Implementation of Secure Multipath Resilience in HIPLS using Mininet and SDN Integration for Dynamic Path Failover

Emma Siklosi
Linköping University
Linköping, Sweden
emmsi015

Fabian Johansson
Linköping University
Linköping, Sweden
fabjo285

Mattias Johansson
Linköping University
Linköping, Sweden
matjo680

Morgan Uvelid
Linköping University
Linköping, Sweden
moruv297

Robin Ngo
Linköping University
Linköping, Sweden
robng725

***Abstract*—In modern networking, security, and resilience is vital for continuous data transmission. With users increasingly demanding faster and uninterrupted network performance, this study explores the integration of the Host Identity Protocol (HIP) with Virtual Private LAN Service (VPLS) into Software-Defined Networking (SDN) to achieve secure multipath resilience and dynamic path failover. Using Mininet for simulation and Ryu as the SDN controller, we evaluate both reactive and proactive failover mechanisms. The reactive approach recalculates routes dynamically in the case of link failures, while the proactive one precomputes and installs backup paths. Key performance metrics analyzed include recovery time and bandwidth utilization. Results show that proactive failover reduces the maximum ping latency by 69.5% compared to the reactive failover mechanism. As the network grows, the reactive SDN controller experiences increased processing time depending on the number of introduced switches, whereas the proactive controller maintains consistent processing time.**

***Index Terms*—SDN, Mininet, Ryu, Path failover, HIPLS**

I. Introduction

In modern networks, ensuring resilience and security is critical, especially when data must traverse multiple paths to reach its destination. The Host Identity Protocol (HIP) combined with Virtual Private LAN Service (VPLS) is called HIP-VPLS (HIPLS). Combining this with Software-Defined Networking (SDN) provides centralized control and dynamic management of network flows, enabling multipath resilience and failover mechanisms. This integration allows for dynamic recovery in case of path failure, making networks more reliable and adaptable to changing conditions. This study explores how secure multipath resilience can be achieved by integrating HIPLS with SDN, using Mininet and Ryu for simulation and testing.

Additionally, this paper examines the differences between reactive and proactive routing strategies in achieving reliable and efficient recovery during link failures. Proactive routing involves pre-computed backup paths activated when ports are detected as down. Reactive routing, on the other hand, computes new paths after a failure is detected. A reactive routing strategy reacts to changes in network conditions as they occur. By comparing these approaches in the context of HIPLS-SDN, this paper will highlight their differences in terms of recovery time, resource utilization, and adaptability to dynamic network environments.

II. Research Questions

1) How can a path fail-over mechanism that detects real-time link failures and attacks be implemented using SDN?
2) To what extent can traffic be dynamically rerouted to alternative paths during network transmission?
3) How can Mininet and Ryu simulate a real-world network scenario to evaluate the recovery time and bandwidth of SDN-enabled routing during link failures?

III. Background

This section provides the background information necessary for understanding the remainder of the project.

A. Host Identity Protocol (HIP)

The Host Identity Protocol (HIP) enhances Internet architecture by introducing a cryptographic namespace and separating host identity from location, known as the identifier or locator split. A Host Identity (HI), represented by a cryptographic key pair, replaces IP addresses for secure host identification. Its compact form, the Host Identity Tag (HIT), is a 128-bit hash of the HI used in protocols and applications.

HIP also supports local scope identities for efficient traffic management within limited domains, such as regional networks or specialized environments like network-on-chip systems [1].

1) HIPv2: Builds on the previously mentioned HIP by enhancing its capabilities and addressing known limitations observed in earlier versions. HIPv2 uses public cryptographic keys as host identifiers, increasing security and accountability [2]. In HIPv2, public cryptographic keys replace the IP addresses at the application level of communications, while IP addresses remain essential for routing processes. It fully supports mobility and multihoming, allowing hosts to change their network attachment points without disrupting ongoing sessions [3]. It also enhances the security by integrating IPsec (Internet Protocol Security).

B. Virtual Private LAN Service (VPLS)

VPLS is a Virtual Private Network (VPN) that runs on Layer 2 making it an L2VPN. It emulates a Local Area Network (LAN) environment over a Wide Area Network (WAN), enabling multiple geographically distributed sites to connect seamlessly. By linking hosts to a multipoint network, VPLS provides a unified and transparent networking experience, as if all connected sites were part of the same LAN [4].

C. Software-Defined Networking (SDN)

Allows networking devices to focus only on data plane functions since the SDN controller handles all control plane functionality. Control plane functionality covered by the SDN controller includes monitoring the network and computing forwarding rules. The SDN controller's most important task is maintaining end-to-end connectivity between nodes. Therefore when a link between nodes fails or breaks the controller has to reconfigure the network so the end-to-end connectivity for all paths are restored [5].

D. Ryu

Ryu is a controller for an SDN network. It is designed as a centralized control platform that separates the control plane from the data plane and provides programmable network configurations [6]. Ryu can handle multiple southbound protocols, such as OpenFlow. These protocols are essential for managing communication between the SDN controller and the network devices responsible for forwarding the data [7].

E. Open Flow

OpenFlow is a protocol that separates the control plane from the data plane. This means that centralized management and dynamic configuration can be used. OpenFlow is used for the communication between the SDN (Software-Defined Networking) controller and the network devices (switches). The SDN controller manages forwarding paths for packets by programming flow tables in the switches with specific rules [8].

The OpenFlow flow table contains fields shown in Table I. In a flow table, some or all fields can be present. The fields are defined as follows [9]:

- **Rule**, identifier for that rule.
- **Match Fields**, defines what fields the switch should map on e.g. IP, HIT, MAC, etc. This could include wildcard characters *.
- **Priority**, defines the order the incoming packets will be matched against the rules, higher priority will be matched first.
- **Action**, how the switch should handle that packet e.g. forward packet to interface two.
- **Counter**, used to track statistics e.g. number of packets matched entry.
- **Timeout**, determines how long the rule is active.
- **Cookie**, an identifier used by the controller to manage and track flow entries.

Rule	Match Fields	Priority	Action	Counter	Timeout	Cookie
R1	IP, Port	1	Forward 2	100	30 s	1
R2	IP, MAC	2	Drop	50	60 s	2
R3	IP	3	Forward 3	75	None	3

TABLE I: A Flow table example

Group tables extend OpenFlow configuration rules from version 1.1, enabling more advanced monitoring and forwarding capabilities. They play an important role in managing network traffic and increasing the flexibility of network operations. For instance, a fast failover group table can be configured to monitor the status of ports and interfaces, ensuring rapid response to link or port failures [5]. Implementing complex forwarding behaviors that standard flow tables would struggle to achieve, group tables enhance network functionality. One of the group table's key benefits is traffic distribution across multiple paths or links, improving resource utilization. In cases where a single packet needs to be delivered to multiple destination group tables aggregate traffic into a single flow table entry per switch, saving significant link resources and reducing the number of flow entries needed. In the event of a link failure, fast failover groups ensure high availability by providing backup paths. If a primary path fails, the group table can quickly switch to a predefined backup path, minimizing downtime and maintaining service continuity [10].

