Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Fachbereich Informatik
Arbeitsbereich Kommunikationsnetze

Masterarbeit Informatik

Design and P4-Based Implementation of Scalable and Resilient BIER and BIER-TE for Large IP Multicast Networks

Design und P4-basierte Implementierung von skalierbarem und ausfallsicherem BIER und BIER-TE für große IP-Multicast Netzwerke

Steffen Lindner

26.03.2019

Gutachter

Prof. Dr. habil. Michael Menth
Fachbereich Informatik
Arbeitsbereich Kommunikationsnetze
Universität Tübingen

Betreuer

Daniel Merling, M.Sc.

Steffen Lindner:
Design und P4-basierte Implementierung von skalierbarem und ausfallsicherem BIER und BIER-TE für große IP-Multicast Netzwerke
Design and P4-Based Implementation of Scalable and Resilient BIER and BIER-TE for Large IP Multicast Networks
Masterarbeit Informatik
Eberhard Karls Universität Tübingen
Bearbeitungszeitraum: 01.11.2018 - 26.03.2019

Erklärung

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbstständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Masterarbeit wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Tübingen, den 26.03.2019

(Steffen Lindner)

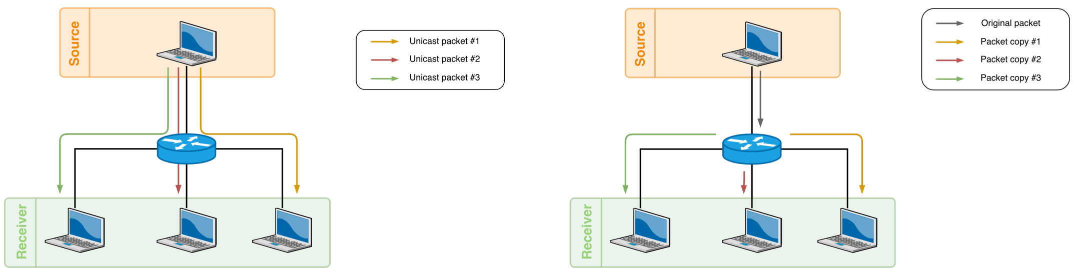
Contents

IP-Multicast communication is a widely used method to address multiple receivers for the same packet without using a copy for each receiver. It enables a minimum number of packets to reach a group of recipients, taking into account the shortest paths. Applications are for example IP-TV, live streams, multicast VPN or telephone conferences. Although classical IP multicast drastically reduces the number of data packets in the network compared to IP-unicast communication, it also has some disadvantages. To avoid these disadvantages BIER and its extension BIER-TE was introduced by the IETF.

In the context of software defined networking, this thesis presents all relevant concepts of BIER and BIER-TE in theory as well as an implementation in P4. After a short overview of underlaying principles, BIER and BIER-TE are introduced and their fast reroute components are discussed. Scalability concerns of BIER and BIER-TE are addressed and a domain based approach to resolve this issue is examined. Basics of software defined networking are covered along with an introduction to P4. For the sake of completeness, related work is reviewed. The resulting implementation of the prototype is outlined and both, aspects of the data plane and the control plane are considered. To facilitate readability, the implementation is described on a high, intuitive level. Summing it up, a short conclusion and an outlook for possible future work is given.

1. Introduction

In classical unicast communication there is a single sender and a single receiver, i.e. a one to one relationship [14]. If a packet has to be delivered to several receivers, one packet is sent for each receiver. This means that identical packets, with regard to their content, may be sent several times over the same link. Figure 1.1(a) illustrates this behavior.



(a) Unicast traffic

(b) Multicast traffic

Figure 1.1.: Unicast vs. multicast traffic.

The first link between sender and router transmits three identical packets, although only one packet would be sufficient. This behavior takes on much greater significance for example for IP-TV. Let us assume that every household in a certain region watches the same channel. Instead of sending one unicast packet to each household in a neighbourhood, it would be sufficient to send a single packet to a nearby distributor and duplicate it there for every household.

This reduces the network load up to the distributor by the factor of the number of receivers. Figure 1.1(b) illustrates this behavior.

Instead of sending three individual packets, the sender transmits one packet, which is duplicated by the distributor and sent to the corresponding recipients. This concept is known as multicast.

1.1. Goal of the Thesis

This thesis examines an alternative transport mechanism for classical IP multicast called BIER and its traffic engineering extension BIER-TE. In the context of software defined networking a P4 based implementation of BIER and BIER-TE was developed. Besides the basic functionality of BIER and BIER-TE, a new extension called BIER(-TE) Scalability was implemented to improve scalability. The Fast

Reroute (FRR) component is also taken into account to ensure reliability. To support scalability, a domain concept based on spectral clustering was developed and tunneling mechanisms such as segment routing were investigated. In addition to the pure data plane implementation in P4, a controller architecture for automatic topology recognition and switch configuration was developed.

1.2. Structure of the Thesis

Chapter 2 introduces basic concepts used in this thesis, such as IP-Multicast, Fast Reroute and Tunneling. Chapter 3 introduces Bit Index Explicit Replication (BIER) and its Traffic Engineering extension BIER-TE. In addition to the essential principles, problems of BIER and BIER-TE are considered. Chapter 4 introduces the new scalability concept of BIER and BIER-TE. Subsequently, chapter 5 explains the SDN context of this work and introduces the used programming language P4. Chapter 6 is devoted to related work, in particular existing multicast implementation in the SDN context. Implementations of the concepts, including BIER(-TE)(-FRR), segment routing and the scalability extension are presented in chapter 7. Chapter 8 then draws a conclusion.

2. Basic Principles

This chapter introduces basic mechanisms that are used more frequently in the course of this thesis. Basics of IP-Multicast, Tunneling and Fast Reroute are covered.

2.1. IP-Multicast

In classic multicast, a sender sends a message to a group of receivers, i.e. a one to many relationship. While the sender address is a unicast address, the receiver address is a multicast address. In IPv4 the multicast address is a class D address in the range 224.0.0.0 – 239.255.255.255 [22] and is called a multicast group.

For the IP-TV application, it quickly becomes clear that terminal devices must be able to subscribe and unsubscribe dynamically from multicast groups, for example when switching channels.

2.1.1. Internet Group Management Protocol

The Internet Group Management Protocol manages the membership of devices in a multicast group. Among other things, it enables the subscription and unsubscription of a multicast group [6]. An IGMP packet is carried inside an IP packet with the protocol number of 2. The implementation resulting from this work uses IGMP in version 2 [10].

IGMPv2 packets have a size of 64 bits and are structured as shown in Figure 2.1 [10].

- An 8 bit *Type*-field indicates the type of the message, e.g. subscription or unsubscription.
- An 8 bit *Max Response Time*-field specifies the time a receiver waits until he sends his membership report. This functionality is not used in this prototype implementation.
- A 16 bit *Checksum*-field builds a simple checksum over the whole IGMP message.

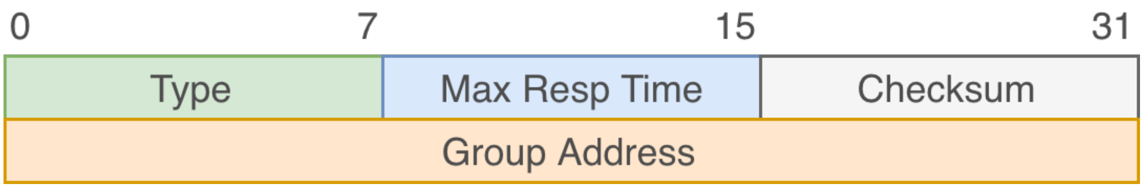


Figure 2.1.: IGMP packet structure.

The 8 bit *Type*- field can have the following values:

Value	Meaning
0x11	Membership query - Not implemented
0x16	Membership report - subscription
0x17	Leave group

Figure 2.2.: Type values in IGMPv2.

When a host wants to subscribe to a multicast group, it sends an IGMP packet to its connected router. The router propagates the information that it wants to receive packets for this multicast group using multicast routing protocols.

2.1.2. Multicast Routing

When a router receives a multicast packet, it must decide whether to forward the packet on only one or on multiple interfaces. This decision depends not only on the destination of the multicast packet, but also on its sender as shown in Figure 2.3 [14].

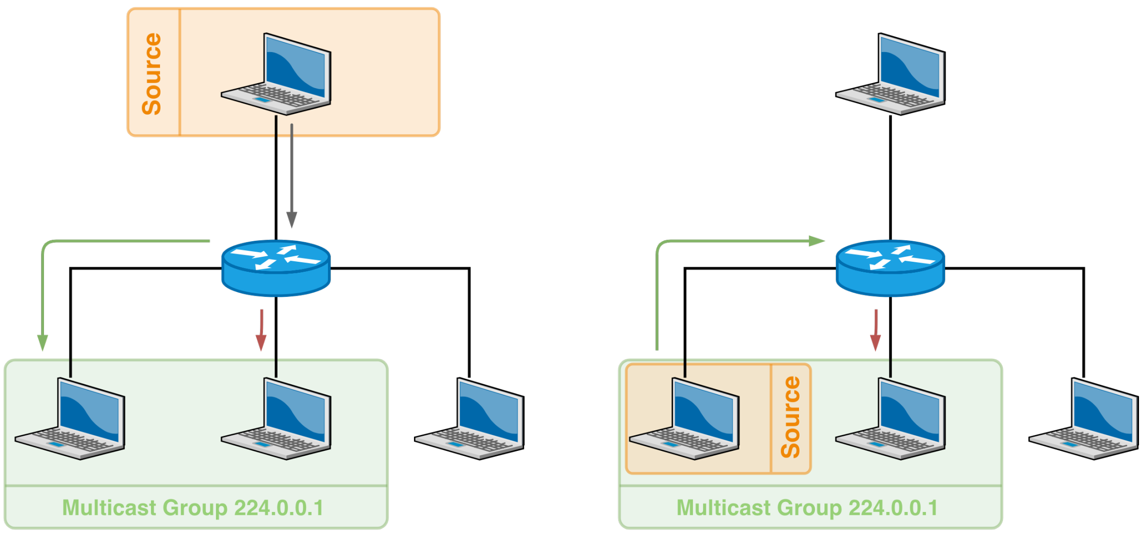


Figure 2.3.: Multicast is sender dependent.

The left side of the image shows the case where an external source sends a packet to the multicast group. In this case, the router must duplicate the packet to the two links connected to the multicast group's receivers. In contrast, the right side shows the case where a member of the multicast group is also the sender. In this case, the router only has to send the packet to the interface that is connected to the other multicast group receiver.

To take this behavior into account, each router creates a so-called multicast tree that determines how a packet is forwarded. To take different sources into account, a router creates $m \times n$ multicast trees for m sources and n groups. These trees are also called source-based trees [14]. Figure 2.4 illustrates two multicast trees on a simple network with one sender and two multicast groups. Each intermediate router has to save its corresponding source-group tree.

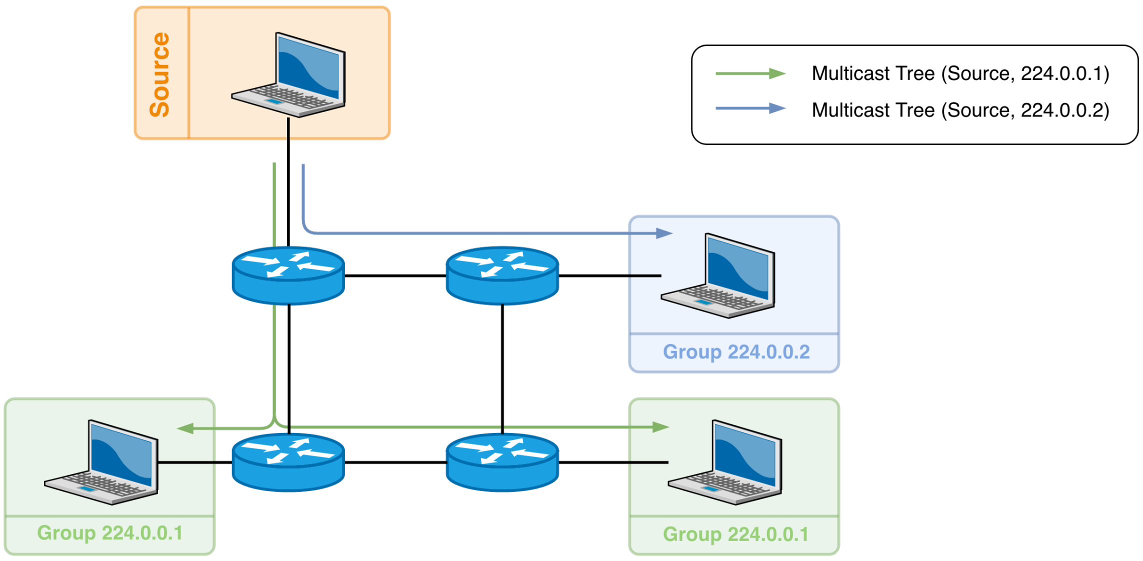

Source
Multicast Tree (Source, 224.0.0.1)
Multicast Tree (Source, 224.0.0.2)
Group 224.0.0.2
Group 224.0.0.1
Group 224.0.0.1

Figure 2.4.: Source based multicast trees.

When a device subscribes or unsubscribe to a multicast group, all routers in the multicast domain must update their local multicast trees. This update requires both time and computational effort on the routers.

The source-based method is the simplest method of multicast. Other methods, such as group shared trees [22], reduce the number of multicast trees required per router. However, the basic problem remains the same: a change in a multicast group takes time and effort until all participating routers in the network have recalculated their multicast trees.

2.2. Tunneling

Tunneling is a widespread concept. The basic idea is to pack an existing packet into an additional packet. A classic use case is the transmission of IPv6 packets in a pure IPv4 network. At the input node of the IPv4 network, the IPv6 packet is packed into a new IPv4 packet. The destination address is an exit node of the IPv4 network. Within the IPv4 network, the tunneled packet is now transmitted using its IPv4 header. At the output node the IPv4 header is removed and the original IPv6 packet can be transmitted. The tunnel concept is also used for many other things like virtual private networks (VPN), bypassing broken links/nodes and much more.

2.3. Resilience for Networks: Fast Reroute

Fast Reroute (FRR) is a mechanism to quickly respond to the failure of a single link or a single node error in critical networks without having to calculate new forwarding rules based on the changed topology. As soon as a device realizes that it can no longer send packets on an interface, it uses predefined rules to send the current packet to its intended destination via another interface. These rules are calculated before the actual failure occurs and allow a very fast response to failures. Fast Reroute distinguishes between two protection mechanisms: link protection and node protection. Link protection ensures that traffic going over a certain link to its neighbor can still reach this neighbor although the link fails. Node protection on the other hand ensures that traffic going over a certain neighbor can still reach its destination although the neighbor fails. In both cases it's intended that the deviation from the original path is as small as possible. The device that detects and bypasses the error is called Point of local repair (PLR).

Since FRR is only a concept, there are several different approaches and the concrete implementation is technology dependent. In the following, FRR is explained in the context of BIER and BIER-TE with reference to implicit and explicit backup paths.

2.3.1. Link Protection

In the link protection case, it is assumed that only the link behind the interface no longer works. In this case, a backup path to the node behind the failed link is calculated.

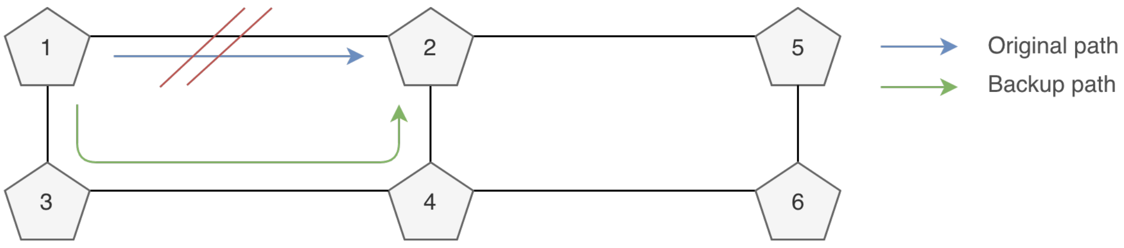


Figure 2.5.: FRR link protection example.

Figure 2.5 illustrates the example of link protection. Suppose node 1 wants to send a packet to node 5. Its normal path would lead directly to node 2. As node 1 detects, that sending packets a.ong the link to node 2 is not possible and since link protection is configured, it assumes that the link between node 1 and node 2 has failed. As a result, the packet is forwarded over the previously calculated backup path using node 3 and node 4 to node 2, where it can normally be forwarded to node 5.

