0.109	0.839	[0.483 1.]
		22002	0.461	0.576	[0.354 0.919]	3502	0.073	0.72	[0.404 1.]
		14401	0.483	0.559	[0.347 0.904]	1651	0.055	0.636	[0.391 1.]
		8121	0.505	0.551	[0.346 0.902]	780	0.048	0.584	[0.359 0.925]
		3900	0.527	0.544	[0.345 0.889]	352	0.048	0.553	[0.345 0.884]
		1479	0.543	0.547	[0.354 0.866]	165	0.061	0.532	[0.353 0.815]
		407	0.525	0.559	[0.364 0.885]	55	0.071	0.532	[0.367 0.785]

Table E-20: Link failure protection - USNET-network - minimized cost

E-12 Pan-EUR link failure protection with minimized cost setting

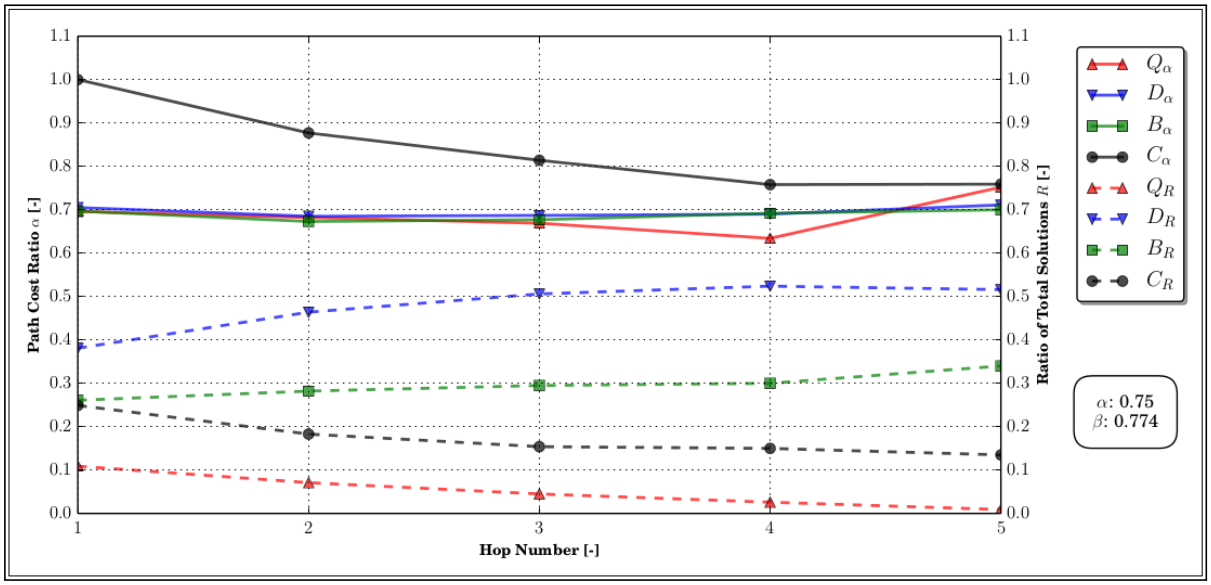


Figure E-20: Link failure protection - Pan-EUR-network - minimized cost

Hop	*Total*	Q_N	Q_R	Q_α	Q_{CI}	D_N	D_R	D_α	D_{CI}
1	67482	4179	0.062	0.815	[0.418 1.562]	17385	0.258	0.85	[0.417 1.721]
2	31882	965	0.03	0.76	[0.41 1.405]	6494	0.204	0.757	[0.398 1.451]
3	10302	166	0.016	0.72	[0.38 1.362]	1607	0.156	0.696	[0.368 1.248]
4	2246	14	0.006	0.595	[0.428 0.78]	252	0.112	0.662	[0.356 1.276]
5	341	0	0	0	['-', '-']	30	0.088	0.714	[0.409 1.147]
		B_N	B_R	B_α	B_{CI}	C_N	C_R	C_α	C_{CI}
		17668	0.261	0.697	[0.4 0.972]	16859	0.249	1	[1. 1.]
		9064	0.282	0.673	[0.394 0.967]	5870	0.183	0.877	[0.508 1.]
		3047	0.295	0.677	[0.395 0.962]	1586	0.154	0.814	[0.486 1.]
		646	0.3	0.692	[0.399 0.969]	324	0.15	0.758	[0.429 1.]
		108	0.34	0.7	[0.371 0.959]	43	0.135	0.759	[0.424 0.987]