F. Open vSwitch (OVS)

An OpenFlow switch consists of flow tables or group tables that perform packet lookups and forwarding. Using an OpenFlow switch the controller can add, update, and delete flow entries. These switches are responsible for forwarding at the data layer. This can refer to a physical or virtualized switch or router. A switch that forwards packets entering the switch based on the flow table, which sets rules for what to do with the incoming traffic. These flow entries are maintained and generated by the controller. The switches are classified into two types based on their support for OpenFlow[11]:

- **Dedicated OpenFlow switch:** Only supports OpenFlow forwarding. It can only process all traffic that passes through it in OpenFlow mode. Can't perform Layer 2 or Layer 3 forwarding on traffic.
- **OpenFlow-compatible switch:** Supports both OpenFlow forwarding and Layer 2/3 forwarding. It also supports OpenFlow features such as flow tables.

G. Network Layers

The network can be divided into multiple layers, and one commonly used method for partitioning is the Open Systems Interconnection (OSI) model. The seven layers of the OSI model are illustrated in Figure 1. The HIP protocol is run between the network layer and the transport layer, hence it is sometimes referred to as running on layer 3,5. Open vSwitches are SDN-enabled devices that run on the data link layer and the network layer, therefore they can be used as both switches and routers.

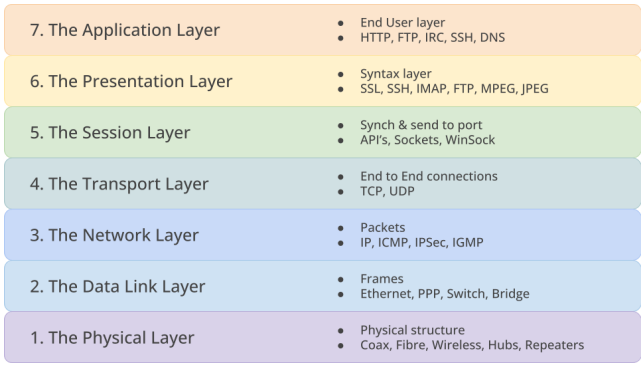

Fig. 1: Network layers

H. IPv6

Is the latest version of IP, replacing IPv4. The key features of IPv6 are:

- **Increased Address Space**: This is crucial for the continued growth of the internet and the increasing number of devices in the context of IoT [12].
- **Seamless Network Management**: Simplifies subnet addresses management and includes features like stateless autoconfiguration, which allows devices to configure themselves automatically when connected to a network [13].
- **Improved Security**: Built-in support for IPsec, which was previously optional for IPv4, now provides end-to-end security for Internet communications [12].

I. Multipath Routing

Multipath routing is the general technique of utilizing multiple routes to balance concurrent traffic across different potential paths in a network. This approach helps distribute traffic load across multiple paths, preventing congestion on a single path, which could otherwise increase latency and cause packet loss. By doing so, it increases the data the network can handle concurrently, thereby improving the overall network performance [14]. It also ensures faster reactions to network failures since it establishes alternative paths for traffic flow ahead of time. With pre-established connections between sources and destinations, in the case of congestion or other suspected issues, the traffic can be redirected to an alternative path without the need to search for new routes, which would otherwise slow down the recovery time [15].

J. Path Failover

Existing failure detection protocols at the data-link layer, such as the Spanning Tree Protocol (STP), are often classified as slow because they can take seconds to detect failures. These protocols work by updating port statuses in switches to maintain the network distribution tree. This process involves monitoring and modifying the state of switch ports - marking them as active, blocked, or forwarding traffic- to ensure that the network avoids loops and continues functioning correctly even when links fail [5].

Precomputing primary and backup paths using OpenFlow and integrating fast failover detection mechanisms like BFD recovery time can be reduced. This approach ensures consistent and minimal recovery time, regardless of the length of the paths or the size of the network [5].

K. Bidirectional Forwarding Detection (BFD)

The BFD protocol checks the liveliness of paths to predetermined endpoints, a monitored link. A session for a monitored link is set up with a three-way handshake and then each node sends control and echo messages. When a control message containing the status of the monitored link is sent to the endpoint, the endpoint replies with an echo of its status. By using control and echo messages BFD can detect path failure. BFD can be implemented over any transport protocol, over OpenFlow switch it can be implemented using UDP/ IP stream [5].

L. Mininet

Mininet is a network emulation tool designed to create and test SDN applications in a controlled environment. It allows the creation of virtual network typologies [16].

M. Dijkstra's

Algorithm 1 shows Dijkstra's algorithm, which computes the shortest path from a source vertex s to all other vertices in a weighted graph $G = (V, E)$. The algorithm is based on a priority queue implementation.

Algorithm 1 Dijkstra's Algorithm [17]

Input: Graph $G = (V, E)$, source vertex s
Output: Shortest distance from s to all vertices in V
Initialize distances: $d[v] \leftarrow \infty$ for all $v \in V$, $d[s] \leftarrow 0$
Initialize priority queue Q and insert $(s, 0)$ into Q
while Q is not empty **do**
  Extract u from Q with the smallest distance $d[u]$
  for each neighbor v of u **do**
    if $d[u] + w(u, v) < d[v]$ **then**
      Update $d[v] \leftarrow d[u] + w(u, v)$
      Insert or update v in Q with priority $d[v]$
    end if
  end for
end while
Return d (shortest distances to all vertices)

The time complexity of Dijkstra's algorithm using a priority queue in the worst case is given by Equation 1, where V is the number of vertices and E is the number of edges.

$$O((V + E) \cdot \log V) \tag{1}$$

N. Link Layer Discovery Protocol (LLDP)

LLDP is a standard protocol operating on layer 2 (data-link) in the OSI model. The protocol is used by switches to advertise information such as the MAC address and port information to its neighbors in the local network. The usage of LLDP allows

network discovery which enables the construction of network topologies [18]. Network discovery is particularly beneficial in SDN network environments where the controller relies on an extensive network map.

LLDP can also detect link failure by sending heartbeats every 16 ± 8 ms over links without an active session. The receiver needs to respond within a 50-150 ms time window. If no response is received, the link is presumed to be down [5].

IV. Related work

This section examines existing approaches to network reliability and recovery in SDN, focusing on key contributions that address backup forwarding rules and recovery mechanisms while highlighting their strengths and limitations.

A. Computing Backup Forwarding Rules in SDN

This paper introduces two algorithms designed for proactive and reactive routing. However, the rerouting algorithms presented are not the most efficient due to several limitations [19]. One significant drawback is their higher computational complexity, as additional processing is required to determine alternative paths during proactive and reactive routing scenarios. This complexity can lead to increased latency, particularly in dynamic environments where rapid changes occur. Moreover, the algorithms may fail to fully optimize the use of available network resources, potentially causing increased congestion due to the use of longer backup paths.

While these algorithms offer valuable alternatives to existing methodologies, they lack the incorporation of advanced techniques such as real-time traffic analysis or machine learning, which could significantly enhance their efficiency and adaptability in routing decisions.

An alternative approach involves precomputing all backup paths for every node by simulating the removal of individual links. This algorithm systematically traverses each node in the network and iterates through all its associated links. The algorithm removes each link from the network and recalculates the shortest path to generate a corresponding backup path. This process is repeated for every node and its links, ensuring all nodes have precomputed backup paths available for any potential link-down scenario.