2.3.2. Node Protection

In contrast, in the node protection case it is assumed that the node itself has failed. Instead of sending the packet via a backup path to the (assumed failed) node behind the interface, the packet is send to the next-hop of the failed node towards the final destination - in this case node 5.

3. Bit Index Explicit Replication (BIER)

This chapter introduces BIER(-TE) and its resilience extension. In addition to the basic functionality, problems with BIER and BIER-TE are also discussed.

Bit Index Explicit Replication (BIER) [41] was developed by the Internet Engineering Task Force (IETF) to solve the problems of classic multicast. The basic idea is that the receiver of multicast groups are no longer stored in the nodes itself but directly in the packet. Due to this change, only the nodes at the edge of the domain need to be updated when the multicast groups change.

3.1. BIER Concept

Bit Index Explicit Replication works as a kind of point-to-multipoint tunnel [27]. BIER-enabled devices are divided into pure forwarding devices (Bit-Forwarding Router - BFR), input nodes for a domain (Bit-Forwarding Ingress Router - BFIR) and output nodes for a domain (Bit-Forwarding Egress Router - BFER). Bit forwarding routers are only responsible for forwarding packets hence they only need to know how to reach any other node within their domain. The input nodes of a domain (BFIR) must encode the receivers (BFERs) of the packet and are the beginning of the point-to-multipoint tunnel. The Bit-Forwarding Egress Routers (BFERs) are the end of the point-to-multipoint tunnel, unpack the original IP-Multicast packet and forward it using classic IP multicast. In order to encode receivers efficiently, each BFER is assigned a bit in a bit string, starting with the least significant bit. The bit string is part of the BIER header and contains n bits for n BFERs. If the corresponding digit is `1`, the BFER must receive a copy of the packet; if the digit is `0`, the BFER must not receive a copy. A BFIR can also be used as a BFER at the same time. Figure 3.1 illustrates the BIER mechanism.

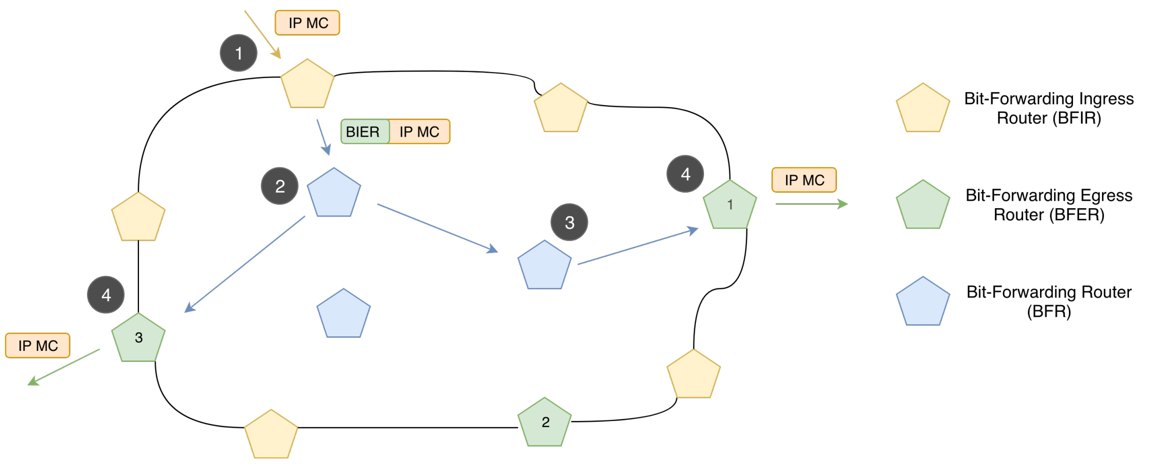

IP MC
1
BIER
IP MC
2
3
4
1
IP MC
4
3
IP MC
2
Bit-Forwarding Ingress Router (BFIR)
Bit-Forwarding Egress Router (BFER)
Bit-Forwarding Router (BFR)

Figure 3.1.: BIER concept. Illustration from [27].

When an IP multicast packet arrives at an ingress node of a BIER domain (BFIR), a BIER header is added to the multicast packet ❶. The BIER header identifies all egress nodes of the domain that must receive a copy of this packet using a bit string, let's say BFER 1 and 3. The corresponding bit string is `101`. The packet is transported via the Bit Forwarding Routers (BFRs) ❷ ❸ to the respective destination nodes (BFERs). There the BIER header is removed ❹ and the IP multicast packet is forwarded according to its destination address. Although in principle only BFERs need an unique identifier, which are used as position in the bit string, the so-called BFR-id, an identifier, can be assigned to all BIER-enabled devices [41].

3.1.1. BIER Forwarding

A BFR always sends only one copy of a packet over a certain link, even if several BFERs are reached using this link. An essential component of multicast is that each recipient receives the packet exactly once. To ensure that this property is maintained a BFR removes all BFERs from the BIER bit string that are reached via another next hop, before forwarding a BIER packet. Without this modification it could happen that the recipient receives more than one copy of the packet.

A key component in the forwarding procedure of BIER packets is the so called Bit Index Forwarding Table (BIFT). For all possible destinations (BFERs), the BIFT contains the corresponding next hop. It also contains the so called forwarding bit mask (FBM). The FBM contains the BFR-ids of the BFERs that can be reached using the same next hop. All other BFERs must be removed from the BIER header to prevent duplicates, as mentioned before [29]. This can be achieved using a bitwise logical AND between the packet bit string and the FBM. This logical combination is also called *applying the FBM* in the following.

Figure 3.2 shows a topology with BIER capable devices. Each device was assigned an unique BFR-id and the shortest path tree for node 1 is visualized. Figure 3.3 illustrates the BIER forwarding procedure.

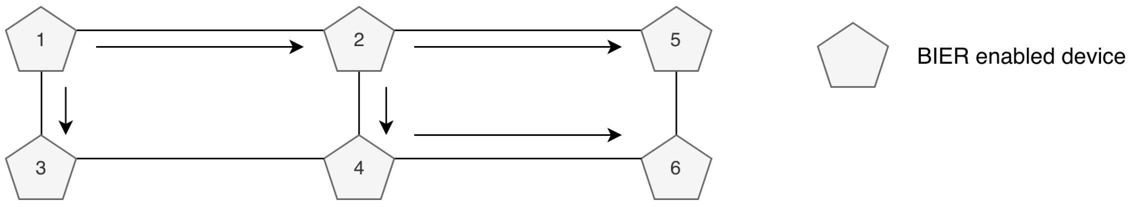


Figure 3.2.: Example topology from [29].

When a BIER packet arrives at node 1, the device iteratively checks whether a bit is set in the bit string for which an entry exists in its BIFT. If an active bit identifies the device itself, the packet is copied, unpacked and passed to the IP layer. This process is called *decap* in the illustration. In the original packet, the device bit is removed and matched to the next active bit again. If the active bit does not identify the device, a copy of the packet is created, the FBM is applied, and the resulting packet is sent to the next hop. This ensures that the bit string of the packet only contains the destinations that are to be reached via this neighbor. In the original packet, the bits activated in the FBM are removed from the bit string and the packet gets matched to the next activated bit. This process is repeated until no bit is active and the packet gets discarded. The standard suggests that the device always matches on the the least significant active bit.

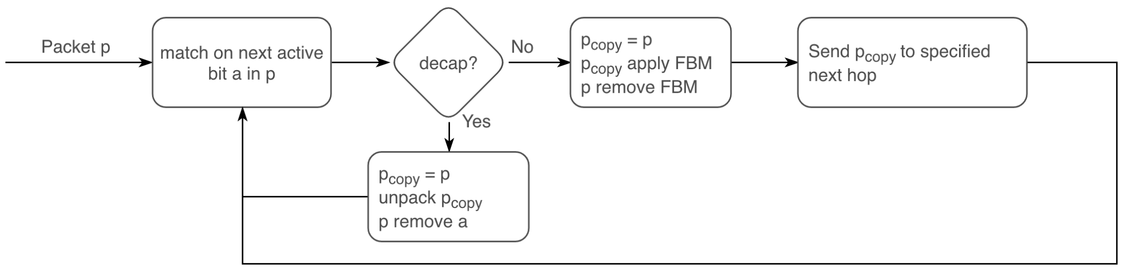


Figure 3.3.: BIER forwarding procedure.

Table 3.1 shows the BIFT for node 1. Nodes 2 and 3 are directly the next hop of node 1. Nodes 4, 5 and 6 can be reached via node 2. The FBM is the logical bitwise OR of all BFR-ids that share the same next hop. BFR 2, 4, 5 and 6 share BFR 2 as next hop, thus the FBM consists of the respective BFR-ids resulting in `11101`. The BIFT of a node can also contain itself, where only its own bit is set in the FBM.

Destination	FBM	Next-Hop
2	111010	2
3	000100	3
4	111010	2
5	111010	2
6	111010	2

Table 3.1.: BIFT for node 1.

The BIER forwarding mechanism only needs the information how a device can reach all other devices. This information is gained by the underlying routing layer [41]. Unlike classic multicast, a forwarding device no longer needs to store $m \times n$ multicast trees for m sources and n groups. Instead, the information for the destination is directly encoded in the packet and the forwarding devices only need to forward the packets leveraging their known forwarding rules. This also eliminates the need for the forwarding devices to constantly update when multicast groups are changed. Only the BFIRs of a domain need to know which BFERs should receive a multicast packet. Accordingly, only the BFIRs need to be updated if the multicast groups change.

3.1.2. BIER-FRR

In its basic RFC 8279 specification [41], a BIER-enabled device must wait until its routing entries are updated to respond to a local link or node error. To ensure that traffic can be redirected correctly, all devices involved in the backup path must be updated to bypass the failed link/node. This procedure takes time, and parts of the traffic cannot be transferred correctly during this process. For this reason, Fast Reroute mechanisms have been proposed for BIER [28].

Since BIER does not support a direct FRR mechanism, the underlying routing layer, e.g. IP, is leveraged. The proposed BIER-FRR mechanism ensures that the bit string of the packet is adapted in such a way that duplicates and cycles are prevented. At the time of a link or node error, a BIER-enabled device can only determine that its next hop is not available for the current packet. It cannot tell whether only the link to this next hop has failed or the next hop itself. It must therefore be decided whether a next hop that cannot be reached is interpreted as a link failure or as a node failure.

3.1.2.1. Link Protection

Link protection assumes that only the link between the PLR and its next hop has failed. If another path to this next hop exists, it would be sufficient to send the current packet to the next hop using this path. BIER-FRR link protection leverages a unicast tunnel over the underlying routing layer, e.g. IP. The routing layer must determine whether and how the next hop can be reached. This means that BIER-FRR relies on the resilience component of the routing layer. Since the BIER packet is tunneled to the original next hop, no additional changes have to be made to the bit string of the packet. Figure 3.4 illustrates BIER-FRR link protection using IP as routing underlay. For simplicity, only the bit string of the BIER packet is shown.

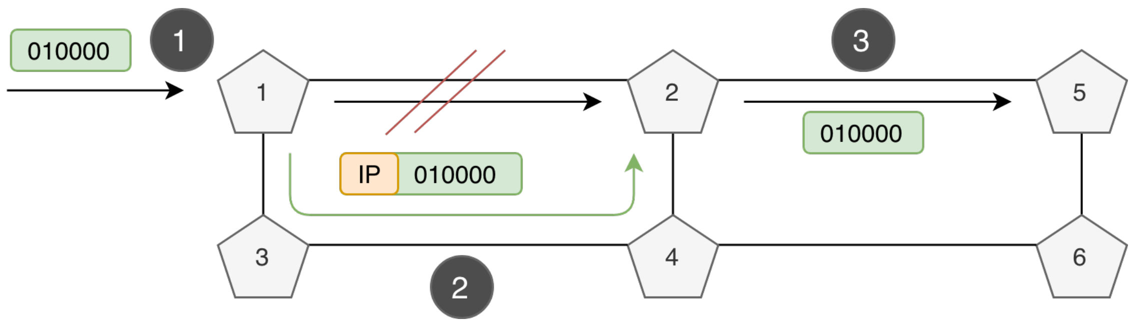


Figure 3.4.: BIER-FRR link protection leveraging IP as routing underlay.

Let's assume that node 1 receives a BIER packet with bit string 010000 ❶ and that the link between node 1 and node 2 has failed. In its BIFT, node 2 is entered as next hop for the active bit, thus node 1 applies the corresponding FBM. If node 1 determines that its next hop is not reachable, it tunnels the BIER packet via the routing underlay, in this case IP, to node 2 ❷, where it is then unpacked and transferred according to the BIER forwarding rules ❸. Node 1 turns into a point of local repair (PLR) in this case. If the routing underlay has a FRR mechanism and there is an alternative path, the packet is transferred to the next hop via this path. The BIER forwarding procedure remains unchanged.

3.1.2.2. Node Protection

Unlike link protection, node protection assumes that the next hop that cannot be reached has failed. The PLR must now ensure that all potential next hops of the failed next hop, the so-called next next hops, receive a customized copy of the BIER packet without producing duplicates at the final destinations. If the next hop would be reachable, the node would forward the packet to any next next hop that needs a copy and would apply the appropriate FBM. Since the next hop has failed, this work must be done by the PLR [28]. If the PLR determines that it cannot send a BIER packet to a next hop, it applies the corresponding FBM and tunnels the packet to the next next hop. Since the next hop could send the packet to multiple next next hops, this process is repeated iteratively until all next next hops are processed. Figure 3.5 illustrates an example for BIER-FRR node protection. Table 3.2 shows the BIFT of node 1 and table 3.3 shows the BIFT of node 2.

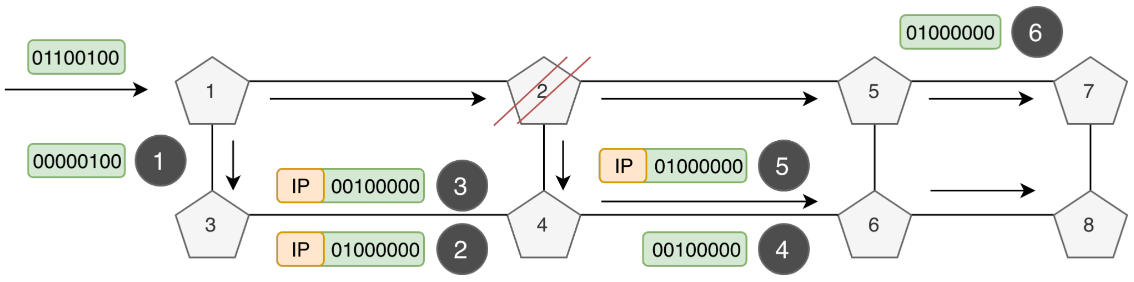


Figure 3.5.: BIER-FRR node protection using IP as routing underlay.

Destination	FBM	Next-Hop
2	11111010	2
3	00000100	3
4	11111010	2
5	11111010	2
6	11111010	2
7	11111010	2
8	11111010	2

Table 3.2.: BIFT for node 1.

Destination	FBM	Next-Hop
1	00000101	1
3	00000101	1
4	10101000	4
5	01010000	5
6	10101000	4
7	01010000	5
8	10101000	4

Table 3.3.: BIFT for node 2.

Let's assume a BIER packet with bit string 01100100 arrives at node 1. The BIER packet must be delivered to nodes $\{3, 6, 7\}$. Let's also assume that node 2 has failed. When node 1 processes the BIER packet and the bit for node 3 is next, node 1 copies the packet, applies its FBM for node 3 (00000100) and sends it to the next hop for node 3, which in this case is directly node 3 ❶. The resulting bit string of the packet copy is 01100100 & 00000100 = 00000100. In the original packet, the bit string is combined with the complementary FBM, 01100100 & 11111011 = 01100000.

In the next step, the bit for node 6 is processed. Node 1 creates a copy of the BIER packet, applies the FBM for node 6 (11111010) and tries to send the packet to the next hop for node 6. The next hop for node 6 would be node 2. Since node 2 cannot be reached, node 1, which is now the PLR, additionally applies the FBM that node 2 would have applied for node 6 (10101000) and tunnels the packet to the next hop of node 2, in this case node 4 ❸. A possible backup path for the IP FRR mechanism would be $1 \rightarrow 3 \rightarrow 4$. The bit string of the tunneled BIER packet is

$$\underbrace{\texttt{01100000}}_{\text{bit string of original packet}} \;\&\; \underbrace{\texttt{11111010}}_{\text{FBM for node 6 from node 1}} \;\&\; \underbrace{\texttt{10101000}}_{\text{FBM for node 6 from node 2}} = \texttt{00100000}$$

At node 4 the IP packet is unpacked and the underlying BIER packet is forwarded to node 6 ❹. The bit string of the original packet is now combined with the complementary FBM for node 6 of node 1 and the complementary FBM for node 6 of node 2, resulting in the bit string 01000000. In the last step, the bit for node 7 is processed and the FBM 11111010 is applied. Subsequently, node 1 determines that node 2 cannot be reached and applies the FBM for node 7 of node 2. The resulting bit string is 01000000 and the packet is sent via IP tunnel to node 5 ❷ ❺. A possible backup path for the IP FRR mechanism would be $1 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 5$. At node 5 the IP packet is unpacked and forwarded to node 7 ❻. The bit string of the original packet after those steps is 00000000. Since no active bit is left, the original packet gets discarded at node 1.