Table E-21: Link failure protection - Pan-EUR-network - minimized cost

E-13 USNET switch failure protection with minimized crankback setting

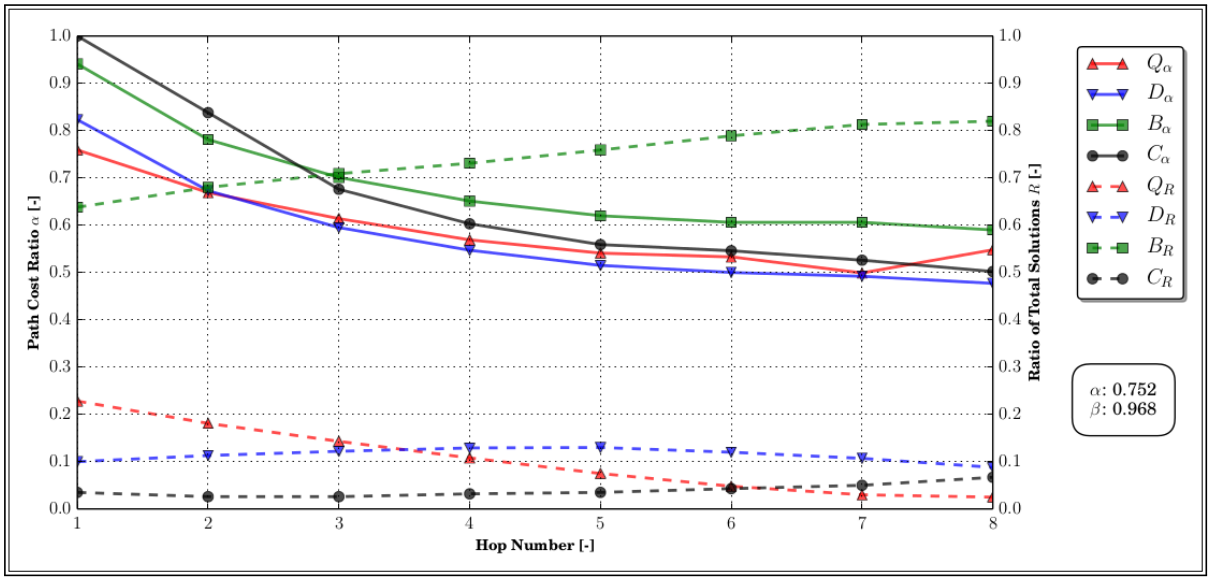


Figure E-21: Switch failure protection - USNET-network - minimized crankback

Hop	*Total*	Q_N	Q_R	Q_α	Q_{CI}	D_N	D_R	D_α	D_{CI}
1	86834	19810	0.228	0.759	[0.392 1.44]	8642	0.1	0.823	[0.376 1.929]
2	67972	12275	0.181	0.669	[0.376 1.222]	7699	0.113	0.673	[0.349 1.4]
3	47821	6846	0.143	0.614	[0.369 1.085]	5842	0.122	0.595	[0.339 1.154]
4	29692	3218	0.108	0.569	[0.351 0.968]	3823	0.129	0.547	[0.335 1.01]
5	16124	1210	0.075	0.541	[0.348 0.925]	2099	0.13	0.515	[0.327 0.909]
6	7373	354	0.048	0.533	[0.355 0.892]	888	0.12	0.5	[0.319 0.941]
7	2760	83	0.03	0.499	[0.333 0.739]	296	0.107	0.492	[0.318 0.891]
8	805	20	0.025	0.548	[0.375 1.006]	71	0.088	0.477	[0.32 0.749]
		B_N	B_R	B_α	B_{CI}	C_N	C_R	C_α	C_{CI}
		55379	0.638	0.941	[0.436 2.286]	3003	0.035	1	[1. 1.]
		46237	0.68	0.781	[0.385 1.771]	1761	0.026	0.838	[0.454 1.614]
		33905	0.709	0.701	[0.372 1.481]	1228	0.026	0.676	[0.366 1.271]
		21707	0.731	0.651	[0.36 1.287]	944	0.032	0.603	[0.38 1.141]
		12245	0.759	0.62	[0.357 1.155]	570	0.035	0.559	[0.367 0.97]
		5814	0.789	0.606	[0.356 1.093]	317	0.043	0.546	[0.369 0.896]
		2243	0.813	0.606	[0.364 1.032]	138	0.05	0.526	[0.355 0.926]
		660	0.82	0.59	[0.371 0.967]	54	0.067	0.502	[0.322 0.784]