B. Fast Recovery in SDN

The study introduces a fast recovery scheme in SDN that leverages per-link Bidirectional Forwarding Detection (BFD) together with preconfigured primary and backup paths managed by an OpenFlow controller [5]. This approach achieves recovery times averaging 3.3 ms, significantly better than traditional methods that often average over 30 ms. Through experimental evaluations, the authors demonstrate that the recovery time remains stable, regardless of the network size or length of the paths involved. The findings emphasize the limitations of conventional failure detection mechanisms, which can lead to longer recovery periods in high-demand and availability networks. Finally, this proposed method improves network resilience and ensures the delivery of high-availability services.

V. Methods

Our method is based on a literature study and scientific experimentation in a simulated environment, with measurements conducted to evaluate the performance of the implemented SDN controllers.

A. Literature Study

A thorough literature review was conducted to prepare for implementation and methods. The literature study involved analyzing materials provided by our supervisor, which included background information, tools, and software programs. To get a more comprehensive understanding we also reviewed additional, relevant, scientific papers and books.

To find research papers, we utilized Google Scholar with keywords such as SDN, fast failover, link failure, fast recovery, and link failure detection. These keywords formed the basis for our paper.

We also took advantage of the university library's database, especially Scopus and Scopus AI, to locate related work. Relevant abstracts and summaries were extracted using queries formulated with keywords that were related to the topic.

B. Experimentation

The lost throughput during link failure was measured with iperf and the elephant flow method. Iperf was used to transfer the maximum amount of data possible between two nodes in the network. Two Mininet nodes (hosts), the pair H1–H2 and H1–H5, were used for this measurement. One node acted as a server using the *iperf -s* command, while the other node was used to send data using *iperf -t 10 -c "IP of the server"*. The *-t* flag was used to specify the amount of transmission time in seconds. Two separate tests were made using iperf, one with the link up and one with the link down. The mean of the tests was computed after 10 runs to limit possible outliers that may have occurred during the tests.

To measure and evaluate ping times during link failure, automated tests were run. The flow method used is often referred to as mice flow, which consists of smaller short-lived data transmissions. The automated script is run with the command *ping "IP of the host" -c 100 -i 0.05* and automatically shut down a link between the involved hosts after 1 second of delay. The flag *-c* specifies the number of pings to send and *-i* is used for the ping interval.

A real-world network scenario was simulated to evaluate the performance of the SDN controller implementation. Two specific scenarios were considered: one with general traffic and another with HIP-VPLS traffic. These experiments assessed the controller's ability to manage dynamic path failover and its multipath resilience capabilities. The simulations were carried out using Mininet, with the Ryu controller managing the topology.

1) Measurements: The metrics of recovery time and bandwidth utilization during link failures, and dynamic path rerouting will be measured for conducting a performance evaluation.

The scenarios that will be evaluated consist of a combination of the following alternatives:

- Reactive vs Proactive SDN controller
  The reactive controller is the SDN controller that reactively reroutes the traffic by removing affected switches' flow entries containing the broken link. The switches will then send a new pack-in and the controller will recompute the shortest path and install those connected flows in the switch. The proactive controller precomputes all paths and utilizes group tables with fast failover and watch port that automatically re-routs to a backup path if the port from the primary path currently is down.
- General flow data vs HIP-VPLS data
  The general flow data will be represented as pings running with delays on network topology, shown in Figure 2. Compared to pings running with delays encrypted and sent as HIP-VPLS packets on network topology as shown in Figure 3.
- Link failure vs no link failure
  The scenarios will be tested both handling link failure and running without failures.

The SDN controller is conceptually placed at the center of the network topology. To simulate this configuration, a delay will be added between the switches and the controller. The delay will vary based on the propagated distance between the switch and the controller. This simulates a scenario where the controller operates remotely instead of locally on the same machine as the network topology is simulated.

The following scenarios will be considered, with the simulated SDN controller being conceptually placed in center within the network topology:

1) Reactive SDN controller sending general flow data without link failure
2) Reactive SDN controller sending general flow data with link failure
3) Proactive SDN controller sending general flow data without link failure
4) Proactive SDN controller sending general flow data with link failure
5) Reactive SDN controller sending HIP-VPLS data without link failure
6) Reactive SDN controller sending hip HIP-VPLS data with link failure
7) Proactive SDN controller sending hip HIP-VPLS data without link failure
8) Proactive SDN controller sending hip HIP-VPLS data with link failure

For each scenario, the following measurements were conducted using 100 packets and a ping interval of 0.05 seconds:

1) Controller processing time
   A timer was implemented within the SDN controller where it starts initialization to measure its processing time, stopping the timer at the packet-in handler method's end. The packet-in handler method is triggered only when a switch communicates with the controller during the initialization or removal of network links.

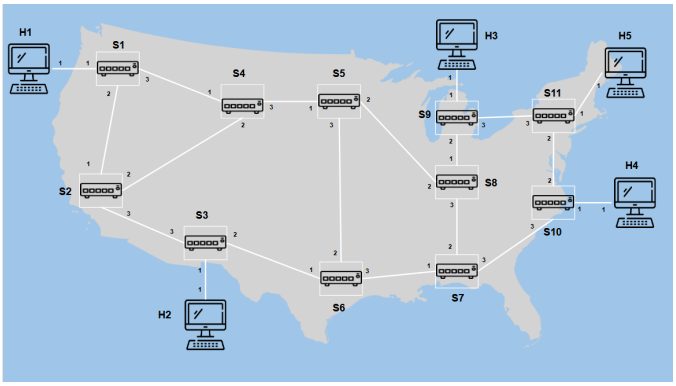

Fig. 2: Network Scenario 1

2) Lost packets during link down
   The link included in the ping path was marked as *down* during the pinging process. An automated script with a delay of 1 second was used to bring the link down after 1 second. This was automated with the same delay to ensure the average time remained consistent across all iterations. The number of packets lost during the link down was measured.
3) Throughput loss during link failure
   Using iperf, one host was set up as a server and one as a client. The client sent the maximum bandwidth available in under ten seconds to the server. Ten tests were run for both link down and link up.
4) Number of flows installed
   A counter was used to count all installed flows within the SDN controller, in the install and delete path methods.

The network scenarios are represented using a map of the USA to give a conceptual overview of the system's structure. The network is based on the American network Abilene [20] and is only used to get a topology over a real network that is currently in use. By real-world standards, one SDN controller is not enough to control a whole country and is only used for evaluation measures in this paper. The scenarios are not modeled as exact real-world scenarios.

2) Network Scenario – general traffic: A scenario with general traffic is shown in Figure 2.

3) Network Scenario – general traffic: A scenario with HIP-VPLS traffic is shown in Figure 3.

4) Experimentation Setup: The experiments were supposed to be conducted with two network traffic scenarios; one with general traffic and another with HIP-VPLS traffic. The Mininet-based network topologies were configured with artificial propagation delays due to distances between switches, hosts, and the controller. The distances were calculated with Google Maps between nodes from the Abilene topology [20]. The distances were then fed into an Ethernet cable propagation calculator tool [21] where corresponding delays were determined. Calculated delays between links are shown in Table II. Delays between switches and the SDN controller are shown in Table III. The chosen delays would reflect around 50-55 ms Round Trip Time (RTT) from the east to the west coast.

