Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Networked Systems
ETH Zürich — seit 2015

Towards a new framework for integration of network planes

Semester Thesis

Author: Siddhant Ray

Tutor: Edgar Costa Molero

Supervisor: Prof. Dr. Laurent Vanbever

March 2021 to Jun 2021

Abstract

We present a new integration system for layer-3 routers in programmable networks, which allows for the creation of a new forwarding node. These nodes retain the traditional routing control plane from layer-3 routers. However, we replace the static data plane of the original routers by a new, programmable data plane. We create networks using these new forwarding nodes, replacing erstwhile layer-3 routers. These new nodes allow for the creation of smarter data planes and customized control planes to implement traditional routing algorithms. We then use our new forwarding nodes, combining route calculations from the control plane and smart forwarding using our programmable data plane, in order to create better network wide routing and traffic management. All in all, we now benefit from the best of each world, as we co-design the control and data planes.

Contents

Chapter 1

Introduction

Computer networks have traditionally used Layer-3 routers to forward IP traffic using several IP forwarding protocols such as Open Shortest Path First (OSPF), Routing Information Protocol(RIP), Border Gateway Protocol(BGP) and such others. We use the term Layer-3 to mean the standard Layer-3 of the network OSI model, which specifically deals with routing and forwarding for IP packets. Several recent advancements have also shown new protocols such as Multi-Protocol Label Switching(MPLS), which can be used to forward IP packets without relying on IP routing structures. However, in this project, we revisit the concept of IP routing and endeavour to create an improved system for IP routing, by distributing the control plane and data plane of the routers into separate entities, which can yield faster and scalable performance over traditional IP routing.

For a long time all routers have been engineered with a control plane , which acts as the "brain" of the router, computes the routing behaviour for all IP prefixes it needs to route to, and a data plane, which describes the forwarding behaviour of the packet, in terms of setting the next hop it must be sent to etc. In the original computer networks, the control plane and the data plane were tightly coupled and integrated into one device, with fixed functionality determined by the vendor i.e. once design, the operator could not reprogram the routing and forwarding behaviour. Then, with the introduction of Software Defined Networking(SDN), it became possible split the control plane and data plane, program the control plane and make it access the data plane using a new OpenFlow protocol [1]. The data plane itself, however still did not leverage any programmability at this point.

Today, in several routing package suites like FRR [2], the control plane computes the routes and populates the forwarding table in the data plane, which is the networking stack of the standard Linux kernel. The problem with this architecture is that the data plane is "static" and cannot make any decisions on its own. The data plane in the Linux kernel can only forward packets as instructed by the control plane. However, recent advancements have shown that data planes can be leveraged to have a certain degree of programmability, one instance of which was shown in P4 [3] switches. This is extremely promising as now, we can make better forwarding decisions at the data plane level, make updates in forwarding states, and if needed, even send feedback to the control plane, about the state of the forwarding table in the data plane. This also allows for the router to have a software based control plane, which can push the forwarding information to a programmable data plane such as a P4 switch, which can run in hardware and lead to forwarding packets at line rate.

In this thesis, we present one use-case we can achieve well with this combined architecture. We know that control planes are generally slower in computing best next hops in cases of link failures, especially protocols like Border Gateway Protocol(BGP), which must updates $O(100k)$ entries in the data plane's forwarding table. In our proposed setup, we can have a scenario in which the data

plane switches to a pre-computed "non-optimal" path in case of a link failure, which is disjoint from the main optimal path. [4] This update will be much faster than relying on the control plane for the path updates. However, since the control plane is in communication with the data plane directly, once it computes the optimal backup path, it can replace the "non-optimal" path chosen by the data plane. This should lead to reduction in loss of traffic (as the recompute and path update are now detached), leading to better performance in times of link failures. One thing we need to account for is this concurrent path update between the control plane and the data plane, must not lead to a race condition, which would be an important factor to consider in the design.

A use-case such as this, shows that it is worthwhile to explore the possibility of having a co-design in the networks routers, which can run traditional software based control planes, along with a programmable data plane. In our thesis, we present a new method to achieve this co-design.

1.1 Task and goals

In this project we attempt to create a co-design between software based control planes and programmable data planes for network forwarding devices such as routers.

1. We create a prototype of a Super-Node instead of a traditional Layer-3 router which runs a Layer-3 routing control plane along with a programmable P4 data plane.
2. We create the Super-Node with the control planes in their own network namespaces, while retaining the data planes in a common network namespace
3. We make our initial Super-Node topology with all nodes within a single Autonomous System(AS) and use the Open Shortest Path Protocol(OSPF) to attain network wide forwarding.
4. Finally, we extend the prototype to networks with more than one AS, use the Border Gateway Protocol(BGP) to route across multiple AS.
5. We also try to use the programmability of the P4 switch data plane to incorporate functions such as load balancing via ECMP.
6. As an additional functionality, our project also provides a method to add FRR routers as complete Layer-3 routers with standard Linux data plane to p4-utils [5], using P4-switches as L2-switches, by making some small modifications.

1.2 Overview

We present this thesis in several sections in order to organise and present our work in a coherent manner.

- In Chapter 2, we present the required background for the project, especially about the P4 switch architecture and the routing protocol suite(FRR) which we use in the project.
- In Chapter 3, we present the detained design of our Super-Node architecture, in which we combine a disjoint control plane with a programmable data plane, along with basic evaluation for our model.
- In Chapter 4, we present a more detailed evaluation and further use cases of running our Super-Node across different networks, along with implementation of functions such as load balancing.

- Finally, in Chapter 5, we analyse the possible impacts of our work and scope of future improvements to our Super-Node prototype based networks.

Chapter 2

Background

Our project attempts to explore the possibility of a joint, efficient design between the control plane and the data plane of the network. In order to achieve this, we make use of several existing tools and software suites, in order to make our network model. We present the required background and preliminaries for the same in section 2.1.

2.1 Background

2.1.1 The P4 Language and Programmable Switch Architecture

Programmable switches have been a key area of research in the field of computer networks recently, as to enable greater control over the data plane of erstwhile "static" data planes, which could not be reprogrammed to make packet level decisions. With the introduction of the P4 programmable language, this completely changed. P4 (named for "Programming Protocol-independent Packet Processors") is a language for expressing how packets are processed by the data plane of a forwarding element such as a hardware or software switch, network interface card, router, or network appliance. [3] Most of the targets which run P4 programs, have a separable control plane and a data plane. P4 provides an implementation to only forwards packets using the data plane i.e. it doesn't provide any implementation for the control plane.

Though P4 programs leverage no control over the network's control plane, it provides a P4Runtime API, which provides an interface to communicate between the control plane and the data plane. Network designers can use this P4Runtime API to communicate instructions from an external control plane, to the P4 programmable data plane, which in turn can make packet forwarding decisions.

The P4 language is developed to work independent of the protocol stack being used on the packet i.e. P4 programs can process packets irrespective of the actual higher level protocol the packets use for routing. We talk about P4 switches in the context of our project, which are switches capable of running P4 programs and carrying our smart packet forwarding decisions. Due to the protocol independent architecture, the switches do not know any information about the protocol and headers on the packet, till it has been processed by the switch program. In the latest version of P4 i.e. P416, P4 programs specify how the various programmable blocks of a target architecture are programmed and connected. In this context, they develop The Portable Switch Architecture (PSA) [6], which is a target architecture that describes common capabilities of network switch devices that process and forward packets across multiple interface ports.