3.2. BIER Traffic Engineering (BIER-TE)

BIER allows multicast packets to be forwarded along the paths of the underlying routing layer. However, it is often desired that certain packets use certain paths that do not correspond to the paths of the routing layer. BIER-TE [7] introduces a traffic engineering extension for BIER that allows to specify paths independent of the routing layer.

3.2.1. BIER-TE Concept

BIER-TE extends the concept of the BIER bit string. In addition to the BFERs, all adjacencies in the network are also represented as bits in the bit string. An active bit either identifies a BFER or identifies an adjacency that has to be used. This concept allows the entire distribution tree to be encoded directly in the packet. Figure 3.6 illustrates the concept of BIER-TE. For simplification, not all adjacencies were assigned a bit position.

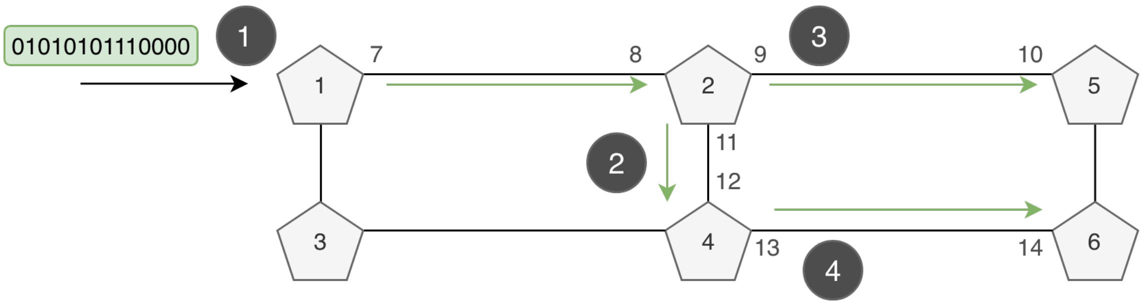


Figure 3.6.: Example for BIER-TE.

Let's assume that the adjacencies of the network are assigned the bit positions shown in Figure 3.6 and that a BIER-TE packet with bit string `01010101110000` arrives at node 1 ❶. The bit string corresponds to the adjacencies $\{7, 9, 11, 13\}$ (marked green) and the BFERs $\{5, 6\}$. Node 1 is connected to adjacency 7. Since this bit is active, node 1 sends a copy of the packet via this adjacency to node 2. Node 2 is in turn connected to adjacencies 8, 9 and 11. Since bit 9 and 11 in the bit string are active, node 2 sends a copy over adjacency 9 and 11 ❷ ❸. Node 4 sends the received BIER-TE packet according to the bit string via adjacency 13 to node 6 ❹. At node 5 and 6 the received packets are unpacked and passed to the IP layer, since the bits identifying these nodes are active.

3.2.2. BIER-TE Forwarding

Each BFR is directly connected to only a part of the adjacencies. This means that only a part of the bits in the bit string are relevant for it. To ensure that no duplicates arrive at the receiver and no loops occur, a BFR must remove all bits in the bit string for which it is responsible before transmitting a packet to a neighbor. These bits are called Bits of Interest (BoI) [27]. Beside the connected adjacencies the own BFR-id also belongs to the BoI.

In large networks, not all devices may support BIER. This may cause two BIER devices to be connected at the BIER level, but not directly at layer 2. In this case the connection is realized by a tunnel of the underlying routing layer, e.g.

IP. A connection realized in this way is called *routed*. In contrast, a direct layer 2 connection is called *connected* [7] [27]. *Routed* connections can also be used for plain BIER. An example for a Bit Index Forwarding Table (BIFT) for BIER-TE is shown in table 3.4.

Bit position	Action
1	decap
2	connected
3	routed

Table 3.4.: BIFT for a BIER-TE BFR.

Figure 3.7 illustrates the BIER-TE forwarding procedure. When a BIER-TE packet arrives at a BFR, it checks iteratively whether a bit is set in the bit string for which an entry exists in its BIFT. When the active bit corresponds to a *decap* operation, identifying the device, it copies the packet, unpacks it and passes it to the IP layer. The bit string of the original packet is adapted, i.e. the *decap* bit is removed. When an active bit corresponds to a *connected* operation, the packet is copied, the bit string of the copy is combined with the complementary BoI bit mask and forwarded on the specified adjacency. This prevents multiple use of adjacencies and thus the unnecessary replication of packets. It further prevents loops and duplicates at the receiver. In the original bit string, the bit identifying the previously used adjacency is removed and the next active bit gets processed. When an active bit corresponds to a *routed* operation, the packet is copied, the bit string of the copy is combined with the negative BoI bit mask and forwarded to the next hop on the BIER layer using a unicast tunnel of the underlying routing layer, e.g. IP. This procedure is repeated until no active bit can be matched anymore.

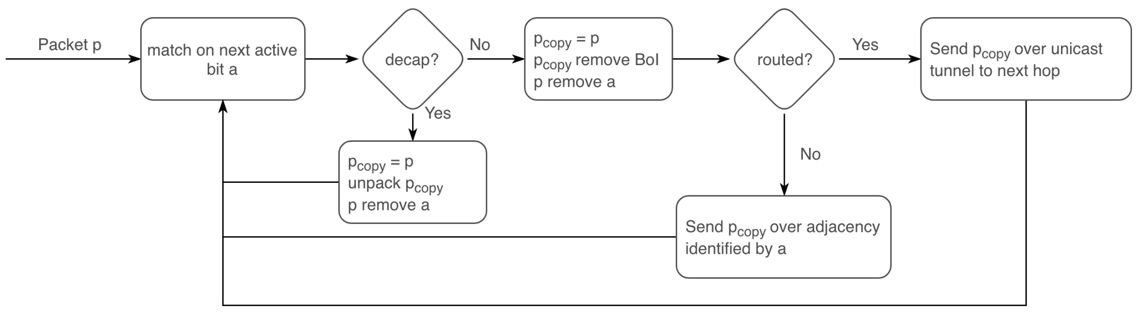

Packet p
match on next active bit a
decap?
No
Yes
p_{copy} = p
p_{copy} remove BoI
p remove a
p_{copy} = p
unpack p_{copy}
p remove a
routed?
Yes
No
Send p_{copy} over unicast tunnel to next hop
Send p_{copy} over adjacency identified by a

Figure 3.7.: BIER-TE forwarding procedure.

3.2.3. BIER-TE FRR

In its basic specification in [7], BIER-TE does not provide any protection mechanism. [8] proposes several protection mechanisms for BIER-TE, including tunneling leveraging RSVP-TE/P2P or segment routing and a native BIER-in-BIER FRR mechanism. This thesis deals exclusively with the latter.

BIER-TE allows to specify explicit paths. This makes it possible to bypass a local link or node error on a BIER-TE path. Although it is in principle possible to customize the original BIER-TE header of a packet to bypass the local error, it is conceptually easier to encapsulate the packet into another BIER-TE packet that contains the backup path. As in the BIER-FRR mechanism, the PLR must also take over the work of the failed next hop with BIER-TE-FRR node protection.

3.2.3.1. Link Protection

As in the BIER-FRR mechanism, the only assumption is that the link between the PLR and its next hop has failed. For this reason, it is sufficient for the PLR to build a tunnel to its next hop. In the BIER-TE-FRR case, this tunnel is realized by BIER-TE, where the backup path is specified and the bit for the tunnel endpoint is set. Figure 3.8 shows an example of BIER-TE FRR link protection. For simplicity, not all adjacencies were assigned a bit position.

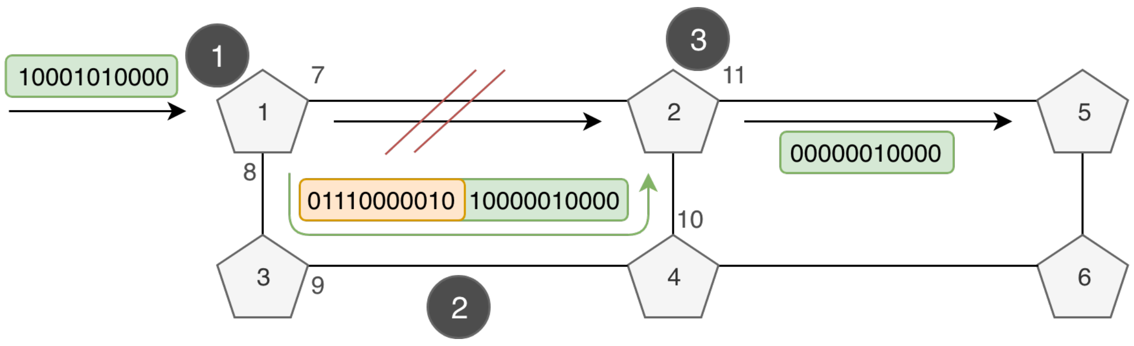


Figure 3.8.: BIER-TE FRR link protection example.

Let's assume that the network adjacencies were assigned the bit position as in Figure 3.8 and that the adjacency between node 1 and node 2 has failed. Let's further assume that a BIER-TE packet with bit string 10001010000 arrives at node 1. This bit string corresponds to the adjacencies {7, 11} and the node {5} ❶. The packet would normally be delivered via the adjacencies 7 and 11 to node 5, which would unpack the packet and process it according to the next header. After node 1 has removed its BoI from the packet copy, it determines that the specified adjacency is not usable. Node 1 now encapsulates the modified packet into a BIER-TE packet whose bit string uses the backup path over the adjacencies 8 $\rightarrow$ 9 $\rightarrow$ 10 to reach node 2. To force node 2 to remove the outer BIER-TE header and forward the actual BIER-TE packet, the bit for node 2 is also set in the outer header. The resulting bit string of the outer BIER-TE packet is 01110000010 $\equiv$ {10, 9, 8, 2} ❷. When the BIER-TE packet arrives at node 2, it unpacks the packet, since the bit for node 2 is set, and processes the inner BIER-TE packet according to the forwarding rules. Since the bit for adjacency 11 is set, node 2 will forward the packet through this adjacency to node 5 ❸.

3.2.3.2. Node Protection

Node protection assumes that the next hop has failed. The PLR must therefore take over the work of the next hop. In the case of BIER-TE, the PLR must additionally remove the BoI of the next hop and create a tunnel to the appropriate next next hop. This tunnel is realized by BIER-TE as well. Figure 3.9 shows an example of BIER-TE FRR node protection. For simplicity, not all adjacencies were assigned a bit position.

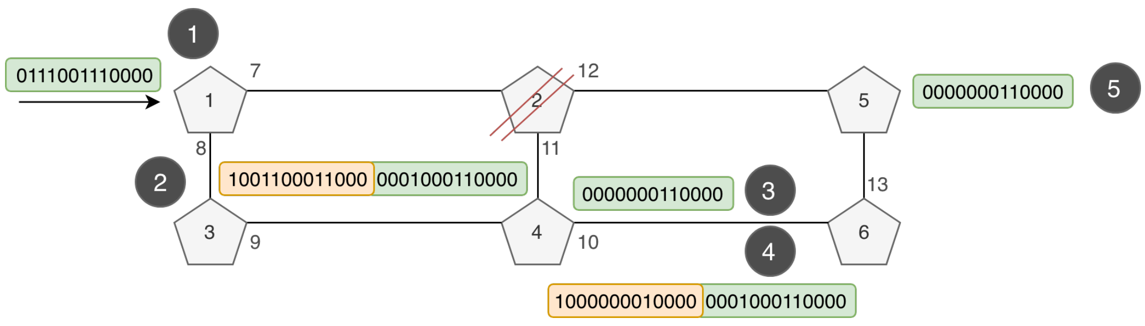


Figure 3.9.: BIER-TE Node protection example.

Let's assume that the network adjacencies were assigned the bit position as in Figure 3.9 and that node 2 has failed. Let's further assume that a BIER-TE packet with bit string `0110001110000` arrives at node 1 ❶. This bit string corresponds to the adjacencies {12, 11, 10, 7} and the nodes {6, 5}. Without the failure of node 2, the packet would take the path through nodes 1 $\rightarrow$ 2 $\rightarrow$ 5 and 1 $\rightarrow$ 2 $\rightarrow$ 4 $\rightarrow$ 6. Since node 2 has failed, the PLR must take over the work of node 2. As node 2 would have forwarded the packet to node 4 and 5, node 1 removes its BoI {8,7,1} and the BoI of node 2 {12,11,2} from the BIER packet and encodes the backup paths to node 4 (adjacency 8, 9) and node 5 (adjacency 8, 9, 10, 13) in a new BIER header ❷. The bits for nodes 4 and 5 are also set to unpack the BIER-in-BIER packet at nodes 4 and 5. When the BIER-in-BIER packet arrives at node 4, a copy is made, the copy is unpacked and forwarded according to the original BIER header via adjacency 10 ❸. In the original BIER-in-BIER packet, the BoI of node 4 are removed and the packet is forwarded to node 6 via adjacency 10 according to the bit string ❹. Node 6 forwards the BIER-in-BIER packet to node 5 via adjacency 13. There it is unpacked and the underlying BIER header is processed ❺. To prevent duplicates to the next next hops in all error cases, the PLR can also delete all incoming adjacencies of the next next hop from the original BIER-TE header.

3.3. BIER(-TE) Issues

BIER and BIER-TE enable the distribution tree to be coded implicitly respectively explicitly in the packet. This means that no additional states are required in the nodes as is the case with classic IP multicast. In order to make this statelessness possible, the bit string was introduced in BIER. Although this has many advantages, the bit string has certain limitations. In the case of BIER, the bit string is n bit long for n BFERs. Considering very large networks, the number of BFERs can be very high. This means that the BIER header also becomes larger the larger the networks are. For this reason, the BIER specification proposes bit strings of variable length [41]. Even though this is theoretically possible, it is usually more efficient if a header has a fixed size. However, a fixed size of the bit string limits the number of BFERs in a network. The more BFERs exist in a domain, the more forwarding entries are needed in the BFRs, so simply increasing the number of bits in the bit string results in larger forwarding tables.

To extend the number of BFERs in a BIER domain, the proposed standard introduces the so-called Set Identifier. The Set Identifier is used to extend the bit string and is part of the BIER header. A bit string with length l, active bit k and Set Identifier n implies, that the packet must be forwarded to the BFER with BFR-id $n \cdot l + k$. If a single Set Identifier is not sufficient to specify all required BFERs, a separate BIER packet must be created for each Set Identifier. Although this method allows to increase the number of BFERs, it also increases the number of required forwarding entries and thus influences the scalability of BIER.

For BIER-TE this problem is even more obvious. In addition to the BFERs, all adjacencies of a network must be additionally encoded. Since a network usually has significantly more adjacencies than BFERs, the number of bits required in the bit string increases considerably. Using BIER-TE FRR not only the BFERs need a BFR-id but all BIER capable devices, because each device can be used as a tunnel endpoint in case of an error.

4. BIER(-TE)-Scalability

This chapter describes an enhancement of BIER and BIER-TE that is intended to solve the problems mentioned above.

A new (currently unpublished) mechanism called BIER(-TE)-Scalability has been introduced to solve BIER(-TE) scaling problems. This mechanism significantly reduces the number of forwarding entries in BFRs as well as the length of the BIER bit string and thus improves the scalability of BIER(-TE).

4.1. Concept

A major problem with BIER is that each BFR must know how to reach every other BFER. This results in n forwarding entries for n BFERs. For topologically close BFERs, it is very likely that the shortest paths to these BFERs are identical or very similar at the beginning. If the BFERs were grouped according to their location, it would be sufficient to send a copy of the BIER packet in the direction of the group of these BFERs. In this case, each BFR on the way would only need one entry per group. Using a tunnel, e.g. IP, for transport to this group, the entries for the group on BIER level would even be omitted. Only the BFRs within a group need to know how to reach each BFER in their group. Across the groups, BFR-ids could be reused, resulting in shorter bit strings. For BIER-TE the main problem is that there exist usually a lot of adjacencies in a network. The extension using the Set Identifier does not help for BIER-TE as the whole path has to be encoded within a single Set Identifier and cannot be changed along the way. Furthermore each BFR needs a BFR-id for BIER-TE.

