technische universität
dortmund

A Comparison of Fast-Recovery Mechanisms in Networks

Implementation and Evaluation of Fast Recovery Mechanisms in Mininet

Frederik Maaßen May 2022

Supervisor:
Univ.-Prof. Dr. Klaus-Tycho Förster
M. Sc. Alexander Puzicha

Technical University Dortmund
Faculty for Computer Science
Chair for Modelling and Simulation (LS-4)
http://ls4-www.cs.tu-dortmund.de

Abstract

The failure of links and devices in networks is a common occurrence for any network administrator. As such modern networks implement global convergence protocols and *Fast Re-Routing* (FRR) mechanisms to quickly restore connectivity. FRR mechanisms tend to create sub-optimal paths containing unnecessary loops which increase latency, disturb other data flows or create other adverse effects on the network until the slower convergence protocol rewrites routings on the network. *Fast Recovery Methods* (FRM) like ShortCut (Shukla and Foerster 2021) optimize existing FRR mechanisms by e.g. removing existing loops in routings created by FRR. However, ShortCut has yet to be tested in further detail. In this work we implement three topologies, a FRR mechanism and ShortCut in a virtual network and run performance tests while introducing artificial failures to the network, evaluating the performance of these two methods. Our results show that ShortCut reduces the effects of sub-optimal paths chosen by a FRR mechanism and in some cases even restores pre-failure conditions on the network.

Contents

1 Introduction

In recent years, especially during the COVID-19 pandemic, network usage has risen exponentially. In Germany alone the per capita data usage on the terrestrial network has risen from 98 GB per month in 2017 to 175 GB in 2020 (Bundesnetzagentur Deutschland 2021).

1.1 Motivation

A large part of the population suddenly had to spent additional time in their homes which has contributed to this rise in data usage. But this development is not limited to the pandemic. Data usage has been constantly rising due to the popularity of streaming services, increased internet usage in daily life and the rising popularity of cloud based services.
Because of the increased usage, failing networks cause an increasingly severe amount of social and economic costs. This is why the reliability of networks is as important as ever.
Failures in networks will always occur, be it through the failure of hardware, failures caused by errors in software or human errors. In addition to this the maintenance of networks will also regularly reduce a networks performance or cause the whole network to be unavailable.
Network administrators use a multitude of ways to increase performance, reduce the impact of failures on the network and achieve the highest possible availability and reliability. Two of these methods include the usage of global convergence protocols like *Open Shortest Path First* (OSPF) (Moy et al. 0004–1998) or similar methods, either on the routers themselves or on a controller in a *Software Defined Network* (SDN), and the usage of FRR (Chiesa et al. 2021) approaches.
The key difference between both is the time they take to become active. Because FRR mechanisms only use the available data on the device they tend to take effect near immediately. Global convergence protocols however are slow, sometimes even taking seconds to converge (Liu et al. 2013). This is due to them collecting information about the network by communicating with multiple devices, recomputing routes for all affected parts of the network and deploying these flows on routers and switches.

Most of the FRR approaches will however create sub-optimal paths which may be already in use or contain loops, effectively reducing the performance of the network.
FRMs like ShortCut (Shukla and Foerster 2021), *Resilient Routing Layers* (RRL) (Kvalbein et al. 2005), Revive (Haque and Moyeen 2018) and Blink (Thomas Holterbach et al. 2019) try to alleviate this issue by removing longer paths from the routings only using data available on the device, bridging the gap between FRR and the global convergence protocol.

1.2 State of the art

Until the global convergence protocol converges it leaves the routing to In-network methods like FRR which will reroute traffic according to pre-defined alternative routes on the network. In some cases however methods like FRR cause routing paths to be longer than necessary which produces additional traffic on the network and adds delay to transmissions.
RRL pre-computes alternative routing tables, switching between routing tables in case of failure, but needs to manipulate packets to inform routers of changed routing tables.
ShortCut uses information about the incoming packet to determine whether or not the packet returned to the router, using already existing FRR implementations. In case a packet returns it will remove the route with the highest priority from the routing table, assuming that the path is no longer available.
Revive installs backup routes pro-actively using an optimized algorithm and controllers, but is prone to loops created by alternative paths as failures are not propagated to routers.
Blink uses *Transmission Control Protocol* (TCP) mechanisms to detect failures in TCP flows on the network, but only works reactively, requiring the network to have already failed before taking effect.
All of these FRMs are described in further detail in section 2.3.5.
Older FRMs have already been evaluated thoroughly and even though they do work in theory they either have yet to see widespread implementation or face limitations in their applicability, be it by requiring a high amount of resources or by using e.g. packet manipulation, excluding networks which by structure are incompatible to such mechanisms.

1.3 Contribution

We provide an introduction to the topic of modern networks, failure scenarios and resilient routing.
In this context we use Mininet (Lantz, Bob and the Mininet Contributors 2021), a tool to create virtual networks, to implement multiple topologies with routings. We then implement a simple FRR mechanism, re-routing returning packets to an alternative path.

We implement ShortCut using *nftables* (Ayuso et al. 2019) and python 3.8 (van Rossum and Drake 2009) that can be installed on any linux based router, and which is used for testing.
We build a testing framework that can be used to automatically create Mininet topologies, formulate tests in python dictionaries using existing measurement functions, set network wide bandwidth limits or delays and to run automatic sets of tests. The framework can be called using an argument based *Command Line Interface* (CLI).
Using this framework we test several topologies with increasingly longer looped paths using FRR with and without ShortCut. We discover that ShortCut is able to reduce latency in our topologies by up to 30%, reduces the amount of packets forwarded on the network by up to 38% for TCP transfers and up to 55% for *User Datagram Protocol* (UDP) transfers and is able to remove bottlenecks created by concurrent data flows on the looped paths, restoring the original functionality of the network.
The test framework, this thesis and all test results can be accessed on our Git repository (Maaßen 05/2022).

2 Basics

Our work evaluates the effectiveness of fast recovery methods. In this context we explain software-defined networks in section 2.1, including controllers and their possible applications. In section 2.2 we then elaborate on common failure scenarios and their causes, for which we show measures network administrators take to alleviate security concerns and achieve high reliability in section 2.3. These are split into different levels of operation and put into perspective for the evaluation of a failure recovery mechanism.
Lastly we provide an overview over our used technologies for testing in section 2.4, including the evaluation criteria for a network, types of measurements we can perform on the network and how artificial failures can be introduced to such a network.

2.1 Modern networks

In our digital society networks have become an essential infrastructure for countries worldwide. A huge part of the population today is in some form reliant on the availability of networks and associated services, be it on their smartphone or their home internet access. As such, the reliability of a network has a huge impact on the economy and social life.
A study in 2015 (Montag et al. 2015) found that their participants used WhatsApp, a messaging service, for around 32 minutes a day, with an overall usage mean of their smartphone of around 162 minutes a day, mostly spent online. Private and commercial users alike cause a huge amount of traffic. The german federal network agency reported a per capita network usage on the terrestrial network of 175 GB per month (Bundesnetzagentur Deutschland 2021) in Germany. The traffic per capita in 2017 was at around 98 GB per month. Together with the rise of e.g. cloud based solutions for companies, network requirements are expected to rise exponentially.
With bigger networks and higher traffic the work of a network administrator is getting more complex by the day. Modern networks need to be flexible, scalable and reliable. Configuration changes should be applied near instantly, failures should be corrected as fast as possible and new components should only have to be connected to the existing network, with all additional configuration being applied automatically or only requiring a minimum amount of manual work.

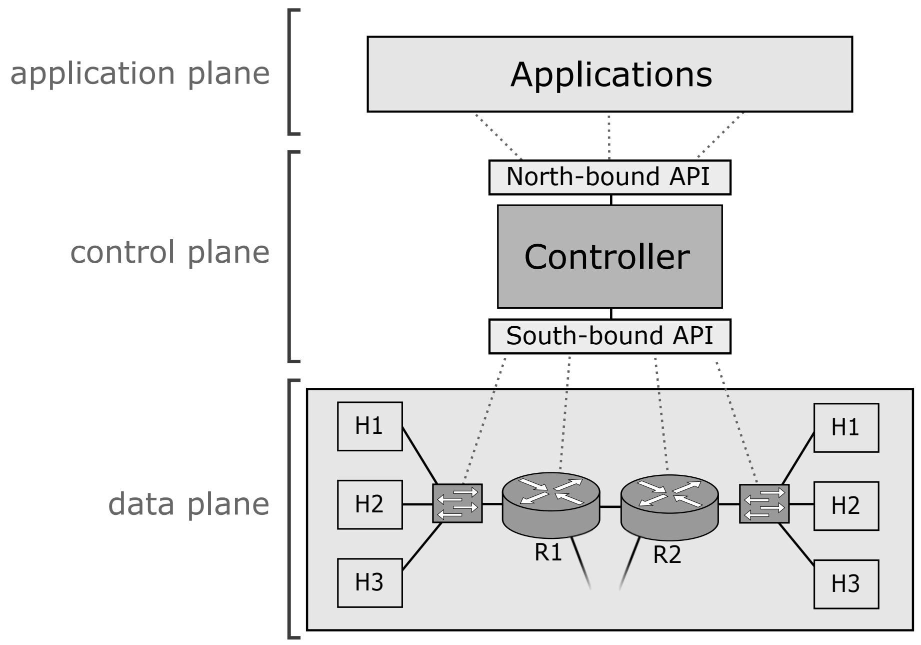


Figure 2.1: SDN network architecture

In the past many of the challenges a network faces in modern times were solved by specialized networking devices which offered their own proprietary configuration capabilities, including interfaces, protocols and software. In many cases this would lead to companies using one supplier for the biggest part of their networks, as well as huge amounts of manual labour configuring each device.

While many networks are still configured manually, the current trend strives towards SDNs, taking complexity from single devices and instead using a component called controller to handle logical operations (Rak and Hutchison 2020), like configuration of new devices, the roll out of configuration changes and monitoring. This allows for a better network management especially in networks with high availability constrains and flexibility requirements e.g. the commercial network of an internet provider. Network administrators receive a real time overview over the network, implement new software controlled features on the fly or create virtual network environments. A cloud provider could create automatic processes using a SDN controller to configure virtual networks for a tenant in seconds.

In fig. 2.1 you can see a typical SDN architecture with the controller as a centralized management component. It is connected to multiple networking devices like routers and switches, and is able to read and write information to these devices. Several protocols can be used to affect routers and switches. Some vendors have their own proprietary protocols, while some protocols follow open specifications like OpenFlow and *Open Virtual Switch Database* (OVSDB). They allow the controller to communicate with the networking devices via their so called *south-bound api*.

In a network with an OpenFlow controller, routers use flow tables for routing. These contain information provided by the controller about the destination of specific packets, identified by a set of rules. If a packet enters the router for which no flow table entry matches, the "table-miss" flow entry is hit and the packet is sent to the controller. The

controller would then use the information attached to the packet to create a new flow entry for similar packets according to the controller's configuration. The controller can then edit the flow tables of the affected routers and e.g. add the newly created flow entry.
Alongside these controllers multiple vendors supplied SDN controller platforms that allow for an enhanced overview over the network, add the ability to use modular plugins for additional functionality and overall simplify the interaction with the network. Lastly, controllers serve as an interface for other applications that can communicate with the controller over its so called *north-bound api*. This includes applications like firewalls, load balancers and even business systems, allowing for a lot of flexibility when designing an automated network.
This causes networks to be logically split by their available information and level of operation. On the one hand there are operations directly on routers and switches, only using available information to the device itself and maybe its neighbours. Mechanisms working under these conditions are part of the so called *data plane*. On the other hand there are operations on the controllers, managing devices and maintaining an overview over the network, or routing protocols using link-state algorithms to collect data about the network. These functions and mechanisms are part of the *control plane*.

2.2 Failure scenarios in modern networks

Computer networks are a complex structure of many different interconnected devices, with each device powered on and working for long times without break. As such, network components will and do fail. In big networks, daily failures of multiple devices are a common occurrence (Gill, Jain, and Nagappan 2011) even though most devices will be optimally stored, cooled and secured against outer influences or human interference. We show the scale of failures occurring in networks in section 2.2.1 using a study by Gill, Jain, and Nagappan (2011), which analyzed failures in a network of data centers over a year and evaluated their causes and possible consequences.

2.2.1 Failures in data centers

The reasons for failures are multifaceted and largely depend on the used network components, network structure and security measures. In 2011 a study by Gill, Jain, and Nagappan (2011) analysed failures that occurred in a network of large professional data centers over a year. They found that over their measurement period a mean of 5.2 devices experienced a failure per day, while a mean of 40.8 links experienced a failure per day. Most of these failures had a small, if any, impact on the network, as many security measures like redundancy or dynamic re-routing were already implemented. Still, many devices and links experienced downtimes and a huge amount of packets and data were lost. The downtimes and lost data could cause costs for the provider as it is common to define a service-level agreement (SLA), an agreement between service-provider and consumer which defines levels of availability and reliability (besides other factors) the provider has to fulfil (Wustenhoff and BluePrints 2002). If the agreed levels of availability and reliability are not met, the provider will pay a penalty. A lower downtime or faster convergence can therefore avoid additional costs.

Failure causes

When analysing the causes of the failures, most device failures were caused by maintenance operations on the network, for example if an aggregation switch was shut down temporarily and each connected switch and router was temporarily unreachable. Link failures on the other hand were caused by many different issues, including software errors in load balancers also causing the link from and to the load balancer to be assumed as faulty, as well as hardware issues requiring the replacement or power cycling of components. Some were discovered to be caused by protocol issues. The misconfiguration of components was only discovered sporadically, but was also a cause for network failures.
Aside from the observed failure causes in this example, there are still factors like environmental interferences, e.g. environmental catastrophes or cosmic radiation, external influences, e.g. cut cables by a building company, or malicious attacks, e.g. denial-of-service attacks against parts of the network, that are able to cause failures in computer networks.
Failures caused by e.g. maintenance work are unavoidable. Most network administrators will therefore introduce a level of redundancy in their network, feathering the impact of e.g. the replacement of a faulty component. In the practical reality many systems work with an active-active architecture. In this case all involved components for redundancy are used in the network for normal operation. Each maintenance operation will cause the backup line to temporarily be under an increased level of load, possibly causing an overload, additional delays or the loss of data. But while active-active architectures are also more expensive to run because of additional running costs for power and maintenance, they increase efficiency of bought devices and provide redundancy.

2.3 Resilient routing in modern networks

The main resiliency goals of a network provider are (1) producing as little failures as possible and (2) reducing the impact of each remaining failure, given that some failures are unavoidable.
Securing a SDN imposes additional challenges but also additional flexibility over a traditional network. In a network without controller most routings will be either entered manually or a routing protocol like OSPF (Moy et al. 0004–1998) or Intermediate System to Intermediate System (IS-IS) (Callon 0012–1990) would be used. In case a routing protocol is used, the routers themselves handle the routing in accordance to the definition of specified routing protocol. In the subsection 2.3.1 we explain one of the traditional routing protocols OSPF and its implications. We put this into contrast in subsection 2.3.2 where we discuss benefits and drawbacks of a SDN. After we established a basic overview over the differences of architectures, we shift our focus

on resilience on the control plane and data plane specifically in subsection 2.3.3 and subsection 2.3.4. While fast recovery methods (FRMs) mostly work on the data plane they take a role somewhere in between the data and control plane and are often used in addition to methods on the data and control plane. This is why we devote the subsection 2.3.5 to the explanation of multiple FRMs and their applications.

2.3.1 Traditional routing protocols by taking the example of OSPF

OSPF is a link-state protocol and therefore shares information contained on a router with its direct neighbours. Routers will be split into different pre-assigned areas, with a cluster of routers designated as *backbone area*. The rest is split up into a multitude of areas and area types, summarized as *nonbackbone* areas. The backbone area is used as a central network point and all traffic from the network that moves between other areas must flow through the backbone area. Because of its design, the backbone area would be a single point of failure if only a single router was used, blocking all traffic between areas in case of a failure. The usage of multiple routers as backbone area serves as additional safety through redundancy, combating this issue.

Each router shares all his information with its directly adjacent neighbours on the condition that the adjacent router belongs to the same pre-defined area. After an introductory period of sharing routing information using the *hello protocol*, each router in an area would hold a database about the network. OSPF uses the *hello protocol* and each router sends regular "Hello" messages to its neighbours according to a predefined *HelloInterval* which defaults to a value of 10 s. In addition to this, there is a *RouterDeadInterval* which should be a multiple of the *HelloInterval* and is set to 40 s by default. If a router did not receive a "Hello" message for as long as the dead interval is configured, the router will assume the neighbouring router to be down and will share this information. In case a network component, e.g. a link, would be detected as faulty, the router would share this information with its adjacent neighbours which will in turn repeat the change to all routers connected to them. This would cause a propagation of a refreshed version of the routing and allows for a relatively fast convergence time.

But communication between routers is still time-consuming and a convergence of a link state protocol like OSPF will also suffer from its fault detection mechanisms. A failing link might be easily and quickly recognized, but a failing router would require the network to first wait for the *RouterDeadInterval*. This would mean a down time of 30 s to 40 s depending on the configuration.

Because routers perform these tasks themselves this also implicates that the internal resources, e.g. processing power and internal storage of each router, need to be used in part for OSPF. The network itself also receives additional traffic because the need of the routers to share information. This would cause the network to break if components were overloaded without any protective measures, as the processing power would not suffice to still maintain OSPF.

2.3.2 Security benefits and drawbacks of software-defined networks

In contrast to this, a software-defined network uses one or more controllers as central management components. They are made up of specialized hardware and software, and use their own fault detection and failure recovery. Routers in such a network have no need to perform complex routing protocol algorithms. Said operations are taken over by the controller.

The main purpose of a controller is the creation of an abstraction layer of the network. Network administrators have an easier time collecting information about the network, as well as deploying new router configurations, granting them a lot of flexibility when deploying safety measures.

SDN is, as any network, susceptible to denial-of-service (DOS) attacks. An e.g. OpenFlow controller adds new variants of DOS attacks. One of such would be sending packets which will by design cause a "table-miss" hit, only to cause each of these packets to be sent to the controller. In an experiment Alharbi, Layeghy, and Portmann (2017) found that the controller ONOS started to drop its performance after only 2000 packets per seconds were used in an orchestrated attack. A SDN without working controller will start dropping packets that cause a "table-miss", preventing new flows to be created and therefore limits the networks functionality.

As such SDNs introduce an additional single-point-of-failure to the network. Controllers should be used redundantly and, if possible, should only control parts of a network, with each their own controller cluster. This prevents failures in controllers to impact the whole network.

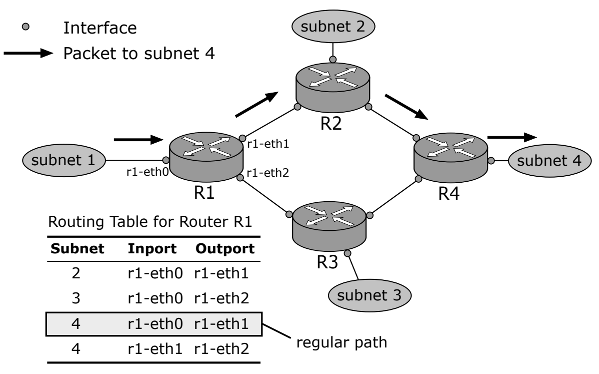
To their advantage controllers allow for the easy implementation of more advanced safety measures. A lot of work has been done toward DOS security in SDNs (Dridi and Zhani 2016, Kuerban et al. 2016).