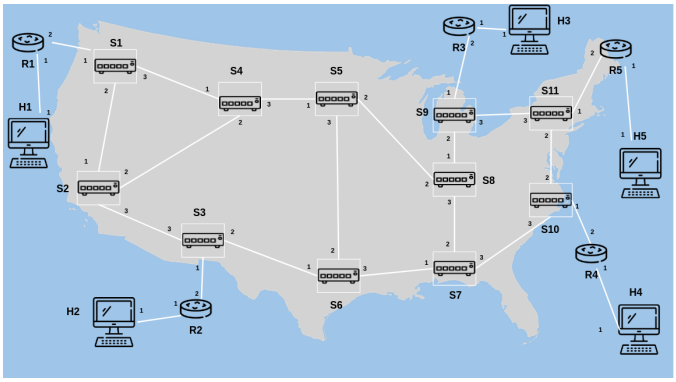

Fig. 3: Network Scenario – HIP-VPLS traffic

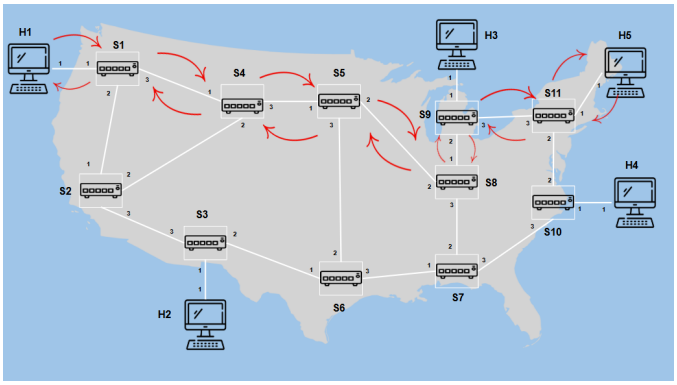

Fig. 4: Network Scenario H1-H5 primary path

Link Endpoint 1	Link Endpoint 2	Delay
H1	S1	1ms
H2	S3	1ms
H3	S9	1ms
H4	S10	1ms
H5	S11	1ms
S1	S2	6ms
S1	S4	9ms
S2	S3	3ms
S2	S4	8ms
S4	S5	5ms
S3	S6	11ms
S5	S6	6ms
S5	S8	4ms
S6	S7	5ms
S8	S7	4ms
S8	S9	1ms
S7	S10	4ms
S9	S11	5ms
S11	S10	1ms

TABLE II: Link delays corresponding to propagated distances including switches and hosts

The controllers were implemented in two configurations, a reactive SDN controller that dynamically responds to events such as flow installations and a proactive SDN controller that pre-configures routes in the switches. The communication paths used during the experimentation are illustrated in Figures 4 and 8. These paths are defined as follows:

Path 1: $\text{Host}_1 \leftrightarrow \text{Host}_2$, Path 2: $\text{Host}_1 \leftrightarrow \text{Host}_5$.

Switch	Delay
S1	14ms
S2	13ms
S3	16ms
S4	5ms
S5	1ms
S6	6ms
S7	8ms
S8	4ms
S9	5ms
S10	11ms
S11	10ms

TABLE III: Link delays corresponding to propagated distances between switches and SDN controller

The controlled variables during experimentation included the network topology configurations (hosts, routers, and switches) and the traffic volume sent through the network. Uncontrolled consistent variables included HIP-VPLS, OpenFlow, and the Open vSwitch type. Mininet served as the simulation environment, while Ryu was used for the SDN controller implementation. OpenFlow facilitated communication between the switches and the controller, enabling dynamic path failover mechanisms.

5) Procedure: Our data collection process was divided into two distinct procedures, one for each of the described controllers. The first procedure, focused on gathering minimum, average, maximum, and standard deviation latency metrics during the ping process, began with configuring the Abilene network topology [20], see Figure 2, in Mininet and linking it to the Ryu controller.

Traffic generation was initiated using an automated script, where we connected to a specific host within the topology and configured a ping command. This command was customized with parameters such as the interval between packets, specified using the *-i* argument, and the total number of packets to be sent, specified using the *-c* argument. The generated ping traffic created a consistent data flow, enabling us to observe the network's behavior under various conditions.

To simulate a link failure, we automatically triggered a link-down event one second after the ping operation started. This deliberate timing ensured that the failure occurred mid-ping, allowing us to capture its impact on traffic flow and get an accurate reading on all relevant metrics. The link failure event and the subsequent re-routing for each communication path are illustrated in Figures 5, 6 and 9.

During the experiments, we recorded key metrics such as packet loss, minimum, average, maximum, and standard deviation of latency and the aggregated controller compute time. These measurements were output to a measurements text file using the Linux pipe utility. By specifying the interval of the ping messages, we calculated the rerouting completion time by multiplying the number of lost packets by the given interval. This process was repeated ten times for each controller to ensure the reliability and consistency of the results.

We implemented two counters in both the reactive and

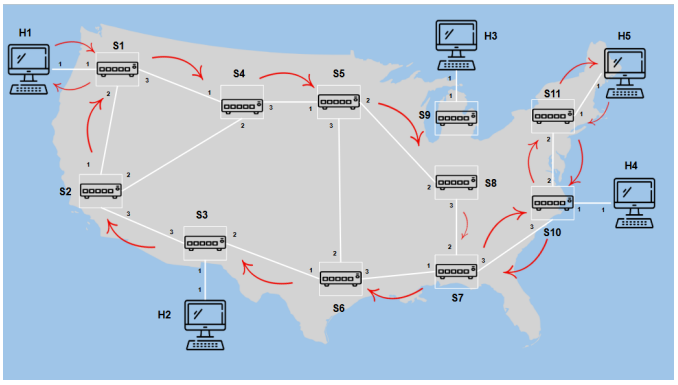

Fig. 5: Network Scenario H1-H5 with proactive-rerouting

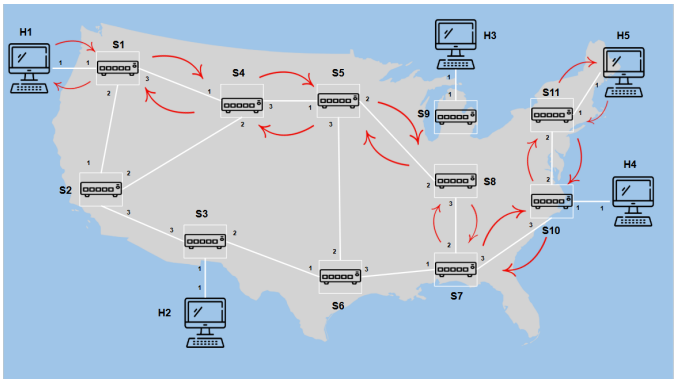

Fig. 6: Network Scenario H1-H5 with reactive-rerouting

proactive controllers to collect additional data on the number of flows affected by the link-down event. One counter tracked the number of flows that were installed, while the other tracked the number of flows that were removed.