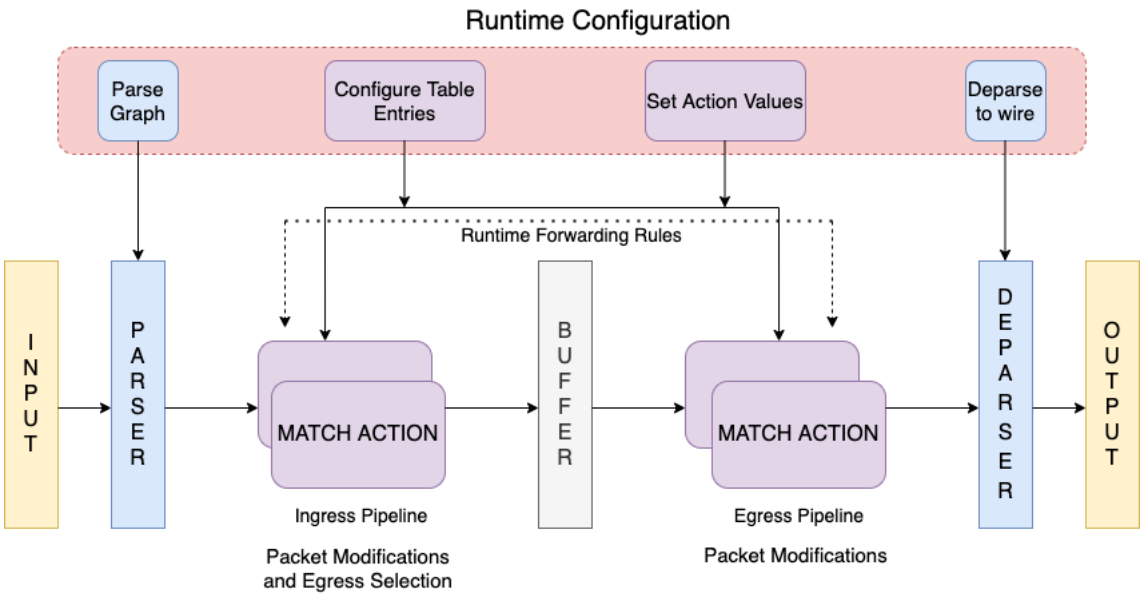

Figure 2.1: P4 Switch Pipeline

Figure 2.1 shows a typical P4 program pipeline. We divide the P4 program into several control blocks as shown in the figure. When a packet enters the P4 switch, it is processed by the following blocks in the given order :

1. Parser: The parser is a state-machine which extracts the required headers from the packet, which in turn decides on which packet headers, the P4 program can perform computation.

2. Match-Action Pipeline: A P4 programs contains structures called tables which use match keys to match onto a particular field in the packet header, port of incoming packet etc. It then sets the values for the actions contained within the table, which decide how the packet should be forwarded, changes made in the packet headers etc.

3. Deparser : Headers which have been parsed by the P4 program, need to be put back onto the output wire, else packets will be dropped and not forwarded by the switch. The deparser is responsible to adding the headers back in the correct order.

Apart from this, P4 allows packets to be processed by the ingress pipeline, replicated i.e. send the packet to multiple egress ports or recirculated i.e. send the packet back to the ingress port. After the packet arrives at the respective egress ports, it is queued, deparsed and forwarded on the required ports corresponding to the forwarding decisions.

The P4 community is open source and it is managed today by the Open Networking Foundation(ONF). The complete source code for P4 can be found here [7].

Software P4 target : The Behavioral Model

The current version of the P4 software target switch is the bmv2 switch (it stands for behavioural model version 2) and can be used for testing and debugging P4 data planes, and also the functionalities of the control plane software which is used with the data plane. The bmv2

model is maintained by p4lang [7] and hosts several implementations of the software switch such as simple_switch, simple_switch_grpc, psa_switch, etc. The input to the bmv2 switch is a JSON file generated from the P4 program by a P4 compiler and the switch interprets it to implement the packet-processing behavior specified by that P4 program.

2.1.2 Mininet and p4-utils

Mininet

Mininet [8] is a rapid, prototyping tool which helps create emulated networks inside a single OS kernel (VM, cloud etc.) Mininet is used to create virtual network topologies and it provides support for creating virtual hosts, switches, links and controllers. Mininet hosts run standard Linux network software. Mininet networks run real code including standard Unix/Linux network applications as well as the real Linux kernel and network stack, which makes it easy to test networking applications on the emulated network topologies and move it to real hardware, with minimal changes.

Mininet provides an implementation to define custom switches by inheriting from its base implementation of the switch node. By default, Mininet is configured in a way to run all switches of the network in same network namespace, however it provides a method to move the nodes into their own network namespaces if needed. Mininet thus creates kernel or user-space switches, controllers to control the switches, and hosts to communicate over the simulated network. The links between hosts and switch pairs in Mininet are defined as virtual ethernet(veth) pairs.

Mininet also provides a topology aware command-line-interface(CLI) which can be used for real time monitoring and debugging, as it provides several CLI methods for interface management, link changes etc. Mininet is an open-source implementation and the entire code for the same can be found here. [9]

p4-utils

p4-utils is an application software which is used to build, develop, debug and test P4 networks. It is an extension to Mininet with new classes of switches which support execution of P4 programs. p4-utils is equipped with the original functions of Mininet, along with a new P4 switch node within Mininet, which uses the simple_switch model from P4's bmv2 switch architecture. p4-utils also includes the P4 compiler to compile the relevant P4 program onto the software target. The current version of p4-utils used in this project defines the topology of the network, with the parameters for the individual nodes in a JSON file, which is used to build the topology. Further, the P4 program written for the p4switches is executed, and several methods are provided for debugging and testing for the programmable data plane created. p4-utils retains the original Mininet CLI and also provides a simple_switch_CLI to connect to the individual p4 switches, in order for faster and smoother evaluation and testing.

p4-utils is an open-source software and its implementation can be found here. [5]

2.1.3 FRRouting(FRR) Suite

FRR is a free, open-source IP software routing suite which provides implementations for several routing protocols such as OSPF, RIP, BGP, ISIS etc. The role of FRR in a networking stack is to exchange routing information with other routers, make routing and policy decisions, and inform other layers of these decisions. In general, the FRR routing policies install routing decisions into the OS kernel, and the kernel networking stack makes the required forwarding decisions by filling

in the forwarding tables. In addition to dynamic routing FRR supports the full range of layer 3 configuration, including static routes, addresses, router advertisements etc. [2]

FRR doesn't implement a single routing program which configures all routing protocols, like many traditional routing software. In FRR, each routing protocol is implemented as an individual daemon which all run in parallel. The routing daemons do not directly communicate with the data plane in this scenario. FRR provides a special daemon called zebra which acts as a middle man and is used to communicate routing information to the kernel and coordinate routing decisions. Also, the zebra daemon is used for basic configuration functions such as interface IP addresses, enabling forwarding etc.m The system architecture for the FRR routing daemons is shown in Figure 2.2.