Table E-22: Switch failure protection - USNET-network - minimized crankback

E-14 Pan-EUR switch failure protection with minimized crankback setting

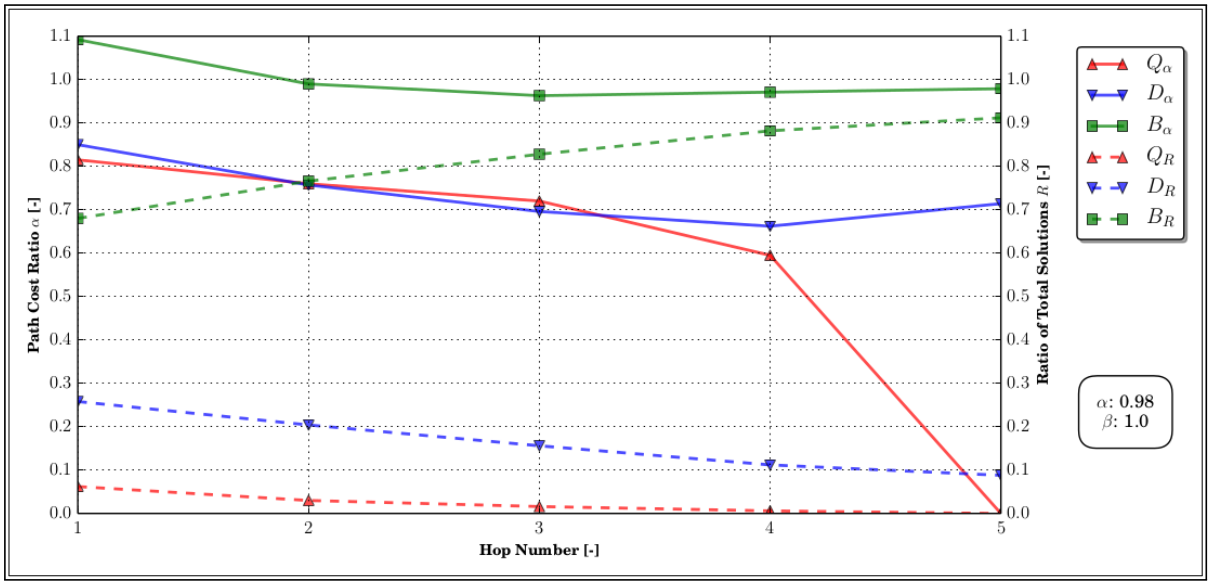


Figure E-22: Switch failure protection - Pan-EUR-network - minimized crankback

Hop	*Total*	Q_N	Q_R	Q_α	Q_{CI}	D_N	D_R	D_α	D_{CI}
1	67453	3256	0.048	0.703	[0.404 0.972]	13144	0.195	0.7	[0.406 0.974]
2	31981	788	0.025	0.668	[0.388 0.963]	5435	0.17	0.665	[0.385 0.973]
3	10332	128	0.012	0.629	[0.411 0.915]	1322	0.128	0.649	[0.372 0.963]
4	2208	11	0.005	0.641	[0.406 0.923]	205	0.093	0.62	[0.364 0.912]
5	326	1	0.003	0.719	[0.719 0.719]	24	0.074	0.617	[0.341 0.963]
		B_N	B_R	B_α	B_{CI}	C_N	C_R	C_α	C_{CI}
		45918	0.68	1.092	[0.456 2.5]	0	0	0	['-', '-']
		24423	0.766	0.99	[0.444 1.962]	0	0	0	['-', '-']
		8529	0.828	0.963	[0.444 1.738]	0	0	0	['-', '-']
		1980	0.882	0.971	[0.462 1.704]	0	0	0	['-', '-']
		311	0.912	0.979	[0.479 1.564]	0	0	0	['-', '-']