Figure 4.1 illustrates the concept of BIER(-TE)-Scalability. The entire domain is divided into so called subdomains. In the example of Figure 4.1, the whole domain is divided into 3 subdomains. In each subdomain only a fraction of the total bit string is needed. Instead of a bit string with length 21, for 21 BFRs, a bit string with up to 8 bits is sufficient. In each subdomain the BFR-ids can be distributed starting with ID 1 and only the forwarding entries to other devices in the same subdomain are needed. When an IP multicast packet arrives at a BFIR, it determines to which BFERs in which subdomains the packet must be delivered ❶. For each subdomain, a separate BIER packet with the corresponding BFERs is created ❷. The packet is then tunneled to a tunnel node in the specified subdomain ❸. At the tunnel node the packet is unpacked and the normal BIER forwarding takes place within the subdomain.

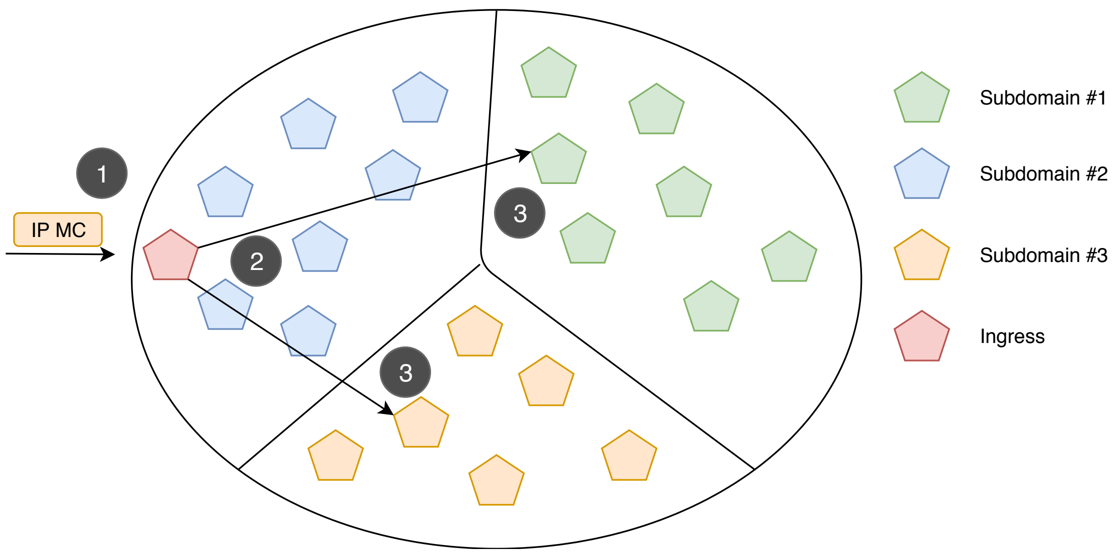


Figure 4.1.: BIER(-TE)-Scalability concept.

The BIER(-TE)-Scalability mechanism combines unicast and multicast. Unicast is used to transport the packets to the respective target domain. The packet can then be distributed within the target domain using BIER multicast.

The number of forwarding entries for BIER and the number of bits required in the bit string for BIER-TE thus scale anti-proportionally with the number of subdomains. At the same time, the number of subdomains determines the unicast to multicast ratio. If a single subdomain is used, no tunnel is required and all traffic is distributed via BIER multicast. If instead a subdomain is used for each BFR, all traffic is sent via unicast.

4.2. Tunnel Mechanism

The unicast tunnel into the subdomains can be realized in different ways. For this work, IP tunnels, segment routing tunnels and a domain forwarding approach without explicit tunnels were implemented. Details to the segment routing implementation are described in the implementation chapter 7.

4.2.1. Domain Forwarding

Within the used BIER header, there is the so called domain field. This field is used to specify the corresponding subdomain. Instead of an explicit tunnel, it would be sufficient to forward a packet to the target domain according to its subdomain identifier. In this way, the packet would be forwarded to the target domain until it reaches a BFR that is part of the target domain. At this point, the normal BIER forwarding takes over. Thus each BFR would need an additional entry for each other subdomain. The forwarding rules in this work were chosen in the direction of the tunnel node. Since, without further calculation, it is not clear which BFR of the target domain receives the packet first, this variant is not possible for BIER-TE since the BIER-TE distribution tree is dependent on the start node.

5. Software Defined Networking (SDN)

This chapter introduces software defined networking (SDN) and the programming language P4, which can be used to program switches.

Classical IP networks are very complex and difficult to manage. Some few suppliers provide a large part of the hardware and software used. Network operators often have to configure each device individually with vendor-dependent configuration tools. The control plane, which decides how a packet is forwarded, and the data plane, which forwards the packet according to the decision of the control plane, are strongly connected within a network device [21]. This strong manufacturer-dependent connection of data and control plane makes it very difficult to impossible to configure new previously unforeseen device behavior. For this reason it often takes several years until a new standard can be supported by devices. This does not only slows down the development of existing networks, but also the development of new technologies, as it often takes years for prototypes to become available [21] [30].

5.1. Concept

Software defined networking is an approach where the data plane and control plane are separated and the device becomes a pure forwarding device. The control logic is outsourced to a logically centralized controller. Figure 5.1 illustrates the architecture of software defined networking.

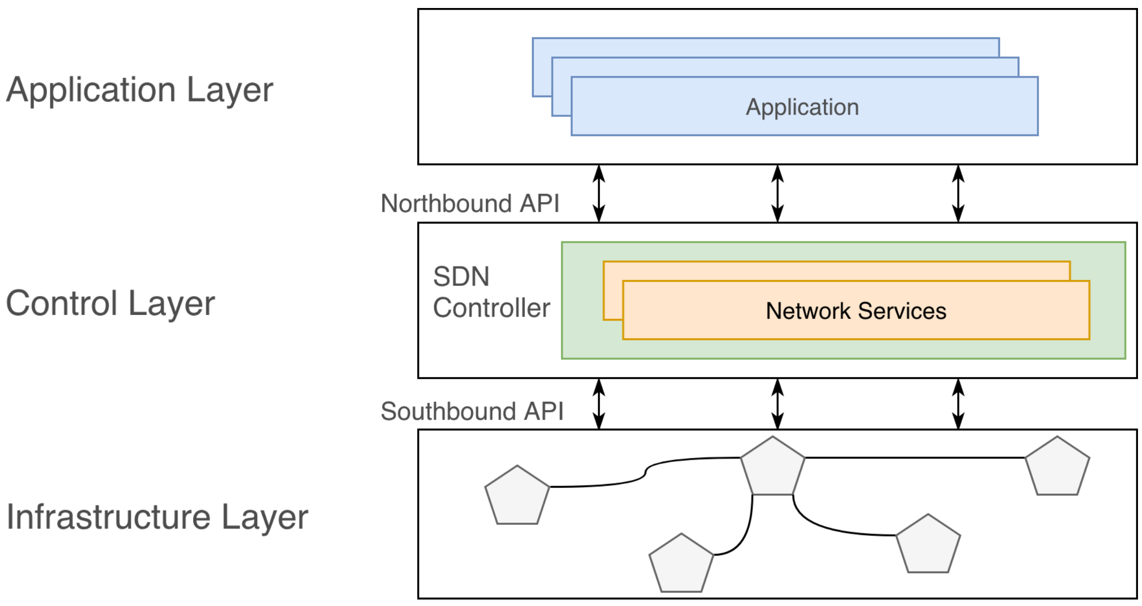


Figure 5.1.: Software defined networking architecture. Illustration according to [30].

The control logic is centralized in a software-based controller. This SDN controller maintains a global view of the network [30] and communicates with the actual devices in the infrastructure layer using a so called southbound API. Typical tasks of the SDN controller are Layer 2 switching and IP routing, which form the network services. The network services thus take over the tasks of the legacy control plane. The devices in the infrastructure layer are responsible for the actual transport and processing of the data according to the decisions of the SDN controller. The basic principle is that the devices simply forward, without their own decision-making process. The application layer contains business software, such as intrusion detection systems or firewalls. The business software can communicate with the SDN controller via a northbound API, e.g. REST, and thus also becomes part of the decision-making process.

5.2. P4

P4 (Programming Protocol-Independent Packet Processors) is a high-level language for programming protocol-independent packet processors [5]. The main goal is the flexible description of the data plane. For this purpose P4 introduces the forwarding model shown in Figure 5.2.

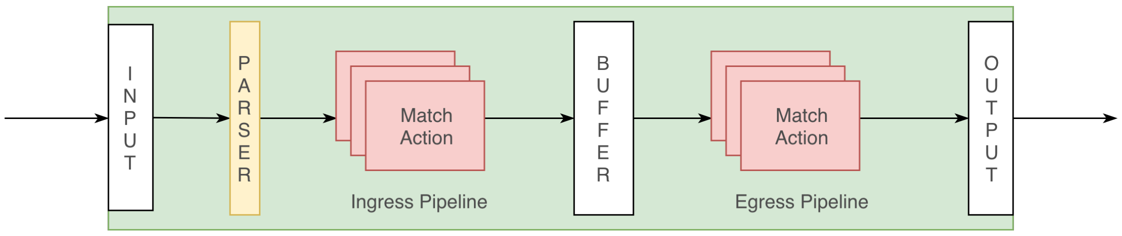


Figure 5.2.: P4 abstract forwarding model. Illustration according to [5].

P4 allows the user to program a parser that can parse any header. This flexibility also allows the user to define new headers and thus implement new technologies. Furthermore, P4 allows to define the way a packet is processed. In addition to basic protocol independent operations (assignment, arithmetic, etc.), P4 allows to define tables that compare the previously parsed header contents to arbitrary rules and, if a rule applies, execute a stored action. This process is also called match+action operation. An action is similar to a function in common programming languages and contains several primitive operations. Inside an action, further actions can be executed. In this way an IP packet can be parsed, the TTL in the IP header can be adjusted, the destination address can be compared with an LPM comparison, and the corresponding forwarding can be performed on an interface. The so called pipeline is divided into two stages, ingress and egress, which are separated by a buffer. Both stages of the pipeline contain a match+action operation, both of which can change the header of the packet. However, the ingress pipeline already specifies the interface on which the packet leaves the device. Based on this decision the packet is for example forwarded, replicated or rejected [5]. The egress pipeline takes over per-instance modification, for example for multicast packets. In addition to the information from the header, so called metadata can also be assigned to a packet. Metadata is specific to a packet, discarded as soon as the packet leaves the device and used as some kind of temporary storage.

One of the basic ideas of P4 is to be independent of specific hardware, also called target. This is achieved using a compiler that translates the target independent P4 program to a target dependent program. Thus it is theoretically possible to write programs in P4 for any target that has a P4 compiler. P4 is currently available in version 16 [31].

5.2.1. Language Definition

Each P4 program consists of the following components:

- Header / metadata definitions
- Parser definition
- Control block
- Deparser definition

Depending on the architecture, additional steps such as checksum calculation can be added.

5.2.1.1. Header Definition

Header definitions specify the structure of incoming packets and consist of individual fields. The contents of the headers are read by the parser. Listing 1 shows an example of a header definition for simplified Ethernet.

```
typedef bit<48> macAddr_t;
header ethernet_t {
    macAddr_t dstAddr;
    macAddr_t srcAddr;
    bit<16>   etherType;
}
```

Listing 1: Example header definition for Ethernet.

A simplified Ethernet header consists of a source mac address, a destination mac address and a type for the following protocol. Mac addresses are classically 48 bit long. Since this type is used several times a new type named *macAddr_t* was created, which consists of 48 bits. The Ethernet type, specifying the next protocol, consists of 16 bits. Each header also has an implicit validity property that is indicated by a *validity* bit. This bit indicates whether the current packet has this header. If a new header is to be added while the packet is being processed, the *validity* bit of this header must be activated. To remove a header, it is sufficient to set the *validity* bit to 0.

5.2.1.2. Parser definition

When all required headers have been defined, the parser can be programmed. It specifies the order of the headers and consists of several states. In each state a transition to a subsequent state is executed. Thus it is possible to parse several headers one after the other, enabling the usage of several headers for later processing. An example parser for Ethernet and IPv4 packets is shown in listing 2.

Suppose we want to program a switch that supports simple IPv4 forwarding. Each incoming packet consists of an Ethernet frame with the previously specified fields, since layer 2 technology is, at least in this case, Ethernet. The first state is the predefined start state. Since we always expect Ethernet frames, it is possible to change from the start state to the state for parsing Ethernet frames. After the Ethernet frame is parsed and one wants to use IPv4 forwarding, the protocol of the next layer has to be determined. The Ethernet type in the Ethernet frame is used for this purpose. If it is not an IPv4 packet, the packet is not parsed any further and the parser switches to the accepting state. Otherwise the state for parsing IPv4 packets is selected.

```
parser packetParser(packet_in packet,
                    out headers hdr,
                    inout metadata meta,
                    inout standard_metadata_t standard_metadata) {

    state start { // entry point
        transition parse_ethernet;
    }

    state parse_ethernet { // parse ethernet packet
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            TYPE_IPV4 :         parse_ipv4;
            default :           accept;
        }
    }

    state parse_ipv4 { // parse ipv4 packet
        packet.extract(hdr.ipv4);
        transition accept;
    }
}
```

Listing 2: Example parser definition.

5.2.1.3. Control Block

As previously noted, the P4 pipeline is divided into an ingress and an egress part. These parts form a so-called control block. Within these control blocks additional control blocks can be invoked. Control blocks are used to encapsulate functionality and form an interface between the other components of the architecture, similar to classes in object-oriented programming [31]. Control blocks allow sequential execution of operations and table matching. The sequence of these operations - and thus also the order in which a packet is processed - is described in a so called *apply* block. Listing 3 shows an example for a simple IPv4 forwarding control block.

```
control IPv4(inout headers hdr,
             inout metadata meta,
             inout standard_metadata_t standard_metadata){

    action ipv4_forward(egressSpec_t port) {
        standard_metadata.egress_spec = port;
        hdr.ipv4.ttl = hdr.ipv4.ttl - 1;  // decrement time to live (ttl)
    }

    table ipv4_lpm { // table for ipv4 unicast match
        key = {
            hdr.ipv4.dstAddr: lpm;
        }
        actions = {
            ipv4_forward;
        }
    }

    apply {
        ipv4_lpm.apply();
    }
}
```

Listing 3: Example for control block.

The control block of the IPv4 unit contains a table that maps the destination address of the IPv4 packet to an action, using longest prefix matching. This action contains the port on which the packet must be sent as parameter. Besides a longest prefix match, P4 supports exact matches, range matches, selector matches and so-called ternary matches in its current specification, which allow the definition of a wildcard mask. A table can contain several possible actions, whereby a table entry is always assigned to one action. The table can be illustrated as follows.

Prefix	Action
134/8	ipv4_forward(1)
134.12/16	ipv4_forward(2)
134.12.23.3	ipv4_forward(3)

Table 5.1.: P4 table for IPv4 forwarding.

If there is a matching entry for the destination address of the current IPv4 packet, the associated action is triggered (ipv4_forward), within the action the TTL is adjusted and the outgoing port is specified. A table can have several actions to choose from and thus take over several functions. Within the apply block several different tables can be matched and actions of the control block can be called.

Since a device often has to support more than one protocol, it makes sense to define each protocol as a separate control block and to call the respective protocol-specific blocks within the ingress control block. Listing 4 shows how specific control blocks can be called inside the ingress control block.

```
control ingress(inout headers hdr,
                inout metadata meta,
                inout standard_metadata_t standard_metadata) {
    IPv4() ipv4_c; // IPv4 control block

    apply {
        if(hdr.ethernet.etherType == TYPE_IPV4 {
            ipv4_c.apply(hdr, meta, standard_metadata);  // apply ipv4 control
        }
    }
}
```

Listing 4: Use of control blocks to isolate functionality.

These nested control blocks allow a good structuring and encapsulation of the functionality. Within the egress control block other control blocks can be called as well.

5.2.1.4. Deparser Definition

The last step before a packet is sent is to add all valid headers. Within packet processing, new headers can be added as well as existing headers removed. The deparser consists of a control block in which all possible headers are added in the correct order using an *emit* statement. Only the valid headers, indicated by the *validity* bit, are added. Listing 5 shows an example of a deparser for Ethernet and IPv4 headers.

```
control deparser(packet_out packet, in headers hdr) {
    apply {
        packet.emit(hdr.ethernet);
        packet.emit(hdr.ipv4);
    }
}
```

Listing 5: Example for a deparser.