2.3.3 Resilience on the data plane

Per definition most methods providing additional network resilience on the data plane are network agnostic and operations executed on the data plane can be assumed to be very fast as overhead produced by e.g. communication protocols is avoided. One widely used method to add resilience are static routes that are installed directly on the components, re-routing traffic in case of failures. These are called local FRR (Nelakuditi et al. (2003)). FRR routes make use of already existing link redundancy in the network.

They mostly use interface dependent routing, identifying returning packets and routing them over an alternative path. Because FRR acts on the data-plane it reacts very fast and will re-route packets in a matter of milliseconds, but by its agnostic nature will often create loops and therefore delays and additional traffic in a network.

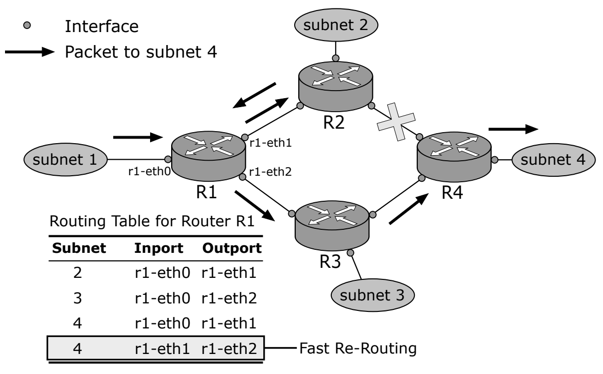
The process is visualized in section 2.3.3. In the shown example, a packet is sent from subnet 1 to subnet 4. The process on each router is exemplary shown for router R1. The routing table shows only relevant entries for this example, additional entries are omitted. It contains the destination *subnet 4* multiple times, depending on the interface that a packet was received on. In this example, if a packet to subnet 4 was received on



Routing Table for Router R1

Subnet	Inport	Outport
2	r1-eth0	r1-eth1
3	r1-eth0	r1-eth2
4	r1-eth0	r1-eth1
4	r1-eth1	r1-eth2

(a) Network with FRR before a failure



Routing Table for Router R1

Subnet	Inport	Outport
2	r1-eth0	r1-eth1
3	r1-eth0	r1-eth2
4	r1-eth0	r1-eth1
4	r1-eth1	r1-eth2

(b) Network with FRR after a failure

the interface *r1-eth0*, which is from subnet 1, the packet is sent over *r1-eth1* to router R2.

In case the connection between router R2 and R4 would be disrupted, router R2 would redirect the packet back to router R1. Because router R1 has a separate routing table entry for incoming packets on *r1-eth1* going to subnet 4, it will now send the packet on interface *r1-eth2*. The new non-optimal route now contains a loop as can be seen in section 2.3.3. While the new route might not be an optimal route, it is taking effect as soon as a router has recognized a failure and redirected a packet back to its entry interface. In this example this would also only add the small delay of 2 transmission times. The additional delay is dependent on the network and the level of redundancy used. Loops have additional implications for the network and are part of the possible optimizations for failures in a network, which is discussed in section 2.3.5.

The conditions for using FRR are easy to fulfil and most networks will already be compatible with it. A certain level of redundancy is required, as components using FRR need alternative routes to take.

2.3.4 Resilience on the control plane

In contrast, a SDN controller will react on failures by collecting information about the network and calculating near-optimal alternative routes. It would then re-write the routing tables on each affected component with protocols like OpenFlow. OSPF also works on the control plane, as the routers collect information about the network.

Operations on the control plane are very thorough as decisions are made based on an overview of the network. The collection of information and the deployment of solutions, however, is very time consuming. A failure in a network that only uses control plane mechanisms for failure handling will be unattended to during the whole process, potentially creating backlog or reducing availability for a longer period of time. Operations on the control plane mostly have convergence times in the dimension of seconds (Liu et al. 2013).

This is the reason why most modern networks will use a combination of mechanisms on the data *and* control plane, e.g. FRR and a global convergence protocol, allowing sub-optimal paths to restore availability while the global convergence protocol provides an optimised routing after some time.

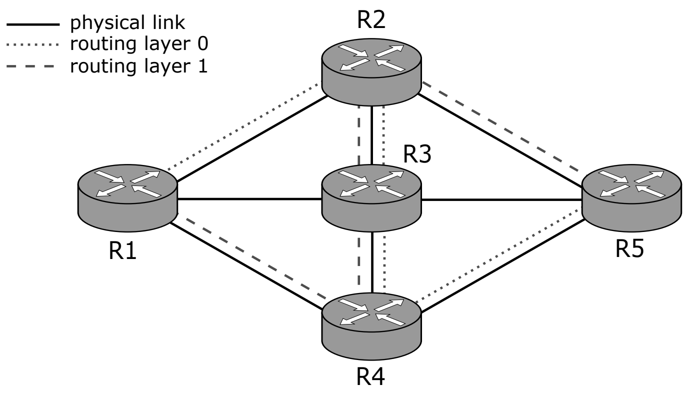


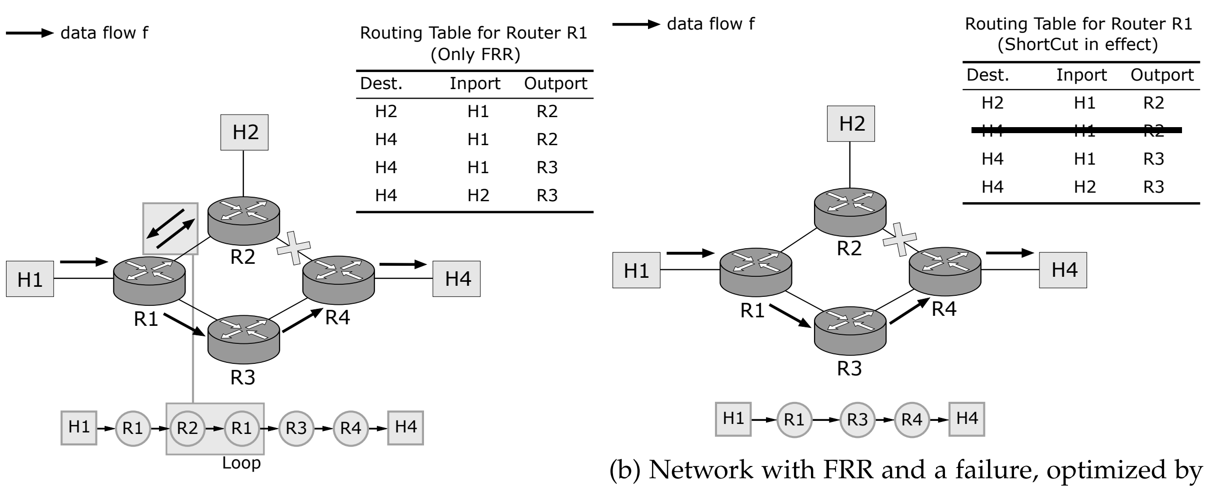
Figure 2.3: Resilient Routing Layers - visualization

2.3.5 Fast Recovery Methods

FRMs are operations that take place on the data plane. A combination of operations on the data *and* control plane inevitably create a delay; while the alternative route was already established in a matter of milliseconds through e.g. FRR, the operation on the control plane, e.g. a controller calculating optimal routings, will take any time in the order of seconds. In this time gap the network uses a sub optimal route for its traffic. Loops created by e.g. FRR will affect the network in this time frame and will not only potentially delay traffic but also reserve scarce link capacity on looped routes.
As such FRMs can be perceived as optimizations of data plane mechanisms like FRR. Because FRR is very prominent in networks, we use FRR and FRMs optimizing FRR as main examples. We chose a few of the existing FRMs and explain them. In Chiesa et al. 2021 you can find a more thorough survey of some of the existing FRMs.

Resilient Routing Layers

One core issue of FRR is that routes created by FRR are inherently agnostic. Routers depend their routing decision on no information other than the incoming interface and the destination. This limits routing options; each combination of incoming interface and destination network can have only one outgoing interface mapped. This can cause loops as seen in section 2.3.3, because each route has to be "checked" and return a packet for the next route to become active.
When solving the issue of sub-optimal paths on the controller however, the calculation and roll-out of a solution can take several seconds. Kvalbein et al. (2005) proposed an alternative solution.
Instead of using one routing for the network, each router would instead persist multiple routing tables, with each routing table belonging to a so called routing layer. All routing layers must be able to reach all entry and exit points of the network.
E.g. a link would be seen as "safe" if there is at least one layer in which the link is not included. This can be extended for devices such as routers. Routing layers protecting links can be seen in the example in fig. 2.3. While e.g. the link from R1 to R2 is included in routing layer 0, it is not included in routing layer 1. If a failure in this link would occur, the network could switch all routers to routing layer 1, circumventing



Routing Table for Router R1 (Only FRR)

Dest.	Inport	Outport
H2	H1	R2
H4	H1	R2
H4	H1	R3
H4	H2	R3

Routing Table for Router R1 (ShortCut in effect)

Dest.	Inport	Outport
H2	H1	R2
~~H4~~	~~H1~~	~~R2~~
H4	H1	R3
H4	H2	R3

(a) Network with FRR and a failure

(b) Network with FRR and a failure, optimized by ShortCut

Figure 2.4: Concept of ShortCut

the failure. This requires routers to manipulate packets and add an identifier to the packet, determining the routing layer that should be used.
Layers can be defined either manually or by algorithms taking configurable parameters. A more robust, low relevance networking section could receive a lower number of layers, while a high availability, high relevance networking section would receive more layers and therefore higher robustness against failures.
Because each router needs a routing table for each layer, the number of routing tables and therefore the allocated memory on the routers grows with the number of layers. If e.g. each router had a safe layer, the amount of routing tables on each router would be equal to the total amount of routers in the network.
The manipulation of packets limits the applicability of this method, as existing parts of the network might not be compatible.

ShortCut

The authors of ShortCut (Shukla and Foerster 2021) propose to remove loops created by FRR and therefore optimize the routes by self-editing existing flows on a router. In addition to the interface specific routing performed by FRR, ShortCut also uses this data to identify packets which were returned to the router. By maintaining a priority list of flows for each port they would then be able to remove invalid entries, e.g. links that failed or routes that returned a packet.
In fig. 2.4a you can see a network with pre-installed FRR routes on router R1, in this example an additional route for packets heading to H4 coming from router R2, and a failure in the link between routers R2 and R4. The returning packet from R2 will be routed, according to the routing table, to router R3. This results in an off-path or loop from R1 to R2, passing R1 twice. In fig. 2.4b the routes on router R1 for packets from H1 are edited. Because ShortCut recognized that packets to H4 return from R2, the route forwarding to router R2 is omitted from the routing table. As entries in a routing table are evaluated from top to bottom, the alternative route on router R1 will now always be used for packets to H4, effectively removing the loop.

This saves (1) the additional transmission times to router R2 and back, as well as (2) link capacity on the link between routers R1 and R2. ShortCut is applicable to most network topologies as well as pre-existing FRR and global convergence stacks.

Blink

Blink (Thomas Holterbach et al. 2019) interprets TCP flows and uses them as an indicator for failures occurring in the network. For this it uses TCP packets arriving at routers implementing blink, analysing a sample of these flows. If it recognizes that e.g. a packet was not acknowledged and this occurs multiple times, it signals a failure and re-routes traffic accordingly. This, however, requires the network to fail first, only restoring connectivity after packets were already lost.

Revive

Revive (Haque and Moyeen 2018) combines the usage of local backup routes with a memory management module, distributing traffic according to pre-defined memory thresholds on switches. All backup routes are implemented by installing flows on the routers/switches with an OpenFlow controller. It also uses OpenFlow's Fast Failover Groups (FFG) for detecting failures. The backup routes, however, are prone to create loops and the addition of re-routing using controller mechanisms is relatively slow, although much faster than a global convergence protocol.

2.4 Creating a test environment

Modern networks achieve high levels of complexity and flexibility to satisfy many different configurations and use cases. To evaluate a network and test new mechanisms and functions, the simulation of network technologies becomes necessary to confirm theoretical concepts.
While using real life components for such a simulation might be the most realistic approach, it quickly becomes infeasible especially in a scientific context, as it not only adds dependencies on different hardware vendors possibly distorting results, but also reduces replicability of said results. Furthermore it also complicates automation of measurements, increasing the overhead in testing.
Most networks already use some sort of virtualization, be it in form of *Virtual Local Area Networks* (VLAN), partitioning already existing physical networks into "virtual" networks, or the usage of virtual routers running on so called *white boxes*, eliminating the need for specialized hardware.
This eliminates some of the bridges between a virtual simulation and a real life network. While the network itself might not exist in the physical world, the software running on these virtual devices is already similar or even the same as in real life networks. In some cases configurations can even be shared between physical and virtual devices, allowing for fast real life confirmation of results.
As such most scientific work in regards to networks is in some way done on virtualized networks. It becomes possible to implement huge network structures and testing

setups on a single computer and allow each reader with a sufficiently powerful home computer to replicate the results.
In the following sections we will first take a look at Mininet (Lantz, Bob and the Mininet Contributors 2021), a network virtualization toolkit in section 2.4.1. We then go on to discuss measurement criteria for a computer network in section 2.4.2. Lastly we discuss methods of introducing failures in Mininet in section 2.4.3.

2.4.1 Mininet

Mininet (Lantz, Bob and the Mininet Contributors 2021) is a tool to create virtual networks running in linux based operating systems, with each component running its own linux kernel on a single system. This is done using network namespaces, which is a feature of the linux kernel allowing for independent network stacks with separate network devices, routes and firewall rules.
Mininet supplies a verbose management console which allows for the creation of virtual networks by using simple commands. These networks can then be evaluated, hosts can be accessed via a terminal and influenced in their behaviour, e.g. by editing their routing tables or implementing packet filtering on them, and pre-defined functions can be used to evaluate the performance of the network or monitor traffic.
Mininet also comes with a python API which can be used to create automatic scripts that will create host, routers, switches and controllers, connect them via links, set their routing configuration and execute functions on the network.
These scripts can be shared and used in Mininets provided virtual machine based on the Ubuntu distribution in several versions of Ubuntu.
Additionally it supports the OpenFlow protocol and the usage of controllers like POX, as well as the P4 language (Bosshart et al. 2014), which can be used to customize packet handling even more. Written controllers can then even be used in real life setups that replicate the created network in Mininet and retain their programmed functionality.
This allows the quick creation and configuration of test environments, as well as replicability of test scenarios, the creation of failure scenarios and the application in real life.
Mininet also provides support for limitations on created networks, which allows users to define throughput limits for links or added delays for packets passing certain links.
Its throughput is dependent on CPU capabilities, but because of its relatively small overhead, it was confirmed to be a realistic approximation for experiments for up to 1500 hosts in Muelas, Ramos, and Vergara (2018).
The virtual machine has most necessary tools pre-installed by default, making Mininet a good option for non-hardware performance testing and experimentation.
One important note about the usage of the Mininet VM is the configuration in *VirtualBox* (Oracle 2022), a virtualization software allowing the operation of virtual machines. The Mininet VM is provided as an image for this software. As such the developers behind Mininet already preconfigured the virtual machine for usage in *VirtualBox*. While implementing and evaluating our network we decided to allocate more host system resources to our VM, increasing the amount of cpu cores from initially one core to four cores. This however caused huge fluctuations in our test results, sometimes

limiting the bandwidth between hosts in a network to around 10% of their expected bandwidth. Only after using one cpu core again would the VM and Mininet function properly.

2.4.2 Measuring performance

When evaluating the performance of a network there are several quantitative and qualitative criteria to be considered, depending on the use case and the type of network. We explain the evaluation criteria in section 2.4.2. In the following sections we then explain possible ways to measure specific criteria. In section 2.4.3 we lastly explain how failures can be introduced and used in a test environment.

Evaluation criteria

The most basic usage of a network is the pure transfer of data without timing or flow constraints. Here, a faster transmission and therefore a higher bandwidth is the main criteria of evaluation. Protocols like TCP also resend dropped packages, so in these cases the bandwidth can also be used to make assumptions about the transfer error rate.
For traffic with timing constraints e.g. *Voice-over-IP* (VOIP) the bandwidth is bound by the number of concurrent transmissions and the quality of sound. In modern networks these transfers should only use a fraction of the available bandwidth. Additionally, in contrast to a TCP data transfer, dropping a few packages is only a minor inconvenience and would most likely not be noticed by the user.
Therefore the most recognizable criteria in VOIP is certainly the delay and the flow of packages. While a delay of a few milliseconds will not be instantly recognizable to the user, it can quickly result in a poor experience, especially if other factors like the usage of a wireless connection to a phone etc. are considered and delays accumulate. VOIP protocols mostly use a jitter buffer to collect packets in a certain time frame and reorder them according to their sequence numbers. This works very well if all packets were received in the correct order and without much delay. However, if a disruptive failure would occur chances are that additional delays and changed routings could cause the jitter buffer to be insufficient and the quality to suffer. As such, the evaluation of the packet flow and the order of transmission should be evaluated as well.
On-demand video streaming mostly uses the UDP and is not timing bound. High bandwidths are certainly required, but the lack of such will only be perceived when the bandwidth becomes unable to sustain a fluid playback of the video content. UDP does not resend dropped packages and a small count of missing packets is not a huge problem. A higher count of lost packets will however reduce quality and in some cases even cause a disruption of the service.
Up until now we only considered the bandwidth as the speed of data transfer from one device to another. In reality networks will forward thousands of data streams and existing links are rarely unused. If a data transfer would e.g. use a larger route through the network caused by a failure and that route contains a loop, the links inside this loop are unnecessarily strained and might influence other traffic in the network.

In summary, a non-specialized network is evaluated by it's bandwidth, link usage, latency, packet loss and packet flow.

Measuring bandwidth

Measuring bandwidth requires sending arbitrary data over the network. Tools like *iperf* allow for easy testing of bandwidth. The process of testing includes starting a server on a receiving device and starting a client on the sending device. Over a certain period data is then sent over the network and the client as well as the server will log the transfer rate by second.
After the pre defined time the transfers will stop and the client instance of *iperf* will shut down, after printing out the average transfer rate. The server instance will have to be shut down manually.
By default *iperf* will use TCP to send data, but when using the "-u" flag it will instead transfer data using UDP. When using UDP *iperf* requires an additional bandwidth parameter, which will specify how much data will be sent over the network. This is done because protocols like TCP use flow control to limit the amount of data sent on the capabilities of the receiving device. A slower device like a mobile phone will e.g. limit the data transfer to not get overwhelmed. Protocols like UDP do not provide any flow control and therefore *iperf* has to limit the used bandwidth itself. Using "-u 0" will cause *iperf* to send as many UDP packets as possible.

Measuring latency

Measuring latency can be done most easily by using the *ping* application included in most operating systems and sending multiple "pings" in a certain interval from a sending device to a receiving device. The sending device will send an *Internet Control Message Protocol* (ICMP) echo request packet to the receiving device over the network and if the packet was received, the receiving device will answer with an ICMP echo reply packet. After each sent packet, the sender will wait for the reply and log the time difference between sending and receiving.
A *ping* test can be run for a given time during which a failure can be introduced.