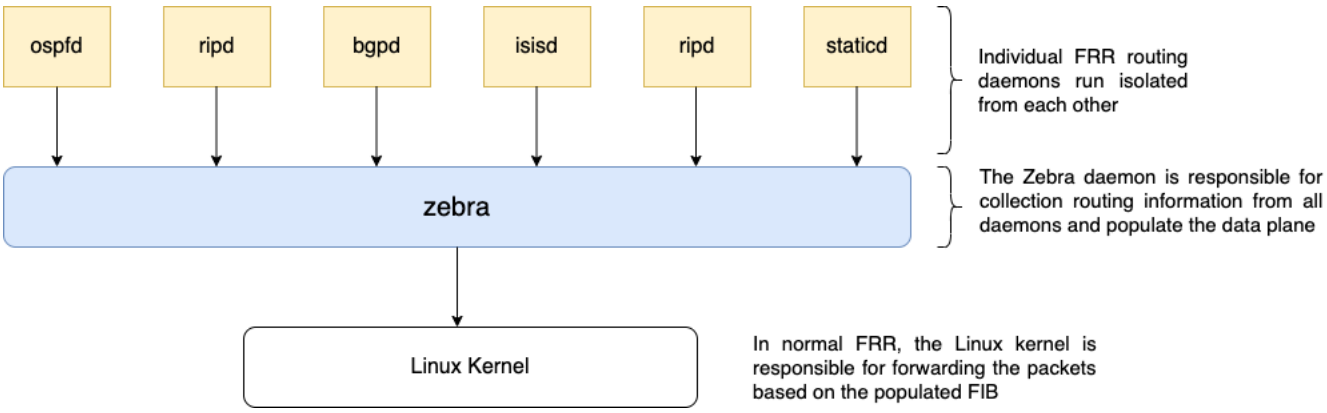

Figure 2.2: FRR Routing Daemons Architecture

The use of individual daemons for every routing protocol in FRR is advantageous as in removes inter dependencies. Individual routing daemons can be started at any time and if one daemon misbehaves, it does not cause the entire routing process to crash. Each daemon can individually be loaded at run time if needed. Daemons can easily be enabled or disabled using the FRR configuration, which can be run either in integrated mode with a single configuration file for all daemons or in distributed mode with an individual configuration file for each daemon.

The zebra daemon is always active as it is responsible for instructing the kernel with the required forwarding behaviour for the packets. FRR also provides the option of running the protocol stacks in the root kernel namespace or in user defined custom namespaces.

Forwarding Plane Manager(FPM)

In FRR, the zebra daemon is responsible for aggregating routing information from different protocols and selecting the best routes to fill the Forwarding Information Base(FIB) of the OS kernel. Zebra passes these routes on to the kernel which uses its IP stack to forward packets as described in the FIB for every IP prefix, which have been computed by FRR.

FPM is the forwarding plane manager of the zebra daemon which is used to receive the forwarding plane with the routes received by zebra from the individual protocol daemons. This can also include additional routes which might have been present in the kernel. FPM acts as a server which receives all the routes sent by the zebra daemon.

Zebra uses a TCP connection to connect to the FPM server periodically, on the FPM port 2620 and server address 127.0.0.1, which can be reconfigured. Once the connection is set up, the entire forwarding table is sent to the FPM server. This behaviour is part of FRR's service which is to

provide the forwarding information computed by Zebra to external applications.

The default message format for the messages sent to the FPM server is the Netlink protocol on Linux. By default, the sending messages to the FPM server is disabled and in order to enable FPM to receive the routes, the zebra daemon must be configured with the *–enable fpm* flag and the module must be started with *-M fpm*. One thing which is key to observe here is that when zebra sends routing messages to the FPM server, it doesn't stop populating the kernel, which continues to receive updates in its FIB. Also, the FPM in FRR replaces routes for the same prefix i.e. if a route is added for a particular IP prefix, followed by another for the same prefix, the first route is replaced with the second route, making FPM contain only the latest updated route for the IP prefix.

2.2 Related Work

The goal of this thesis is to develop a new network node with a router control plane and programmable data plane, which replaces the static data plane of the original router. We present some cases where similar research has been carried out, using the FPM module in FRR. In [10], they present a system of a Point-to-Point-over-Ethernet(PPPoE) plugin for Vector Packet Processing(VPP), in which they use FPM messages from FRR's Zebra daemon to co-create a daemon running BGP, with VPP.

The system of Super-Node's that we create, can be used for offloading functions from the control plane to the data plane. We show some related studies on offloading network functions between these network planes.

Offloading functions in network nodes between the control plane and the data plane has been an ongoing area of research for a long time. Most of the research in the field, comes down to one of two directions. The first is moving functions which need to have faster compute, from the control plane to the data plane, and the advancements in P4 programmable switches enable us now to run functions directly in the data plane. In [11], Edgar et al. show this by offloading functions such as failure detection, connectivity retrieval directly to the hardware data plane in order to achieve faster performance. In [12], Naga et al. show the possibility of caching most hit switch table rules in the TCAM, which allows faster information retrieval in the data plane, and leaving the less frequently hit rules to be processed by the control plane. Offloading network functions also exist in the other direction. In [13], Wintermeyer et al. show as one of their results that rarely used functions in the data plane can be pushed up to the control plane, without any significant overhead in latency.

Chapter 3

Design

Our project aims at building a co-design network plane which consists of an independent control plane which computes the routing behaviour for our network. It has independent routing protocols running which compute the best possible routes which routes to reach the hosts in our network. For our routing control plane, we use the FRRrouting suite, which acts as a tradtional control plane. However, for our data plane, we no longer rely on the kernel's static data plane, which can only receive routes from the control plane, update its FIB and forward packets, without being able to make any decisions on its own. We leverage the recent advancements made in programmable switches, and attempt to replace the forwarding data plane of the kernel with a P4 programmable data plane, the concepts of which were introduced in sub-section 2.1.1.

3.1 Network Super-Node Design

The routing and forwarding device in our network consists of a control plane which runs the FRR routing daemons and the data plane which is a P4 bmv2 simple_switch architecture, and we will refer to this node as a Super-Node as it has detached control and data plane implementations. The basic network topology from which we started developing the idea of this super node consisted of a pure FRR router, which had FRR for its control plane and Linux kernel's IP stack as the data plane, which was populated by the zebra daemon.

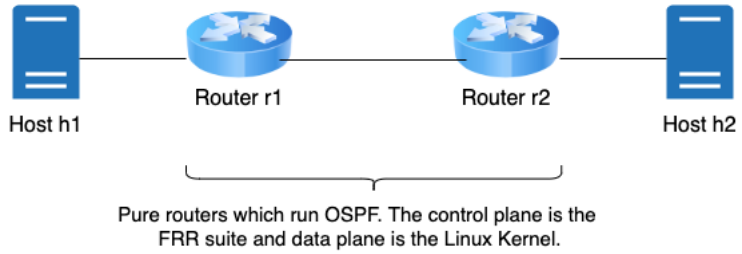

Figure 3.1: Simple router setup

We setup this test architecture, using virtual nodes in the kernel, in Figure 3.1 in order to study the possibility of replacing the Linux kernel as the data plane with a P4 programmable data plane, as the Linux kernel only provides a static data plane. However, we realised that FRR doesn't provide a good way to stop populating the FIB in the Linux kernel's data plane, if the routers are physically connected, the link state is up and forwarding of packets is possible. Another problem

we discovered is that if the kernel's data plane FIB is populated, it will always use the IP stack in the kernel for forwarding between the nodes. We explored the possible of directly using the copy of the forwarding received from the zebra daemon's FPM server, however as it doesn't prevent the Linux kernel's FIB from being updated either, FPM alone did not provide a solution.