Table E-23: Switch failure protection - Pan-EUR-network - minimized crankback

E-15 USNET link failure protection with minimized crankback setting

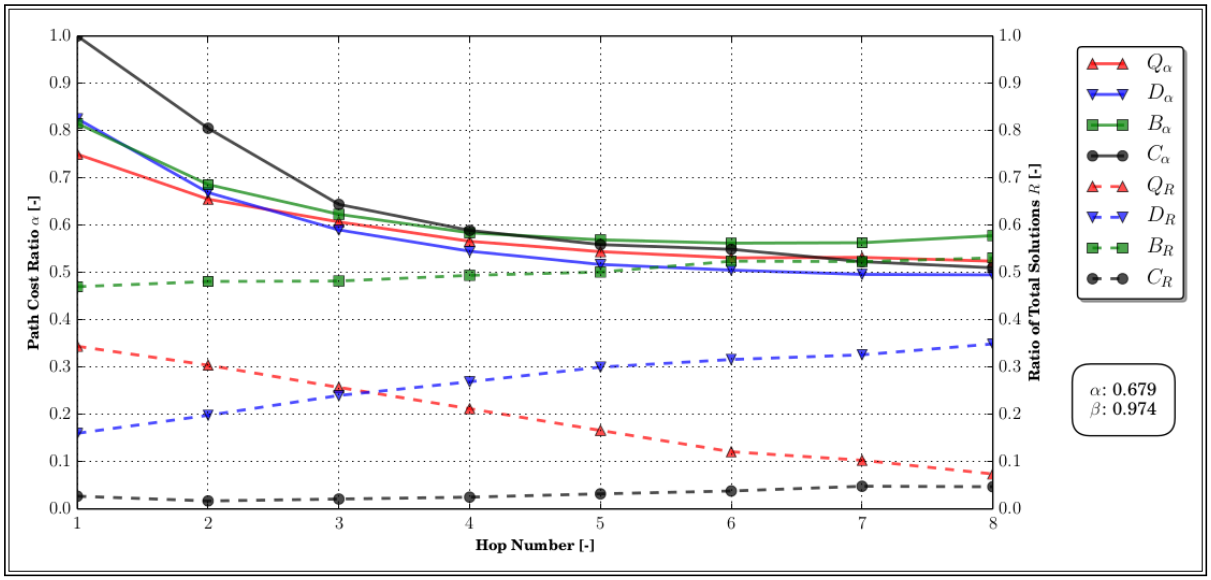


Figure E-23: Link failure protection - USNET-network - minimized crankback

Hop	*Total*	Q_N	Q_R	Q_α	Q_{CI}	D_N	D_R	D_α	D_{CI}
1	86740	29825	0.344	0.75	[0.39 1.405]	13875	0.16	0.825	[0.382 1.85]
2	67861	20603	0.304	0.655	[0.367 1.181]	13469	0.198	0.669	[0.353 1.403]
3	47818	12312	0.257	0.607	[0.362 1.071]	11454	0.24	0.59	[0.349 1.085]
4	29813	6315	0.212	0.566	[0.351 0.952]	8030	0.269	0.545	[0.338 0.972]
5	16000	2663	0.166	0.544	[0.35 0.895]	4799	0.3	0.517	[0.334 0.875]
6	7346	892	0.121	0.531	[0.351 0.873]	2324	0.316	0.505	[0.342 0.82]
7	2680	277	0.103	0.532	[0.37 0.919]	874	0.326	0.496	[0.344 0.781]
8	769	57	0.074	0.524	[0.353 0.811]	268	0.349	0.495	[0.349 0.721]
		B_N	B_R	B_α	B_{CI}	C_N	C_R	C_α	C_{CI}
		40735	0.47	0.815	[0.416 1.721]	2305	0.027	1	[1. 1.]
		32622	0.481	0.686	[0.373 1.37]	1167	0.017	0.805	[0.472 1.489]
		23027	0.482	0.623	[0.358 1.256]	1025	0.021	0.644	[0.371 1.178]
		14727	0.494	0.584	[0.349 1.118]	741	0.025	0.589	[0.382 1.07]
		8023	0.501	0.569	[0.346 1.074]	515	0.032	0.559	[0.37 0.866]
		3849	0.524	0.562	[0.344 1.02]	281	0.038	0.549	[0.368 1.033]
		1401	0.523	0.563	[0.357 0.973]	128	0.048	0.523	[0.334 0.859]
		408	0.531	0.578	[0.338 0.96]	36	0.047	0.51	[0.35 0.699]