Measuring influence of link usage

One approach to measure the influence of two data flows on each other is to start two *iperf* measurements simultaneously, with one using host H1 as client and host H4 as server, and the other using host H2 as client and host H1 as server. When a failure is introduced, both data flows will pass the link between routers R1 and R2 in the same direction as can be seen in fig. 2.5. Traffic that is sent in opposing directions does not influence each other as ethernet uses a duplex connection.
This would cause the connection between router 1 and router 2 to be strained by both data transfers. They would most likely try to use the maximum amount of bandwidth they can acquire as they are unable to exactly specify which transfer should be prioritized or if they should split the available bandwidth equally.

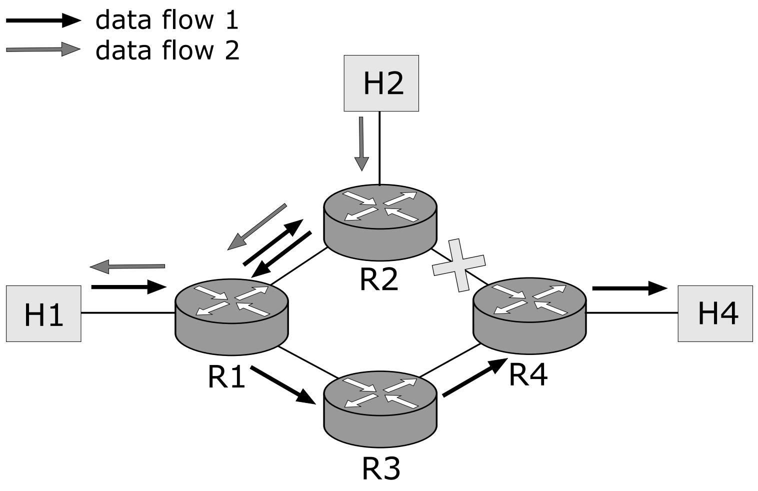


Figure 2.5: Link usage

Measuring packet loss

In a virtual network packet loss will not be caused by faulty devices on an otherwise unknown route as each component is assumed to work correctly. The main causes of packet loss in such a network would be either an overflow of packet queues on the routers, where full queues would cause additional packets to be dropped, or the introduction of failures and resulting waiting times from these configuration changes. In most networks packet queues should be configured according to the network requirements and will most likely be large enough to hold enough packets for the throughput that has to be achieved. This is why we focus on packet loss caused by failures.
We have to keep in mind that some of this packet loss might be caused by the configuration changes which are used to simulate a link failure in the virtual network. As such packet loss will most likely only occur shortly after a failure is introduced, or, in case UDP is used in a bandwidth measurement, if the bandwidth that has to be achieved overloads the packet queues of the routers.

Monitoring packet flow

Network mechanisms like ShortCut influence the routing and therefore the flow of packets in the network. To gain an overview over the routes packets take, we can analyse this packet flow by measuring the amount of packets belonging to a specific protocol, e.g.. This can be done with tools like *nftables*, which besides its functionality to create rules for packets acting as a firewall is also able to use these rules to deploy counters. These counters can be accessed in regular intervals to confirm if e.g. a FRM was successfully triggered and traffic is re-routed, skipping a router.

2.4.3 Introducing failures

In related studies failures are often split into definitions of **device failures** and **link failures**. Depending on the network structure link and device failures have different

impacts on the network. If e.g. a top of the rack switch fails, the whole rack loses connection to the network. A single link failure might only affect a single component. Nonetheless, a link failure and a device failure are not so different. While a link failure requires the network and its administrators to take different measures than if it was a device failure, the data plane of the network will react quite similarly.
Failures and measures taken by the network can be split into two phases. In the first phase a network component, be it a link or device, breaks. This can be either a failure in parts (e.g. specific packets dropped, protocols not working etc.) or a complete failure which constitutes a complete breakdown of functionality. The network uses timeouts and fault detection protocols on layer 1 (physical) and 2 (ethernet) to detect faulty interfaces and links. Most complete failures will be recognized near instantly, while partial failures might take longer to detect because they require e.g. timeouts to run out. These partial failures will most likely be caused by device failures, as link failures are less complex. Even a partially working link will most likely produce faulty packets which will be recognized quickly. The first phase contains all operations done until a failure was recognized.
When the failure was detected, the second phase starts. This is when the network becomes active and reacts to the failure, e.g. by using FRR to quickly reroute affected traffic flows and inquiring the controller to take further action. This is the phase of the failure in which the type of failure becomes near irrelevant for the operations on the data plane; the router recognizes a failing route, e.g. because the router behind a link is broken, and reacts accordingly. This reaction is very deterministic as only data on the router is used.
The FRMs in this work always presume clean failures that already have been detected. Fault detection in phase 1 is therefore excluded from this work and we always assume that a device or link has failed completely. Because we only focus on the fault mitigation on the data plane we can also assume that a faulty devices constitutes a device on which all links are broken. A failure in the only link connecting a router for example is the same as a completely broken router.
This is why we opt to simulate failures by deactivating interfaces on routers. The router will instantly recognize the error and routing table entries referring to the affected interface will be ignored. Every occurring fault and measurements will be limited to the fault mitigation phase.

3 Implementation

In the following chapter we implement an examplary network in Mininet, including routing between hosts and routers. We also implement fast re-routing as well as ShortCut. In section 3.1 we explain the test framework that we built for performing tests. In section 3.2 we then explain how we implemented FRR in the test framework. Lastly we talk about our implementation of ShortCut in section 3.3.
All implementations, this thesis and the test result can be accessed on our Git repository (Maaßen 05/2022).

3.1 Implementation of a testing framework

We describe the setup of a custom test framework which we will continue to use for the tests on Mininet networks. The framework is a collection of pre-configured networks, tests, routings and scripts, which can be used as a base for automatic tests. The framework is implemented in python 3.8.
To perform tests for multiple topologies and failure scenarios a structurized framework should be implemented. The core component is a main script called *mininet_controller*, responsible to perform all operations on the Mininet network using the Mininet python API.
Topologies can be created and added to the framework by creating a python module in the package "topologies", inheriting from the "CustomTopo" class described in *CustomTopo* in section 3.1.1. They then have to be referenced in the "topos" dictionary in the *mininet_controller* with the name of the topology as key. The corresponding value is a dictionary which in turn contains the class, the module name, i.e. the file name without extension containing the class, and a description of the topology as items.
Each topology implements a *build* function which is a predefined function for Mininet topologies and contains the setup of links, hosts and switches. When creating a Mininet network an object of the topology can be passed to Mininet, which will call the *build* function to create network components.
If routers should be used in a topology they can be added as nodes. This is done by calling the *addNode* function and passing it the "cls" parameter with our custom router class as value. The router class is described in *LinuxRouter* in section 3.1.1.

Each topology also provides its own routing, IP policies for FRR and tests, as each of these have to be manually adjusted for each topology. We use python dictionaries to store these configurations. The configuration of routings is described in section 3.1.2, the configuration for IP policies is described in section 3.1.3 and the configuration of tests is described in section 3.1.4.
The test framework already provides measurement commands which can be used in test configurations. These are described in section 3.1.5. Results from these measurements can be used to plot graphs using our plotting commands described in section 3.1.6.
Lastly we also provide users with a parameter based CLI, which we describe in section 3.1.7.

3.1.1 Base components

In the following sections we describe the two added base classes which extend the behaviour of existing Mininet classes for our needs.

LinuxRouter

Mininet does not provide a router class by default. This is easily circumvented as linux already provides tools to configure routing logic on a linux host or, in case of Mininet, on a Mininet *Node*. To make use of this functionality the IPv4 forwarding has to be activated using the *sysctl* utility.
Because this process would have to be repeated with every router that is created, the Mininet creators already provided an example for a *LinuxRouter* class, which will on instantiation execute the necessary command to enable the IP forwarding.
We use this example as our router class used for all routers implemented in the framework.

CustomTopo

Because each topology should provide some additional functionality we added a *CustomTopo* class. Every topology that should be used in our test framework needs to inherit from this class.
When instantiating an object from our class the user has the option to pass additional custom limit and delay parameters. These are set as attributes on the topology and will be used in the extended *addLink* function to set limits and delays on each link created in the topology. The user can, however, overwrite global limits and delays on links by providing his own configurations in his function calls of the *addLink* function in his topology.
In it we add some abstract functions for policy, test and routing configurations which should be extended by topology classes. In addition to this we added utility functions which will create dictionaries with e.g. routing tables mapped to interfaces which are used in the deployment and testing of Mininet networks.

3.1.2 Routing configuration

The configuration for routings is saved in a python dictionary. Each key specifies a router which has to match the names specified for the routers in the *build* method. This splits the configuration into dictionaries by router.

Each router can contain multiple routing tables specified in the key by name. This table name will be used to create additional routing tables automatically. Lastly, there exists a list of routings for each routing table. A routing is a 4-tuple/quadruple consisting of the route with subnet mask, a gateway address, an outgoing interface and the metric priority, in that particular order.

Each of these routings will be added automatically to their respective routing tables on the specified routers.

3.1.3 IP policy configuration

The configuration of IP policies for the implementation of FRR follows a similar approach to the routing configuration. It is a python dictionary split by routers as keys. Each router has a key "rules" which contains a list of dictionaries, each containing a specific rule.

A rule consists of a table it should be applied to, an incoming port and a list of destination addresses. For each destination address an entry will be added to the IP policies, redirecting traffic to the specified table.

3.1.4 Test configuration

The test configuration is a python dictionary with test names as keys. Each test is defined by a python dictionary, containing information about the test.

A test has two phases, the pre-execution phase and the execution phase. In the pre-execution phase a list of commands pre-defined by the Mininet controller can be used to prepare the network for a test. These commands can be specified under the key *pre_execution*. This was implemented because a network that was just created might have additional delays on the first execution of functions caused by e.g. *Address Resolution Protocol* (ARP) handling.

The execution phase contains the actual testing. A command can be executed and its results will be written to the Mininet console.

Additionally failures can be introduced to the network. The key *failures* contains a list of failures, defined by a type and a list of commands. The types of implemented failures are "intermediate" and "timer". Intermediate failures will be executed after a first run of the execute command, which will be repeated after the failure function was called. A failure that is of the type "timer" will start a timer in the beginning of the measurement, which will execute the defined command after a delay, which is provided through an attribute named "timing".

3.1.5 Implemented commands for measurements

A command can either be a lambda which is only dependent on the net, which will be passed into the lambda by default, or a function definition which is a tuple of the name of the function and a second nested x-tuple, depending on the function to call, containing all arguments that need to be passed to the function. The functions are defined in the *mininet_controller* and can't be called directly, because the topologies are loaded dynamically and will not know existing function definitions until they are loaded. Because of this the *mininet_controller* contains a dictionary called "functions" which has the function name as key and an attribute called "callable" containing a lambda with the function call using the provided arguments. In the following we list the existing functions and explain their functionality, which files are created and which output can be used for further testing and evaluation.

connection_shutdown

The function *connection_shutdown* deactivates one interface each on two specified Mininet hosts, effectively removing a connection between those two hosts. It can, however, be used to deactivate any interface on any device in Mininet and is oblivious to the network context and existing connections. This function will use a connection which is by definition a list with 2 elements, containing the names of the hosts/nodes which are linked together, a list with 2 elements containing the names of both components for printing and a list with 2 elements containing the interfaces which should be shut down. It will then access the components in the network and use the cmd function provided by Mininet to execute an "ifconfig down *interface*" on the component, which will cause the interface to be deactivated. This is done on both sides of the connection to make sure that each router will be able to recognize the missing connection instantly.

measure_bandwidth

The function *measure_bandwidth* is used to measure the bandwidth between two Mininet hosts in a Mininet network using *iperf*, logging and parsing results after the measurement. Results are then sent to one of the plotting functions described in section 3.1.6. This function will use a 2-element list of hosts, a 2-element list of IPs, a length parameter that defines how long the test will run in seconds, an interval parameter defining the interval between each log entry of *iperf*, a unique test name for naming a created graph or log entry, a graph title in case a graph should be created, a flag that defines whether *iperf* should use TCP or UDP as transfer protocol and a bandwidth to limit the transfer rate of *iperf*.
The command starts an *iperf* server. While experimenting we sometimes experienced unexpected behaviour causing tests to fail. There seemed to be an issue with the timing of the *iperf* server and client commands which were executed on the corresponding devices. Because the *iperf* server and client were started detached and the python script executed both commands directly one after another, the client seemed to try to connect to the server while the server was still in its startup process, denying the

connection. This is why we added an additional delay between server and client command execution.

measure_link_usage_bandwidth

The function *measure_link_usage_bandwidth* is used to start two separate *iperf* measurements between two host-pairs and will log and parse results of both measurements. The results are then passed to the multi-plotting function described in section 3.1.6.
The second *iperf* measurement will use the port 5202 instead of the default port 5201 in case two servers are started on the same device. The function reuses the *measure_bandwidth* function described in section 3.1.5. We call the measurement done by the *measure_bandwidth* function the "main" measurement, the additional measurement used to evaluate the influence of another file transfer on the network the "additional" measurement.
The measurements are configurable by providing an interval in which *iperf* will log results, as well as a length parameter to specify how long an *iperf* measurement should be run. While the main measurement is run for the exact specified time the additional measurement is run for a slightly longer time due to timing issues.
Because we reuse the *measure_bandwidth* function, the additional measurement is started with a delay. The *measure_bandwidth* functions introduces a sleep time of one second between the execution of the *iperf* server command and the client command, which would cause the additional measurement to be executed a second early to the main measurement. This is why we considered this additional second in the execution and the parsing process of the *iperf* output, executing the additional measurement for a longer period of time, omitting the entry for the first second and shifting all time values one second ahead. Doing this we create a log output that is nearly synced only adding the delay created by the Mininet and python overhead.
Both results are then passed to the multi-plotting function referenced in section 3.1.6, to create a plot containing both bandwidth measurements. The original graph created by the *measure_bandwidth* function is also saved. The combined graph receives the subtitle "_combined".

measure_latency

The function *measure_latency* is used to start a *ping* latency test between two hosts in a Mininet network. It will log the results, parse them and then pass them to the plotting function described in section 3.1.6. This function will use a sender element in the network, a destination IP, a length parameter that defines how many ping packets should be sent, an interval parameter defining the delay between each ping, a unique test name, a list containing the range for the y-axis of a created graph and a graph title which will be used in naming the files of tests.
After creating a test name out of the components of the measurement, including sender and destination, which latency measurement function was used (currently only ping is used), the defined unique test name, e.g. containing information whether the test was started before or after introducing a failure, as well as whether or not ShortCut was

used. It will then start a ping from the defined sender to the destination IP and log the results to a file in the "/tmp/" directory. The ping command is started in detached mode using the bash target "&". Output is directly written to a file using the ">" operator in the bash command. The python script will then wait for the corresponding time, adding some seconds to make sure all executions were fully run. After that, the produced ping output is parsed using bash tools, including *more*, *head* and *awk*, to create a single line command which will create a file with time-value pairs, separated by line.

The output file will then be extended by a zero value. All files created are saved in the "/tmp/" and can be used to further inspect results. They are named according to the test name. Files containing console output or parsed versions of said output use the file extension ".out".

After successfully parsing the output files the information about the test, including information about the test name is passed to the plotting function further explained in section 3.1.6, which will create a plot in the "/tmp/" directory under the same test name using the file type ".eps", which can be implemented directly in a LaTex document for documentation.

measure_packet_flow

This function will use a client and server parameter to start an *iperf* transfer and will implement packet counters using the filtering capabilities of *nftables*. All devices referenced in the flow measurement targets provided in the parameters of this function will receive a separate counter. Depending on the "flag" parameter they will count all packets entering the device which belong to the specified protocol.

When the packet counters are created information about them will be stored in a global packet counter memory. This memory is later used to access and read the values of the counters. The script saves the current state of packet counters globally in case the script is run again during the lifetime of the network to avoid the multiple implementation of the same counter and thus errors that could occur. The test uses this state to act depending whether or not the counters were already initialized. If no counters exist yet it will create them, if they were already created during runtime they are reset to a value of zero instead.

There are python libraries for reading *nftables* entries. Because an implementation of these would take additional time and the usage of network namespaces of Mininet adds some complexity to the configuration, we decided to instead manually check and parse the output of the *nftables* counter using python.

After starting a bandwidth test using *iperf* on the client device, the packet counters are started which will start a python thread for each of the measurement targets. In each of these threads the bash command for displaying counters is used to access the current count. The output is saved in python, parsed and then saved to a log file which is named after the device that is being logged, including the current time of execution in a fitting format for *gnuplot*.

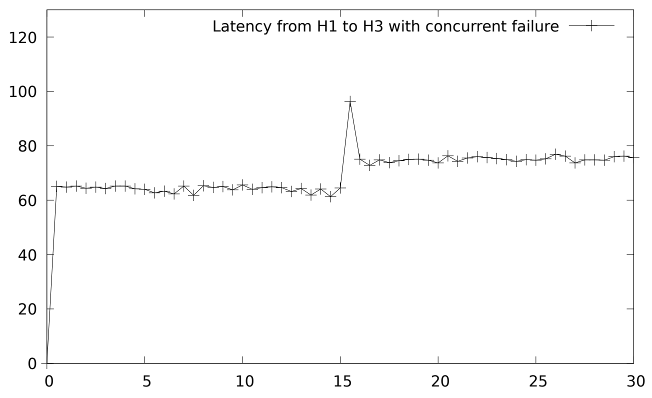


Figure 3.1: Exemplary latency test run with automatic plotting

After stopping the *iperf* server the created log files are passed as a dictionary with the corresponding label for the data to the multi-plotting function explained in section 3.1.6.

3.1.6 Plotting functions

Plotting results helps with visualizing differences, but performing a high number of tests and creating graphs manually quickly becomes tedious and will take a lot of time. This is why we used *gnuplot* to automatically create graphs. Each function that produces a log output containing results will parse its own results to only contain a time-value and a value, separated by a space, with additional values listed line-by-line. This data is then passed to *gnuplot*, which will produce an eps file containing a simple plot of our data. A test run with *iperf* would look like seen in figure 3.1. We use the "with linespoints" option, as well as the passed graph title for additional information on the plot. A call of the plotting function also allows to specify the labels for the x- and y-axis, as well as a range for the y values. If no range was provided the test framework will use the minimum and maximum values of the results instead.
In addition to plotting a single line in a graph we also implemented a function to plot multiple data files in *gnuplot* automatically. The function uses a greyscale as line colors and different dash styles for differentiating plots. It adds a defined label to each dataset. This can be used to e.g. plot the packet flow of multiple devices. A plot created with this method will look something like can be seen in fig. 3.2.