As there was no way to stop the kernel's forwarding plane from being populated, the only way to prevent the kernel from forwarding was to was to remove the links between the routers, and forward all route updates and messages only via the P4 switch's data plane. This allows us to fool the router's control plane to believe that there are connected links to other routers in the network, which in reality do not exist. Following this, we present our Super-Node based network topology, which incorporates this idea in Figure 3.2.

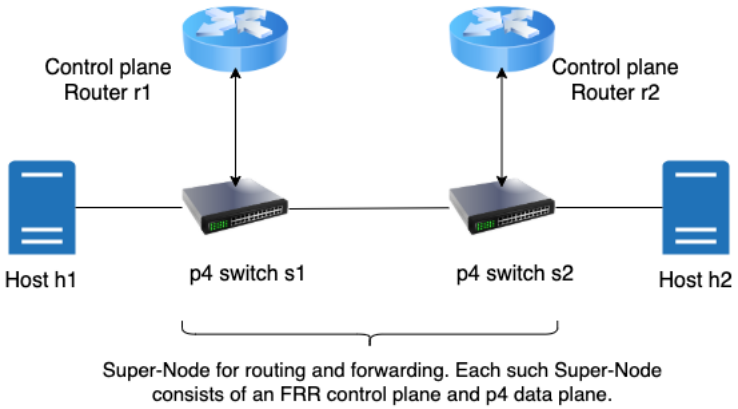

Figure 3.2: Super-Node setup

In our Super-Node, we run an FRR control plane for computing the routes and a P4 data plane for forwarding behaviour. As the routers no longer have physical connections, even if the kernel FIBs get updated, they cannot forward as they are not connected to the hosts or the other routers. We will call the router part of our Super-Node as a control plane router as it can only compute routes but cannot push the routes to the kernel's data plane. However, for computing the routes, it still needs to exchange messages with the other routers, for example, if running Open Shortest Path First(OSPF), OSPF hello messages, link state update messages etc. will need to be exchanged in order for the control plane router to compute the routes which will now be done via the P4 switches. We will mention the details of this exact design in a later section. However, with the Super-Node setup, we manage to fool the control plane routers to generate routes by making it believe that it is directly connected to the other control plane routers.

3.2 Network Topology Setup and Design

We create a virtual topology for our network with the Super-Nodes. As there did not exist any universal way to achieve this, this task was not as trivial as envisioned. We realised that replicating the complete behaviour of the control plane routers in our Super-Nodes will not be straightforward, as we would need to monitor the complete behaviour of the router, all different kind of messages sent in a real router network, and then decide what the P4 switches would need to forward in order to achieve the same performance. For this, we first created a base setup with just the routers as shown in Figure 3.1. To do this, we used Mininet [8] to create a 2-router, 2-host network topology.

As Mininet does not provide an implementation to create Layer-3 routers, we added a custom router definition on a Mininet switch. To do this, we created a new virtual Router node which inherits from the Switch class of Mininet. One thing we needed to ensure here is that every router created must be assigned to its own namespace i.e. having a complete network stack to itself. By default, Mininet runs all the switches in the default root namespace as the switches only perform Layer-2 forwarding. However, Mininet provides a way to move nodes into nameless namespaces, which are linked to the process ID(PID) of the created node. We moved the router to this namespace for our setup.

Mininet on its own, does not provide any routing functionality on our created routers as there is no implementation for the same. Hence, on our created Mininet virtual routers, we ran instances of routing daemons provided by FRR. On each Router node in our 2-router topology, we ran the routing daemons and configured the daemons using FRR configuration files. For our initial tests, we only configured the zebra daemon for configuring the IP addresses of the router and its interfaces, and enabling forwarding and the OSPF daemon on the routers to compute the routes for achieving connectivity in our topology. The hosts were configured as Mininet hosts, with IP addresses and default routes which allowed packets to reach the routers. With this setup, we were able to achieve a topology with running FRR daemons on routers created in Mininet, and using OSPF, achieve network-wide connectivity. In this base setup, the Linux kernel's data plane was responsible for forwarding IP packets.

For our Super-Node, we decided to use p4-utils [5] as it already provided good utilities to integrate P4 switches with Mininet. However, the current implementation of p4-utils only allows topologies to be created using JSON files, instead of using Python scripts which we were using in our base Mininet topology. Also, the p4-utils implementation has existing IP assignment strategies which assign IP addresses to the hosts and P4 switches together, in order for them to be used for Layer-3 forwarding if needed. We did not want this, as for our P4 switches, we did need IP addresses, as our control plane was the FRR daemon. In our base setup, we had kept our router definition as minimal and were running instances of the FRR routing daemons while we after we created the topology. This design idea did not fit into the p4-utils structure and we had to change it. One more problem we initially discovered that since P4 switch node and router node, both inherit from the Mininet switch class, it led to overlap on several occasions, leading to problems in creating the topology. To fix this, we did several things:

- We added a new manual IP assignment strategy to be used with Super-Nodes which differentiates between routers, P4 switches and hosts, and configures IP addresses only for hosts.
- We add running the instance of our FRR daemon directly inside our router definition, in order to prevent unwanted clash with the rest of the topology
- We treat P4 switches and FRR control plane routers as entirely different objects, configure them separately to prevent any overlap
- Finally, we add one link between the control plane router and the P4 switch which creates our Super-Node

These steps created our topology, with only one problem still left. The control plane routers in our topology created via Mininet had just one interface which connected to the P4 switch. In order to configure them with our FRR configurations, the interfaces which "should connect" to the others, had to be configured in order for the zebra daemon to recognize them. However, we know that the control plane router should have as many interfaces as many there are interfaces on P4

switches connecting to each other in the topology. This is because the control plane router and the P4 switches have a 1 to 1 mapping, which means there must be the same number of interfaces on control plane routers. To fix this problem, we add a dictionary of CP-interfaces to the router's parameters, which we compute by enumerating the switches and counting the interfaces which connect to other switches and hosts, and make the router node add these CP-interfaces for the control plane routers in the router definition using Linux IP commands.

3.3 Control Plane and Data Plane Co-Design

Once our setup was designed for Super-Nodes, our objective was to use the P4 switches (which was the data plane of our Super-Node) and the FRR control plane routers (which was the control plane of our Super-Node) to do the following:

1. Act as a transparent forwarding node for the control plane router's messages in order for them to exchange messages and compute routes based on the protocol daemon
2. The control plane routers must send the complete routing information to its FPM server
3. The P4 switch must read the FPM server, select the useful routes and write table entries for forwarding packets in the network

3.3.1 Generating Routing Messages

Most routing protocols rely on some kind of message exchange to determine neighbours, link-state etc. in order to compute the routes it must use to reach them. For example, OSPF relies on a hello message exchange which lets a pair of neighbours determine if they are connected, by periodically sending and receiving hellos. Following this, they send link-state update and acknowledgement messages in order to establish that they are connected via an active link, and then rely on hello messages exchanged to determine the state of the link thereafter. We initially used only OSPF in an attempt to generate routes and achieve connectivity. In our base setup, this was done via the direct links from the routers.