5.2.2. OpenFlow vs. P4

OpenFlow is one of the most popular southbound interfaces for SDN architectures and defines a protocol to program flow tables of different switches and routers. In its original definition, an OpenFlow switch consisted of at least three components: A flow table that assigns an action to each flow entry, a secure channel that connects the switch to its controller, and the OpenFlow protocol that defines a communication standard between the controller and the switch [26]. In its current specification, OpenFlow specifies headers that can be used to match on packets. Initially, OpenFlow 1.0 supported only a few header types, including Ethernet, TCP and IP. Meanwhile, OpenFlow 1.5.1 supports several header types, including MPLS and many more [5]. This restriction causes that new, yet unsupported, protocols cannot be used with OpenFlow. Since BIER is a relatively new protocol, there is currently no native OpenFlow support.

In contrast, P4 is not a protocol for communication between switch and controller, but a programming language that makes it possible to program the data plane. In P4 it is possible to implement arbitrary new protocols and to use the newly defined headers for match+action operations. This makes P4 much more suitable than OpenFlow in the context of this thesis.

5.2.3. P4 Runtime

While P4 specifies the way a packet is processed by the switch, P4 Runtime [39] allows a controller to fill the tables defined in P4. In the SDN context, the P4 Runtime forms the southbound interface between the switch and the controller. The communication between switch and controller uses the gRPC protocol [1], a remote procedure call system based on HTTP/2 and protobuf [40]. Figure 5.3 illustrates the P4 Runtime architecture.

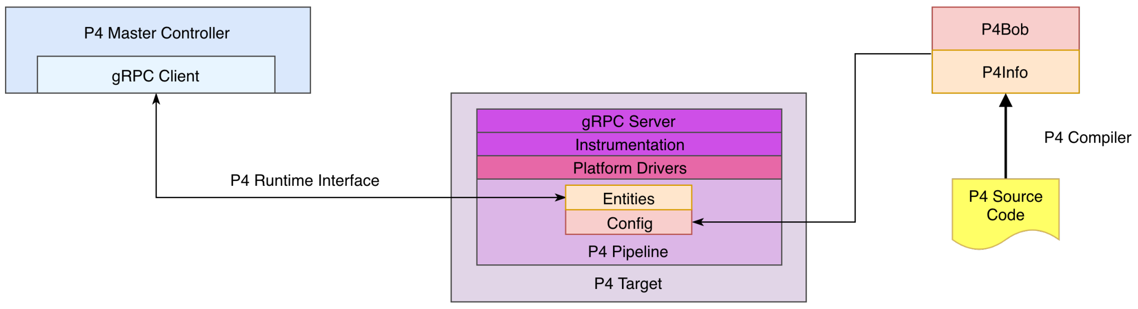


Figure 5.3.: Simplified P4 Runtime architecture representation with reference to [39].

Figure 5.3 shows a P4 device, which is controlled by a single master controller. The P4 Runtime interface between the controller and the device is used to write table entries. The Instrumentation section of the P4 Runtime server enables the translation from target independent P4 Runtime API calls to target dependent instructions. In principle, the P4 Runtime supports the use of multiple controllers, of which only a single controller has write access. The so called slave controllers can be used for statistics or similar matters.

6. Related Work

The following chapter deals with related work in the area of multicast and BIER in the SDN context. An alternative BIER implementation in OpenFlow will be discussed briefly, followed by some work on multicast and SDN.

6.1. Bit Index Explicit Replication using OpenFlow

The authors of [16] presented the first implementation of BIER using OpenFlow and an external controller. Due to the limitations of OpenFlow, described in Chapter 5, it is not possible to implement an independent BIER header. Instead, Giorgetti et al. used MPLS to encode the bit string as an MPLS label and used OpenFlow group tables to implement the desired forwarding behavior. One of the basic principles of BIER is statelessness, which means that intermediate nodes do not need to be updated when multicast groups change. Due to the limitation of OpenFlow, an MPLS BIER implementation offers two possibilities to realize the BIFT. Either the controller updates the BIFT to cover the currently possible bit strings or all possible bit strings, represented as MPLS labels, are stored in the BIFT. The first approach is contrary to statelessness and therefore does not differ from classical multicast. The second approach leads to exponentially growing table sizes - with a bit string of n bits, 2^n possible bit strings exist. In theory, since OpenFlow 1.5, recirculation of packets is possible, which would allow an iterative processing of the bit string. This could significantly reduce the number of BIFT entries - linear in the number of bits. In practice, there is no hardware that supports OpenFlow 1.5, mostly only OpenFlow 1.3 is supported.

6.2. IP-Multicast using SDN

The survey [19] provides a broad overview of multicasting in SDN. They provide many aspects for SDN based multicast like tree planning, multicast routing and reliability. Most work using SDN for multicast relies on explicit flow specific multicast trees, using an external controller for tree computation and SDN as a tool to spread the calculated rules. The authors of [36], [35] use OpenFlow to enable an efficient, scalable and transparent delivery of over-the-top streams, leveraging multiple trees for load-balancing of multicast traffic and efficient group subscription changes. The authors of [23] use OpenFlow in combination with an intelligent controller to compute multiple shared trees, which improves the scalability compared to source dependent trees. The controller clusters the multicast sources into groups, enabling the selection of a centralized rendezvous point (RP). This RP is used to construct shortest-path multicast trees for each cluster. Hu et al. [17] propose two

$(2 + \epsilon)$-approximation methods for the uncertain multicast problem, i.e. enabling the use of uncertain sources. The uncertain multicast problem aims to construct a forest, which ensures that each destination connects to one source through a path in the forest. The objective of the uncertain multicast problem is to minimize the cost of the forest, which is the sum of all edge costs. They use a centralized SDN controller for computation. [18] propose modified Steiner trees, called Branch-Aware Steiner Tree (BST), which try to minimize the number of used edges and branch nodes in a tree. They further show that this problem is NP-hard by reducing it to integer linear programming (ILP), which is shown to be NP-complete. Afterwards they design an 1.55-approximation algorithm to solve this problem, which lies in APX[1] and thus can be deployed computationally efficient using SDN.

Kotani et al. [20] propose a management method for multicast trees that supports failure recovery and dynamic group membership changes. A centralized OpenFlow controller calculates multiple multicast trees and, upon a failure, checks which pre-installed tree is not affected. The unaffected tree gets activated through a new rule on the root switch (a switch connected to a sender), which forwards the traffic using the backup tree. The authors of [34] provide resilience for one-link failures using OpenFlow. Each multicast tree is associated with a backup tree, which gets used on a link failure leveraging the fast failover mechanism from OpenFlow. They also propose a method, to extend their solution, to handle more than a single link failure.

[1]APX: Set of NP hard optimization problems that can be approximated within deterministic polynomial time bounded by a constant approximation ratio

7. Implementation

The following chapter describes the implementation of scalable and resilient BIER(-TE), as well as the implementation and architecture of the control plane. Even though the main focus of this work was on the implementation of the concepts presented so far, a detailed code-based description is omitted in favor of readability and instead the implementation is described on an abstract level. A detailed code-based documentation can be found on the enclosed CD.

7.1. Behavioral-Model Version 2 (bmv2)

The bmv2 framework makes it possible to program different architectures for software switches, implementing the target on general purpose CPUs. In this work the so called *simple switch* architecture is used, which corresponds to the architecture shown in Figure 5.2 [32]. The P4 architecture, which determines the set of intrinsic metadata as well as available externs and describes the pipeline, leveraged in this work is the so called *v1model*. Figure 7.1 shows the pipeline of the *v1model*, which is almost the same as shown in Figure 5.2.

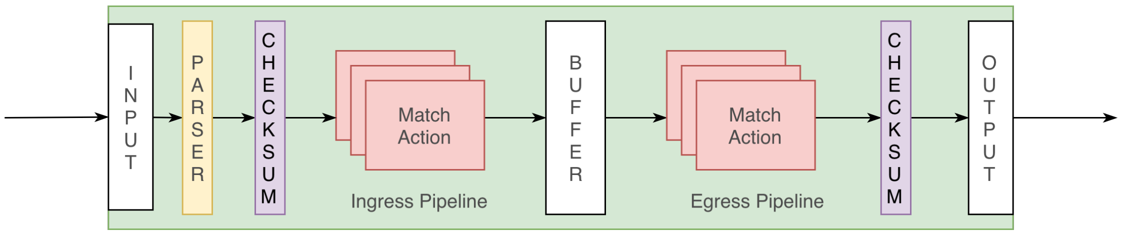


Figure 7.1.: V1model pipeline.

The *v1model* adds a checksum check to the pipeline. After a packet is parsed, it checks if the checksum is valid. At the end of the pipeline, before the packet is sent, a new checksum is calculated and added to the packet.

7.2. General Implementation Notes

As already mentioned in section 5.2.1.3, control blocks are well suited for separating functionality from each other. This concept was extensively used for the P4 based implementation. The ingress and egress pipelines are used to call control blocks that handle the actual packet processing. A separate control block was implemented for each protocol. If a control block is called within the ingress or egress pipeline, a separate copy is created for each of the tables it contains. This means that table entries can be created specifically for the ingress or egress flow, which allows an abstract assignment of tasks. Within the ingress flow all decapsulation operations and header manipulations are performed. In contrast, the egress flow is mostly used for encapsulation, such as the construction of a tunnel. The P4 based implementation consists of the elements shown in table 7.1.

Control block	Task
Port	Manages port information and assigns it to metadata
MAC	Manages the customization of MAC addresses when a packet is forwarded
IPv4	Manages classic IPv4 forwarding
BIER	Manages BIER and BIER-FRR
BIER-TE	Manages BIER-TE and BIER-TE-FRR
Tunnel	Manages all tunnel related operations, such as decap and encap operations, e.g. adds a BIER header to a IP Multicast packet
Topology-Discovery	Processes topology packets and sends them to neighbors / the controller
Subdomain	Manages the subdomain information and adjusts cloned subdomain packets
Segment Routing	Manages SR forwarding and SR-FRR

Table 7.1.: Implemented control blocks.

7.3. Fast Reroute Requirements

A main aspect of this work was to implement resilient BIER and BIER-TE. To support an effective FRR mechanism, the device needs to know which ports are available and which are not. P4 does not support this functionality by default, i.e. there is no mechanism in P4 to check if a port is available or not. For this reason, this information has to be external and made available in the pipeline. How this information is obtained is described in section 7.10. At this point it is assumed that this information can be made available and stored in a table of the switch. The port information is stored in a bit string, similar to BIER. The bit string `1011` means that port 1, 2 and 4 are available, while port 3 is not. As soon as a packet enters the ingress pipeline, the port information is retrieved from the corresponding table and stored in a metadata field by the port control block. Afterwards, each table can use this metadata field to distinguish a normal entry from a backup entry.

7.4. BIER Implementation

RFC 8296 [42] specifies a BIER header for MPLS networks, including the bit string, TTL, version field and many more. For this implementation a simplified BIER header shown in Figure 7.2 was used.

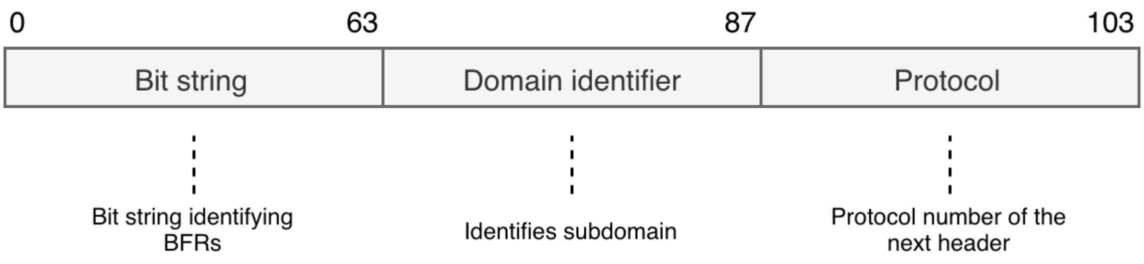


Figure 7.2.: BIER header used for implementation.

The used BIER header supports a bit string of 64 bits and a domain identifier of 24 bits which is used for the assignment of the subdomains. Additionally the header contains a 16 bit protocol field used to identify the next protocol.

In the standard specification [41] an iterative processing of the BIER header is suggested in which every active bit is processed, starting with the least significant bit. If an active bit is matched, a copy of the packet is created and the original packet is used again for matching. This procedure can be implemented very similarly in P4. P4 does not support native loops, only the re-entrance of a packet into the pipeline. There are two different methods: recirculation and resubmit. Figure 7.3 shows the difference of the methods.

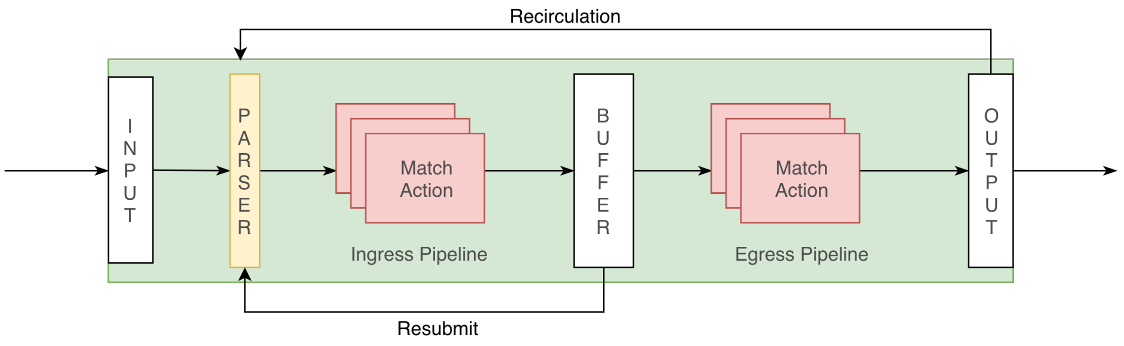


Figure 7.3.: Recirculation vs. resubmit.

If the recirculation operation is executed on a packet, the packet first runs through the entire pipeline before it is led into the parser again. This also means that the packet passes through the deparser and all header changes are performed. On the contrary, in the resubmit operation, the packet is reloaded into the parser after the ingress flow. All header changes made in the ingress are thereby discarded. For this reason the recirculate operation is used for BIER processing. P4 also supports a cloning operation that allows to copy packets. P4 v16 supports two different types, ingress-to-egress (I2E) and egress-to-egress (E2E), which differ in the fact that the packet is copied once at the end of ingress and once at the end of egress pipeline. Although the cloning operation is performed at the end of the corresponding pipeline, the clone corresponds to the packet at the beginning of the ingress, respectively the beginning of the egress pipeline. As a result, in an I2E clone, all header changes made in the ingress are discarded. It is then inserted at the beginning of the egress pipeline. A cloned BIER packet is afterwards recirculated and used again for BIER forwarding.

7.4.1. BIER Forwarding Procedure

The implemented BIER forwarding is almost identical to Figure 3.3. Contrary to the explanation in RFC 8279 it does not necessarily start with the least significant bit. However, RFC 8279 explicitly mentions that the BIER forwarding only has to come to the same result and must not be implemented identically. Matching to an active bit in the bit string is performed using the *ternary* match of P4. A *ternary* match consists of a target value and a mask. The header or metadata field is combined using a bitwise AND with the mask and compared with the target value. If the target value equals the result, there is a match. For instance, if we want to check whether the least significant bit is set in a BIER header containing the bit string `011`, it is sufficient to specify a *ternary* match with the binary value `001` and the mask `001`. If the least significant bit is set, the combination with the mask results in `001`, otherwise `000`. In this way, n BFERs require n table entries in the BIFT, which is equivalent to the specification in RFC 8279. Figure 7.4 illustrates the BIER related pipeline of the implementation.

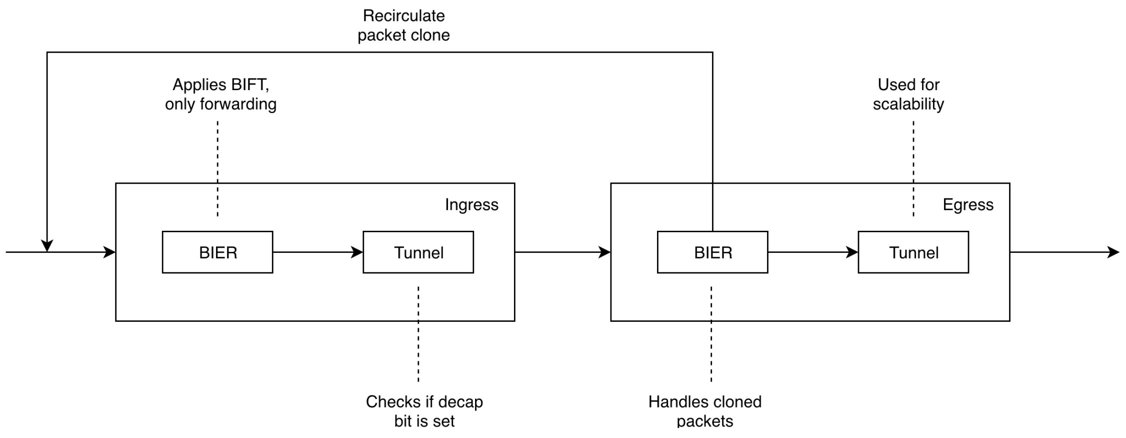


Figure 7.4.: P4 BIER related pipeline.