Table E-24: Link failure protection - USNET-network - minimized crankback

E-16 Pan-EUR link failure protection with minimized crankback setting

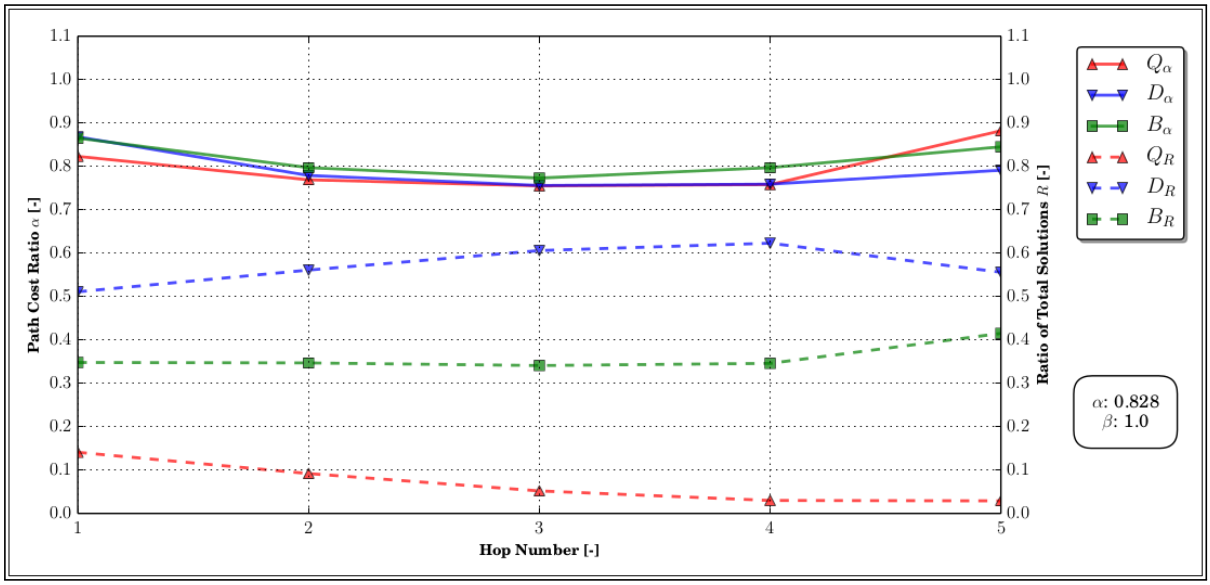


Figure E-24: Link failure protection - Pan-EUR-network - minimized crankback

Hop	*Total*	Q_N	Q_R	Q_α	Q_{CI}	D_N	D_R	D_α	D_{CI}
1	67885	9601	0.141	0.823	[0.411 1.597]	34686	0.511	0.868	[0.424 1.787]
2	32327	2969	0.092	0.769	[0.405 1.42]	18133	0.561	0.779	[0.41 1.45]
3	10473	549	0.052	0.755	[0.392 1.372]	6350	0.606	0.756	[0.415 1.327]
4	2217	67	0.03	0.758	[0.343 1.308]	1382	0.623	0.759	[0.404 1.285]
5	313	9	0.029	0.882	[0.452 1.606]	174	0.556	0.791	[0.433 1.256]
		B_N	B_R	B_α	B_{CI}	C_N	C_R	C_α	C_{CI}
		23598	0.348	0.865	[0.413 1.824]	0	0	0	['-', '-']
		11225	0.347	0.797	[0.403 1.519]	0	0	0	['-', '-']
		3574	0.341	0.773	[0.397 1.383]	0	0	0	['-', '-']
		768	0.346	0.797	[0.392 1.395]	0	0	0	['-', '-']
		130	0.415	0.845	[0.401 1.385]	0	0	0	['-', '-']

Table E-25: Link failure protection - Pan-EUR-network - minimized crankback

Bibliography

[1] Open Network Foundation. (2013, apr) Openflow switch specification openflow version 1.3.2 (wire protocol 0x04). [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.2.pdf

[2] ——. (2013, dec) Sdn architecture overview (version 1.0). [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/SDN-architecture-overview-1.0.pdf

[3] ITU-T, "Itu-t recommendation g.1010: End-user multimedia qos categories," ITU Telecommunication Standardization Sector, Tech. Rep. ITU-T G.1010. [Online]. Available: http://www.itu-t.org

[4] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "Openflow: meeting carrier-grade recovery requirements," *Computer Communications*, 2012.

[5] H. Kirrmann, M. Hansson, and P. Muri, "Iec 62439 prp: Bumpless recovery for highly available, hard real-time industrial networks," in *Emerging Technologies and Factory Automation, 2007. ETFA. IEEE Conference on.* IEEE, 2007, pp. 1396–1399.

[6] M. Rostan, "Industrial ethernet technologies: Overview," in *ETG Industrial Ethernet Seminar Series, Nuremberg*, 2008.

[7] N. L. van Adrichem, C. Doerr, and F. A. Kuipers, "Opennetmon: Network monitoring in openflow software-defined networks."

[8] J. Postel, "Internet Protocol," RFC 791 (INTERNET STANDARD), Internet Engineering Task Force, Sep. 1981, updated by RFCs 1349, 2474, 6864. [Online]. Available: http://www.ietf.org/rfc/rfc791.txt

[9] ITU-TX, "Information technology–open systems interconnection–basic reference model: The basic model," International Organization for Standardization / International Electrotechnical Commission, Tech. Rep. Recommendation 200 (1994)| ISO/IEC 7498-1: 1994, 1994.

[10] IEEE, "Ieee 802.3 standard for ethernet," Institute of Electrical and Electronics Engineers, WG802.3 Ethernet Working Group, Tech. Rep. 802.3-2012, 2012.

[11] ——, "Ieee standard for local and metropolitan area networks: Media access control (mac) bridges," Institute of Electrical and Electronics Engineers, WG802.1 Bridging and Management Working Group, Tech. Rep. 802.1D-2004, 2011.

[12] ——, "Ieee standard for local and metropolitan area networks–media access control (mac) bridges and virtual bridged local area networks," Institute of Electrical and Electronics Engineers, WG802.1 - Higher Layer LAN Protocols Working Group, Tech. Rep. 802.1Q-2011, 2011.

[13] J. Case, M. Fedor, M. Schoffstall, and J. Davin, "Simple Network Management Protocol (SNMP)," RFC 1098, Internet Engineering Task Force, Apr. 1989, obsoleted by RFC 1157. [Online]. Available: http://www.ietf.org/rfc/rfc1098.txt

[14] F. Kuipers and F. Dijkstra, "Path selection in multi-layer networks," *Computer Communications*, vol. 32, no. 1, pp. 78–85, 2009.

[15] F. A. Kuipers, "Quality of service routing in the internet. theory, complexity and algorithms," 2004.

[16] M. Mendonça, B. N. Astuto, X. N. Nguyen, K. Obraczka, T. Turletti *et al.*, "A survey of software-defined networking: Past, present, and future of programmable networks," 2013.

[17] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown, "Elastictree: Saving energy in data center networks." in *NSDI*, vol. 3, 2010, pp. 19–21.

[18] K.-K. Yap, M. Kobayashi, R. Sherwood, T.-Y. Huang, M. Chan, N. Handigol, and N. McKeown, "Openroads: Empowering research in mobile networks," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 1, pp. 125–126, 2010.

[19] N. Feamster, "Outsourcing home network security," in *Proceedings of the 2010 ACM SIGCOMM workshop on Home networks*. ACM, 2010, pp. 37–42.