In our Super-Node setup, the control plane routers had been configured identically to the base setup, with the same interface configuration, without real links. Thus, the control plane routers attempt to send out the same messages on these interfaces, which it would when it was connected to other real routers. In order to forward these messages via the P4 switches, we first created packet sniffers using Scapy [14], which allowed us to capture the packets sent out on the control plane router's interfaces, and re-inject them onto the interface which connects the control plane router to the P4 switch. The switch was programmed to forward the packets without making changes, so that the other control plane router receives the message sent for routing.

However, this was a problem as routers do not process a packet if it receives it on an unexpected interface. Since the control plane router expected to receive the packet on its interface which should have been connected to the other router, but instead received the packet on the interface connected to the P4 switch, it did not process the packet with the OSPF hello message. To fix this, we would need to re-inject the packet back onto the the intended interface, using another packet sniffer. However, double re-injection would make the entire system much slower and complex to handle, as packet sniffers needed to filter much more to ensure correct re-injection of packets.

Hence, we decided to change our Super-Node to include more links between the control plane router and P4 switch, in such a way that every interface of the control plane router is connected

to a new "dummy interface" on the P4 switch. Figure 3.3 shows us the new Super-Node with additional links to dummy interfaces.

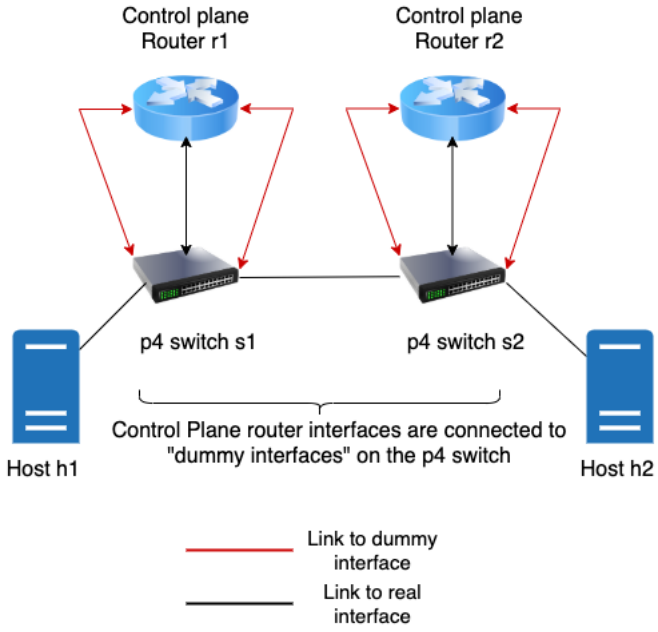

Figure 3.3: P4 switches with dummy interfaces

This was helpful was now we did not need to re-inject packets as the control plane routers automatically send the hello messages on the link to the dummy interfaces, which the P4 switch could forward. However, there was a one small issue here.

Our topology was built using Mininet, which does not allow for multiple links to be set between a pair of nodes, hence we had to look for another way to do this. Also, since the control plane routers are in their own namespace and hence the CP-interfaces we created before in 3.2 were inside the router namespaces. On the other hand, the P4 switches were in the root namespace, and hence we could not directly create the dummy interfaces and add links from them to the control plane routers. To solve this, we created the CP-interfaces and the dummy interfaces for the P4 switches in the root namespace, added a virtual ethernet link(veth) using Linux IP commands and moved the CP-interface to the router namespace. As Mininet namespaces are unnamed, we used the PID of the namespace in order to move the CP-interface.

With all these steps in place, we were able to forward the correct messages for OSPF from one control plane router to another, enable the router to generate routing information and receive the same on the router's FPM server.

3.3.2 P4 Data Plane Design

The P4 switch in our Super-Node acts as the data plane for forwarding packets to the destination IP prefix. Our P4 data plane contains tables to match on certain fields of the packets, and actions which set the outgoing port of the IP packet. We also need to make a distinction here between packets which must be routed from control plane routers to each other, to determine the routes for the IP packets. Apart from this, we will have tables to match on IP packets and forward them between hosts.

Pipeline for control plane router forwarding

We need to ensure that packets from the control plane router CP-interfaces, reach the correct CP-interface of the other control plane router via the P4 switch. As we have dedicated links now between the CP interfaces and the dummy interfaces on the P4 switch, which have a 1 to 1 mapping, we have a P4 table which matches on the incoming port on the P4 switch, which the dummy interface. On matching on the port, we have written a simple Layer-2 forwarding action, which sets the output egress port for the packet. As these interfaces only receive packets meant for the control plane routers, it is enough to keep simple forwarding rules for this. We identified that two kinds of packets need to be forwarded for the control plane routers to set up OSPF routes, i.e. Address Resolution Protocol(ARP) packets and OSPF packets containing hello message, link-state-update etc packets. We could achieve this by checking the kind of packet, and then hitting the appropriate tables which contain actions to forward these packets to the required egress port. We write rules for these tables as static entries in the switch as they are simple, constant and must be the same forever. As we have complete control over programming the data plane, we can ensure that tables get hit only if a certain kind of packet is received by the P4 switch.

Pipeline for IPv4 forwarding

We have a second kind of forwarding required for the P4 switches, which must forward IPv4 packets sent from one host to another. The routes computed by the control plane routers contain the forwarding information for IPv4 which must be written as table entries in the P4 switch. As the routes computed by OSPF already have end to end connectivity in the network i.e. the routing information has a route to reach every host from every host, we use a simple IPv4 forwarding logic which updates the MAC addresses for every packet, and sets the egress port based on the forwarding rules in the routes, which the control plane computed and sent to the FPM server. We write rules into the tables using a controller, which we will explain in the next section.

Apart from plain IPv4 forwarding, we also have a Equi-Cost MultiP-Path(ECMP) for load balancing across multiple paths. ECMP carries out load balancing on multiple paths between the source and the destination, if the paths have equal cost. Our control plane already returns us multi-path routes for OSPF, as FRR's OSPF daemon calculates ECMP on its routes, and returns all routes for equal cost paths. We write an action in our P4 program to calculate hash values for packets to be put on the same path. As ECMP ensures packets of the same flow are put on the same path, we compute a 5-tuple hash value for the packet given as:

$$\text{ECMP\_hash} = HashAlgorithm(ipv4.Src, ipv4.Dst, tcp.Src, tcp.Dst, proto) \mod \#OP \quad (3.1)$$

where ipv4.Src is the source IPv4 address, ipv4.Dst is the destination IPv4 address, tcp.Src is the TCP source port number, tcp.Dst is the TCP destination port number, proto is the IPv4 protocol used and #OP is the number of output ports, determined by the number of paths of equal cost. In our P4 code, we have an table ecmp_to_nhop, which matches on the ECMP has value, calculated if multi-path is present, and sets the egress port as a different output port for each different hash value calculated. The hash algorithm we use is the CRC16 Hash algorithm, as it is available as a P4 extern function.

A summary of all the tables and match-action logic we use in our data plane program is given in table 3.1.