When a BIER packet enters the ingress pipeline, the BIER control block checks whether there is a forwarding entry in the BIFT. If this is the case, the packet is copied and the FBM is applied. Since the packet is not copied until the end of the ingress pipeline, the adjusted bit string for the recirculated packet must be stored in metadata and, after the packet has passed through the pipeline again using recirculation, restored. Otherwise it wouldn't be possible to handle two packets (original & copy) with different bit strings. After the BIER control block the tunnel control block is invoked. If the decapsulation bit is set, the packet is unpacked and copied. Since the copy is made at the end of the ingress pipeline, the original header is restored as soon as the cloned packet is recirculated and the packet can be further processed. The order in which the bits are processed is therefore irrelevant.

In the egress flow, the BIER control block simply takes over the task of recirculating cloned packets. Contrary to the description in the BIER RFC, the cloned packet is always used to process the remaining header bits. This difference is caused by implementation specific details and does not affect the forwarding process. The tunnel control block is used for the scalability extension, which is described later. The forwarding entries are calculated by the controller using shortest paths.

7.4.2. BIER FRR

The BIER FRR mechanism leverages an IP tunnel for failure bypassing. In order to detect the failure of a port and then select a different action, the BIFT not only matches the bit string of the BIER header, but also the available ports. For this reason, there are two table entries for each BFR-id. A default entry that is used when the outgoing port to the next hop is available, and a backup entry that is used when the outgoing port to the next hop is not available. The backup entry contains a tunnel action that copies the packet, wraps the original BIER packet in an IP packet, adjusts the bit string of the BIER header - in case of node protection - and sends it to the next hop or next next hop in the case of node protection. The normal BIER forwarding routine remains unaffected because the packet copy is processed normally. Default and backup entries can be distinguished with a ternary match to the currently available ports. The default entry checks if the corresponding port bit, which indicates the port to the desired next hop, is set. The backup entry checks if this bit is not set. Both, default and backup entry are calculated by the controller. The tunnel action can also be used to create *routed* connections between BFRs.

7.5. BIER-TE Implementation

The BIER-TE implementation is very similar to the BIER implementation and the same header is used. Likewise, a *ternary* match is used for the different bits in the BIER-TE bit string. The BIER-TE control block is not responsible for unpacking the packet, this is done by the tunnel control block. The implementation supports both *connected* and *routed* connections.

7.5.1. BIER-TE FRR

With BIER-TE FRR, the packet is tunneled over BIER-TE to bypass the local error. In the case of node protection certain bits must be removed from the original BIER header depending on the next next hops. To avoid that the BIFT consists of all possible combinations of bit strings to execute all necessary changes in one step, an iterative procedure is used. The so called BIER-TE Adjacency FRR Table (BTAFT) is leveraged for this purpose. The BTAFT is similar to the BIFT and matches on the remaining bit string of a BIER-TE header and the currently available ports. For node protection one can check to which next next hops the failed next hop would have forwarded the packet. Figure 7.5 illustrates the combination of BTAFT and BIFT in the ingress pipeline.

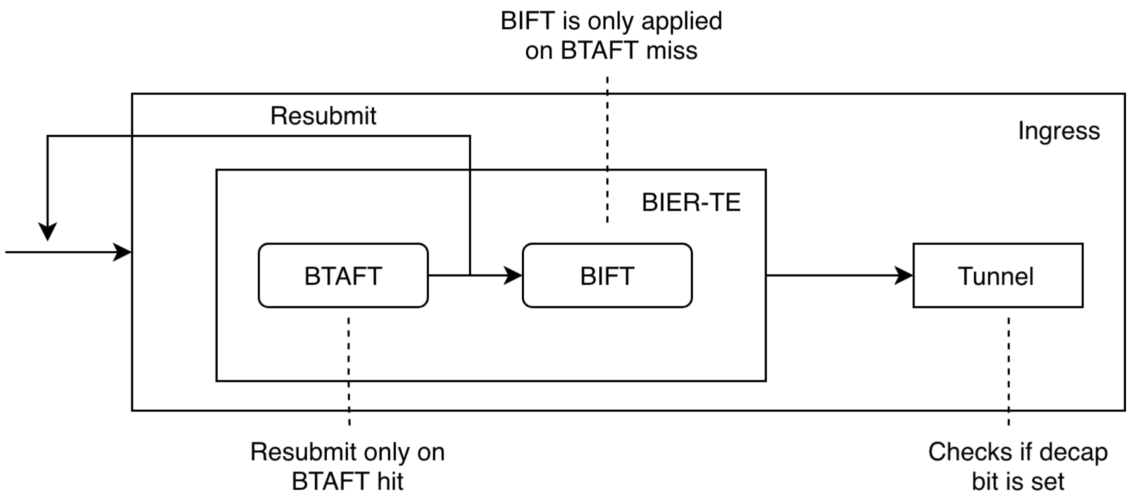

BIFT is only applied on BTAFT miss
Resubmit
Ingress
BIER-TE
BTAFT
BIFT
Tunnel
Resubmit only on BTAFT hit
Checks if decap bit is set

Figure 7.5.: BIER-TE ingress pipeline.

A failed next hop can be detected by the available ports, provided by metadata through the port control block, and a used next next hop by the remaining adjacency bits in the BIER-TE header. Thus for each next next hop to which the packet could go, one can specify a backup path as BIER-TE bit string, the so called add bitmask, and the bits which must be removed from the original header, the so called reset bitmask. The combination of all matching add bitmasks results in the header for the BIER-TE tunnel. The combination of all reset bitmasks results in the bit string that has to be removed from the original BIER-TE header. In the case of link protection the reset bitmask contains only the failed adjacency. Since this procedure requires multiple matches to the same table the resubmit operation is used. The add and reset bitmasks that are matched during this process are stored in metadata. As soon as all next next hops are processed a flag is set and the tunnel control block creates the BIER-TE tunnel.

7.6. Topology Discovery

For known static topologies it is very easy to fill the tables with the corresponding static entries. However, as soon as the implementation is to be used on unknown or dynamic topologies, the topology of the network must be recognized. For this reason, a simple proprietary detection mechanism was implemented using the header shown in Figure 7.6.

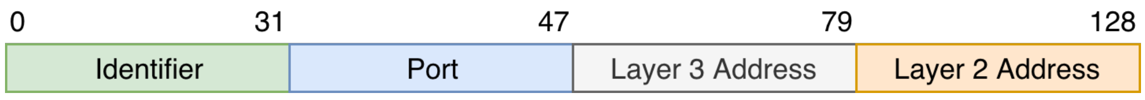


Figure 7.6.: Topology header used for topology discovery.

The header starts with a 32 bit identifier indicating the sender followed by a 16 bit port field. Layer 3 (IP) and layer 2 (MAC) addresses are also part of the header.

All devices periodically send these topology packets to all their neighbors. When a device X receives a topology packet with identifier Y on port Z, it sets the port field in the header to Z and sends the packet to the controller. This way the controller knows that X is connected to Y via port Z.

7.7. IGMP Implementation

To achieve a realistic handling of multicast groups, the IGMPv2 protocol is supported. As described in section 2.1.1, the IGMP protocol is used to subscribe and unsubscribe to multicast groups. End users can dynamically subscribe or unsubscribe from multicast groups and the sent IGMP packets are forwarded from the switch to the controller, which manages the groups and updates the corresponding entries in the BFIRs.

7.8. BIER(-TE) Scalability

The scalability extension of BIER(-TE) is used to reduce the number of table entries and bits in the BIER header. For this purpose, topologically close BFRs are grouped together and considered as subdomains. While this problem is easy to solve for known static topologies, a dynamic method must be used for unknown topologies. To find such groups of closely spaced BFRs, clustering is a suitable method.

7.8.1. Clustering

Clustering is the task of finding groups of objects so that objects within a group are similar to each other and objects in different groups are dissimilar [38]. Although this definition seems very clear at first, the clustering problem is highly non-trivial. First, in most cases the 'similarity' relation is not transitive. On the other hand, it is very difficult to find a so called ground truth[1] because clustering is a classic unsupervised learning[2] problem. The cluster problem is defined as follows [38]:

Input:	A set of elements $\mathcal{X}$ and a distance function d, $d : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}_+$, that is symmetric, satisfies $d(x, x) = 0$ for all $x \in \mathcal{X}$ and calculates the 'similarity'. Some cluster algorithms also require the number of clusters.
Output:	A partition of the domain set $\mathcal{X}$ into subsets $C = (C_1, ..., C_k)$, such that $\bigcup_{i=1}^{k} C_i = \mathcal{X}$ and $C_i \cap C_j = \emptyset, \forall i \neq j$.

There are many clustering algorithms, for example k-means, kernel k-means and linkage algorithms. Nevertheless, spectral clustering has become one of the most

[1]Ground truth: Result taken as best/correct solution

[2]Unsupervised learning: Unsupervised learning, in contrast to supervised learning, does not involve so called labels which represent the desired result

popular clustering algorithms in recent years [24]. Due to its efficiency and superiority over other clustering algorithms, spectral clustering was used for subdomain classification.

7.8.1.1. Spectral Clustering

Spectral clustering solves the problem of clusters in graphs. The input therefore consists of a graph $G = (V, E)$ and the goal is to find clusters such that there are many edges within the clusters and as few edges as possible between the clusters. If the input is not a graph, i.e. arbitrary data points, a k-nearest neighbor graph can be computed. Intuitively the problem described before corresponds to a classic min-cut algorithm, which could be solved with the Max-Flow-Min-Cut theorem using the Ford/Fulkerson algorithm in polynomial time. The problem with a pure min-cut algorithm is that mostly outliers end up in their own cluster and the clusters tend to be unbalanced. The goal would therefore be a so called balanced min cut. However, the balanced min cut problem is NP hard.

For this reason spectral clustering uses a balancing term with later relaxation [25]. Instead of the normal definition of a cut

$$cut(A, B) = \sum_{i \in A, j \in B} w_{ij}$$

where w_{ij} corresponds to the weight assigned to the edge ij, a so called RatioCut is used

$$RatioCut(A, B) = cut(A, B) + \left(\frac{1}{|A|} + \frac{1}{|B|} \right)$$

which forces equally sized partitions in order to minimize this cut. However, finding a global minimum of RatioCut is still NP hard.

Graph Laplacian

Given an undirected graph with non-negative edge weights w_{ij}, W is called the weight matrix and $D := diag(d_1, ..., d_n)$ the degree matrix of the graph. The unnormalized Graph Laplacian L is defined as:

$$L = D - W$$

It's not difficult to show that for all $f \in \mathbb{R}^n$ the so-called key property holds:

$$f^t L f = \frac{1}{2} \sum_{i,j=1}^{n} w_{ij} \; (f_i - f_j)^2$$

Unnormalized Spectral Clustering

In order to find two clusters of the same size, we want to solve the RatioCut problem:

$$\min_{A \subset V} RatioCut(A, \overline{A})$$

By defining a vector $f \in \mathbb{R}^n$ with

$$f_i = \begin{cases} +(|\overline{A}|/|A|)^{\frac{1}{2}} & i \in A \\ -(|A|/|\overline{A}|)^{\frac{1}{2}} & else \end{cases}$$

we can rewrite the RatioCut as $RatioCut(A, \overline{A}) = f^T L f$ and the RatioCut problem as:

$$\underset{f}{min}\ f^T L f \text{ subject to } f_i \text{ of the form given above}$$

Any f of the form given above also satisfies $\sum_{i \in V} f_i = 0$, which is equivalent to $f \perp \mathbb{1}$. The hard constraint for f_i can be relaxed by simply enforcing $f_i \in \mathbb{R}$ which gives rise to the following problem

$$\underset{f}{min}\ f^T L f \text{ subject to } f \perp \mathbb{1} \text{ and } f_i \in \mathbb{R}$$

Using the Rayleigh-Ritz theorem [24], the solution to this minimization problem is the second smallest eigenvector f of L. The clusters can then be assigned according to the sign of f_i, i.e. $A = \{i |\ f_i \geq 0\}, \overline{A} = V \setminus A$.

The basic idea of the algorithm can be extended for several clusters. The smallest k eigenvectors of L encode the cluster structure. To obtain the cluster structure a so called spectral embedding is performed and the obtained points are clustered with a simple algorithm, e.g. k-means. The spectral embedding simply takes the matrix V, defined by the first k eigenvectors, and uses each row as a new data point.

In practice, mostly the so called normalized spectral clustering is used. The difference is that the balancing term does not use $|A|$, but the volume of A. This leads to the fact that the eigenvectors of the so called random walk Laplacian are used, which usually results in better solutions. In the implementation the scikit library [33] was used.

7.8.2. Subdomain Matching

Once the subdomains are defined, it must be determined which subdomains require a copy of an incoming IPv4 multicast packet. Each subdomain requires a separate BIER(-TE) header. To that end, the tunnel control block of the egress pipeline contains a table which matches IPv4 multicast addresses to a bit string representing all subdomains that need a copy of the packet. For example, the bit string `011` corresponds to the subdomains 1 and 2. Figure 7.7 shows an abstract example for the subdomain matching procedure.

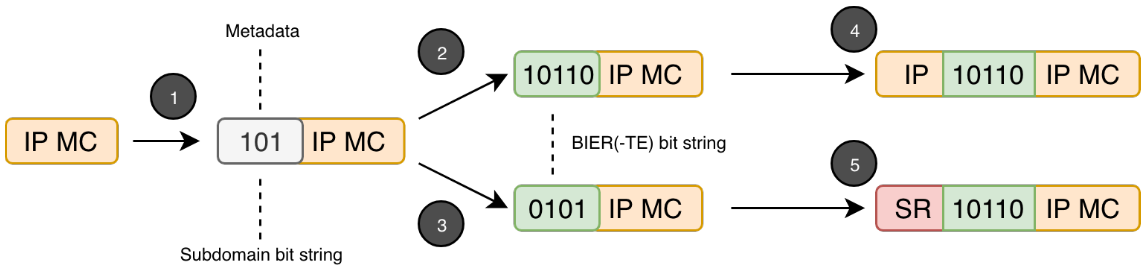


Figure 7.7.: Example for subdomain matching procedure.

As soon as an IP multicast packet arrives at a BFIR, the destination address is used to determine through a table which subdomains require a copy of this packet. The subdomain information is then stored in metadata ❶ and each [subdomain-bit, IP multicast address] pair can be mapped to the corresponding BIER(-TE) header ❷ ❸. The mapping is again performed leveraging a table that compares the current subdomain metadata bit string and the packet IP multicast address. The corresponding action puts the appropriate BIER header on the packet and removes the matched subdomain bit from the subdomain metadata. This mechanism, similar to normal BIER forwarding, is performed iteratively using a *ternary* match, *clone* operation and *recirculation*. Afterwards, each BIER(-TE) packet can be tunneled to its appropriate tunnel node using the subdomain identifier in the BIER header, either through a plain IP tunnel ❹ or a segment routing tunnel ❺. For BIER, an implicit tunnel leveraging the domain identifier can be used as described in section 4.2.1.

7.9. Segment Routing

RFC 8402 [12] proposes a standard for a mechanism called segment routing (SR), which follows the source routing[3] paradigm [11]. In this way, SR can be used to, for example, bypass local errors or creating traffic engineered explicit tunnels.

7.9.1. Concept

The SR mechanism introduces so called Segment Identifiers (SIs) or short segments. A segment represents an arbitrary instruction, which either has local meaning (local segment) for a certain node, or a global meaning (global segment) within the entire domain. Ingress nodes push ordered lists of segments on packets, which determine the performed actions. An intermediate node can remove or add segments to that list. A possible instruction for a segment is a forward command, which allows the specification of implicit/explicit paths depending on whether local or global segments are used. Other possibilities for instructions would be for example QoS handling or processing via a virtual machine. Thus, SR enables per-flow routing, whereby only the ingress nodes need to store a per-flow state [11]. In the context of this work, only forwarding instructions are relevant.

[3]Source routing: The sender specifies the entire path used by the packet [14].