3.1.7 Command line interface

To make use of our test framework we added a *command line interface* (CLI), allowing the user to deploy existing topologies and run tests on them by providing arguments to the call of the *mininet_controller* script.
By calling the script using the "-h" or "–help" argument a list of possible arguments is displayed, including a description of the arguments. Using the argument "–topos" a list of all configured topologies, including a description, is displayed. When the name of a topology is provided to the "–topo [topology]" option, it will be used to create a

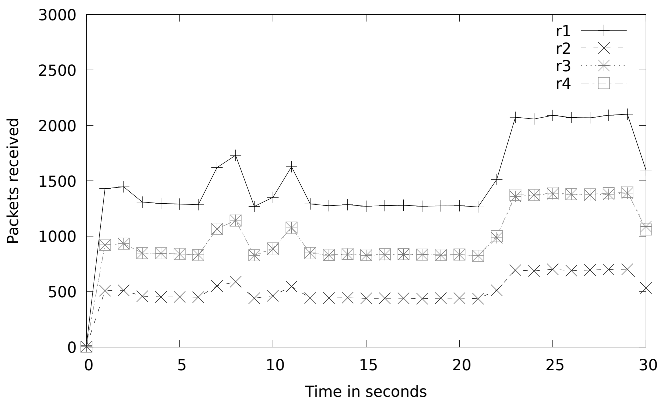


Figure 3.2: Exemplary packet flow test run with automatic plotting of multiple graphs in a figure

Mininet and will boot into the Mininet CLI. Using the "–topo [topology]" argument with a specific topology and adding the "–tests" argument will print all existing test names. When providing a specific topology and the argument "–test [test]" with a specific test name, the Mininet network is created according to the topology and the specified test is run according to the tests configuration. After the execution of the test finished, the network is shut down.
The argument "–limit_bw [bandwidth in Mbps]" can be used to limit the bandwidth on all links in the network by default, only using different bandwidth limitations on links for which another bandwidth limitation has been explicitly configured in the topology. This is also true for the argument "–delay [delay in ms]", which will add the specified delay per default to all created links.
When using the argument "–shortcut" the ShortCut mechanism is installed on the network.
Lastly we added arguments for the easy execution of multiple tests. The argument "–produce_set" will execute the with "–test [test]" specified test on the topology specified with "–topo [topo]" with and without our implementation of ShortCut. The argument "–full_suite" will execute a given test on all members of a pre-defined list of topologies as long as the test uses the same name in all included topologies. Each test will be repeated twice for each topology, once with and once without our implementation of ShortCut.

3.2 Fast Re-Routing

To implement FRR it is required to identify returning packets.
Depending on the target address and the interface of entry we can determine whether a packet is taking its regular route or if it is a returning packet. In case of a returning packet it would need to take an alternative route if possible.
In our simple network this would be the case in a scenario where a packet is going from H1 to H3, but the link between R2 and R4 would be unavailable. R2 then returns the packet to R1. The information that a packet to H3 should not normally be received on

the ethernet device linked to R2 can be used to re-route this packet by using additional routing tables, referenced by ip policy rules.
The function *ip rule* is part of *iproute2* (Alexey Kuznetsov 2011) and allows for the addition of policy rules that decide for each packet coming into the router which routing table to use, depending on the destination, source, incoming interface or outgoing interface. It also provides additional options and configurations, but in this context only using the destination and the incoming interface will suffice.
By specifying additional routing tables for each router and each input, and adding alternative routes, we effectively implement FRR behaviour.

3.3 ShortCut Implementation

ShortCut uses knowledge about the device it is run on as well as sent and received packets. It should then be able to manipulate routing table entries in case certain conditions are met, e.g. if a specific ip table entry was hit, using the data accessible.
The routing table manipulation has to take effect as fast as possible. Furthermore the performance impact of the ShortCut implementation has to be evaluated.
We begin by discussing possible options to detect failures, including the identification of returning packets in section 3.3.1. After we select the identification method using *nftables* we go on to explain the implementation in section 3.3.2.

3.3.1 Identifying failures

To determine which route should be deleted from the routing table, ShortCut has to gather knowledge about the packets forwarded by the router. The already implemented FRR explained in section 3.2 adds routing tables which will be used depending on the interface the packet was received on. These alternative routes are also added to the default routing table with a lower priority metric, in case the link directly connected to the router would fail.
To identify a packet that is returning and therefore a failure we already use the incoming interface when implementing FRR. If we would however be able to execute a function when such a packet is received, we would also be able to delete the old invalid routing table entry. For this there are several approaches which could be used.
The programming language P4 (Bosshart et al. 2014) can be used to write router logic and compile it so that a Mininet routers behaviour could be changed completely. This would also allow us to execute additional functionality in certain cases, e.g. if a specific ip route table entry is hit, but would require a manual implementation of core router functionalities like ARP handling. In the context of this work this is not feasible.
Another possible solution is the usage of low-complexity controllers, but to be able to accurately identify returning packets, the controller needs in depth knowledge about the routings and policy rules of a router. The easiest way to achieve this would be the implementation of subnets, routes and most of the logic inside the controller through flow table entries, but this would require a re-implementation of most of the routing logic as well.

A far more easily implemented solution is packet filtering, which allows us to use most of the pre-existing functionality and logic while still being able to react to certain packets. In most linux distributions this functionality is already supplied using the *nftables* package, which is mostly used for firewalls and therefore has the ability to specify rules for packets which will then be inserted to a multitude of targets, e.g. log files or a so called "netfilter queue".

3.3.2 Implementation using nftables

Netfilter tables (*nftables*) is a packet filtering tool for implementing firewalls in linux kernels and uses tables to store chains, which are a set of rules for incoming packets hooked to different parts of the network stack. These hook points are provided by the Netfilter kernel interface.
It is certainly possible to write software using these hooks directly, but in the scope of this work we will use *nftables* to expose packets using *nftables* rule definitions and their possible targets. The targets include log files in which information about a received packet can be stored. An outside software could then monitor said log files and react to entries fitting a certain rule. *nftables* even provides the option to write log prefixes to identify entries.
When trying this method it was quickly discovered that even though this would be achievable in a realistic environment with linux kernel based routers, using this method in a namespaced networking environment like Mininet is not possible because logging to the default log file, which is a system log file, is disabled in the kernel. This measure was taken because logging to a system log, which is shared by each networking component in a namespaced network, could cause the host system to suffer from a self-inflicted denial of service attack.
A far more sophisticated and also achievable approach in a namespaced networking environment is the usage of "netfilter queues". These were created for the purpose of exposing packets to a userspace software, which is exactly what we are trying to do. Netfilter queues can be numbered and are unique to their namespaced component, so that the netfilter queue 1 on router 1 will be different from the netfilter queue 1 on router 2. This is important because it saves us the manual distinction of netfilter queues. A packet hitting a netfilter table rule can then be added to a queue, which can be monitored by e.g. a python script. The python script would then be able to manipulate the ip route tables.

4 Testing

In this chapter we define and perform the tests for our evaluation. For this we first explain created topologies and their routings in section 4.1, followed by an explanation of our measurements which are taken of every topology in section 4.2. We then go on to explain the addition of failures, FRRs and FRMs to the tests in section 4.3.

4.1 Topologies and Routing

We define a set of topologies, which we later use in our tests. This includes a minimal network explained in section 4.1.1 and two failure path networks in section 4.1.2.

4.1.1 Minimum sized network

This network is used for tests in an environment that fulfils the minimum requirements for the implementation and reliable testing of a FRM. For this we determine the minimum requirements for a network when testing ShortCut. ShortCut removes loops from alternative paths selected by FRR after a failure was detected. To create a test case

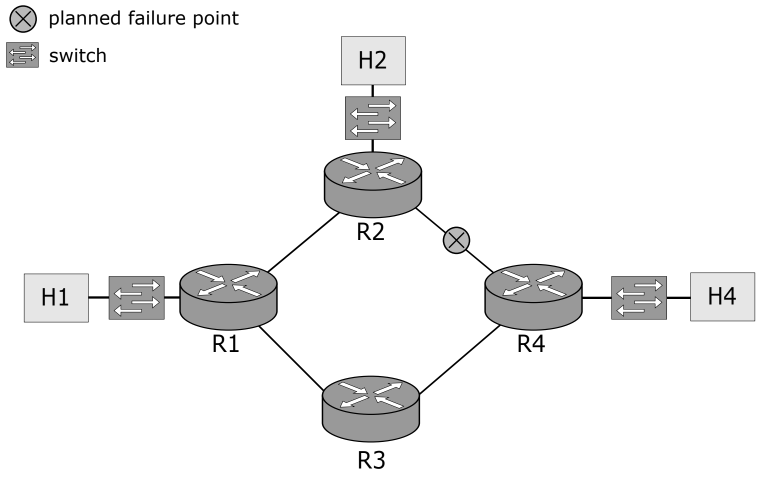


Figure 4.1: A minimal testing network using 4 routers, 3 switches and 3 hosts

in which ShortCut is able to optimize a FRR we need at least 2 paths/routes which are connected and functionally independent, going to the same destination.
Two routes are functionally independent if no link is shared between start and end point. The routes can be used as alternatives for each other if they share a common start and end point.
We also need 2 hosts which can be used for sending and receiving packets. In a virtual network environment like Mininet routers could also be used as sending and receiving devices. Because we want to stay comparable to real life examples, we decided to still use extra hosts. This is also the reason why we use switches on each router. In our example all hosts connected to a router share the same subnet.
These conditions can be satisfied by a topology containing three routers, two switches and two hosts. All routers are connected to each other and each router is connected to a switch connected to one host. The longest path is one hop, the shortest path is a direct connection to the destination router. To create a scenario in which a FRR route would reroute after a failure, one route between routers would have to be cut.
Suppose there are three routers A, B and C. They are connected to each other. If A wanted to send a packet to C, it could either use the direct link to C or send the packet to B, which would route them to C. A loop could only occur if A sent its packets to C taking a detour over B by default. By cutting the connection between B and C, packets leaving for C to B would return to A and then be sent over the direct link. The new path now contains an additional hop when compared to the original route, one to B and one back to A. Using ShortCut to remove the loop would now cause the network to use the direct path. In theory, the network will now add less delay to the transfer of packets than before the failure.
FRMs and other protocols that focus on reliability of networks were created to restore the original functionality of the network. Most networks will already use an optimized layout and configuration. As such a suboptimal route will not be chosen unless deemed necessary. Hence it is unlikely that in a real life network the alternative router chosen after a failure occurred is faster than the old route. Because of these concerns we decided to add a fourth router to the topology.
In addition to this we added a third and fourth host. This is optional, but when testing the added strain on the network caused by a second data flow passing a loop created by FRR, we want to add the option to use a real data flow instead of a simulated data flow as described in section 2.4.2.
The whole topology can be seen in fig. 4.1.

Routing

We combined this small network with a simple routing. All routers except for router R3 create three subnets: one host subnet and two subnets connected to the other routers. In case of router R3 only two subnets are needed as no host is connected.
The address of a subnet was chosen according to the router number. The subnet for hosts connected to e.g. router R1 is 10.1.0.0 with a subnet mask of 24 or 255.255.255.0, the gateway address is 10.1.0.1.

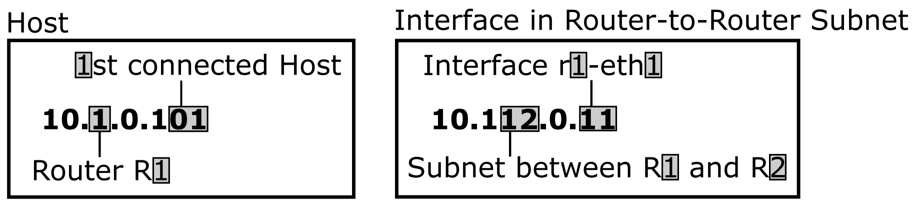


Figure 4.2: IP address selection of (a) a host address and (b) an interface address in a router to router subnet

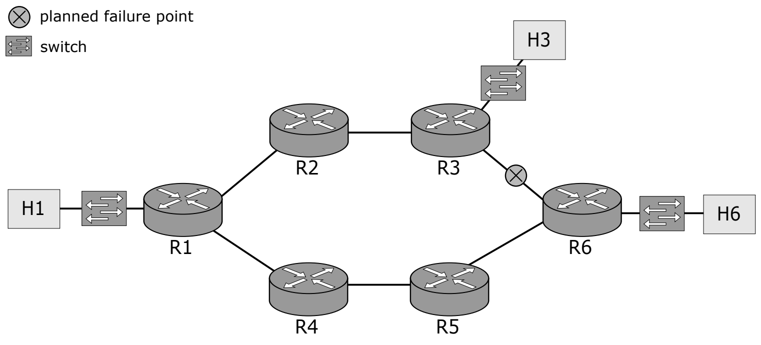


Figure 4.3: A network with a failure path length of 2 links and 2 routers

The subnets between routers were given IP addresses in a similar fashion as shown in fig. 4.2. The IP address for the subnet between routers R1 and R2 therefore is 10.112.0.0. When choosing addresses we wanted to be able to identify the router and the interface connected to such a subnet only by its IP. As only routers are part of the subnets between routers, and we have a very limited amount of routers, we added the router number, as well as the interface number to the IP address. We always attached all hosts to the first interface and counted fro

4.1.2 Failure path topologies

FRMs advantages include the removal of longer paths. In theory, these should reduce delays and strain on the network. To evaluate which consequences and advantages FRMs have, we use two topologies with a similar design to our minimal network, but additional hops on both paths. Both paths contain the same amount of hops to create comparable values.

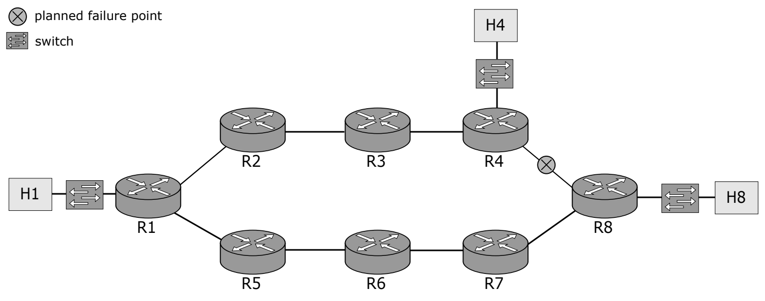


Figure 4.4: A network with a failure path length of 3 links and 3 routers

Routing

The routing is an extension of the routing in the minimal network. The naming convention shown in fig. 4.2 was used in both longer failure paths networks as well.

4.2 Measurements

To evaluate the performance of a network we established a list of criteria in section 2.4.2. Our measurements should reflect these criteria, which is why we implemented corresponding measurement functions in our test framework as described in section 3.1.5.
In the following sections we describe the implemented performance tests in detail, beginning with the bandwidth test in section 4.2.1. We go on to explain the process of our latency measurements in section 4.2.2. A test evaluating the influence of two concurrent data transfers on looped paths is explained in section 4.2.3. To evaluate the distribution of packets on the network while running a data transfer either with TCP or UDP we implement a test described in section 4.2.4.
An evaluation of the results of the tests described here will be given in chapter 5.

4.2.1 Bandwidth

The tests measuring bandwidth are one of the most basic tests in our testing framework. It uses the "measure_bandwidth" function described in *measure_bandwidth* in section 3.1.5.
As each virtual network that is created in Mininet starts with empty caches on all devices, we start a short *iperf* test on the Mininet network prior to each bandwidth measurement which will cause most network handling like packets sent by the ARP to be finished before the actual test starts. This reduces the impact these protocols and mechanisms would otherwise have on the first seconds of the bandwidth measurement.
The bandwidth tests are run using *iperf*. Each test is performed for 30 s with a log interval of 0.5 s.
For all of our bandwidth measurements, H1 is used as the *iperf* client. The server is chosen depending on the topology at hand which is H4 for the minimal topology described in section 4.1.1 and shown in fig. 4.1, H6 for the first network used to evaluate longer failure paths described in section 4.1.2 and shown in fig. 4.3, and H8 for the second network used to evaluate longer failure paths described in section 4.1.2 and shown in fig. 4.4.

4.2.2 Latency

Latency tests are run using the "measure_latency" function described in *measure_bandwidth* in section 3.1.5. Prior to every measurement a separate *ping* test is run between the hosts used for measurement. This is done to fill the caches of the routers, e.g. the ARP cache.
After this initialization the test is run for 30 s with a log interval of 0.5 s.

All latency tests were run with H1 as the sender. For each topology the receiving device changes, using H4 for the minimal topology described in section 4.1.1 and shown in fig. 4.1, H6 for the first network used to evaluate longer failure paths described in section 4.1.2 and shown in fig. 4.3, and H8 for the second network used to evaluate longer failure paths described in section 4.1.2 and shown in fig. 4.4.

4.2.3 Bandwidth link usage

This test uses two concurrent *iperf* measurements to measure the influence on the distribution of bandwidth between two data flows in the network while the actual routing of the packets is changed due to the introduction of a failure. It uses the function "measure_link_usage_bandwidth" described in *measure_link_usage_bandwidth* in section 3.1.5.
Before each test run a separate *iperf* measurement is run to fill caches on the routers, e.g. the ARP cache.
The test is run for 30 s with a log interval of 1 s.For the first running measurement host H1 is used as *iperf* client in all topologies, while the server changes depending on the topology, similar to the configuration in section 4.2.1
The second measurement however always uses host H1 as an *iperf* server, shifting the client depending on the topology. The client is always the host attached to the router on the top path nearest to the failure point, which is host H2 in the minimal topology described in section 4.1.1, host H3 in the first failure path network described in section 4.1.2 and host H4 in the second failure path network, which is also described in section 4.1.2.

4.2.4 Packet flow

The test uses the "measure_packet_flow" function described in *measure_packet_flow* in section 3.1.5.
For each topology we choose four routers for which the packet counters should be implemented. This will cover all routes packets could take as some routers are connected in series, forwarding each packet they receive.
There are only four possible different results for the number of packets forwarded in our networks. First is the router R1 in each topology, as this router is the entry point to our loop after introducing a failure. Router R1 is in the unique position that it will receive all packets sent over the loop twice, but will also receive all packets sent back to host H1, e.g. acknowledgements sent by TCP.
The end point of the created loops is our second point of interest. This is router R2 for our minimal topology, router R3 for the first failure path network and router R4 for our second failure path network. They will receive all packets passing the loop only once as they return them back to the sending router.
All routers in between our start and end point of the loop will forward each packet passed into the loop twice, once "upwards" and once "downwards" to the start point. As such each router in this position will also forward the exact same amount of packets. In case of our minimal topology there is no router that fits this case. Router R2 is the

only router in this position on our first failure path network. For our second failure path network we get to choose between routers R2 and R3. We choose router R2.
The fourth and final point of interest is a router on the alternative path, which will receive all packets in case of a failure. For our minimal topology this is either router R3 or R4 as both receive the same amount of packets. Because we only have four routers in our minimal topology, we just start counters on all of them. In the first failure path network we can choose between routers R4, R5 and R6 and in our second failure path network we can choose between routers R5, R6, R7 and R8. We choose router R5 for the first and router R7 for the second failure path network.
We execute a basic *iperf* bandwidth measurement prior to our packet flow measurement to fill caches.
All packet flow measurements are run for 30 s with a log interval of 1 s. We add an additional test using the same configuration but setting the flag in the *measure_packet_flow* function to "udp".

4.3 Failures, FRR and FRMs

Each test is run for each topology in two versions, one with an intermediate failure and one with a concurrent failure. Tests using an intermediate failure define two commands for measurement, one before a failure and one after a failure. The test will execute the first measurement, introduce the failure by running the "connection_shutdown" command described in *connection_shutdown* in section 3.1.5 and then run the second measurement automatically. In case of a concurrent failure the measurement is started and the failure is introduced during the run-time of the measurement.
Each of these tests is also run once only using FRR and once using FRR and our implementation of ShortCut.