C. Limitations

All implementations were done on IPv4 instead of IPv6. The networks are simulated on a single computer. Running the same network configuration on a real network could induce unexpected issues, such as increased latency, packet loss, or unoptimized performance due to hardware or environmental changes. Each switch has at most three ports. We are solely developing for Ubuntu 22.04. Our proactive solutions do not implement back-propagation. Our implementation does not consider backtracking making it possible for our network packets to get stuck in an infinite loop, or until TTL is achieved, effectively cutting off communication.

VI. Software Analysis

The current implementation of Dijkstra's algorithm is optimized to minimize the number of hops in the network. The rationale behind this decision is to prioritize reducing the number of intermediary nodes a packet traverses, which can lead to lower latency and simpler routing paths. If one instead wants to prioritize latency that could be implemented in the Dijkstra algorithm by changing the cost consideration without further modifications of the code base.

An asymmetric latency is introduced while rerouting using the proactive SDN controller as shown in Figure 5. This occurs since the Dijkstra only computes the least hops required to the destination. If the same amounts of hops are required then the algorithm could return a different path depending on the direction. This could lead to TCP performance degradation as discussed in the paper by Hari B et al. [22] among other potential problems. If this is considered crucial, modification to the proactive SDN controller software is needed.

Larger networks need more than one controller due to the amount of overwhelming traffic that can occur [23]. Due to this reason, the Abilene scenario [20] with one controller is not suitable for real-life usage.

The HIP integration with Abilene [20] was not able to be completed during the time frame. The introduction of HIP routers broke links in the network and pinging between hosts was unreliable, making testing impossible. Adding to this, introducing link failures completely crashed the simulated network. The cause of this is still unknown and needs to be further explored. Although the Abilene scenario [20] failed, the example scenario, see Figure 7, worked without problems. This may be due to misconfiguration of the Abilene scenario [20], using a larger network, or the controllers need rework to properly handle HIP traffic on a larger scale.

A. Example HIP Scenario

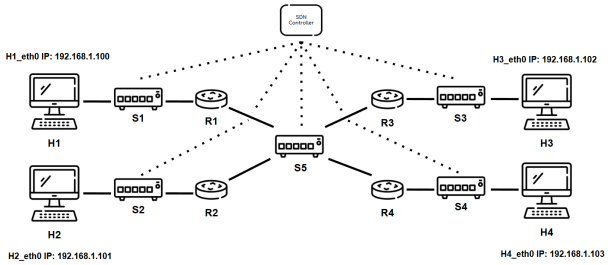

Fig. 7: Example HIP scenario

VII. Results and Analysis

In the following sections, we will present and discuss the implementation of our dynamic path failover using Mininet and SDN.

A. Link Failure Detection

To detect when a link between two network devices goes down, Ryu generates a link-down event. This event is listened for in the controller and allows actions to be made such as removal, installation, and modification of any affected flows. In the controller, a data structure is implemented to keep track of the current network topology. The data structure is used to map pairs of network devices and the port that connects them. On each event where the topology might change, the controller rediscovers the links and devices and updates this internal data structure. This is not limited to just link-down events; changes to switches and ports are also watched.

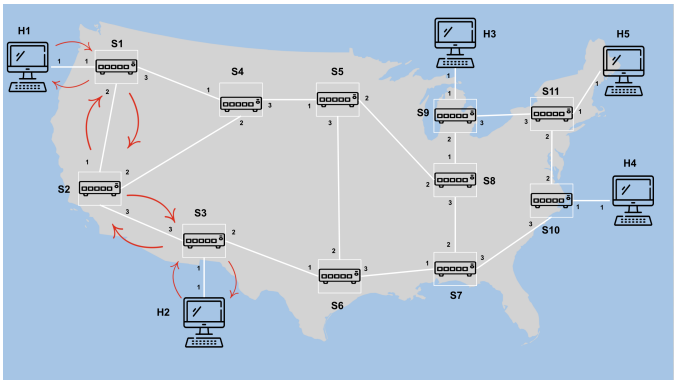

Fig. 8: Network Scenario H1-H2 with primary path

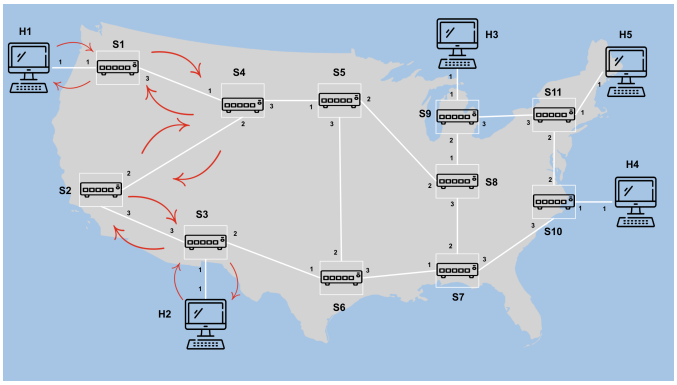

Fig. 9: Network Scenario H1-H2 with rerouting. Both controllers resulted in this rerouting path

B. Throughput loss during link failure

Table IV presents the impact on throughput during a link failure with the reactive controller, while Table V shows the corresponding results for the proactive controller.

Hosts	Average Throughput Before Link Failure	Average Throughput After Link Failure
H1-H5	1.748 Gbits/s	1.421 Gbits/s
H1-H2	6.853 Gbits/s	2.091 Gbits/s

TABLE IV: Throughput before and after link failure with reactive controller

Hosts	Average Throughput Before Link Failure	Average Throughput After Link Failure
H1-H5	1.751 Gbits/s	1.457 Gbits/s
H1-H2	6.852 Gbits/s	2.158 Gbits/s

TABLE V: Throughput before and after link failure with proactive controller

C. Rerouting

After a link failure has been detected, a new path must be calculated through the network. The controller uses Dijkstra's algorithm with a hop-based cost to accomplish this. The distance, equal to the number of hops, is calculated from the source to every other node in the network. This creates a graph that is traversed to find the path from each node to the host. This path is reversed to find the actual path from the source to the destination. Furthermore, default flows are added to each switch during the controller initialization. This flow states that the packet should be forwarded to the controller if no other flow is matched.

1) Reactive Rerouting: In the reactive controller, the routing algorithm is run each time a new host is discovered in the network. This occurs when the first packet from the host enters the network. A data structure in the controller is used to keep track of the MAC address of each host. This is mapped to their neighboring switch and the port on which their packets are received. When a new host is discovered, it is added to the data structure. If the destination of the packet is not yet known, the network is flooded until it is found.

When both the source and destination are known, Dijkstra's algorithm is run to discover the shortest path between them. This path is traversed, and a flow is installed in each affected switch accordingly. The packet is matched on the source and destination MAC addresses, and the receiving port on the switch.

The algorithm described here can be seen in Algorithm 2. If a link-down event is observed, the controller takes action. Each path calculated during initialization was saved to be used during this step. Each flow that is included in a path through the affected link is deleted. The next time a switch receives a packet that is used to take one of these paths, no flows are matched and it is forwarded to the controller. The controller reruns Dijkstra's algorithm and installs new flows corresponding to the new path.