RFC 8402 proposes two data plane instantiations of SR: SR over MPLS (SR-MPLS) and SR over IPv6 (SRv6) [12]. SR-MPLS uses a stack of MPLS labels for the encoding of the segments. The currently active segment is on the top of the label stack. As soon as the segment instruction is performed, the related label is popped from the stack and the next segment can be processed. [13] proposes a new routing header for IPv6 called SR header (SRH). A segment is encoded as IPv6 address and then into a list of segments in the routing header.

Both SR-MPLS and SRv6 require a large header overhead. For this reason, a new proprietary segment routing header was introduced.

7.9.2. Implementation Details

To enable a more efficient processing of the segment routing header a new SR header was introduced. Figure 7.8 shows the used segment routing header. The header consists of a 128 bit long bit string encoding the list of segments. One segment consists of 8 bits, respectively a list of 16 segments can be encoded. The 8 leftmost bits correspond to the first segment. Additionally a 16 bits field indicates the protocol number of the next header.

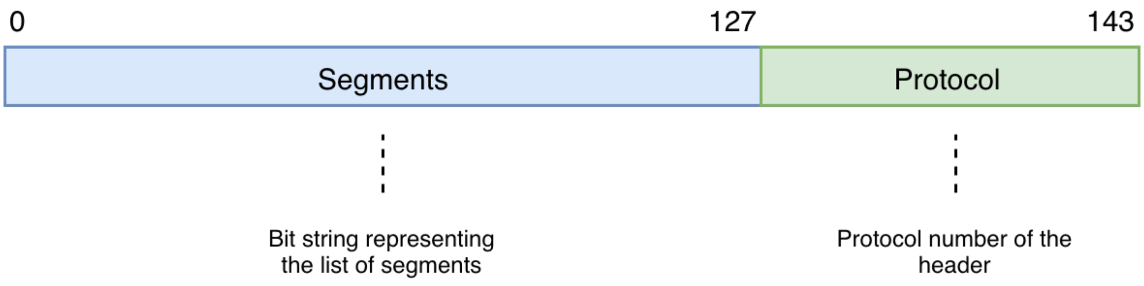

0
127
143
Segments
Protocol
Bit string representing the list of segments
Protocol number of the header

Figure 7.8.: Proprietary segment routing header.

If a node receives a packet with a segment routing header, it is checked whether an instruction is stored for the first segment, represented by the 8 leftmost bits. If this is the case, the instruction is executed and the top segment may be removed depending on the type of the segment. Otherwise, the packet is discarded. The first segment can be identified by a *ternary* match with mask `111111110000...`, containing `1`s in the first 8 bits. The removal of the first segment can be done with an 8 bit left shift.

Forwarding Procedure

The forwarding process differs for local and global segments. Local segments are used for hop-by-hop routing. Global segments represent a destination node similar to an IPv4 address. This distinction allows a loose source routing path, i.e. a path that switches between explicit hops and implicit hops. One part of the path can be specified explicitly, another part only implicitly using a target node. Global forwarding entries, for example, can be selected leveraging shortest paths. The so-called null segment, containing a zero bit string, represents the end of the segment list and will cause the SR header to be removed. Figure 7.9 illustrates the forwarding mechanism.

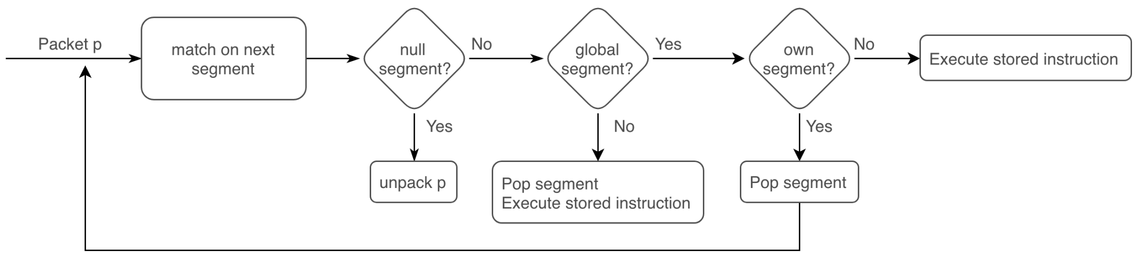


Figure 7.9.: SR forwarding procedure.

When a device receives a packet with a SR header, the next active segment is matched using a *ternary* match. Thereby it is checked whether the current segment is a null segment. If this is the case, the SR header is removed and the packet is recirculated to be able to process the underlying header. Otherwise, the system checks whether the next segment is a local or global one. In the case of a local segment, the segment is removed using a shift operation and the stored action is executed. For a global segment, the system checks whether this segment identifies the current node. If so, the global segment is removed and the packet is recirculated to process the next segment. In other cases, the stored action is executed. The removal of the SR header in case of a null segment is done by the tunnel control block.

7.9.3. Segment Routing FRR

To ensure reliability for segment routing, the authors of [15] propose a FRR mechanism adapting the IP-LFA [4] mechanism. However, since segment routing allows the specification of explicit paths, it is sufficient to encode a backup path that bypasses the error.

7.9.3.1. Link Protection

In the case of link protection, it is assumed that the link between the PLR and its next hop has failed. To bypass this error, the packet must be sent to the next hop on a different path. Since segment routing supports both local and global segments, a different action must be performed depending on whether the segment is local or global. If the segment is local, it is sufficient to remove this segment and insert a list of segments encoding a backup path to the next hop at the beginning of the segment list. For global segments instead, is is sufficient to just add the backup list.

7.9.3.2. Node Protection

In the node protection case, the packet must be sent to the appropriate next next hop via a backup path. As in the BIER(-TE)-FRR case, the work of the failed next hop must also be performed by the PLR and the packet has to be sent to the next next hop. While maintaining a global view, the controller can identify the respective next next hop leveraging the combination of first and second segment. In the following it is assumed that the path from the PLR to the next next hop was determined by the controller and that the first segment on the list implies the usage of the broken link/node. For the FRR behavior, a number of cases need to be distinguished:

- **The first segment is global and identifies the failed next hop.**

  In this case, the second segment must be considered. If the second segment is local, the first two segments are removed and a backup path to the next next hop is specified. If the second segment is global, the packet must be sent via a backup path to the next next hop and only the first global segment must be removed.

- **The first segment is global and does not identify the failed next hop.**

  In this case it is sufficient to send the packet via a backup path to the next next hop. No segment has to be removed.

- **The first segment is local.**

  If the first segment is local and the second global, the local segments has to be removed and the packet has to be sent using a backup path to the next next hop. If the second segment is local as well, both segments have to be removed and the packet has to be send to the next next hop.

All cases can be covered using *ternary* matches on the first two segments. For each entry the controller defines the backup path, identified by a bit string representing the backup segment list and the number of segments that must be removed.

7.10. Controller Implementation

The P4 implementation covers the data plane implementation. However, in order to use the implementation with unknown or dynamic topologies, the table entries must be calculated for the respective network topology. This task is performed by the controller that implements the control plane.

The P4 runtime enables communication between the P4 switch and the central controller. Since the communication is based on gRPC and protobuf, it is possible to write the controller in any language that supports these two protocols. In this work Python 2.7 was used, which corresponds to the used Python version of the P4 working group.

7.10.1. Architecture

In the course of this work, a distributed controller concept was developed. The controller architecture is illustrated in Figure 7.10.

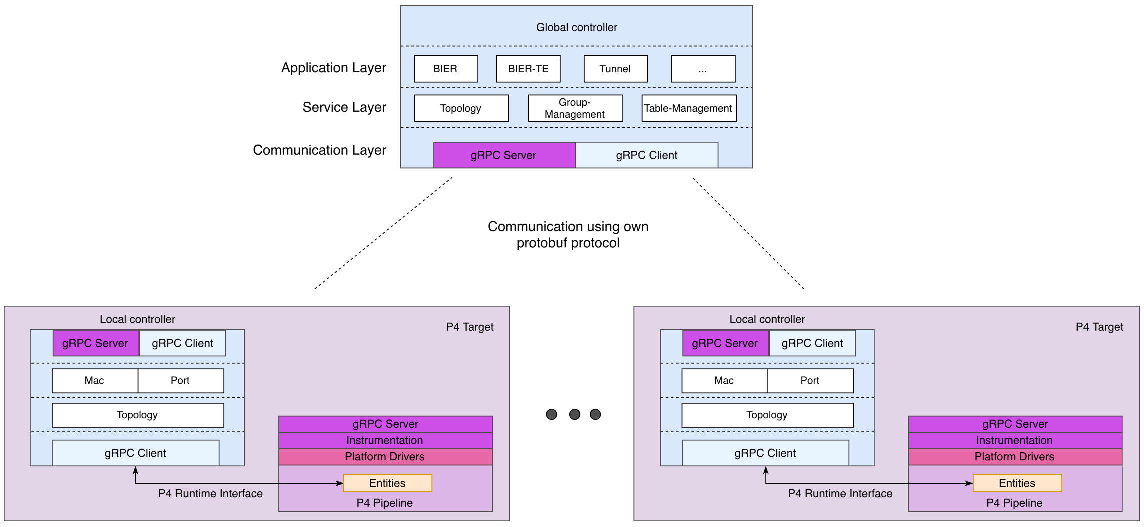


Figure 7.10.: Implemented controller architecture.

Each P4 switch is controlled by its own local controller, which communicates with the switch using the P4 Runtime interface. The local controller only maintains a local view of the network, i.e. it only knows its neighbors. Each local controller communicates with a single global controller, which has a global view of the network and knows the entire topology.

Local Controller

The local controller is responsible for the following tasks:

- **Topology Discovery**

  The data plane written in P4 cannot create packets by itself. For this reason, the local controller generates the corresponding topology packet, according to section 7.6, which identifies the switch and sends it to the device using the P4 Runtime interface. The switch forwards this packet to all its neighbors. If the switch receives a topology packet from its neighbor, the associated local controller can recognize its local topology. The incoming topology packet is then forwarded to the global controller.

- **MAC Application**

  As shown in table 7.1, the switch adjusts the MAC address of the outgoing packet. Without this adjustment, the receiving device would discard the packet. Since the switch only implements the data plane, the information about how a packet has to be adapted must be determined by the controller. This task is performed by the MAC application. The topology manager of the local controller receives the information about connected ports and the MAC addresses behind them via the topology packets. The MAC application uses this information to calculate the corresponding table entries and sends them to the switch using the P4 Runtime interface.

- **Port Application**

  As mentioned in section 7.3, the information about the available ports must come from outside. This task is performed by the port application. The bmv2 framework creates a so called nanomsg channel [2], on which a message is sent when the status of a port changes. The local controller receives this message and can thus update the associated internal port status. The port application then updates the switch's associated table, which provides the port information for the data plane. The way the port information is provided depends on the target. On real hardware, this task would have to be performed by the switch itself.

Global Controller

The architecture of the global controller can be divided into 3 layers as shown in Figure 7.10. The communication layer is responsible for the communication with the local controllers. Since the P4 Runtime only allows one controller with write access, only the local controller communicates directly with the switch. All changes calculated by the global controller are sent to the local controller using a separate channel. The local controller forwards the changes to the switch using the P4 Runtime interface. The service layer provides services for the application layer. This includes topology information, managing multicast groups, and table management on the switches. The application layer calculates the actual table entries. In principle, there is a separate application for each control block in the P4 pipeline. The BIER application calculates the BIER(-FRR) entries, the BIER-TE application calculates the BIER-TE(-FRR) entries, and so on.

Service Layer

The main tasks of the service layer include

- **Topology Management**

  Based on the topology packets received from the local controllers, the topology manager can generate the entire topology of the network. On the basis of this topology the subdomains can be determined by spectral clustering and shortest and backup paths can be calculated. The representation of the topology was implemented with the help of the networkx library [37].

- **Group Management**

  The local controllers forward all IGMP packets to the global controller while the group manager maintains the state of the multicast groups. This information can be used to determine which BFER needs which IP multicast packet.

On this basis, the tunnel application can define the rules for encapsulating IP packets.

- **Table Management**

  Since the topology is determined dynamically and can change, previously calculated table entries may no longer be valid. The table manager allows the creation of table entries and the automatic removal of invalid entries.

Application Layer

The application layer manages the tables of the P4 pipeline. This includes: BIER(-FRR), BIER-TE(-FRR), scalability extension, tunnel, segment routing and IPv4. Applications can register to certain event routines. As soon as the service layer notices a change, for example in the topology or multicast groups, these routines are called. The IPv4 application, for example, can register for a topology change and re-calculate its IPv4 forwarding rules as soon as a topology change is reported. All invalid table entries are automatically removed by the table m anager.

The global controller supports all concepts introduced so far. All forwarding rules are calculated based on shortest paths.

Communication Layer

As noted above, the communication layer is responsible for communication between the global controller and local controllers. According to the communication of the P4 Runtime, a combination of gRPC and protobuf was chosen.

Protobuf [40] allows the definition of message formats and is comparable to XML. The communication between the global controller and local controllers is bidirectional. The global controller sends calculated table entries to the local controllers, the local controllers in turn forward IGMP and topology packets to the global controller. The global controller also sends requests to delete existing table entries. Since gRPC is based on HTTP/2 and therefore does not allow a real persistent bidirectional connection on one channel, both the local and the global controller create a gRPC server. The gRPC server of the local controller receives the messages from the global controller, the gRPC server of the global controller receives the messages from the local controller.

7.10.1.1. Communication

While the local controllers run on the switches, the global controller can run on any device in the network. Since the global controller can change its position flexibly, it is a hindrance if the local controllers need to know to whom they are connecting. For this reason, the global controller first reports to the local controllers using a Hello Message, which includes its IP address and his gRPC listen port. Listing 6 shows the definition of the Hello Message in protobuf.

```
message HelloMessage {
  // indicates IP of global controller
  string ip = 1;
  // indicates gRPC server port
  uint32 port = 2;
}
```

Listing 6: Hello Message from global controller.

```
message SwitchInfo {
  string name = 1;
  string ip = 2;
  string mac = 3;
  uint32 bfr_id = 4;
}
```

Listing 7: Response from local controller.

Using the IP address and the gRPC port contained in the Hello Message, the local controllers can connect to the global controller. To let the global controller know which switch is connected, the switch sends its name, IP address, MAC address and BFR-id in response to the Hello Message. The BFR-id is changed if the scalability extension is used. Listing 7 shows the response to the Hello Message.

Since only the local controller can communicate directly with the switch, the global controller must send all calculated table entries to the local controller. As table entries differ greatly in their P4 Runtime interface definition, a table entry object is transferred as a serialized string. Otherwise many different definitions for different table entries would be needed. Listing 8 shows the definition of a table entry message.

```
message TableEntry {
  // represents a table entry
  // table_entry is a serialized pickle string containing a TableEntry Object
  // used to send a table entry from global to local controller
  string table_name = 1;
  string table_entry = 2;
}
```

Listing 8: Message format for table entry.

Similarly, IGMP and topology packets are sent from the local to the global controller.

7.10.2. Configuration

The global controller can be individually configured to support different extensions. A JSON [9] file as shown in Listing 9 can be used to configure the deployed technologies.

```json
{
  "switches": [
    {
      "name": "s1",
      "ingress": false,
      "local_controller_port": 30051
    },
    {
      "name": "s2",
      "ingress": true,
      "local_controller_port": 30052
    }
  ],
  "mode": "BIER-TE",
  "debug": true,
  "cluster": 2,
  "tunnel": "IPv4",
  "listen_port": 53020,
  "protection": "None"
}
```

Listing 9: Example for configuration file for the global controller.

Listing 9 shows a configuration file for a network with 2 switches that the global controller should manage. In addition to the name of the switches for internal identification, it is indicated whether the switch is an ingress node of the BIER domain. Furthermore, the port of the local controller is specified so that the global controller can connect to it. In addition to the switch information, options can be specified for different applications. Table 7.2 shows the used controller options.