5 Evaluation

In this chapter we evaluate tests that were run using our test framework in Mininet. The tests were performed as described in 4 with a bandwidth limit on each link of 100 Mbps. We use an additional delay of 5 ms per link in our latency tests - other tests do not use a delay.

The evaluations are sorted by topology. For each topology we measured the bandwidth, bandwidth with a concurrent data flow, latency, TCP packet flow and UDP packet flow. We execute each test once with FRR active in the corresponding section *With FRR* and once with FRR and our implementation of ShortCut active in the corresponding section *With FRR and ShortCut*.

We start with our minimal network in section 5.1, followed by the evaluation of two networks with longer "failure paths", measuring the influence of additional nodes in looped paths in section 5.2. Lastly we discuss our results in section 5.3.

5.1 Minimal network

In this section we will evaluate the tests performed on our minimal topology, starting with an evaluation of the bandwidth measurements in section 5.1.1. We continue to evaluate the tests measuring the influence of an additional data flow on the looped path in section 5.1.2. In section 5.1.3 we then evaluate our latency tests. Lastly we evaluate our packet flow measurements using a TCP transfer in section 5.1.4 and a UDP transfer in section 5.1.5.

5.1.1 Bandwidth

We performed multiple tests of influences to the bandwidth with occurring failures. These were run using *iperf* and a logging interval of 0.5 s. All data was collected from the output of the *iperf* server.

The first tests evaluated were run using only a FRR mechanism in *With FRR*. We then evaluate the tests run with a FRR mechanism and an implementation of ShortCut in *With FRR and ShortCut*.

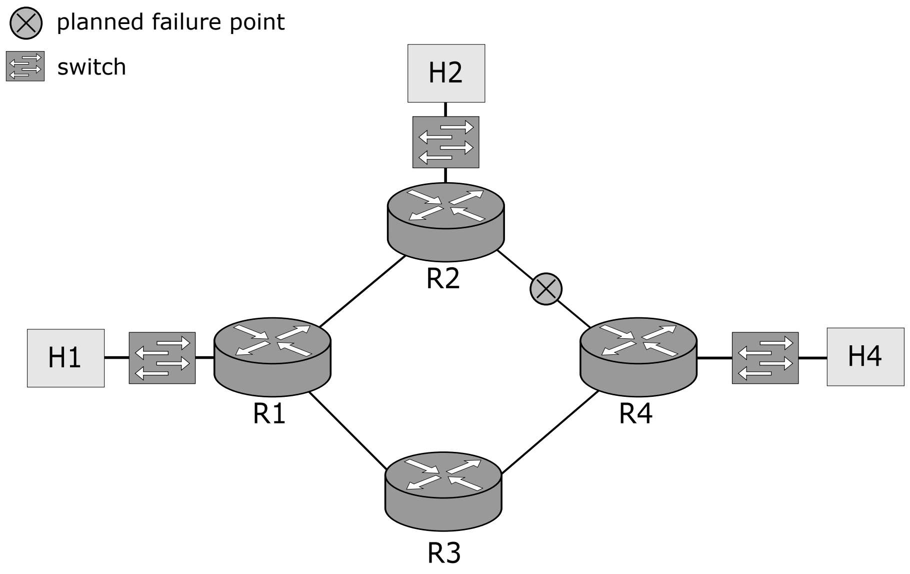
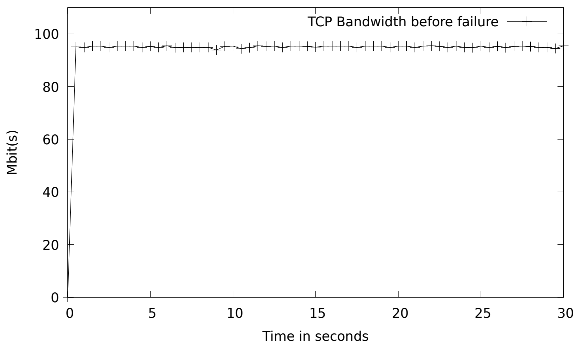
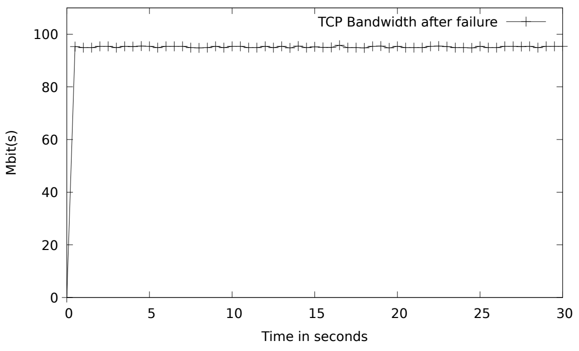


Figure 5.1: Minimal network



(a) Bandwidth before a failure



(b) Bandwidth after a failure

Figure 5.2: Bandwidth measured with *iperf* from H1 to H4

With FRR

We performed a TCP bandwidth test on the minimal network, a topology with 4 routers and 3 hosts. The failure occurred after the first run and before the second run of the test in fig. 5.2. The bandwidth does not change between runs. This is to be expected as additional hops on the path of the packet do not influence the total throughput that can be achieved, and while the looped path is passed by the traffic in both directions, the duplex nature of ethernet connections does not impose any limitations in this regard.

In fig. 5.3 however we introduced the failure while the bandwidth test was running. The test was run for 30 s and the failure was introduced at around 15 seconds, which caused no visible performance drop. However in some executions of this test the performance dropped when introducing the failure and the log output of the sending client reported the need to resend up to 100 packets. Because this behaviour is only occurring sporadically, we assume this to be a timing issue.

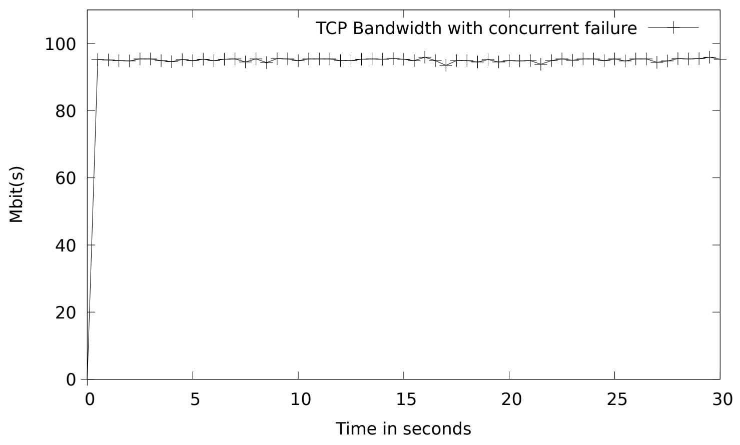
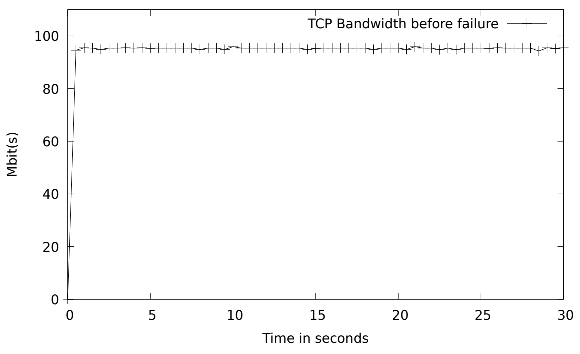


Figure 5.3: Bandwidth measured for 30 seconds, introducing a failure after 15 seconds

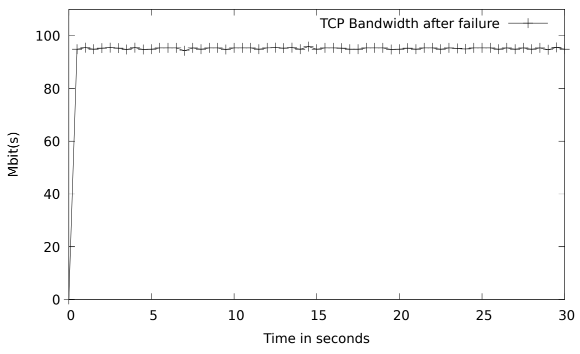
When the connection between routers is cut, our test framework uses the Mininet python API to deactivate the corresponding interfaces on both affected routers. This is done in sequence. In this example the interface on router R2 was deactivated first and the interface on router R4 was deactivated second. We implemented this behaviour after observing the default behaviour of the Mininet network. If the connection between e.g. router R2 and router R4 was only cut by deactivating the interface on router R4, router R2 would not recognize the failure and would loose all packets sent to the link. Because we deactivate the interfaces in sequence and the Mininet python api introduces delay to the operation, the interface on R2 will be deactivated while the interface on R4 will continue receiving packets already on the link and will continue sending packets to the deactivated interface on R2 for a short period of time. All packets sent to R2 in this time period will be lost. But because the *iperf* server itself does not send any actual data, but only *Acknowledgements* (ACK) for already received data, only ACKs are lost in the process.
TCP (Information Sciences Institute - University of Southern California 1981) however does not necessarily resend lost ACKs, and the client does not necessarily resend all packets for which he did not receive an ACK. Data for which the ACKs were lost could still be implicitly acknowledged by the server if they e.g. belonged to the same window as following packets and the ACKs for these packets were received by the client. This could cause a situation in which the server already received data, but the client only receives a notification of the success of the transfer with a delay.
This could cause some test runs to produce more packet loss than others, with most transfers experiencing no packet loss at all.
In our further tests we observed that the bandwidth alone does not change heavily when using different types of topologies. This is why we omit the evaluation of the bandwidth in further topologies.

With FRR and ShortCut

As can be seen in fig. 5.4 and fig. 5.5, using ShortCut had no further influence on the achieved throughput. This is to be expected, as longer or shorter paths will only



(a) Bandwidth before a failure



(b) Bandwidth after a failure

Figure 5.4: Bandwidth measured with *iperf* from H1 to H4 using ShortCut

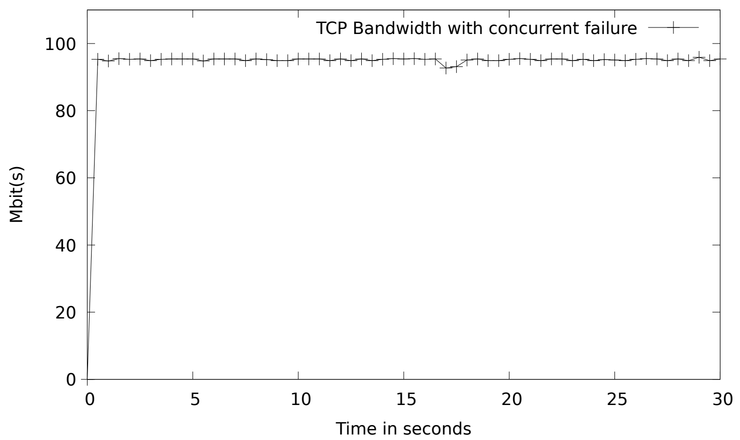
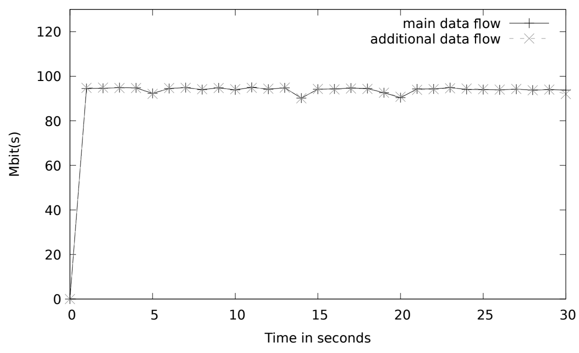
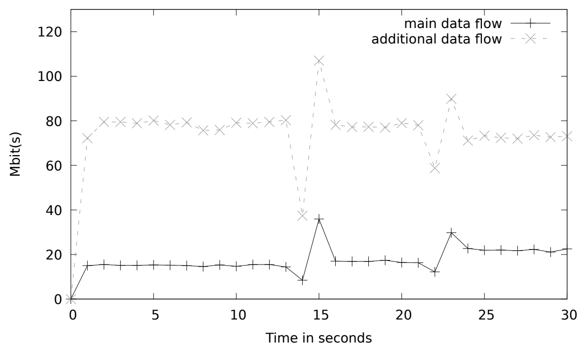


Figure 5.5: Bandwidth measured for 30 seconds, introducing a failure after 15 seconds using ShortCut



(a) Bandwidth before a failure



(b) Bandwidth after a failure

Figure 5.6: Bandwidth with concurrent data transfer on H2 to H1

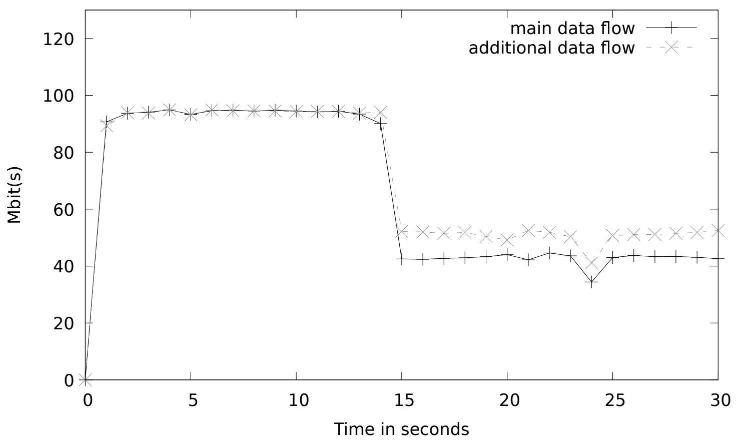


Figure 5.7: Bandwidth H1 to H4 with concurrent data transfer on H2 to H1 - failure occuring after 15 seconds

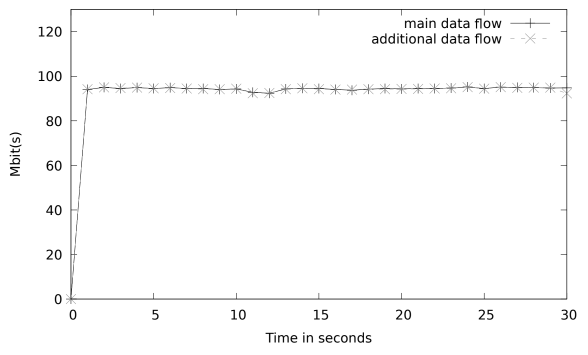
influence throughput if e.g. a link with a lower bandwidth is contained in an additional path.

5.1.2 Bandwidth with concurrent data flow

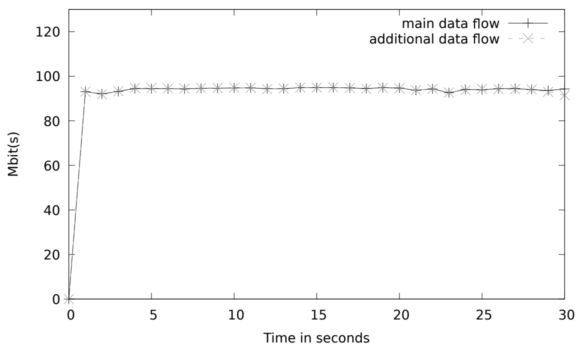
In this test we evaluated the bandwidth between hosts H1 and H4 with a concurrent data transfer between hosts H2 and H1. Both transfers were run with a limitation of 100 Mbps, which constitutes the maximum allowed bandwidth in this test.
The first tests evaluated were run using only a FRR mechanism in *With FRR*. We then evaluate the tests run with a FRR mechanism and an implementation of ShortCut in *With FRR and ShortCut*.

With FRR

Before a failure, as can be seen in fig. 5.6a, the throughput is at around 100 Mbps which is our current maximum. While the additional transfer between hosts H2 and H1 does in fact use some of the links that are also used in our *iperf* test, namely the link between routers R1 to R2 and host H1 to R1, it does so in a different direction.



(a) Bandwidth before a failure



(b) Bandwidth after a failure

Figure 5.8: Bandwidth with concurrent data transfer on H2 to H1 using ShortCut

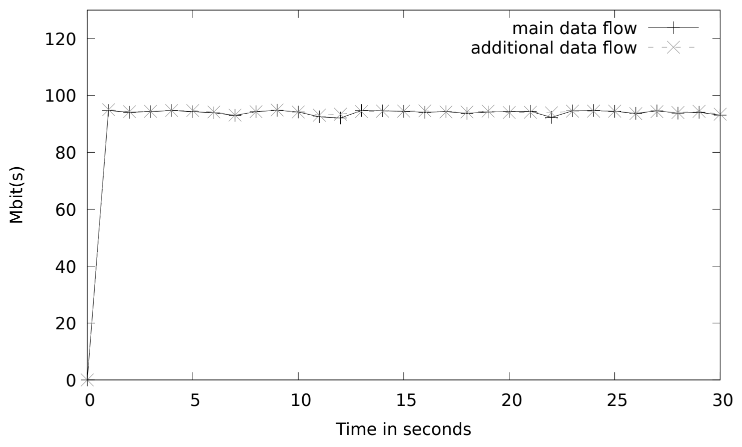
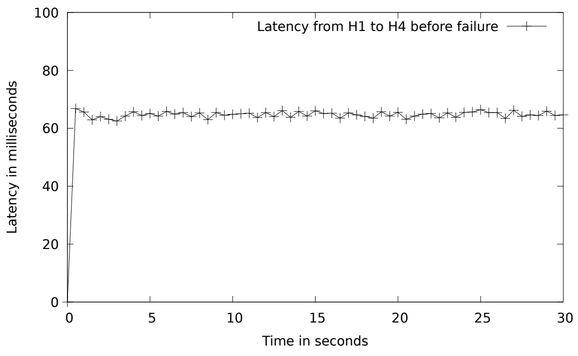


Figure 5.9: Bandwidth H1 to H4 with concurrent data transfer on H2 to H1 - failure occuring after 15 seconds using ShortCut

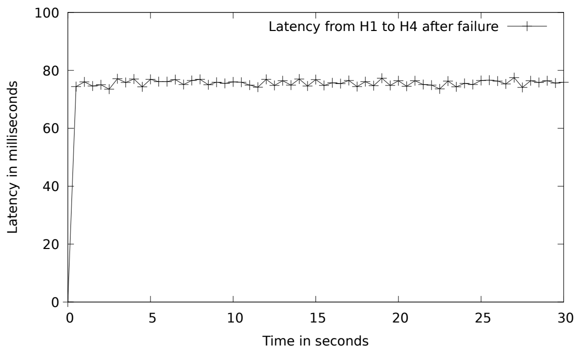
While the data itself is sent from hosts H1 to H4 over H2, only the TCP ACKs are sent on the route back. Data from hosts H2 to H1 is sent from routers R2 to R1 and therefore only the returning ACKs use the link in the same direction, not impacting the achieved throughput.
If a failure is introduced however, traffic from host H1 loops over router R2 using up bandwidth on a link that is also passed by the additional data flow. Therefore we experience a huge performance drop to around 20 Mbps to 30 Mbps, while the additional data flow drops in performance to around 80 Mbps as can be seen in fig. 5.6b. From a network perspective, this results in a combined loss of 50% throughput. While the amount of traffic sent through the network before the failure amounted to 200 Mbps, it now dropped down to a combined 100 Mbps.

With FRR and ShortCut

When activating our implementation of ShortCut no significant change of values can be observed. This is due to the removal of the looped path, effectively allowing both data transfers to run on full bandwidth. This completely restores the original traffic throughput achieved by both data transfers of 200 Mbps.