Table	Match-Key	Action	Type
ARP	Ingress Port	Forward ARP packets	Control Plane
OSPF	Ingress Port	Forward OSPF packets	Control Plane
IPv4	IPv4 address	Forward IPv4	Data Plane
ECMP	ECMP group, ECMP hash	Forward IPv4 using ECMP	Data Plane

Table 3.1: Summary of our P4 program's match-action pipeline

3.3.3 FPM and P4 Switch Controller

Our controller for the P4 switches consists of two sections, an FPM server which contains the routes which should be populated in our P4 forwarding tables and a python controller, which works with p4-utils in order to connect to the P4 switches and populate the table entries for forwarding IPv4 packets. In the normal scenario, with simple P4 switches and no external FRR control plane, it is possible to have a single controller for all the switches as all the switches are in the same root namespace. However, for our scenario, an instance of the controller must be run inside every router namespace, as every router has an FPM server inside its namespace, which cannot be accessed from outside the router namespace.

To solve this, we connect to the P4 switch using the controller from inside the router namespace. For our P4 simple_switches, it is possible to connect to the switch using a controller in the same namespace, as long we have the thrift port of the switch. For connecting to the P4 switch, from inside the router namespace, we assign the interfaces which connect the control plane router and P4 switch (the real link for the real interfaces here, not the dummy interfaces) a pair of IP addresses, as for the simple_switch architecture, it is possible to connect to the switch from outside the namespace via a thrift IP address. This IP address pair for the P4 switch becomes our control interface, which helps the data plane of the switch connect to the control plane, similar to the CPU port in a real switch. We took care that the IP address assigned to the control interfaces, doesn't overlap with the IP addresses of the hosts and control plane router's CP-interfaces. We illustrate this in Figure 3.4.

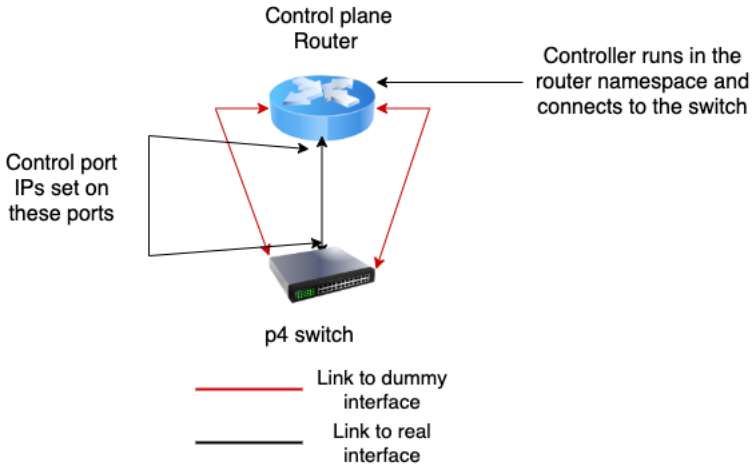

Figure 3.4: Controller to connect to P4 switch

The FPM server receives routes from the control plane router's zebra daemon in the form of Linux Netlink messages. To understand the messages, we wrote a Python script to decode the raw

data bytes of these messages into readable form for the switches to understand. This is possible as FPM frames all data with a header to help the decoder understand how many bytes it has to read in order to read the full route message. Also, we realised that since we had moved the CP-interfaces into the router namespace from the root namespace, we also needed to map the output interface values returned by the FPM server (as they got altered in the root namespace), to real output port numbers, using our controller. We present one sample decoded FPM route message for clarity:

```
{'family': 2, 'dst_len': 24, 'src_len': 0, 'tos': 0, 'table': 254, 'proto':
11, 'scope': 0,'type': 1, 'flags': 0, 'attrs': [('RTA_DST', '10.2.0.0'),
('RTA_PRIORITY', 11), ('RTA_GATEWAY', '10.0.1.2'), ('RTA_OIF', 26715)],
'header': {'length': 60, 'type': 24, 'flags': 1025, 'sequence_number': 0,
'pid': 0}}
```

With reference to Figure 3.2, this routing message shows how a packet from the host connected to the $r1 - s1$ Super-Node, must reach the $r2 - s2$ Super-Node. This is an OSPF learnt route for forwarding and we select the necessary parts of this route to populate the forwarding tables in the P4 switch. We consider the *dst_len*, which gives us the IP prefix, the *RTA_DST* which is the destination host IP prefix and the *'RTA_OIF'* which gives us the output port for forwarding packets(we mapped this port to the actual port for forwarding as mentioned above).

Using this information in the routes received from the FPM server inside the control plane router's namespace, we write the entries to the actions inside the tables we mentioned in section 3.3.2. Doing this, we are able to achieve end to end connectivity between the hosts, using our Super-Nodes for forwarding. For developing our setup and architecture, we had used a 2 host , 2 Super-Node topology as depicted in Figure 3.2.

Chapter 4

Working on Different Use-Cases

For creating our Super-Node design, we have only worked with OSPF as the routing protocol, for a 2 host, 2 Super-Node topology, with a single path for connectivity. We need to test our design and check its working on larger topologies, with multi-path routing if present. Also OSPF is an Interior Gateway Protocol(IGP) which only routes between routers in the same Autonomous System(AS). We also need to evaluate how our Super-Node design works with control plane routers across multiple ASes, which are connected via an Exterior Gateway Protocol(EGP) such as BGP. In this section, we present running both multi-path routing and routing across various ASes as evaluation and use-cases for our Super-Node based networks.

4.1 Testing on Multi-Path Topologies

For setting up our Super-Node design and testing to check for end to end connectivity, we had used OSPF on a 2 host, 2 Super-Node topology as shown in Figure 3.2. As we mentioned in our design, we had programmed the P4 data plane to perform ECMP load balancing on paths of equal cost, if the packets belonged to the same flow. To test this and check if multi-path routing works on the required topology, we set up a new 3 Super-Node network in form of a clique as shown in Figure 4.1 . Note that we remove the links between the CP-interfaces and the P4 switch's dummy interfaces in the figures, for the sake of visual clarity, they are still present.

We set the OSPF costs on the output interfaces of the control plane routers in such a way that there are two paths of equal-cost from host $h1$ to host $h2$. The first path is $h1 \leftrightarrow s1 \leftrightarrow s2 \leftrightarrow h2$ has a cost of 4 and the second path, $h1 \leftrightarrow s1 \leftrightarrow s3 \leftrightarrow s2 \leftrightarrow h2$ also has a cost of $3 + 1 = 4$. Thus, IP packets from $h1$ to $h2$ and vice-versa must be load balanced by ECMP on both the paths.

In order to test this, we built a custom packet generator using Scapy [14], where we crafted TCP packets to be sent from $h1$ to $h2$ and the reverse also from $h2$ to $h1$. Then, we monitored the interfaces on each of the link to confirm that packets were indeed load balanced on different links between the pair of hosts $h1$ and $h2$. As the number of multi-paths for us, is decided by the control plane routers who learn the topology, we use this information to set the number of output ports accordingly for the packets to put on multi-path when needed. Our P4 switch program checks if the router computes multi-path routes, and sets multiple forwarding ports using the ECMP hash table, mentioned in section 3.3.2.