Algorithm 2 Reactive link failure

INPUT:
G: Network topology graph with switches and their link data
$link$: The link that is detected as down.
$path$: The current path from src to dst
$source$: The source node in the network.
$destination$: The destination node in the network.

if link is down then
  Remove $link$ from G
  for all $switch \in affected\_switches$ **do**
    Remove flow entries in $switch$ related to $link$
  end for
  $shortest\_path \leftarrow Dijkstra(G, start\_node = source, end\_node = destination)$
  Update flow entries along $shortest\_path$
end if

2) Proactive Rerouting: In the proactive controller, the routing algorithm is also run at initialization. However, it calculates the paths differently. Just as in the reactive controller, a flood is generated if the destination is unknown. Once both relevant hosts are discovered, the paths are calculated.

For a packet to be able to take a backup path, each switch must know how to reach each host. Dijkstra's algorithm is run twice from both the source and destination if the hosts are previously unknown. These paths are traversed from both hosts to each switch. For each switch, a unique group table is generated to accommodate the backup paths. Two buckets are installed in each group table: One for the primary path and one for the backup. In contrast to the primary path, the backup is not calculated in its entirety with Dijkstra's algorithm. The controller simply checks its available ports, excluding the primary and ones leading out of the network, and selects the first one. Each bucket is assigned a watch-port equal to the flow out-port. Algorithm can be seen in Algorithm 3.

Algorithm 3 Proactive link failure

INPUT:
G: Network topology graph with switches and their link data
$hosts$: List of known hosts in the network.
$detect\_new\_host()$: Function to detect newly added hosts.

INITIALIZE: $known\_hosts \leftarrow \emptyset$
while True do
  $new\_host \leftarrow detect\_new\_host()$
  if $new\_host \neq$ None $\wedge$ $new\_host \notin known\_hosts$ **then**
    Add new_host to $known\_hosts$
    for all $switch \in G$ **do**
      $shortest\_path \leftarrow Dijkstra(G, start\_node = switch, end\_node = new\_host)$
      if $shortest\_path$.next == new_host **then**
        $update\_f(switch, new\_host, shortest\_path)$ {Update flow table}
      else
        $update\_p(switch, new\_host, shortest\_path)$ {Update group table primary bucket}
        if Another port available from $switch$ **then**
          $update\_b(switch, new\_host, shortest\_path)$ {Update group table backup bucket}
        end if
      end if
    end for
  end if
end while

D. Measurements

The baseline of the measurements is given in Figures 10 and 11 in Appendix A. These metrics are compared to Figures 12, 13, 14 and 15 in Appendix A, where link failures occur for the reactive and proactive controller. The proactive approach installs significantly more flows in conjunction with considerably higher processing times compared to the reactive approach. This is expected as the reactive algorithm installs backup paths alongside the primary path per switch for every new host. The reactive algorithm on the contrary, only installs flows in the switches along the computed path.

Measurements with link-down events are presented in Figures 12, 13, 14 and 15 in Appendix A. The proactive approach remains similar in installed flows and processing time. Due to precomputed backup paths, each switch can act accordingly without additional assistance from the controller. The reactive approach on the other hand exhibits greatly increased processing time and number of flows modified. The explanation is that the reactive controller reactively deletes existing links in the affected path and installs new ones each time a link failure event is detected. This process entails intervention from the controller. Due to controller delay an additional RTT of around 100ms is added to the reactive max ping time. Additionally the processing time for computation of a new path is also incorporated in the max ping. The average is therefore slightly higher in the reactive controller due to the massive spike in the max ping time.

VIII. Conclusion

This section answers and summarizes the research questions using the results discussed in the chapters above.

A. How can a path fail-over mechanism that detects real-time link failures and attacks be implemented using SDN?

An SDN-based path failover mechanism can be implemented either reactively or proactively. Different link failure detection methods can be most effective for each approach.

Reactive Approach: The SDN controller detects link failures in real time by monitoring LLDP packets. The controller dynamically removes flow entries from the affected switches' flowtable upon identifying a failure. The affected switches in this case are all switches that are a part of a path containing the broken link. All flow entries connected to those paths in the affected switches are removed.

Proactive Approach: The SDN controller configures group tables with the "watch port" property in advance. These group tables monitor specific ports for failure and automatically reroute traffic to backup paths without requiring interaction with the controller.

B. To what extent can traffic be dynamically rerouted to alternative paths during network transmission?

In SDN, traffic rerouting can be achieved through reactive or proactive path failover mechanisms:

Reactive Approach: The controller dynamically removes flow entries from affected switches' flowtable if a link failure is detected. The switch will send a new packet-in to the controller that will recalculate the shortest path using the Dijkstra algorithm, using a hop-based cost metric. This means the shortest path is determined by the number of hops. The controller then installs the shortest path by updating flow entries in the affected switches' flowtables, enabling the rerouting of traffic.

The provided reactive algorithm in this paper works for all network scenarios as long as a path to the destination

exists, however, the algorithm only works if one link is down at a time. Another flaw is that a processing time is induced whenever a link goes down, this is due to the reactive recomputation of paths. The processing time will be dependent on the number of switches involved in the path, more switches equals higher processing time.

Proactive Approach: When a new host is discovered in the network, the controller precomputes the shortest paths from each switch to this host as a destination using Dijkstra's algorithm with a hop-based cost metric. These paths are installed in the switches group tables with the primary path set as the default action for routing traffic to the host. If an additional output port is available, a backup action is added in another bucket with less priority. The group table uses the fast failover mechanism and the watch port functionality to monitor the status of the port related to each bucket. If the primary port is active, traffic is routed through it. However, if the primary port fails, the group table automatically switches to the backup path.

The given proactive algorithm does not handle backtracking or provide any method for neighboring nodes to detect link failures occurring elsewhere in the network. As a result, depending on the packet path, it may end up in a loop, making this approach unsuitable for certain network topologies.

The proactive approach has zero processing time since it does not rely on communication from the controller and has all backup paths preinstalled. Consequently, rerouting will solely depend on the time it takes for the watch bucket to detect port failure. The proactive algorithm can handle multiple link failures in contrast to the reactive algorithm.

C. How can Mininet and Ryu simulate a real-world network scenario to evaluate the recovery time and bandwidth of SDN-enabled routing during link failures?

Simulations of real-world network topologies can be done using Mininet. This virtual Mininet network should include hosts, switches, and links configured to emulate the desired network. There are also options to specify bandwidth limitations and latencies between the links. These options could be used to represent real-world scenarios artificially. The Ryu controller is then integrated to control the network, providing programmability for custom routing logic. Ryu can also integrate rerouting algorithms and failure detection mechanisms by installing, modifying, and removing flows in affected switches.

Once the setup is ready, traffic flows are generated between hosts using tools like iperf or ping, accessed through the Mininet CLI or API, to simulate traffic. Link failures are introduced in the network by disabling specific links within Mininet, which triggers Ryu's failure event. Ryu captures link failure events and actions can be taken accordingly.

During the simulation, key metrics are monitored. Recovery time is the delay between the link failure event and the successful traffic restoration on an alternate path. Bandwidth performance is assessed by observing throughput before, during, and after the failure to understand how efficiently the network adapts to changes. The collected data is analyzed to evaluate how effectively the SDN-enabled network responds to link failures.