(a) Latency before a failure



(b) Latency after a failure

Figure 5.10: Latency measured with ping

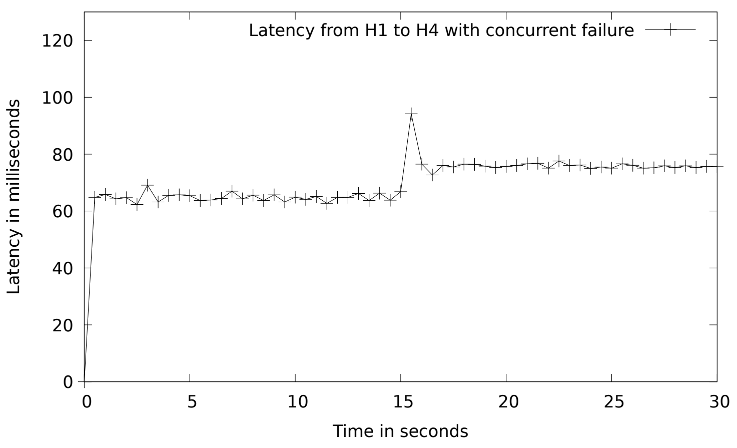


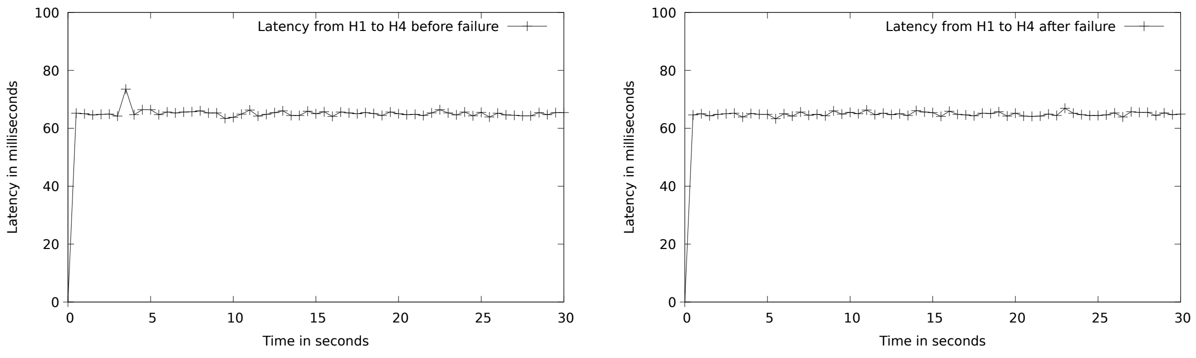
Figure 5.11: Latency with a concurrent failure after 15 seconds

5.1.3 Latency

In the following sections we evaluate the latency measurements run on the minimal topology with 4 routers and 3 hosts first with only FRR in section *With FRR* and then with our implementation of ShortCut running in section *With FRR and ShortCut*.

With FRR

As each link adds 5 ms of delay and *ping* logs the difference in time between sending a packet and receiving an answer, the approximate delay would be the amount of links passed N multiplied with the delay per link. In our test network there are 6 links between hosts H1 and H4. Because these links are passed twice, one time to host H4 and one time back to host H1, this results in an approximate delay of 60 ms.
The test run confirmed these assumptions. As can be seen in fig. 5.10a a ping on the network without failure took an average of around 65 ms with slight variations. The additional 5 ms are most likely caused in the routing process on the routers.
When introducing a failure however, additional links are passed on the way from host H1 to H4. Instead of 6 links passed per direction, the network now sends the packets on a sub-optimal path which adds 2 passed links from router R1 to R2 and back. These



(a) Latency before a failure

(b) Latency after a failure

Figure 5.12: Latency measured with ping using ShortCut

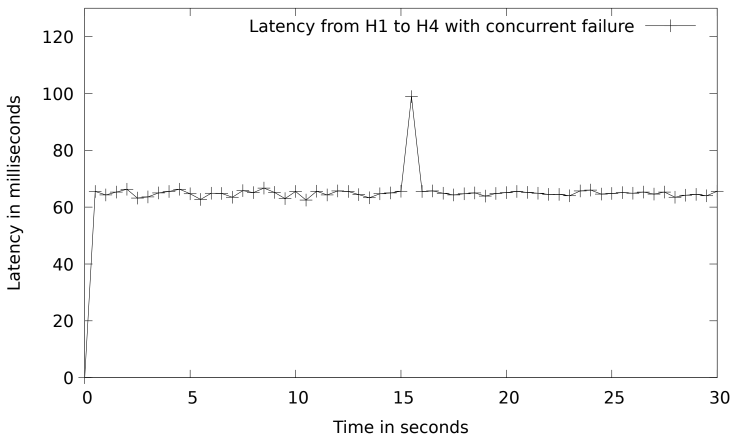
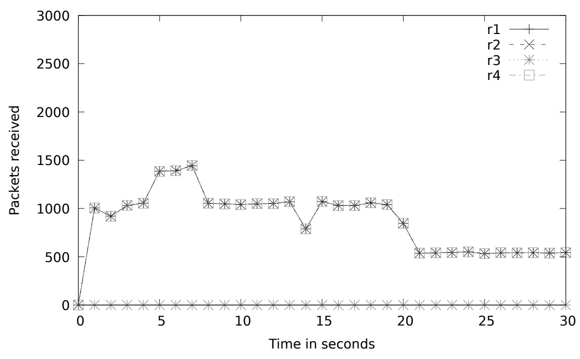


Figure 5.13: Latency with a concurrent failure after 15 seconds with ShortCut

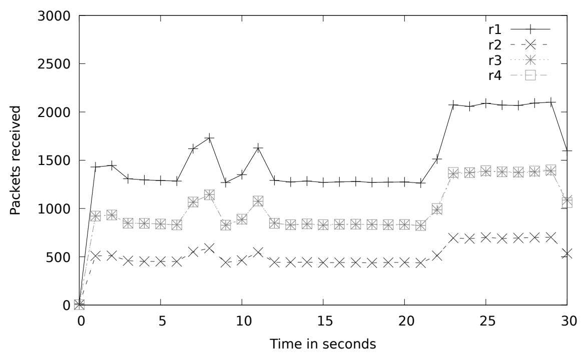
are only passed when sending packets to host H4, packets returning from H4 will not take the sub-optimal path. This would, in theory, add around 10 ms of delay to our original results.

As can be seen in fig. 5.10b this is also the case. With an average of around 76 ms of latency the results show an additional delay of around 11 ms when taking the sub-optimal path. The discrepancy between our assumption of 10 ms and the actual added 11 ms might be caused by the additional router that is passed in the direction to host H4.

When the failure is introduced concurrent to a running test, the latency spikes to around 94 ms for one packet as can be seen in fig. 5.11. This might be caused by the deactivation of interfaces using *ifconfig* and a packet arriving just at the moment of reconfiguration, as packets are sent every 0.5 s and the failure is introduced exactly 15 s after starting the measurement. Depending on the time *ifconfig* takes to reconfigure this would cause the packet to remain in the queue until the reconfiguration is finished, adding to the latency measured in this one instance.



(a) TCP packets flow before a failure



(b) TCP packets flow after a failure

Figure 5.14: TCP Packets on all routers measured with *nftables* counters

With FRR and ShortCut

Our implementation of ShortCut using *nftables* does not seem to add any additional delay to packet transfers as is evident when comparing the average delay produced before a failure in fig. 5.10a and fig. 5.12a.
When introducing the failure the latency does not change as can be seen in fig. 5.12. This is caused by the removal of the looped path and therefore the additional delay each packet would be subjected to.
The spike in latency which can be seen in fig. 5.13, occurring when the failure is introduced can be attributed to the same possible scenario as explained in section 5.1.3, which is most likely a timing issue with the introduction of the failure on the routers R2 and R4 and simultaneously sent ICMP packets.

5.1.4 Packet flow - TCP

To show the amount of TCP packets being forwarded on each router, we measured the packet flow on all routers of this topology. This is done by counting TCP packets with *nftables* while a concurrent data transfer is started from host H1 to H4. The results include the amount of packets forwarded on each router per second. This was done with an intermediate and concurrent failure for a network with FRR in *With FRR*, as well as a network with an additional implementation of ShortCut in *With FRR and Shortcut*.

With FRR

The results in the network before a failure are as to be expected and can be seen in fig. 5.14a. Each router on the route from host H1 to H4, which includes routers R1, R2 and R4, report the same amount of packets at each point of measurement. While the packet count fluctuates during the measurement no packet loss was reported and the bandwidth was at an average of 95 Mbps during the whole run of the test. This is why we assume that the fluctuations can be attributed to the mechanisms used in *iperf*.
After a failure all four routers receive packets as can be seen in fig. 5.14b, but router R1 now receives most packets with an average of around 1500 packets while routers R3

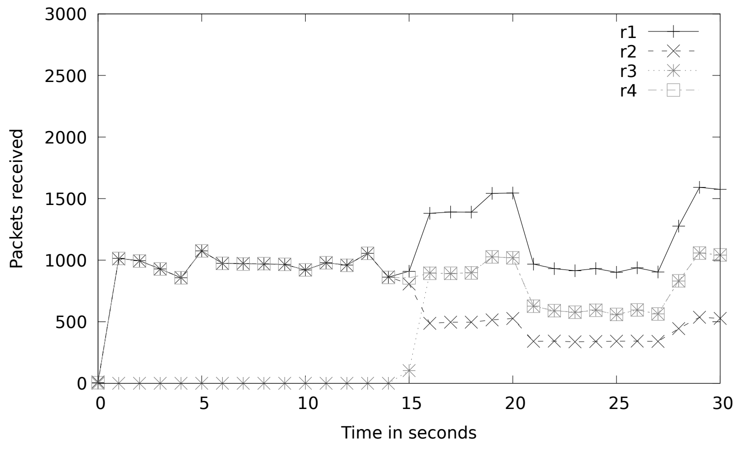


Figure 5.15: TCP packet flow on all routers with failure after 15 seconds

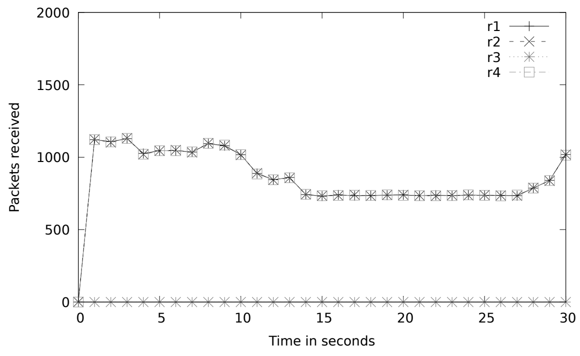
and R4 receive roughly the same amount of packets as before the failure at an average of around 1000 packets. Router R2 receives the least packets with an average of around 500 packets.

This is most likely caused by the looped path and the implications for packet travel this has. Router R1 receives all packets that are sent to host H4 from H1 twice, once sending them to router R2 and the second time when receiving the packets back from router R2 to send them to router R3. But while all packets sent from host H1 pass router R1 twice, ACKs sent back by the *iperf* server on H4 will only pass R1 once, as R1 would not send packets with H1 as destination to R2. Router R2 on the other hand only receives packets sent to H4 but none of the ACKs sent back. This is why, when compared to the average packet count of all routers in fig. 5.14a, R2 receives roughly half of all packets a router would normally receive as TCP specifies that for each received packet TCP will send an ACK as answer. This also explains why router R1 forwards an average of around 1500 packets per second, forwarding data packets with around 500 packets per second twice and forwarding acknowledgement packets once with also 500 packets per second, producing an additional 50% load on the router. Aside from the changed path and therefore the inclusion of router R3 in this path, routers R3 and R4 are unaffected by the failure, forwarding each packet once.

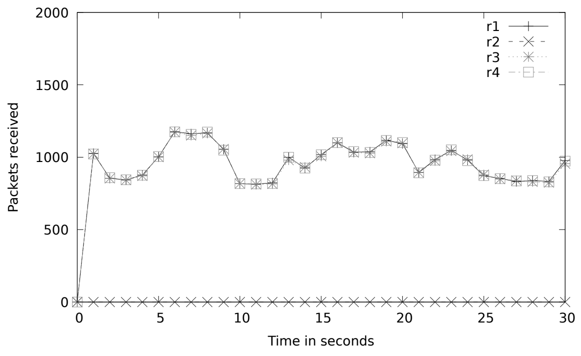
When causing a failure while the bandwidth measurement is running, the failure itself will cause a sudden drop to 0 packets forwarded for a short amount of time. This can be attributed to the time the routers take to change their configuration. The *nftables* counter uses the "forward" netfilter hook which is called in a pre-defined phase in the network stack of linux. Packets which are logged in the forwarding state already received a routing decision, but because Mininet needs some time to reconfigure the interface to shut down a connection, imitating a failure, the packets have to wait for the router to be ready again.

This behaviour has also been observed when measuring the latency and introducing a failure concurrently in fig. 5.11, adding delay to packets to be delivered in the moment of failure.

Reconfiguration of routers in Mininet does not reset the *nftables* counters either, which was confirmed in a quick test counting packets of an *iperf* transfer and shutting down



(a) TCP packet flow before a failure



(b) TCP packet flow after a failure

Figure 5.16: TCP Packets on all routers measured with *nftables* counters using Shortcut

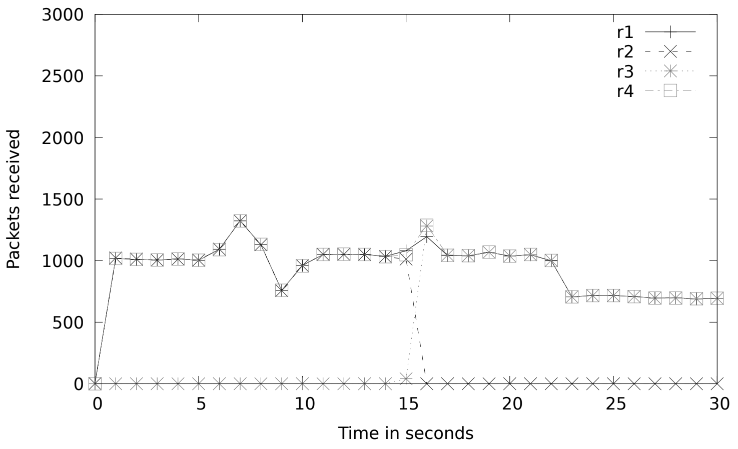


Figure 5.17: TCP packet flow on all routers with failure after 15 seconds using ShortCut

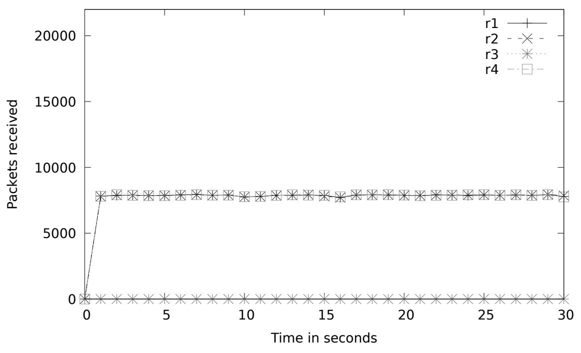
an interface on the same router. The packet count did not change after shutting down the interface.

With FRR and ShortCut

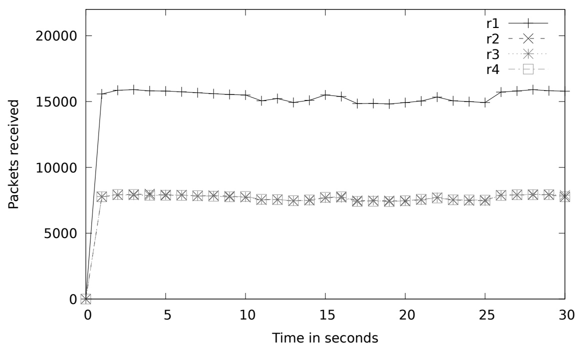
When running the TCP packet flow measurements with an implementation of ShortCut running on the network however, the results change drastically, and as expected all packets sent by the *iperf* transfer are forwarded by router R2 on the original route. After the failure was introduced the router R2 does not forward any packets. ShortCut has effectively cut out router R2 from the route, forwarding packets from router R1 to R3 directly. All remaining routers R1, R3 and R4 now receive all packets and no router forwards any packets twice.

5.1.5 Packet flow - UDP

We repeated the packet flow test in section 5.1.4 using UDP to inspect the differences caused by the two protocols.



(a) UDP packet before a failure



(b) UDP packet flow after a failure

Figure 5.18: UDP packet flow on all routers measured with *nftables* counters

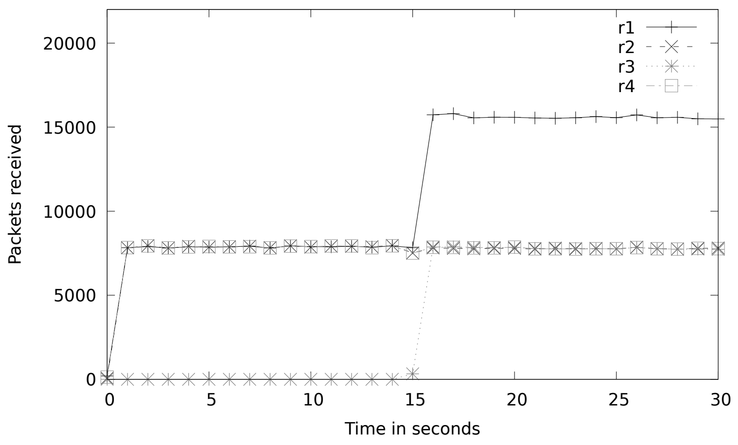
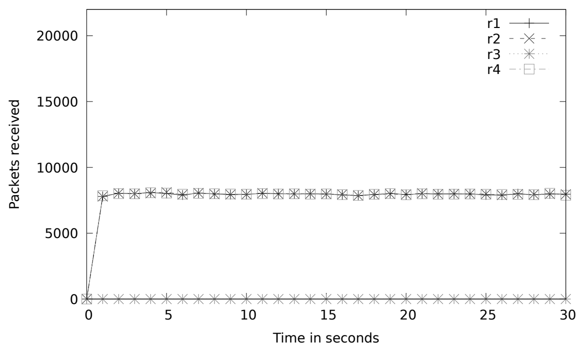


Figure 5.19: UDP Packet flow on all routers with failure after 15 seconds

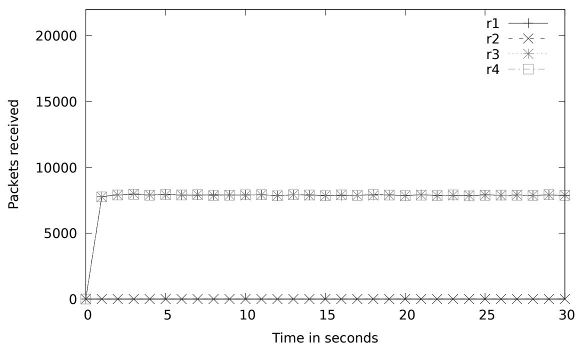
The first tests evaluated were run using only a FRR mechanism in *With FRR*. We then evaluate the tests run with a FRR mechanism and an implementation of ShortCut in *With FRR and ShortCut*.