[20] A. Doria, J. H. Salim, R. Haas, H. Khosravi, W. Wang, L. Dong, R. Gopal, and J. Halpern, "Forwarding and Control Element Separation (ForCES) Protocol Specification," RFC 5810 (Proposed Standard), Internet Engineering Task Force, Mar. 2010. [Online]. Available: http://www.ietf.org/rfc/rfc5810.txt

[21] A. Lara, A. Kolasani, and B. Ramamurthy, "Network innovation using openflow: A survey," 2013.

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

[23] Open Network Foundation. (2013, dec) Open vswitch manual - ovs-vswitchdb.conf.db(5) 2.0.90. [Online]. Available: http://openvswitch.org/ovs-vswitchd.conf.db.5.pdf

[24] Stanford University. (2014) Pox openflow controller wiki. [Online]. Available: https://openflow.stanford.edu/display/ONL/POX+Wiki

[25] Nippon Telegraph and Telephone Corporation. (2014, jan) Ryu sdn controller. [Online]. Available: http://osrg.github.io/ryu/

[26] Project FloodLight. (2014, jan) Floodlight open sdn controller. [Online]. Available: http://www.projectfloodlight.org/floodlight/

[27] Linux Foundation. (2014, jan) Opendaylight project. [Online]. Available: http://www.opendaylight.org/

[28] NEC-Corporation. (2014) Quantum plug-in for openstack cloud computing software. [Online]. Available: https://wiki.openstack.org/wiki/Neutron/NEC_OpenFlow_Plugin

[29] S. H. Yeganeh, A. Tootoonchian, and Y. Ganjali, "On scalability of software-defined networking," *Communications Magazine, IEEE*, vol. 51, no. 2, pp. 136–141, 2013.

[30] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: Scaling flow management for high-performance networks," in *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4. ACM, 2011, pp. 254–265.

[31] 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*. USENIX Association, 2010, pp. 3–3.

[32] S. Hassas Yeganeh and Y. Ganjali, "Kandoo: a framework for efficient and scalable offloading of control applications," in *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 2012, pp. 19–24.

[33] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama *et al.*, "Onix: A distributed control platform for large-scale production networks." in *OSDI*, vol. 10, 2010, pp. 1–6.

[34] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Flowvisor: A network virtualization layer," *OpenFlow Switch Consortium, Tech. Rep*, 2009.

[35] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. ACM, 2010, pp. 267–280.

[36] C. Hopps, "Analysis of an Equal-Cost Multi-Path Algorithm," RFC 2992 (Informational), Internet Engineering Task Force, Nov. 2000. [Online]. Available: http://www.ietf.org/rfc/rfc2992.txt

[37] F. A. Kuipers, "An overview of algorithms for network survivability," *ISRN Communications and Networking*, vol. 2012, p. 24, 2012.

[38] P. Fonseca, R. Bennesby, E. Mota, and A. Passito, "A replication component for resilient openflow-based networking," in *Network Operations and Management Symposium (NOMS), 2012 IEEE*. IEEE, 2012, pp. 933–939.

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

[40] IEEE, "Ieee standard for local and metropolitan area networks–station and media access control connectivity discovery," Institute of Electrical and Electronics Engineers, WG802.1 - Higher Layer LAN Protocols Working Group, Tech. Rep. 802.1AB-2009, 2009.

[41] D. Plummer, "Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware," RFC 826 (INTERNET STANDARD), Internet Engineering Task Force, Nov. 1982, updated by RFCs 5227, 5494. [Online]. Available: http://www.ietf.org/rfc/rfc826.txt

[42] D. Katz and D. Ward, "Bidirectional Forwarding Detection (BFD)," RFC 5880 (Proposed Standard), Internet Engineering Task Force, Jun. 2010.

[43] A. Sgambelluri, A. Giorgetti, F. Cugini, F. Paolucci, and P. Castoldi, "Openflow-based segment protection in ethernet networks," *Optical Communications and Networking, IEEE/OSA Journal of*, vol. 5, no. 9, pp. 1066–1075, 2013.

[44] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "Fast failure recovery for in-band openflow networks." DRCN, 2013.

[45] S. Scott-Hayward, G. O'Callaghan, and S. Sezer, "Sdn security: A survey," in *Future Networks and Services (SDN4FNS), 2013 IEEE SDN for*. IEEE, 2013, pp. 1–7.

[46] S. Shin, P. Porras, V. Yegneswaran, M. Fong, G. Gu, and M. Tyson, "Fresco: Modular composable security services for software-defined networks," in *Proceedings of Network and Distributed Security Symposium*, 2013.

[47] R. Braga, E. Mota, and A. Passito, "Lightweight ddos flooding attack detection using nox/openflow," in *Local Computer Networks (LCN), 2010 IEEE 35th Conference on*. IEEE, 2010, pp. 408–415.

[48] S. Son, S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "Model checking invariant security properties in openflow."

[49] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu, "A security enforcement kernel for openflow networks," in *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 2012, pp. 121–126.

[50] E. M. Spiegel and T. Murase, "An alternate path routing scheme supporting qos and fast connection setup in atm networks," in *Global Telecommunications Conference, 1994. GLOBECOM'94. Communications: The Global Bridge., IEEE*, vol. 2. IEEE, 1994, pp. 1224–1230.