In order to ensure more robust testing, we also repeated the experiment with different OSPF costs, and performed the same experiment with the host pairs $h1$ and $h3$ and $h2$ and $h3$, where multi-path does not exist. We saw that the load balancing with ECMP works only where then are multiple paths of equal cost in our setup, from the control plane router's point of view, else normal

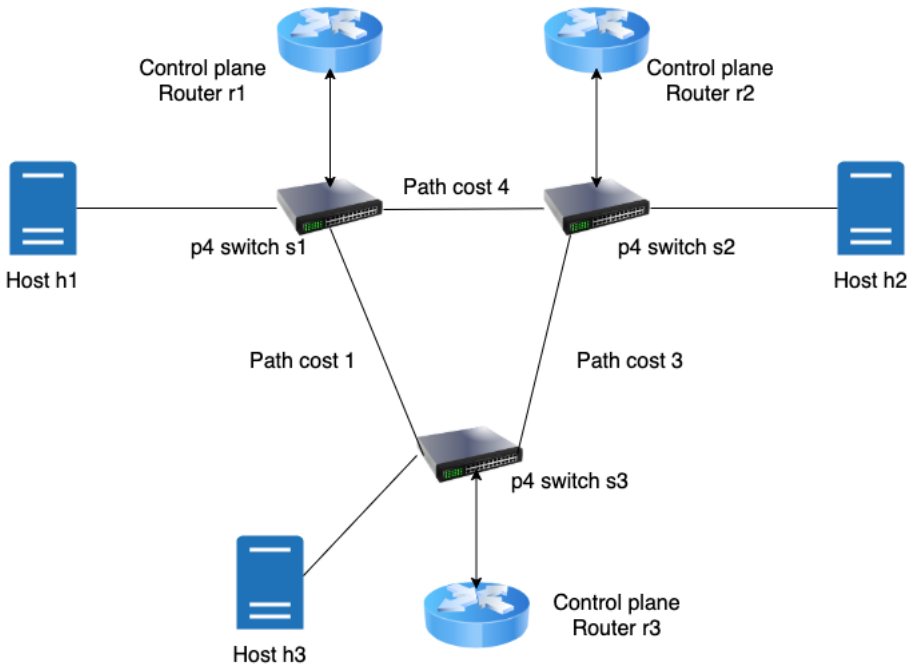

Figure 4.1: 3 Super-Node Multi-Path

IPv4 forwarding is performed by the P4 switches. As ECMP works well in our setup, in future this can also be extended to other load balancing methods such as weighted-ECMP, round robin or FlowLet [15] based multi-path by making appropriate modifications in the P4 program.

4.2 BGP and Multiple Autonomous System Topologies

So far, we have only considered OSPF routing within one Autonomous System(AS). We now try to extend our Super-Node based networks across multiple ASes, and try to run Border Gateway Protocol(BGP) daemons on the routers, in order to communicate routes across multiple ASes. One thing we need to keep in mind is that as we have detached the control plane and data plane in our Super-Node, the P4 data planes are all in the same root namespace and do not have any concept of different ASes. To build our new network, we now run eBGP peering on the border Super-Node's control plane routers, and run iBGP in the Super-Nodes which have a their control plane router's within the same AS. We also run OSPF as an Interior Gateway Protocol(IGP), independently within the AS as of now. We thus, create a network topology with 2 AS, by having identical configuration(as Figure 4.1 within each AS as of now, for testing. Figure 4.2 shows our system with 2 AS, which are as per the control plane routers, the switches are all in the same namespace, and do not know the difference.

A few things change when we introduce the concept of multiple AS for our setup.

1. The border Super-Nodes now behave differently as their control plane routers have more CP-interfaces to connect to other border routers, and hence the P4 switch correspondingly has more dummy interfaces. This needs to be recognised by our controller.

2. IP packets to be routed within the AS have different routing prefixes as compared to packets

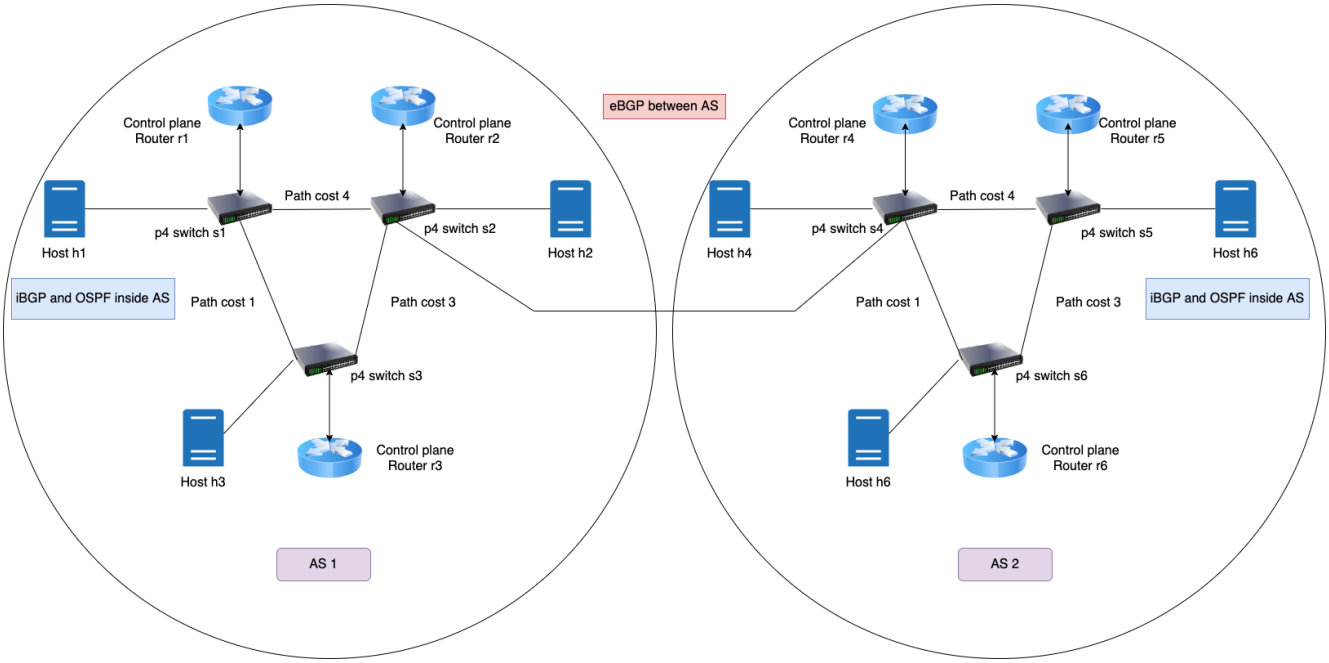

Figure 4.2: 3 AS with eBGP between the AS

to be routed outside the AS. This also needs to be recognised by our controller entity and handled.

Finally, BGP doesn't work on a hello message exchange using IP packets to discover its neighbours unlike OSPF, it relies on a TCP connection set up on TCP port 179 to establish a BGP peer. After this, the routes are exchanged over this connection and this connection is kept up by further sending keep-alive messages, after the connection has been set up. Our design needed to be careful here as we could not let any match-action in our P4 program, change any field in the TCP header, which would cause the connection to not establish.