With FRR

When running the packet flow test measuring UDP packets the amount of packets was much higher compared to TCP packets. *iperf* uses different packet sizes for each protocol, sending TCP packet with a size of 128 kB and UDP packets with only a size of 8 kB (Dugan et al. 2016). The same amount of data transmitted should therefore produce a packet count roughly 16 times higher when using UDP compared to TCP. TCP however, as can be seen in fig. 5.14a, causes the routers to log around 1000 packets per second when running a bandwidth measurement limited by the overall bandwidth limit on the network of 100 $\mathrm{Mbit\,s^{-1}}$. A naive assumption would be that UDP should sent 16000 packets per second over the network, but that does match with our test results seen in fig. 5.18a, where only around 7800 packets per second are logged on the routers.
The reason for this is also the key difference between TCP and UDP: TCP uses ACKs to confirm the transmission of packets. These are packets returning from the *iperf*



(a) UDP packet flow before a failure



(b) UDP packet flow after a failure

Figure 5.20: UDP packet flow on all routers measured with *nftables* counters using ShortCut

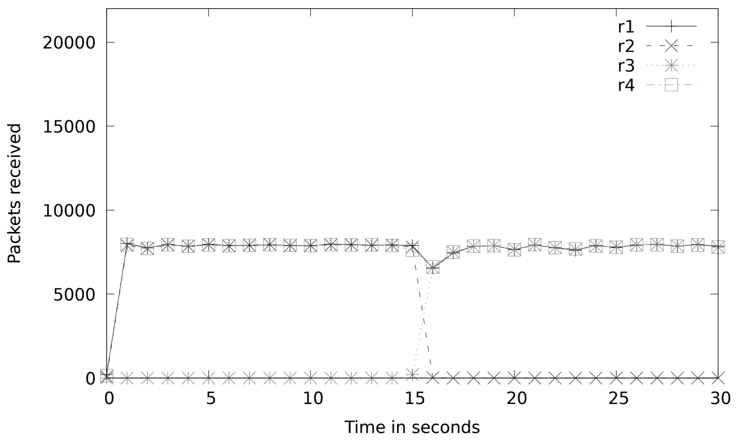
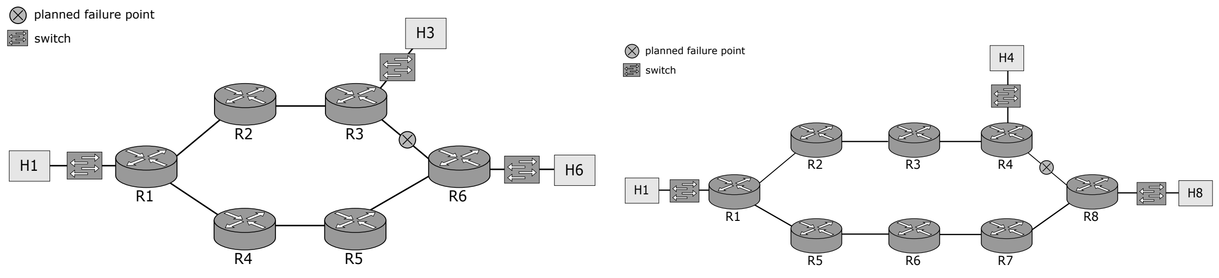


Figure 5.21: UDP packet flow on all routers with failure after 15 seconds using ShortCut

server to the client. For each received data package, the server will send back an ACK. If no ACK is sent for a packet, the client will resend the missing packet. This causes the network to transmit twice the amount of packets, one half containing the actual data and one half only containing ACKs. UDP however will just blindly send packets on their way and does not evaluate whether they actually reached their destination. Therefore all UDP packets contain data and no additional packets for confirmation or congestion control etc. are sent over the network.
Because UDP does not send ACKs the results observed after a failure in fig. 5.18b are very telling, with routers R2, R3 and R4 all forwarding the same amount of packets and router R1 forwarding exactly double the amount of packets.

With FRR and ShortCut

When using ShortCut in a UDP packet flow measurement, the negative consequences of the failure disappear. While in fig. 5.18a routers R1, R2 and R4 receive all packets on the original route, the load switches after a failure from router R2 to R3. As expected, the ShortCut implementation has cut out the looped path and restored the original functionality on an alternative route.



(a) Failure path with 2 hops (b) Failure path with 3 hops

Figure 5.22: Networks with longer failure paths

5.2 Failure path networks

In this section we evaluate the results for our two failure path networks seen in fig. 5.22. The networks were created so that the additional hops on a looped path would be longer, effectively simulating a more severe failure in terms of affected routes.

Most tests however did not produce significantly different results to the minimal network, evaluated in section 5.1, which is why we will focus on differences between the two topology classes.

In section 5.2.1 we evaluate the influence of longer failure paths on achieved bandwidths in the network. Furthermore we investigate the influence of two concurrent data flows on the looped path in section 5.2.2. The impact of a longer looped path on the latency in the networks is evaluated in section 5.2.3. Lastly we inspect packet flow in our failure path networks using a TCP transfer in section 5.2.4 and using an UDP transfer in section 5.2.5.

5.2.1 Bandwidth

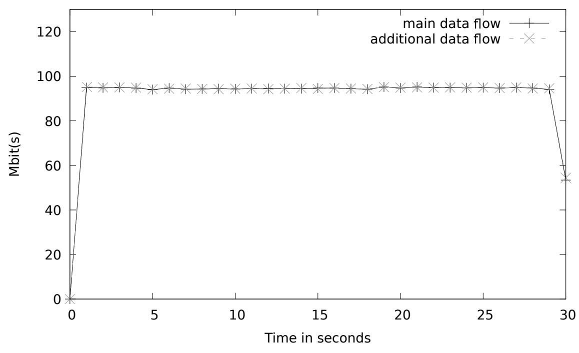
When measuring the bandwidth of our networks with longer failure paths the results were similar to those of the bandwidth measurement in the minimal network, described in section 5.1.1.

The addition of hops to the failure path did not have an effect on the bandwidth. It has to be noted however that the packets sent through the looped path will potentially block other data flows, reducing the overall bandwidth of both data flows. This is evaluated in section 5.2.2.

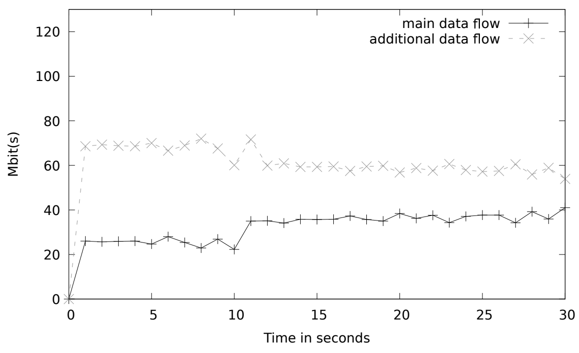
5.2.2 Bandwidth with concurrent data flow

We started two concurrent data flows using *iperf*. In case of a failure, these two data flows would influence each other.

The first tests evaluated were run using only a FRR mechanism in *With FRR*. We then evaluate the tests run with a FRR mechanism and an implementation of ShortCut in *With FRR and ShortCut*.



(a) Bandwidth before a failure



(b) Bandwidth after a failure

Figure 5.23: Bandwidth with concurrent data transfer on H3 to H1

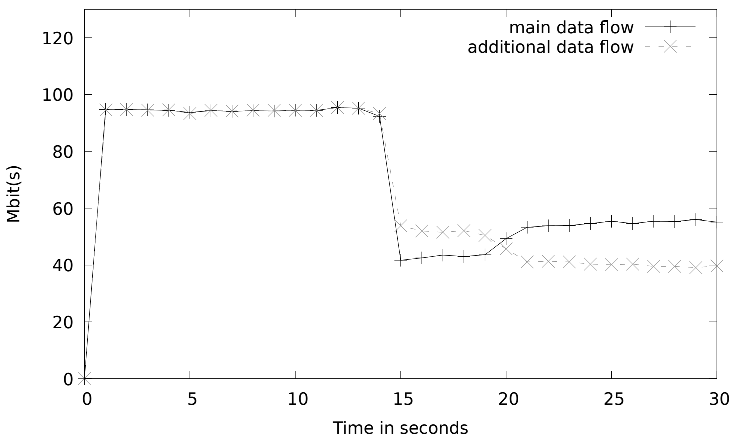
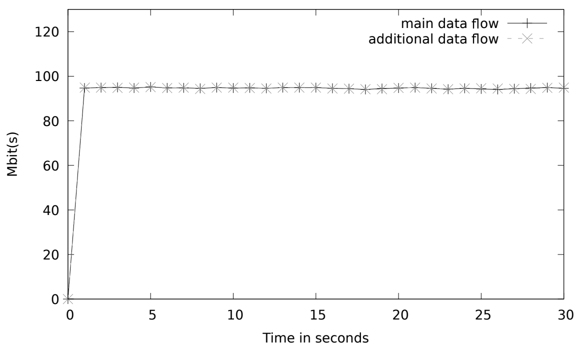


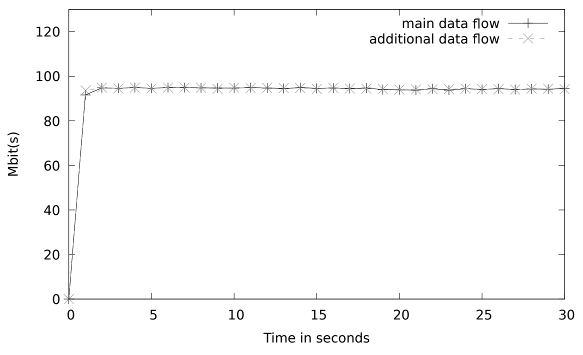
Figure 5.24: Bandwidth H1 to H6 with concurrent data transfer on H3 to H1 - failure occuring after 15 seconds

With FRR

Similar to the the results for our minimal network in section 5.1.2, the addition of a second measurement running concurrently on the looped path does not reduce throughput for both data flows as can be seen in fig. 5.23a.
When introducing a failure however the two data flows use all links from router R3 to router R1 simultaneously. They effectively have to split the available bandwidth, reducing the overall throughput by 50% as can be seen in fig. 5.23b, but there is no impact on the available bandwidth unique to the longer failure paths. This is why we only added the graphs for the smaller variant shown in fig. 5.22a.
Introducing the failure concurrently to the data transfer causes both bandwidths to abruptly drop, which can be seen in fig. 5.24. Although the two data flows distribute the bandwidth differently, they achieve an overall throughput of 100 Mbps. We assume that the incoherent distribution of bandwidth is caused by the timing of the data transfers.
The data transfer in fig. 5.23b already starts with the failure in place. Because the *iperf* instance producing the additional data flow is started slightly before our main data flow, Mininet seems to allocate more bandwidth to this transfer. The graph in fig. 5.23b



(a) Bandwidth after failure - 1st network



(b) Bandwidth after failure - 2nd network

Figure 5.25: Bandwidth with concurrent data transfer using ShortCut

also suggests that both bandwidths approximate each other, suggesting that Mininet tries to, over time, allocate both transfers the same bandwidth.
Our measurement with a failure occurring concurrent to our data transfers however evens the playing field. Both *iperf* instances already send data over the network. This could explain the overall more evenly distributed bandwidth, as well as the main data flow even overtaking the additional data flow.

With FRR and ShortCut

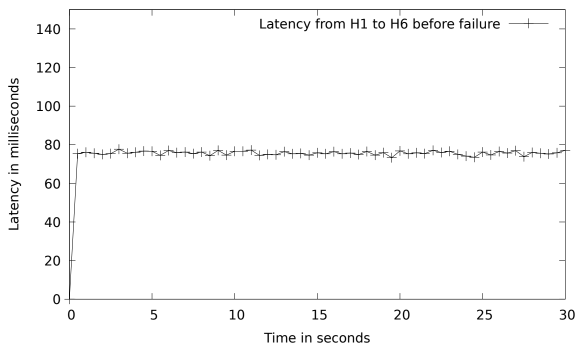
Using ShortCut allows both topologies to restore the full bandwidth for both transmissions, as the looped path is cut, causing the data transfers to not interfere with each other.

5.2.3 Latency

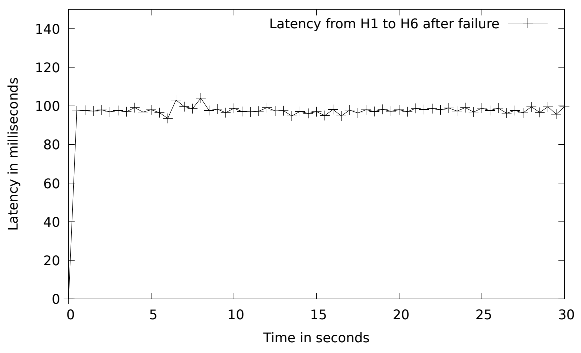
We measured the latency between host H1 and H6 for our first failure path network and between host H1 and H8 for our second failure path network.
The first tests evaluated were run using only a FRR mechanism in *With FRR*. We then evaluate the tests run with a FRR mechanism and an implementation of ShortCut in *With FRR and ShortCut*.

With FRR

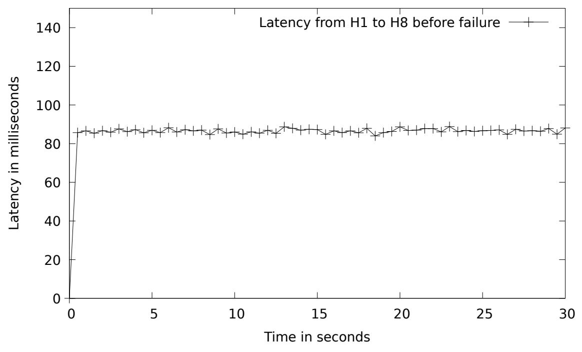
The additional hops in our failure path networks add, as expected, latency to the measurements, as each Mininet link in our tests add a delay of 5 ms. When compared to our minimal network evaluated in section 5.1.3 the first failure path network adds exactly 5 ms, shown in fig. 5.26a for our additional hop. The second failure path network adds an extra 5 ms on top, which can be seen in fig. 5.26c.
When introducing a failure our first failure path network adds around 20 ms of latency as can be seen in fig. 5.26b. fig. 5.26d shows that the second failure path network adds an additional 10 ms to the latency in case of a failure, adding 30 ms in total. This is caused by the additional links on the longer path. Because only ICMP echo requests



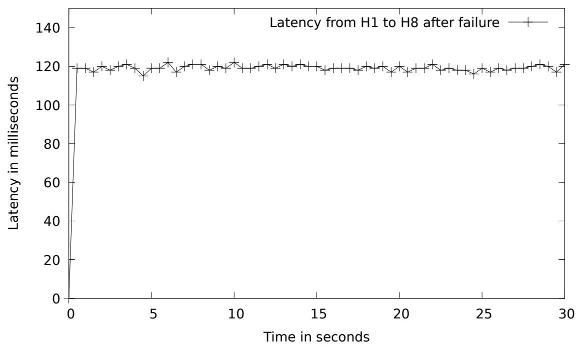
(a) Latency before a failure - 1st network



(b) Latency after a failure - 1st network

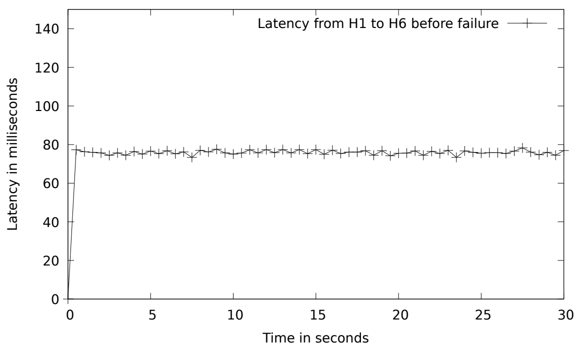


(c) Latency before a failure - 2nd network

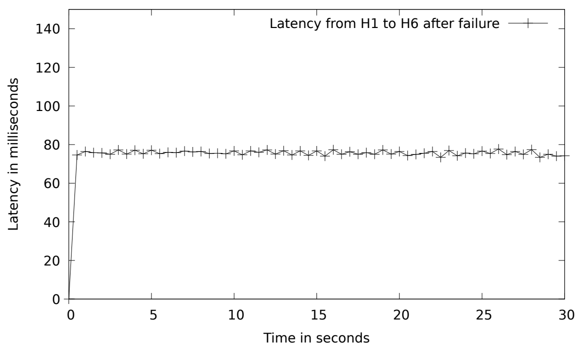


(d) Latency after a failure - 2nd network

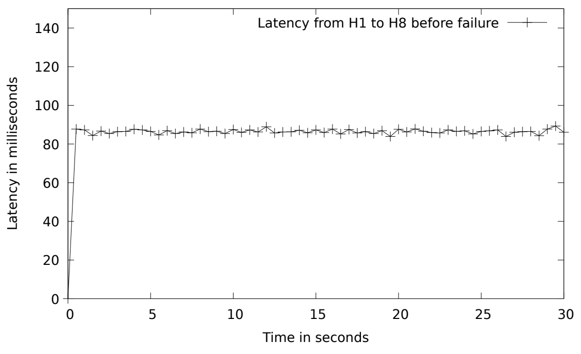
Figure 5.26: Latency measured with *ping* on both failure path networks



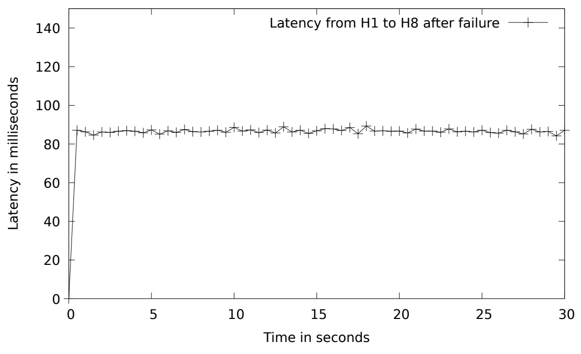
(a) Latency before failure on 1st failure path network using ShortCut



(b) Latency after failure on 1st failure path network using ShortCut



(c) Latency before failure on 2nd failure path network using ShortCut



(d) Latency after failure on 2nd failure path network using ShortCut

Figure 5.27: Latency measured with *ping* on both failure path networks using ShortCut

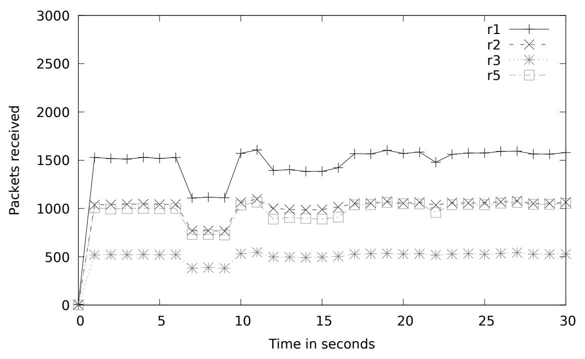
and not replies use the looped path, as packets returning from either host H6 or H8 are not forwarded to router R2 when arriving on router R1, the additional latency on the network will always be 10 ms for each link contained on the looped path. The additional link is passed twice by each ICMP echo request.