[51] L. Wang, R.-F. Chang, E. Lin, and J. C.-s. Yik, "Apparatus for link failure detection on high availability ethernet backplane," Aug. 21 2007, uS Patent 7,260,066.

[52] D. Levi and D. Harrington, "Definitions of Managed Objects for Bridges with Rapid Spanning Tree Protocol," RFC 4318 (Proposed Standard), Internet Engineering Task Force, Dec. 2005.

[53] 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

[54] D. Katz and D. Ward, "Bidirectional Forwarding Detection (BFD) for IPv4 and IPv6 (Single Hop)," RFC 5881 (Proposed Standard), Internet Engineering Task Force, Jun. 2010. [Online]. Available: http://www.ietf.org/rfc/rfc5881.txt

[55] MPLS-TP.com. (2012, jan) Mpls-tp.com - a collection of information on networking - bfd (bi-directional forwarding detection). [Online]. Available: http://mpls-tp.com/bfd/#axzz2qNhXzuhS

[56] Y. Rekhter, T. Li, and S. Hares, "A Border Gateway Protocol 4 (BGP-4)," RFC 4271 (Draft Standard), Internet Engineering Task Force, Jan. 2006, updated by RFCs 6286, 6608, 6793. [Online]. Available: http://www.ietf.org/rfc/rfc4271.txt

[57] J. Postel, "Transmission Control Protocol," RFC 793 (INTERNET STANDARD), Internet Engineering Task Force, Sep. 1981, updated by RFCs 1122, 3168, 6093, 6528.

[58] V. Jacobson, "Congestion avoidance and control," in *ACM SIGCOMM Computer Communication Review*, vol. 18, no. 4. ACM, 1988, pp. 314–329.

[59] D. D. Clark, "Window and acknowledgement strategy in tcp," 1982.

[60] P. v. Mieghem, *Data Communications Networking*. Purdue University Press, 2006.

[61] M. Médard, S. G. Finn, and R. A. Barry, "Redundant trees for preplanned recovery in arbitrary vertex-redundant or edge-redundant graphs," *IEEE/ACM Transactions on Networking (TON)*, vol. 7, no. 5, pp. 641–652, 1999.

[62] G. Enyedi, P. Szilágyi, G. Rétvári, and A. Császár, "Ip fast reroute: Lightweight not-via without additional addresses," in *INFOCOM 2009, IEEE*. IEEE, 2009, pp. 2771–2775.

[63] R. M. Karp, *Reducibility among combinatorial problems*. Springer, 1972.

[64] J. Suurballe, "Disjoint paths in a network," *Networks*, vol. 4, no. 2, pp. 125–145, 1974.

[65] R. Bhandari, *Survivable networks: algorithms for diverse routing*. Springer, 1999.

[66] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[67] R. Bellman, "On a routing problem," DTIC Document, Tech. Rep., 1956.

[68] S. Trajanovski, F. A. Kuipers, P. Van Mieghem, A. Ilic, and J. Crowcroft, "Critical regions and region-disjoint paths in a network," in *IFIP Networking Conference, 2013*. IEEE, 2013, pp. 1–9.

[69] A. Beshir and F. Kuipers, "Variants of the minsum link-disjoint paths problem," in *16th Annual Symposium on Communications and Vehicular Technology (SCVT)*, 2009.

[70] C.-L. Li, S. T. McCormick, and D. Simchi-Levi, "The complexity of finding two disjoint paths with min-max objective function," *Discrete Applied Mathematics*, vol. 26, no. 1, pp. 105–115, 1990.

[71] A. Bley, "On the complexity of vertex-disjoint length-restricted path problems," *computational complexity*, vol. 12, no. 3-4, pp. 131–149, 2003.

[72] B. H. Shen, B. Hao, and A. Sen, "On multipath routing using widest pair of disjoint paths," in *High Performance Switching and Routing, 2004. HPSR. 2004 Workshop on*. IEEE, 2004, pp. 134–140.

[73] P. Van Mieghem, "A lower bound for the end-to-end delay in networks: Application to voice over ip," in *Global Telecommunications Conference, 1998. GLOBECOM 1998. The Bridge to Global Integration. IEEE*, vol. 4. IEEE, 1998, pp. 2508–2513.

[74] Y. Guo, F. Kuipers, and P. Van Mieghem, "Link-disjoint paths for reliable qos routing," *International Journal of Communication Systems*, vol. 16, no. 9, pp. 779–798, 2003.

[75] J. Y. Yen, "Finding the k shortest loopless paths in a network," *management Science*, vol. 17, no. 11, pp. 712–716, 1971.

[76] J. W. Suurballe and R. E. Tarjan, "A quick method for finding shortest pairs of disjoint paths," *Networks*, vol. 14, no. 2, pp. 325–336, 1984.