In spite of ensuring that the P4 switches only have transparent forwarding for TCP and BGP packets i.e. they merely forward by checking the ingress and forward to the egress port, the TCP connection did not setup. On further investigation, it was seen that there was an incorrect checksum field in the TCP packet, which caused it to be dropped when it was received by the control plane router's CP-interface. This issue with the checksum happened even without any modification in the packet header or payload. To solve this, we created the same setup with 2 AS using just FRR routers its original Linux data plane using Mininet. We then checked the behaviour of packets at the interfaces of the FRR routers and found the the same incorrect checksum issue persisted here too, however the router accepted the packet. Since this is an issue with FRR's implementation most likely, we decided to ignore it for now and do the same in our Super-Node setup.

In our control plane router definition, we added a new command to remove checksum offloading for TCP to the network adapter, and compute it using the CPU instead, which solved the problem. With this, we were able to set up the TCP connection between the eBGP peers, learn the routes to the other AS and make our P4 switches forward IP traffic to the required hosts. Along with this, we also create our ECMP load balancing to load balance traffic across different ASes.

Chapter 5

Outlook

In our design for the Super-Node, we have relied on FRR being the traditional control plane for our routers, which pushes computed routes to our programmable P4 data plane. In this section, we review the key advantages and disadvantages of FRR for our control plane router, in order to evaluate whether it really works well and makes sense to rely on it to the extent we do. We also present possible ideas for extending our project in future, new functionalities which can be added, more use-case where our Super-Node based networks may provide faster solutions and so on.

5.1 Is using FRR as a control plane good?

Let us first re-analyse why FRR works well and why we chose it for our control plane router:

- FRR is a production grade, universally available routing suite
- FRR has existing implementation for a variety of routing protocols like OSPF, BGP, ISIS, RIP etc.
- FRR works with all standard *NIX kernels, making it easy to work across diverse kernel networking stacks such as Linux, BSD etc.

The only problem we see with FRR is that it is rigid in its working i.e. it expects links to the CP-interfaces in order for it to send packets. In case, a real link is not present on the CP-interfaces, FRR doesn't send packets out on the interface. Also, the FRR control plane router must receive packets on the intended CP interfaces, else they are dropped. Hence, we needed the CP-interfaces to connect to fake interfaces on the P4 switch, which does lead to doubling of interfaces on the data plane node.

However, based on our working with the Super-Node architecture, we feel that this doubling of the interfaces at this stage doesn't lead to any problem in the forwarding plane as such. Also, as it is possible in the P4 data plane to isolate FRR control packets(for eg. OSPF hellos) from actual traffic by setting conditions to hit only required tables, we do not see it causing a problem for forwarding normal IP traffic. Hence, we do see FRR routing control planes to be a possible universal control plane architecture for our Super-Node design.

5.2 Future Work

In this project we present a network with a new Super-Node design, comprising of a detached control plane coupled with a programmable P4 data plane. We see our work as the start in the

area of using these Super-Node topologies to make smarter, efficient routing and traffic management decisions in networks. As of this project, we have envisioned the architecture of such a network node, and tested it with few standard Layer-3 routing protocols such as OSPF and BGP. We see the possibility of extension of our project in several of , but not limited to, the following areas:

1. We have designed our setup to work with routing protocols such as BGP and OSPF, in future we also see possible extensions to other routing protocols which can be added on to our base setup.

2. As we offload our forwarding plane to a programmable P4 entity, we see it possible to add extensions in future for fast recompute of forwarding states in case of link failures in networks using protocols such as BGP. We see this by leveraging the programmability of the data plane to devise a method to update the forwarding states, without having to learn the entire BGP prefix table for the new routes.

3. We see it possible to improve our data plane architecture by adding more Quality-of-Service(QoS) measures for better congestion control in our data plane, something which we didn't focus on in the scope of this project.

4. Finally, as mentioned in section 4.1, it may also be a good feature to include different load balancing techniques, using our programmable data plane , instead of vanilla ECMP in future implementations of our work.

Chapter 6

Summary

In this project, we present a framework for co-design of a software control plane and a programmable data plane for networking applications. We introduce a method to create networks with a new kind of forwarding node, which comprises of a layer-3 routing control plane, which pushes the forwarding states to a programmable P4 data plane. We refer to this architecture as a Super-Node, which is capable of making intelligent forwarding decisions. We use our Super-Node's data plane instead of using the networking stack of the kernel, to forward packets from end to end in our network. We run traditional routing protocols such as OSPF and BGP to achieve connectivity, and by using our programmable data plane, we are able to achieve far greater control over the packet forwarding behaviour. We include multi-path routing like ECMP and routing across multiple ASes, in order to further test the performance of our system. Finally, we present our system with the new Super-Node design which can be used to make networks faster by using the P4 programmable data plane to speed up network applications such as faster recovery from link failure, customised routing and much more.

Bibliography

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, p. 69–74, March 2008. [Online]. Available: https://doi.org/10.1145/1355734.1355746

[2] "Frrouting." [Online]. Available: https://docs.frrouting.org/

[3] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *ACM : Computer Communication Review*, vol. 44, no. 3, p. 87–95, July 2014. [Online]. Available: http://dx.doi.org/10.1145/2656877.2656890

[4] T. Holterbach, E. C. Molero, M. Apostolaki, A. Dainotti, S. Vissicchio, and L. Vanbever, "Blink: Fast connectivity recovery entirely in the data plane," *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, p. 161–176, 2019.

[5] E. Molero, A. Dietmüller, R. Meier, F. Schleiss, and J. Müller, "p4-utils." [Online]. Available: https://github.com/nsg-ethz/p4-utils/

[6] The P4 Organization Architecture Working Group, "P416 portable switch architecture (psa)." [Online]. Available: https://opennetworking.org/wp-content/uploads/2020/10/P416-Portable-Switch-Architecture-PSA.html#sec-target-architecture-modell

[7] P4Lang, "P4 programmble switch." [Online]. Available: https://github.com/p4lang

[8] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," *Hotnets '10*, October 2010. [Online]. Available: https://conferences.sigcomm.org/hotnets/2010/papers/a19-lantz.pdf

[9] B. Lantz and B. Heller, "Mininet." [Online]. Available: https://github.com/mininet/mininet

[10] S. Zaikin, "Make bgp work again (in vpp)." [Online]. Available: https://github.com/zstas/pppcpdl

[11] E. C. Molero, S. Vissicchio, and L. Vanbever, "Hardware-accelerated network control planes," *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, p. 120–126, 2018. [Online]. Available: https://doi.org/10.1145/3286062.3286080

[12] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Cacheflow: Dependency-aware rule-caching for software-defined networks," *Proceedings of the Symposium on SDN Research*, 2016. [Online]. Available: https://doi.org/10.1145/2890955.2890969

[13] P. Wintermeyer, M. Apostolaki, A. Dietmüller, and L. Vanbever, "P2go: P4 profile-guided optimizations," *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, p. 146–152, 2020. [Online]. Available: https://doi.org/10.1145/3422604.3425941

[14] P. Biondi, G. Valadon, P. Lalet, and G. Potter, "Scapy." [Online]. Available: https://scapy.readthedocs.io/en/latest/

[15] E. Vanini, R. Pan, M. Alizadeh, P. Taheri, and T. Edsall, "Let it flow: Resilient asymmetric load balancing with flowlet switching," *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*, March 2017. [Online]. Available: https://www.usenix.org/system/files/conference/nsdi17/nsdi17-vanini.pdf