With FRR and ShortCut

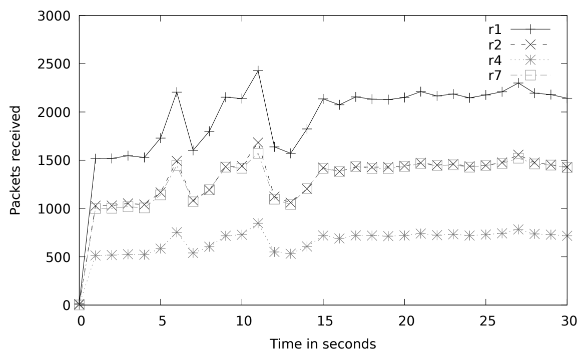
Similar to our results when measuring the minimal topology in section 5.1.3, ShortCut is able to restore the original latency after a failure, independent of the length of the cut looped path, as can be seen in fig. 5.27.

5.2.4 Packet flow - TCP

We measure the amount of TCP packets forwarded in our failure path networks. For this we attach *nftables* counters to for routers in each of these topologies.
The first tests evaluated were run using only a FRR mechanism in *With FRR*. We then evaluate the tests run with a FRR mechanism and an implementation of ShortCut in *With FRR and ShortCut*.



(a) TCP packet flow after a failure - 1st network



(b) TCP packet flow after a failure - 2nd network

Figure 5.28: TCP Packets on all routers measured with *nftables* counters

With FRR

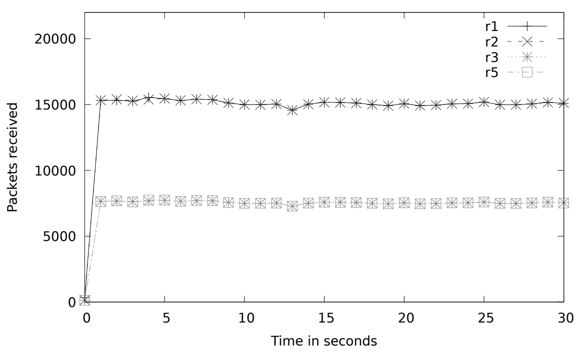
Similar to our observations in section 5.1.4 only TCP packets bearing data are passed into the loop, as returning ACKs from the *iperf* server are forwarded to the host H1 directly. Router R1 however forwards an additional 50% of packets, forwarding packets containing data twice on their way to the *iperf* server and forwarding returning ACKs once back to the host H1. The router R3 in our first failure path network and the router R4 in our second failure path network both only forward each TCP packet bearing data once. Both of these results are shown in fig. 5.28.
The implications for the network are however quite more severe the more routers are contained on the looped paths. Because each router that is not the start and end point of our loop path forwards each packet in the loop twice, the amount of packets forwarded relative to the total amount of packets forwarded in a healthy network will steadily increase.
For our first failure path this means an additional 50% of packets forwarded, assuming that routers R4 and R6 forward the same amount of packets as router R5.
In our second failure path network the amount of packets forwarded even increases by 60%, assuming that router R3 forwards the same amount of packets as router R2 and that routers R5, R6 and R8 forward the same amount of packets as router R7.

With FRR and ShortCut

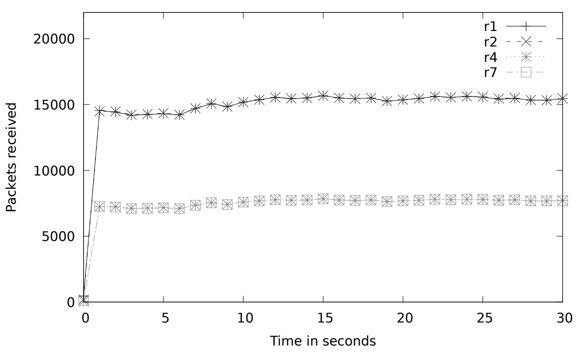
ShortCut was able to cut off the loop and therefore reduce the amount of packets forwarded to the original amount. Because this is similar to the behaviour in section 5.1.4 we omitted additional graphs from our results.

5.2.5 Packet flow - UDP

We used the same test function as in section 5.2.4 but with the "udp" flag to run a packet flow measurement with UDP.
The first tests evaluated were run using only a FRR mechanism in *With FRR*. We then evaluate the tests run with a FRR mechanism and an implementation of ShortCut in *With FRR and ShortCut*.



(a) UDP packet flow after a failure - 1st network



(b) UDP packet flow after a failure - 2nd network

Figure 5.29: UDP packet flow on four routers for both failure path networks after a failure

With FRR

Falling in line with our results in our TCP packet flow measurement in section 5.2.4, a longer failure path adds to the overall workload of the network. This is shown in fig. 5.29. Because UDP does not send ACKs back to the sender, all packets produced by the UDP data transfer are passed through the looped path.

Because router R1 and all routers up until the endpoint of the loop will forward the UDP packets twice, the amount of packets forwarded on the first failure path network increases by 100%. When the failure path is extended as is the case in our second failure path network, the amount of packets forwarded on the network even increases by 120%.

With FRR and ShortCut

Similar to the behaviour of ShortCut in our TCP packet flow measurements, it was able to restore the network to its original amount of packets forwarded on the routers. Because this behaviour does not add any room for interpretation, we omitted graphs for these measurements from our results.

5.3 Discussion

In this section we discuss our results in the previous measurements. We proceed by comparing the results of different measurement types using the three topologies. For each measurement type we collect the implications of a failure for the network and whether ShortCut is able to enhance results. We start with the bandwidth in section 5.3.1, continuing to the bandwidth with a second data flow in section 5.3.2. After that we talk about our latency measurements in section 5.3.3 followed by our packet flow measurements using TCP and UDP in section 5.3.4.

5.3.1 Bandwidth

A failure in our topologies did not have an impact on our bandwidth measurement results. The throughput of a network and therefore the bandwidth that can be achieved on a path is not influenced by additional hops on a route, even though these might increase latency. This is, of course, also true for longer failure paths. As long as no additional data flows are sent through the network the bandwidth is not impacted and ShortCut has no need to restore performance.

5.3.2 Bandwidth with concurrent data flow

Measuring the achieved bandwidth with a concurrent data flow shows that in case a second data flow is running on the network, the looped path now has a real performance impact. The throughput is split between both data flows on all links that are looped. This causes an overall throughput loss of 50%, as the links included in the looped path create a bottleneck for both data flows.
Using ShortCut under such circumstances will fully restore performance to the state without failure. It has to be noted that, although in our measurements both data flows achieved a maximum throughput with ShortCut, the alternative route still could be used by other data flows which might be influenced. This is however also true for the network when not using ShortCut.
Longer failure paths showed no additional impact on the performance and ability of ShortCut to restore the throughput. A longer failure path will have an adverse effect in a realistic network. Because more links experience additional traffic, even more data flows might be affected by the failure. The usage of ShortCut would therefore be more beneficial the more links can be removed from looped paths.

5.3.3 Latency

Sending packets over additional hops will without a doubt increase the latency. This is confirmed in our results. Longer failure paths would of course increase this additional delay. Because ShortCut was able to cut off the looped path, it was able to restore the original latency on our test networks.
As such ShortCut provides a reliable way to optimize alternative routes, especially for time sensitive data like VOIP.

5.3.4 Packet flow

Our packet flow measurements using a TCP data transfer showed the amount of packets forwarded on each router. A naive assumption would be that the routers on a looped path would have to forward each packet twice and therefore have a 100% increased load on them. This is, however, not true for TCP as only packets sent to the receiving device are forwarded through our loop in our topologies. ACKs returning from the *iperf* server are not sent through the looped path. Because of this the routers on the looped path actually only forward half of all packets. All routers on the failure

path, except for the router at the end, will forward all packets sent into the loop twice. While the actual amount of packets does not change for these routers, as ACKs are not sent into the loop, they add onto the overall load of the network. The router directly attached to the broken link will just forward packets once.
It has to be noted that these packets all contain data and therefore do have full impact on the bandwidth of links involved in the looped path as discussed previously in section 5.3.2.
The entry point of the loop, which in our topologies is always router R1, forwards each packet containing data twice and in addition to this the acknowledgements once, increasing the workload for the router by 50%. Longer failure paths do not influence this behaviour, they do however add more devices which will be affected by the failure and therefore also the amount of packets forwarded on the network in total.
This adds up to 33% more packets forwarded in our minimal network, 50% more packets forwarded in our first failure path network and a 60% increase in forwarded packets in our longer second failure path network.
When using UDP for our data transfer and measuring the packets forwarded on routers, the differences between TCP and UDP become quite obvious. As UDP does not send ACKs on successful transmission, all looped routers except for the router directly attached to the failure, e.g. router R4 for the second failure path network, will forward all packets twice.
For our minimal network this meant an increase in packets forwarded on the network by 50%. With increasing length of the failure path the impact on the network grew worse, with a 100% increase of packets forwarded for our first failure path network and a 120% increase for our longer second failure path network.
For all transfers, be it using TCP or UDP, ShortCut is able to cut the looped path and therefore restore the network to a state in which only a minimum amount of routers forward the same amount of packets. No more additional packets are forwarded, alleviating the burden placed on the network. ShortCut proves to be a reliable way to reduce traffic caused by failures.

6 Conclusion

In this chapter we conclude our findings in section 6.1. We furthermore discuss further extensions of this work and how our results could be made more robust for interpretation in section 6.2.

6.1 Results of this work

We provide an entry point into the topic of modern networks, resilient routing and FRMs.
To this end we explain the implementation of a test framework using Mininet, incorpirate a simple FRR mechanism and provide a simple example for an implementation of ShortCut.
We also show that FRMs can have a meaningful impact on networks, restoring bandwidth, removing additional delays and reducing the unnecessary workload for routers through testing in our test framework. ShortCut has proven to be an effective method in relieving additional burden imposed on networks after a failure, reducing delays by up to 38% and additional packet forwarding by up to 120%.

6.2 Future work

During our work on this thesis we discovered more ways to further investigate FRMs and extend our existing test framework. In addition to this we also discovered more methods to produce more robust results. We discuss possible extensions of our test framework in section 6.2.1. The addition of CPU usage measurements on Mininet hosts discussed in section 6.2.2 could also provide additional insights into our results.

6.2.1 Testing framework

We provide a test framework which can be used to performing tests on Mininet networks automatically. The framework is, however, still a prototype and could be extended for broader usage.

One easy extension would be the addition of tests using UDP. While we already implemented an UDP data transfer for a packet flow measurement, additional implementations using UDP in e.g. bandwidth testing could prove worthwhile as UDP has no congestion control, possibly causing packet queue overflows in routers.
This includes the addition of a selection of measurement tools. All measurements were done using one specific tool, depending on the type of measurement, namely *iperf* for bandwidth measurements and the production of data streams and *ping* for latency tests. *iperf* could be replaced with a multitude of software packets and some members of the Mininet community have suggested that e.g. *netperf* (Jones and Contributors 2015) would provide more accurate results. This could be evaluated in further detail.
Because of the time constrains of this work we were unable to test in high volumes, even though we experienced some fluctuations in our measurements. To increase the reliability of our results the tests could be run e.g. a hundred times. This could also be integrated into the testing framework with an additional argument specifying in which quantity the test should be run.
We evaluate three pretty similar topologies, as well as a simple implementation of FRR and an implementation of the FRM ShortCut. Depending on the requirements of a network, an e.g. full topology with all routers inter-connected might be a good starting point for further testing. This should go hand in hand with the implementation of an automatic routing and a more strategic deployment of FRR and FRMs.
As described in section 2.3.5 there are also many different FRMs which could be implemented in Mininet and tested using our test framework.
To further professionalize the implementation of FRMs they could be implemented using P4, overwriting the default behaviour of Mininet routers. This would provide a version of e.g. ShortCut that could be deployed in a real network in a short amount of time.

6.2.2 Measuring CPU usage of hosts in Mininet

Tests in this work are done with a limit of 100 Mbps imposed on the links, in tests without a limit to the bandwidth values of around 40 Gbps were reached. While this should remove any fluctuations that could be caused by additional operations either on the virtual machine or the host system, it does not completely ensure proper distribution of processing power. For this we could run CPU usage measurements while the actual tests are running. This would enable us to further interpret results and possible spikes in delay or bandwidth.

List of Figures

Bibliography

Alexey Kuznetsov (2011): *iproute2*. URL: `https://wiki.linuxfoundation.org/networking/iproute2` (visited on 05/16/2022).

Alharbi, T., S. Layeghy, and M. Portmann (2017): "Experimental evaluation of the impact of DoS attacks in SDN". In: *2017 27th International Telecommunication Networks and Applications Conference (ITNAC)*, pp. 1–6.

Ayuso, P. N., J. Kadlecsik, E. Leblond, F. Westphal, A. B. González, and P. Sutter (2019): *nftables*. n.p. URL: `https://netfilter.org/projects/nftables/` (visited on 05/10/2022).

Bosshart, P., D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker (2014): "P4: Programming Protocol-Independent Packet Processors". In: *SIGCOMM Comput. Commun. Rev.* 44.3, pp. 87–95.

Bundesnetzagentur Deutschland (2021): *Jahresbericht 2020*. n.p. URL: `https://www.bundesnetzagentur.de/SharedDocs/Mediathek/Jahresberichte/JB2020.html` (visited on 05/16/2022).

Callon, R. (12–1990): *OSI IS-IS for Routing in TCP/IP and Dual Environments*. URL: `https://datatracker.ietf.org/doc/html/rfc2328` (visited on 05/16/2022).

Chiesa, M., A. Kamisiński, J. Rak, G. Rétvári, and S. Schmid (2021): "A Survey of Fast-Recovery Mechanisms in Packet-Switched Networks". In: *IEEE Communications Surveys Tutorials* 23.2, pp. 1253–1301.

Dridi, L. and M. F. Zhani (2016): "SDN-Guard: DoS Attacks Mitigation in SDN Networks". In: *2016 5th IEEE International Conference on Cloud Networking (Cloudnet)*, pp. 212–217.

Dugan, J., S. Elliott, B. A. Mah, J. Poskanzer, and K. Prabhu (2016): *iperf3*. n.p. URL: `https://iperf.fr/` (visited on 05/16/2022).

Gill, P., N. Jain, and N. Nagappan (2011): "Understanding network failures in data centers: measurement, analysis, and implications". In: *Proceedings of the ACM SIGCOMM 2011 Conference*, pp. 350–361.

Haque, I. and M. A. Moyeen (2018): "Revive: A Reliable Software Defined Data Plane Failure Recovery Scheme". In: *2018 14th International Conference on Network and Service Management (CNSM)*, pp. 268–274.

Information Sciences Institute - University of Southern California (1981): *Transmission Control Protocol: RFC*. URL: `https://www.rfc-editor.org/rfc/rfc793`.

Jones, R. and Contributors (2015): *netperf*. URL: `https://github.com/HewlettPackard/netperf` (visited on 05/11/2022).

Kuerban, M., Y. Tian, Q. Yang, Y. Jia, B. Huebert, and D. Poss (2016): "FlowSec: DOS attack mitigation strategy on SDN controller". In: *2016 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pp. 1–2.

Kvalbein, A., A. F. Hansen, T. Cicic, S. Gjessing, and O. Lysne (2005): "Fast recovery from link failures using resilient routing layers". In: *10th IEEE Symposium on Computers and Communications (ISCC'05)*, pp. 554–560.

Lantz, Bob and the Mininet Contributors (2021): *Mininet*. URL: `http://mininet.org/` (visited on 05/10/2022).

Liu, J., A. Panda, A. Singla, B. Godfrey, M. Schapira, and S. Shenker (2013): "Ensuring connectivity via data plane mechanisms". In: *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pp. 113–126.

Maaßen, F. (5/2022): "Comparison of Fast Recovery Methods: Git Repository". Bachelor. Dortmund: TU Dortmund. URL: `https://git.cs.tu-dortmund.de/frederik.maassen/ComparisonOfFastRecoveryMethodsInNetworks` (visited on 05/16/2022).

Montag, C., K. Błaszkiewicz, R. Sariyska, B. Lachmann, I. Andone, B. Trendafilov, M. Eibes, and A. Markowetz (2015): "Smartphone usage in the 21st century: who is active on WhatsApp?" In: *BMC Research Notes* 8.1, p. 331.

Moy, J. et al. (4–1998): *OSPF version 2*. URL: `https://datatracker.ietf.org/doc/html/rfc2328` (visited on 03/15/2022).

Muelas, D., J. Ramos, and J. E. L. de Vergara (2018): "Assessing the Limits of Mininet-Based Environments for Network Experimentation". In: *IEEE Network* 32.6, pp. 168–176.

Nelakuditi, S., S. Lee, Y. Yu, and Z.-L. Zhang (2003): "Failure insensitive routing for ensuring service availability". In: *International Workshop on Quality of Service*, pp. 287–304.

Oracle (2022): *VirtualBox*. URL: `https://www.virtualbox.org/` (visited on 05/11/2022).

Rak, J. and D. Hutchison, eds. (2020): *Guide to disaster-resilient communication networks*. Ed. by J. Rak and D. Hutchison. Springer Nature.

Shukla, A. and K.-T. Foerster (2021): "Shortcutting Fast Failover Routes in the Data Plane". In: *Proceedings of the Symposium on Architectures for Networking and Communications Systems*, pp. 15–22.

Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever (2019): "Blink: Fast Connectivity Recovery Entirely in the Data Plane". In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, pp. 161–176.

URL: `https://www.usenix.org/conference/nsdi19/presentation/holterbach`.

van Rossum, G. and F. L. Drake (2009): *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace.

Wustenhoff, E. and S. BluePrints (2002): "Service level agreement in the data center". In: *Sun Microsystems Professional Series* 2.

Eidesstattliche Versicherung

(Affidavit)

Maaßen, Frederik	193017
Name, Vorname (surname, first name)	Matrikelnummer (student ID number)

■ Bachelorarbeit (Bachelor's thesis) [✓] ☐ Masterarbeit (Master's thesis)

Titel
(Title)

A Comparison of Fast-Recovery Mechanisms in Networks

Ich versichere hiermit an Eides statt, dass ich die vorliegende Abschlussarbeit mit dem oben genannten Titel selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

I declare in lieu of oath that I have completed the present thesis with the above-mentioned title independently and without any unauthorized assistance. I have not used any other sources or aids than the ones listed and have documented quotations and paraphrases as such. The thesis in its current or similar version has not been submitted to an auditing institution before.

Dortmund, 17.05.2022	[illegible]
Ort, Datum (place, date)	Unterschrift (signature)

Belehrung:
Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -).

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird ggf. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin") zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

Official notification:
Any person who intentionally breaches any regulation of university examination regulations relating to deception in examination performance is acting improperly. This offense can be punished with a fine of up to EUR 50,000.00. The competent administrative authority for the pursuit and prosecution of offenses of this type is the Chancellor of TU Dortmund University. In the case of multiple or other serious attempts at deception, the examinee can also be unenrolled, Section 63 (5) North Rhine-Westphalia Higher Education Act (*Hochschulgesetz, HG*).

The submission of a false affidavit will be punished with a prison sentence of up to three years or a fine.

As may be necessary, TU Dortmund University will make use of electronic plagiarism-prevention tools (e.g. the "turnitin" service) in order to monitor violations during the examination procedures.

I have taken note of the above official notification:*

17.05.2022	[illegible]
Ort, Datum (place, date)	Unterschrift (signature)

***Please be aware that solely the German version of the affidavit ("Eidesstattliche Versicherung") for the Bachelor's/ Master's thesis is the official and legally binding version.**