[77] Q. Ma and P. Steenkiste, "Routing traffic with quality-of-service guarantees in integrated services networks," in *Proceedings of Workshop on Network and Operating Systems Support for Digital Audio and Video*, 1998.

[78] P. Van Mieghem and F. A. Kuipers, "Concepts of exact qos routing algorithms," *Networking, IEEE/ACM Transactions on*, vol. 12, no. 5, pp. 851–864, 2004.

[79] F. Kuipers, "Qos protocol and algorithm join forces," in *Communications and Networking in China, 2006. ChinaCom'06. First International Conference on*. IEEE, 2006, pp. 1–5.

[80] NetworkX Developer Team. (2014, feb) Networkx - high-productivity software for complex networks. [Online]. Available: http://networkx.github.io/

[81] Python Software Foundation. (2014, feb) Python - programming language. [Online]. Available: http://www.python.org/

[82] Spyder Developement Team. (2014, feb) Spyder - scientific python development environment. [Online]. Available: https://code.google.com/p/spyderlib/

[83] P. Erdos and A. Reyni, "On random graphs i." *Publ. Math. Debrecen*, vol. 6, pp. 290–297, 1959.

[84] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *science*, vol. 286, no. 5439, pp. 509–512, 1999.

[85] A. Jirattigalachote, C. Cavdar, P. Monti, L. Wosinska, and A. Tzanakaki, "Dynamic provisioning strategies for energy efficient wdm networks with dedicated path protection," *Optical Switching and Networking*, vol. 8, no. 3, pp. 201–213, 2011.

[86] R. He and B. Lin, "Dynamic power-aware shared path protection algorithms in wdm mesh networks." *Journal of Communications*, vol. 8, no. 1, 2013.

[87] P. Van Mieghem, H. De Neve, and F. Kuipers, "Hop-by-hop quality of service routing," *Computer Networks*, vol. 37, no. 3, pp. 407–423, 2001.

[88] Open vSwitch. (2014, jun) Open vswitch - manual ovs-vsctl. [Online]. Available: http://openvswitch.org/cgi-bin/ovsman.cgi?page=utilities%2Fovs-vsctl.8

[89] ——. (2014, jun) Open vswitch - manual ovs-ofctl. [Online]. Available: http://openvswitch.org/cgi-bin/ovsman.cgi?page=utilities%2Fovs-ofctl.8

[90] R. Olsson, "pktgen the linux packet generator," in *Proceedings of the Linux Symposium, Ottawa, Canada*, vol. 2, 2005, pp. 11–24.

[91] Wireshark Foundation. (2014, jul) Display filter reference: Linux kernel packet generator. [Online]. Available: http://www.wireshark.org/docs/dfref/p/pktgen.html

[92] T. Ylonen and C. Lonvick, "The Secure Shell (SSH) Transport Layer Protocol," RFC 4253 (Proposed Standard), Internet Engineering Task Force, Jan. 2006, updated by RFC 6668. [Online]. Available: http://www.ietf.org/rfc/rfc4253.txt

[93] M. Huynh, S. Goose, P. Mohapatra, and R. Liao, "Rrr: Rapid ring recovery submillisecond decentralized recovery for ethernet ring," *Computers, IEEE Transactions on*, vol. 60, no. 11, pp. 1561–1570, 2011.

[94] M. German, A. Castro, X. Masip-Bruin, M. Yannuzzi, R. Martínez, R. Casellas, and R. Muñoz, "On the challenges of finding two link-disjoint lightpaths of minimum total weight across an optical network," *Proceedings of NOC/OC&I*, pp. 217–224, 2009.

[95] P. Pieda and J. Spicer, "Using opnet to evaluate diverse routing algorithms."

[96] R. W. Floyd, "Algorithm 97: shortest path," *Communications of the ACM*, vol. 5, no. 6, p. 345, 1962.

[97] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *Journal of the ACM (JACM)*, vol. 34, no. 3, pp. 596–615, 1987.

[98] J. Vuillemin, "A data structure for manipulating priority queues," *Communications of the ACM*, vol. 21, no. 4, pp. 309–315, 1978.

Glossary

List of Symbols

Abbreviations

ARP Address Recovery Protocol
AS Autonomous System
BA Barabási-Albert random network

BFD Bidirectional Forwarding Detection
BGP Border Gateway Protocol
ECMP equal-cost multipath routing

ER Erdös-Rényi random network

IP Internet Protocol
LLDP Link Layer Discovery Protocol
LoS Loss of Signal
MAC Media Acces Control
OSI Open Systems Interconnection
OSPF Open Shortest Path First
OVS Open vSwitch
Pan-EUR Pan-European COST239 network
RF Remove Find
RPR Resilient Packet Ring
RSTP Rapid Spanning Tree Protocol

RTO Re-transmission Time-Out
RTT Rount-Trip-Time
SAMCRA Self Adapting Multi Constraint Routing Algorithm

SDN Software Defined Networking
SNMP Small Network Management Protocol
SSH Secure Shell
SSL Secure Socket Layer
STP Spanning Tree Protocol
TCP Transmission Control Protocol
USNET Sample US Network