ACKNOWLEDGMENT

E. F. M. M. R. would like to thank Mohammad Borhani for patiently answering all our questions. We are also grateful to Andrei Gurtov for his research guidance and Ulf Kargen for making this opportunity possible.

REFERENCES

[1] Yong-Gang Pan, Li-Ping Ni, Xiao Liu, and Zeng-Xiang Li. "Innovational network protocol - HIP". In: *Zhongshan Daxue Xuebao/Acta Scientiarum Natralium Universitatis Sunyatseni* 45.SUPPL. (2006), pp. 164–166.

[2] Pekka Nikander, Andrei Gurtov, and Thomas R. Henderson. "Host Identity Protocol (HIP): Connectivity, mobility, multi-homing, security, and privacy over IPv4 and IPv6 networks". In: *IEEE Communications Surveys and Tutorials* 12.2 (2010), pp. 186–204. DOI: 10.1109/SURV.2010.021110.00070.

[3] Sang-Il Choi and Seok Joo Koh. "Enhanced mobility management schemes in HIP-based mobile networks". In: 2013, pp. 306–311.

[4] Kuntal Gaur, Anshuman Kalla, Jyoti Grover, Mohammad Borhani, Andrei Gurtov, and Madhusanka Liyanage. "A Survey of Virtual Private LAN Services (VPLS): Past, Present, and Future". In: *Computer Networks* 196 (2021).

[5] Niels L. M. Van Adrichem, Benjamin J. Van Asten, and Fernando A. Kuipers. "Fast Recovery in Software-Defined Networks". In: *Third European Workshop on Software Defined Networks* (2014), pp. 61–66. DOI: 10.1109/ewsdn.2014.13.

[6] Yubaraj Gautam, Bishnu Prasad Gautam, and Kazuhiko Sato. "Experimental Security Analysis of SDN Network by Using Packet Sniffing and Spoofing Technique on POX and Ryu Controller". In: 2020, pp. 394–399. DOI: 10.1109/NaNA51271.2020.00073.

[7] Grace T. Shalini and S. Rathnamala. "A RYU-SDN Controller-Based VM Migration Scheme Using SD-EAW Ranking Methods for Identifying Active Jobs in the 5G Cloud Framework". In: *International Journal of Cloud Applications and Computing* 13.1 (2023). DOI: 10.4018/IJCAC.319031.

[8] Xuan Thien Phan, Nam Thoai, and Pierre Kuonen. "A collaborative model for routing in multi-domains OpenFlow networks". In: 2013, pp. 278–283. DOI: 10.1109/ComManTel.2013.6482405.

[9] R. Gopakumar, A. M. Unni, and V. P. Dhipin. "An adaptive algorithm for searching in flow tables of OpenFlow switches". In: 2016. DOI: 10.1109/NATSYS.2015.7489115.

[10] Rafael George Amado, Kim-Khoa Nguyen, and Mohamed Cheriet. “OpenFlow rule placement in carrier networks for augmented reality applications”. In: 2022, pp. 952–959. DOI: 10.1145/3477314.3507101.

[11] Huawei Technologies Co., Ltd. *OpenFlow: A Key Component of SDN*. Webpage. Accessed: 2024-11-26. 2024.

[12] Nuno Miguel Carvalho Galego, Rui Miguel Pascoal, and Pedro Ramos Brandão. “IPv6 in IoT”. In: *Lecture Notes in Networks and Systems* 773 (2024), pp. 89–94. DOI: 10.1007/978-3-031-44131-8_9.

[13] Melvyn Wray. “The sixth protocol”. In: *New Electronics* 41.22 (2008), pp. 43–44.

[14] Soonyong Sohn, Brian L. Mark, and John T. Brassil. “Congestion-triggered multipath routing based on shortest path information”. In: 2006, pp. 191–196. DOI: 10.1109/ICCCN.2006.286271.

[15] Pascal Merindol, Jean-Jacques Pansiot, and Stéphane Cateloin. “Providing protection and restoration with distributed multipath routing”. In: 2008, pp. 456–463.

[16] Jason Liu, Cesar Marcondes, Musa Ahmed, and Rong Rong. “Toward scalable emulation of future internet applications with simulation symbiosis”. In: 2016, pp. 68–73. DOI: 10.1109/SLSA.2016.020.

[17] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. Second. MIT Press and McGraw-Hill, 2001. Chap. 24.3: Dijkstra’s algorithm, pp. 595–601. ISBN: 0-262-03293-7.

[18] Deben Bhattarai. “Link Layer Discovery Protocol”. In: *Cisco Learning Network* (2020). Accessed: 2024-12-09. URL: https://learningnetwork.cisco.com/s/article/link-layer-discovery-protocol-lldp-x.

[19] Niels L. M. Van Adrichem, Farabi Muhammad Iqbal, and Fernando A. Kuipers. “Computing backup forwarding rules in Software-Defined Networks”. In: *CoRR* abs/1605.09350 (2016). arXiv: 1605.09350. URL: http://arxiv.org/abs/1605.09350.

[20] Simon Knight. *Topology Zoo*. https://topology-zoo.org/gallery.html. Accessed: 2024-11-18. Mar. 2011.

[21] Omni Calculator. *Propagation Delay Calculator*. https://www.omnicalculator.com/other/propagation-delay. Accessed: 2024-12-09.

[22] Hari Balakrishnan, Venkata N. Padmanabhan, and Randy H. Katz. “The effects of asymmetry on TCP performance”. In: *Mobile Networks and Applications* (1999). URL: https://link.springer.com/article/10.1023/A:1019155000496.

[23] Gang Wang, Zhifeng Zhao, Jialiang Peng, Rongpeng Li, and Honggang Zhang. “An approximate algorithm of Controller configuration in multi-domain SDN architecture”. In: *9th International Conference on Communications and Networking in China* (Aug. 2014). DOI: 10.1109/chinacom.2014.7054366.

APPENDIX

A. Graph – Scenario 1 and 3: Reactive and Proactive SDN controller sending general flow data without link failure

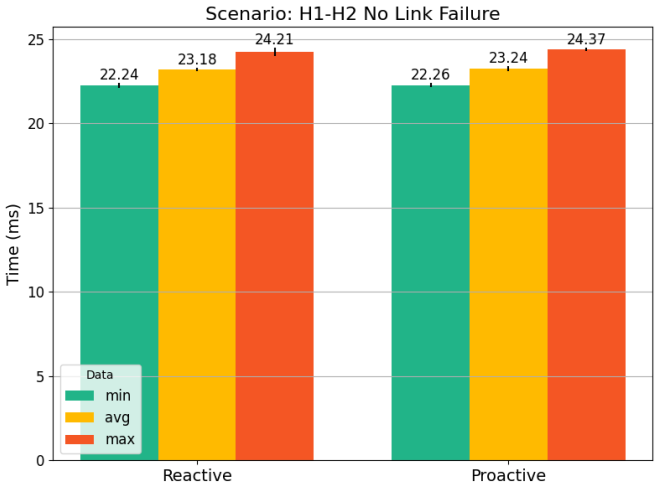

Fig. 10: Scenario 1 and 3 H1-H2

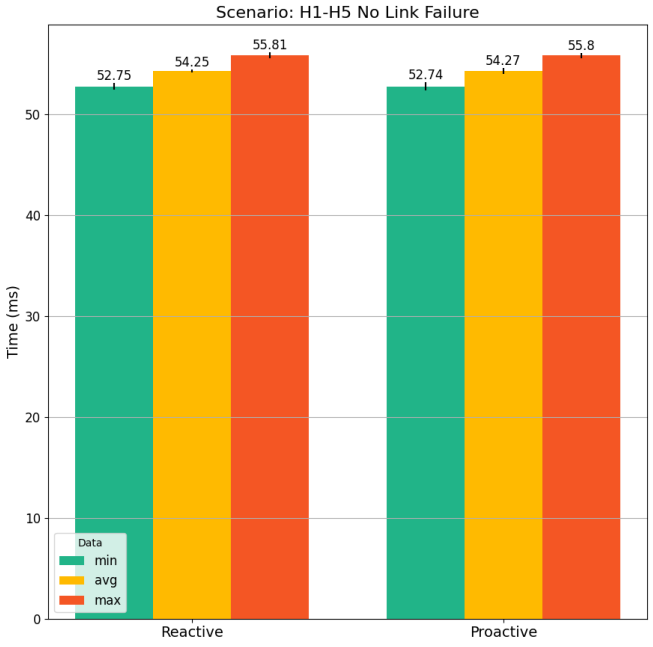

Fig. 11: Scenario 1 and 3 H1-H5

	Reactive		Proactive	
	H1-H2	H1-H5	H1-H2	H1-H5
Flows installed	6	12	74	80
Flows deleted	0	0	0	0
Processing time	8.82 ms	16.35 ms	28.45 ms	20.95 ms
Amount of lost packets	0	0	0	0

TABLE VI: Scenario 1 and 3

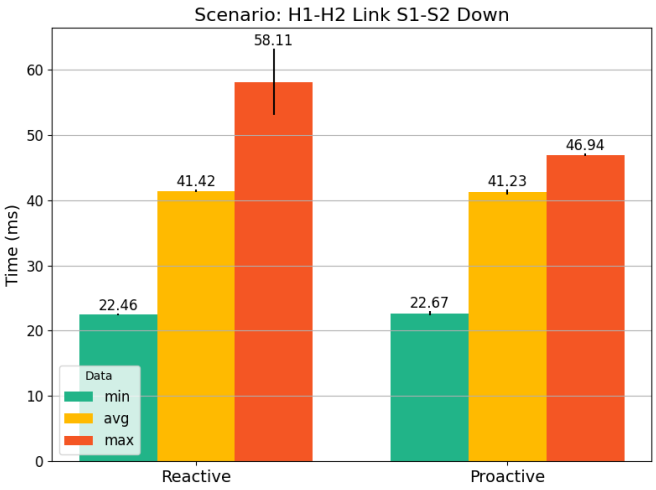

Fig. 12: Scenario 2 and 4 H1-H2 without controller delay

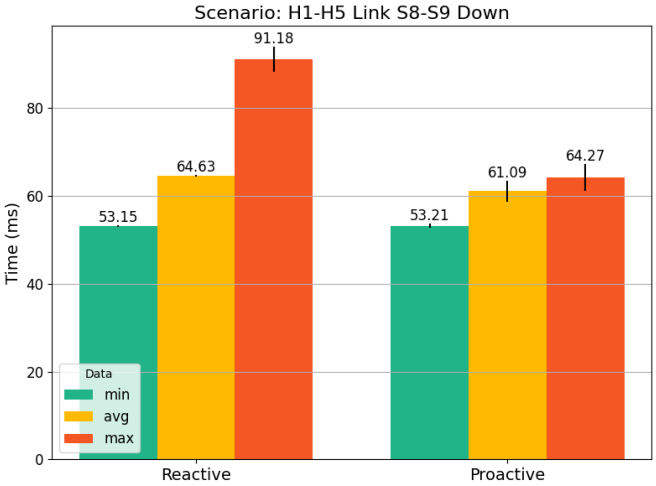

Fig. 13: Scenario 2 and 4 H1-H5 without controller delay

B. Graph – Scenario 2 and 4: Reactive and Proactive SDN controller sending general flow data link failure

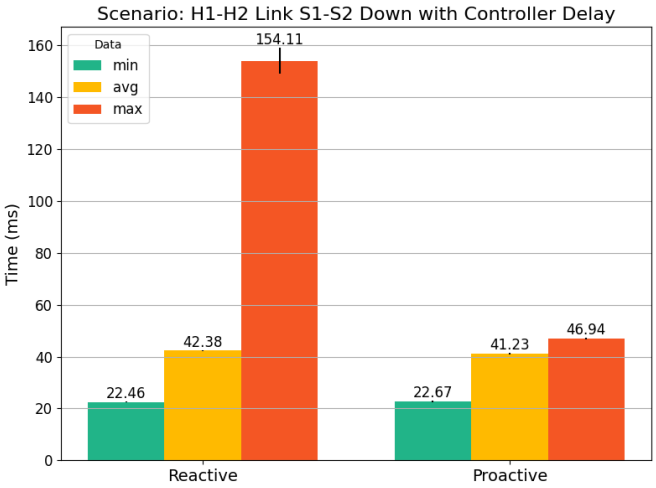

Fig. 14: Scenario 2 and 4 H1-H2 with controller delay

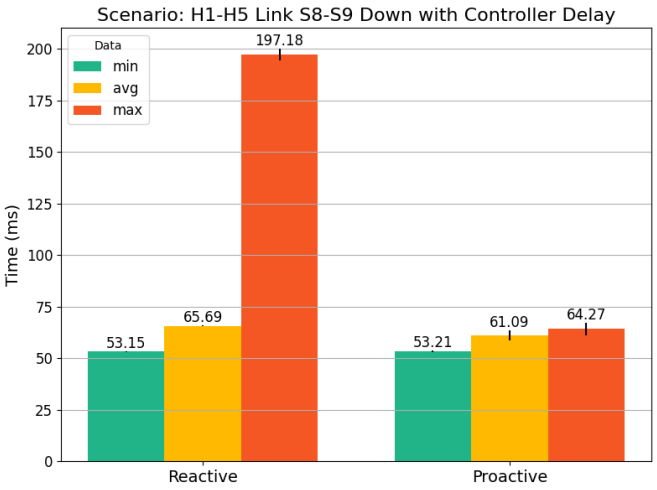

Fig. 15: Scenario 2 and 4 H1-H5 with controller delay

	Reactive		Proactive	
	H1-H2	H1-H5	H1-H2	H1-H5
Flows installed	14	26	74	80
Flows deleted	4	10	0	0
Processing time	14.05 ms	32.71 ms	29.23 ms	20.93 ms
Amount of lost packets	0.7	1.2	0.2	0.7
Percentage of lost packets	0.7%	1.2%	0.2%	0.7%

TABLE VII: Scenario 2 and 4

C. Scenario 5, 6, 7 and 8: Reactive and Proactive SDN controller sending HIP-VPLS data with and without link failure

It was not possible to obtain results with HIP-VPLS within the time frame of the project.