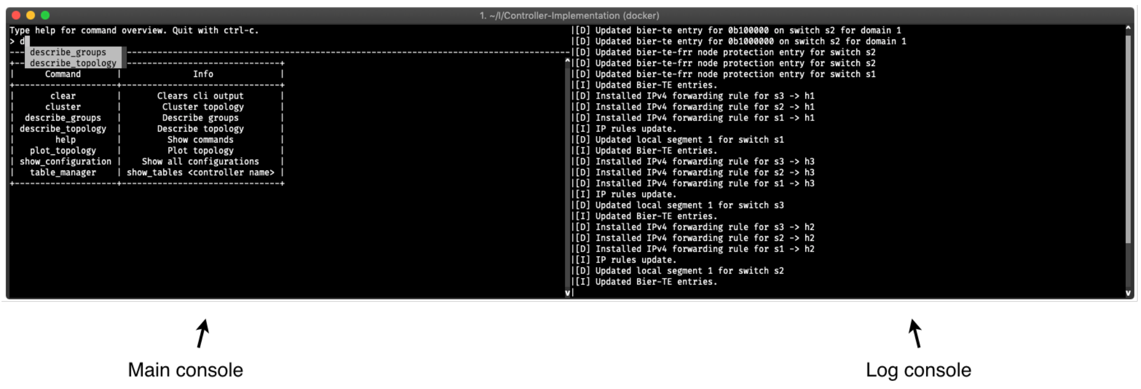
Option name	Possible values	Usage
mode	BIER, BIER-TE	Specifies whether BIER or BIER-TE is used.
debug	true, false	Enables debug mode on the controller CLI.
cluster	1, 2, 3, ...	Specifies the desired number of clusters for BIER(-TE) scalability.
tunnel	None, IPv4, SR, SR-local	Specifies the type of tunnel to the subdomains. None corresponds to the domain forwarding approach. SR uses only a single global segment (similar to IPv4). SR-local uses a hop-by-hop path using local segments.
listen_port	49152, 49153, 49154, ...	Port used for gRPC server on the global controller.
protection	None, Link, Node	Specifies if link or node protection is used. None corresponds to no protection scheme.

Table 7.2.: Supported controller options.

7.10.3. Command Line Interface (CLI)

Global Controller

Based on the event system, all table entries are automatically calculated and updated if there is a topology or group change. In addition, the global controller supports a command line interface and a detailed log system. Figure 7.11 shows the command line interface of the global controller.



```
Type help for command overview. Quit with ctrl-c.
> d
--- describe_groups  ------------------------------
+-- describe_topology ----------------------------+
|      Command       |           Info             |
+--------------------+----------------------------+
|       clear        |      Clears cli output     |
|      cluster       |      Cluster topology      |
|  describe_groups   |      Describe groups       |
| describe_topology  |      Describe topology     |
|        help        |       Show commands        |
|   plot_topology    |       Plot topology        |
| show_configuration |   Show all configurations  |
|   table_manager    | show_tables <controller name> |
+--------------------+----------------------------+
```

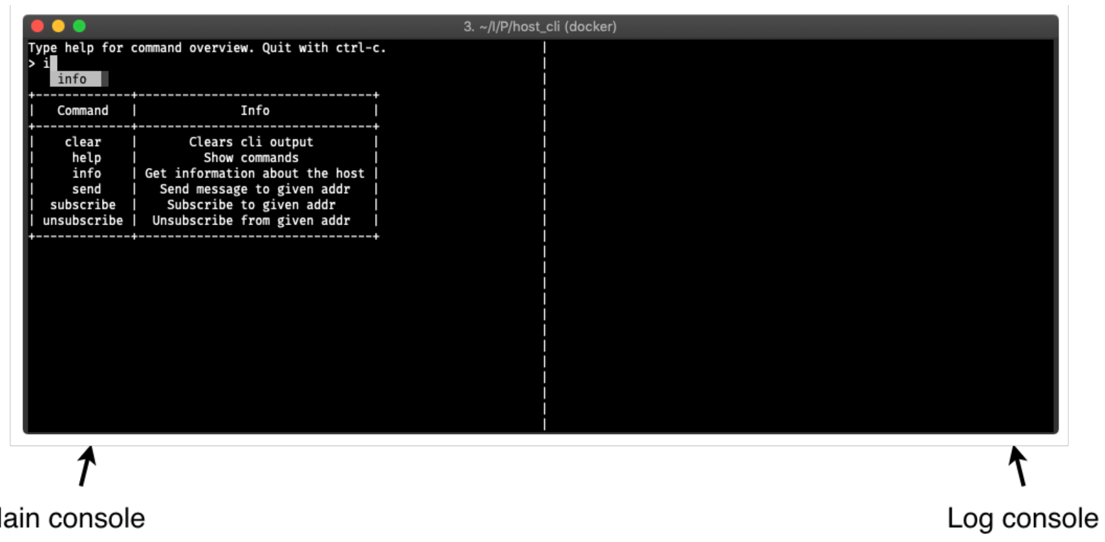
```
|[D] Updated bier-te entry for 0b100000 on switch s2 for domain 1
|[D] Updated bier-te entry for 0b1000000 on switch s2 for domain 1
|[D] Updated bier-te-frr node protection entry for switch s2
|[D] Updated bier-te-frr node protection entry for switch s2
|[D] Updated bier-te-frr node protection entry for switch s1
|[I] Updated Bier-TE entries.
|[D] Installed IPv4 forwarding rule for s3 -> h1
|[D] Installed IPv4 forwarding rule for s2 -> h1
|[D] Installed IPv4 forwarding rule for s1 -> h1
|[I] IP rules update.
|[D] Updated local segment 1 for switch s1
|[I] Updated Bier-TE entries.
|[D] Installed IPv4 forwarding rule for s3 -> h3
|[D] Installed IPv4 forwarding rule for s2 -> h3
|[D] Installed IPv4 forwarding rule for s1 -> h3
|[I] IP rules update.
|[D] Updated local segment 1 for switch s3
|[I] Updated Bier-TE entries.
|[D] Installed IPv4 forwarding rule for s3 -> h2
|[D] Installed IPv4 forwarding rule for s2 -> h2
|[D] Installed IPv4 forwarding rule for s1 -> h2
|[I] IP rules update.
|[D] Updated local segment 1 for switch s2
|[I] Updated Bier-TE entries.
|
```

Figure 7.11.: Global controller command line interface.

The CLI is divided into two parts. On the left side, commands can be entered and information about the current status of the connected switches can be displayed. On the right side, information from the individual applications are displayed. Depending on whether the debug mode is activated, more or less detailed information is displayed.

Hosts

To enable hosts to actively subscribe and unsubscribe to multicast groups, a command line interface has been implemented for the hosts. The host CLI is similar to the CLI of the global controller and is also divided into two parts. Figure 7.12 shows the CLI of the hosts. The left side is used to enter commands, such as subscribing to a multicast group, or sending a packet to a multicast group. The right side displays additional information, such as when the host receives a multicast packet.



```
Type help for command overview. Quit with ctrl-c.
> i
   info

+-------------+--------------------------------+
|   Command   |              Info              |
+-------------+--------------------------------+
|    clear    |       Clears cli output        |
|    help     |         Show commands          |
|    info     | Get information about the host |
|    send     |   Send message to given addr   |
|  subscribe  |    Subscribe to given addr     |
| unsubscribe |  Unsubscribe from given addr   |
+-------------+--------------------------------+
```

Figure 7.12.: Host command line interface.

8. Conclusion

The purpose of this work was the design and implementation of scalable and resilient BIER and BIER-TE. To this end, the data plane was implemented in P4 and a distributed controller concept was developed to automatically configure the data plane. The data plane implementation supports many different protocols including Ethernet, IP, IGMP, BIER, BIER-TE and segment routing. Limitations of P4 were resolved using a local controller. The resulting prototype allows dynamic management of multicast groups, automated topology recognition and fast reaction to link and node errors. Resilience on the BIER layer is guaranteed by BIER-FRR and BIER-TE-FRR. To improve the scalability of BIER and BIER-TE, a domain concept based on spectral clustering was developed and segment routing including a FRR component was implemented as tunnel mechanism. In addition, the implementation supports IP tunnels and domain forwarding without explicit tunneling.

8.1. Future Work

The implementation presented in this thesis features a fully functional prototype for resilient and scalable BIER and BIER-TE including controllers. However, there are still some interesting aspects that could be covered in the future.

8.1.1. Evaluation on Real Hardware

The entire implementation was tested and verified using the bmv2 software switch. However, performance evaluations on software switches are not representative. For this reason a performance evaluation of BIER compared to classic IP multicast on real hardware would be of interest. In particular, the impact of BIER's iterative process on throughput and utilization would be relevant. Since P4 is in principle platform-independent and the implementation only uses elements of the core language, it should be possible to transfer the implementation created in this work to real hardware without too much effort.

8.1.2. Specialized Clustering for Subdomains

In this thesis spectral clustering was used to assign the clusters. A decisive criterion for spectral clustering are components of equal size. It would be interesting to investigate whether the scalability extension of BIER and BIER-TE could benefit from more specific clustering. One could consider to put the main focus of the clusters, for example, on the shortest paths.

8.1.3. Comprehensive Segment Routing Implementation

A separate header for segment routing was introduced in this thesis, which makes it possible to support the segment routing concept. Existing proposals use IPv6 or MPLS for this, which is accompanied by a large header overhead. For this reason, the segment routing implementation could be extended in future work to support all features of segment routing so that a complete standalone segment routing header could be presented.

A. Content of the Enclosed CD

The enclosed CD contains the source code developed in this work as well as a detailed code documentation. The structure is as follows.

The folder **Implementation** contains the following subfolders.

- **Implementation/P4-Implementation**

  This folder contains the P4 based data plane implementation, as well as the implementation of the host CLI. The *examples* folder contains the network topologies used for evaluation. A description of the usage of those topologies can be found inside the documentation. The folder *src* contains the P4 code of the individual components. *host_ cli* contains the implementation of the host CLI and *host_ scripts* additional scripts that can be used within the hosts.

  The *utils* folder contains the build script for the P4 implementation, i.e. the automatic process of creating the mininet topology, starting the P4 switches and the local controllers. This build script is based on the official p4lang tutorial at `https://github.com/p4lang/tutorials`.

- **Implementation/Local-Controller**

  This folder contains the Python implementation of the local controller. The local controller automatically starts with the P4 switch. The *topology.json* inside each specific topology example determines the local controller which is started.

- **Implementation/Controller-Implementation**

  This folder contains the Python implementation of the global controller. The global controller must be started separately and gets configured using a *json* file as described in section 7.10.2.

- **Implementation/Topology-Generator**

  This folder contains a simple topology generator that creates a random topology based on a so called *growing network* (GN). Links between nodes are added based on certain probabilities.

- **Implementation/Documentation**

  This folder contains the documentation of the P4-Implementation, Controller-Implementation and Local-Controller implementation. The documentation is based on sphinx [3], a documentation tool for various applications. To build the documentation, it is sufficient to run *make html* or *make latex* inside the **Implementation/Documentation** folder.

B. Environment Setup and Usage

B.1. Setup

The **Install** folder contains installation scripts for two use cases.

- **Docker**

  Install/Docker contains a Dockerfile to setup a docker container with all needed packages. The docker container is based on an Ubuntu 18.04 image. To create the container run *docker build -t <name> .* inside the *Install/Docker* folder.

- **Generic**

  Install/Generic contains setup scripts to install all dependencies for a linux distribution. The installation has been tested and verified using Ubuntu 18.04.2 LTS. Run *./setup.sh* inside the *Install/Generic* folder.

The recommended installation variant is Generic.

B.2. Usage

To instantiate a topology run *make run TOP=XXX* inside the P4-Implementation folder. *XXX* is the name of the topology, e.g. BIER/simple, 5Nodes, 10Nodes, BIER/rfc-example.

In another terminal, go into the Controller-Implementation folder and start the global controller using

./controller.py –config ../P4-Implementation/examples/XXX/config.json

, where *XXX* is the name of the topology. For further documentation see the documentation.pdf.

Bibliography

[1] grpc open-source universal rpc framework. `https://grpc.io`.

[2] nanomsg: a messaging protocol library. `https://nanomsg.org/`.

[3] Sphinx, a documentation tool. `http://www.sphinx-doc.org/en/master/`.

[4] A. Atlas and A. D. Zinin. Basic Specification for IP Fast Reroute: Loop-Free Alternates. RFC 5286, Sept. 2008.

[5] 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. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[6] B. Cain, D. S. E. Deering, B. Fenner, I. Kouvelas, and A. Thyagarajan. Internet Group Management Protocol, Version 3. RFC 3376, Oct. 2002.

[7] T. Eckert, G. Cauchie, W. Braun, and M. Menth. Traffic Engineering for Bit Index Explicit Replication BIER-TE. http://tools.ietf.org/html/draft-eckert-bier-te-arch, Nov. 2017.

[8] T. Eckert, G. Cauchie, W. Braun, and M. Menth. Protection Methods for BIER-TE. https://tools.ietf.org/html/draft-eckert-bier-te-frr-03, Mar. 2018.

[9] Ecma International. The json data interchange format. Standard ECMA-404, October 2013.

[10] B. Fenner. Internet Group Management Protocol, Version 2. RFC 2236, Nov. 1997.

[11] C. Filsfils, N. K. Nainar, C. Pignataro, J. C. Cardona, and P. François. The segment routing architecture. *2015 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, 2014.

[12] C. Filsfils, S. Previdi, L. Ginsberg, B. Decraene, S. Litkowski, and R. Shakir. Segment Routing Architecture. RFC 8402, July 2018.

[13] C. Filsfils, S. Previdi, J. Leddy, S. Matsushima, and daniel.voyer@bell.ca. IPv6 Segment Routing Header (SRH). Internet-Draft draft-ietf-6man-segment-routing-header-16, Internet Engineering Task Force, Feb. 2019. Work in Progress.

[14] B. A. Forouzan. *Data Communications and Networking*. McGraw-Hill, Inc., New York, NY, USA, 5 edition, 2013.

[15] P. Francois, C. Filsfils, A. Bashandy, S. Previdi, and B. Decraene. Segment Routing Fast Reroute. Internet-Draft draft-francois-sr-frr-00, Internet Engineering Task Force, July 2013. Work in Progress.

[16] A. Giorgetti, A. Sgambelluri, F. Paolucci, P. Castoldi, and F. Cugini. First demonstration of sdn-based bit index explicit replication (BIER) multicasting. In *EuCNC*, pages 1–6. IEEE, 2017.

[17] Z. Hu, D. Guo, J. Xie, and B. Ren. Multicast Routing with Uncertain Sources in Software-Defined Network. In *2016 IEEE/ACM 24th International Symposium on Quality of Service (IWQoS)*, pages 1–6, June 2016.

[18] L. H. Huang, H. J. Hung, C. C. Lin, and D. N. Yang. Scalable and Bandwidth-Efficient Multicast for Software-Defined Networks. pages 1890–1896, Dec 2014.

[19] S. Islam, N. Muslim, and J. W. Atwood. A survey on multicasting in software-defined networking. *IEEE Communications Surveys and Tutorials*, 20(1):355–387, 2018.

[20] D. Kotani, K. Suzuki, and H. Shimonishi. A Multicast Tree Management Method Supporting Fast Failure Recovery and Dynamic Group Membership Changes in OpenFlow Networks. *Journal of Information Processing*, 24(2):395–406, 2016.

[21] D. Kreutz, F. M. V. Ramos, P. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-Defined Networking: A Comprehensive Survey. *Proceedings of the IEEE*, 103(1):63, 2015.

[22] J. F. Kurose and K. W. Ross. *Computer Networking: A Top-Down Approach.* USA, 6th edition, 2013.

[23] Y.-D. Lin, Y.-C. Lai, H.-Y. Teng, C.-C. Liao, and Y.-C. Kao. Scalable Multicasting with Multiple Shared Trees in Software Defined Networking. *Journal of Network and Computer Applications*, 78:125–133, 2017.

[24] U. Luxburg. A tutorial on spectral clustering. *Statistics and Computing*, 17(4):395–416, Dec. 2007.

[25] U. Luxburg. Lecture notes in machine learning: Algorithms and theory, July 2018.

[26] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, Mar. 2008.

[27] M. Menth, D. Merling, and W. Braun. P4-Based Impelmentation of BIER-TE and BIER-TE-FRR for Scalable and Resilient Multicast with Traffic Engineering Capabilities. Dec. 2018. Submitted to TNSM.

[28] D. Merling and M. Menth. BIER Fast Reroute. Internet-Draft draft-merling-bier-frr-00, Internet Engineering Task Force, Mar. 2019. Work in Progress.

[29] D. Merling, M. Menth, N. Warnke, and T. Eckert. An Overview of Bit Index Explicit Replication (BIER). Mar. 2018.

[30] ONF. Software-defined networking: The new norm for networks. Technical report, Open Networking Foundation, April 2012.

[31] P4 Language Consortium. P4-16 Language Specification. `https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.pdf`, 2017. Accessed: 2019-03-01.

[32] p4lang. p4lang/behavioral-model. `https://github.com/p4lang/behavioral-model/blob/master/docs/simple_switch.md`. Accessed: 2019-03-05.

[33] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[34] T. Pfeiffenberger, J. L. Du, P. B. Arruda, and A. Anzaloni. Reliable and Flexible Communications for Power Systems: Fault-tolerant Multicast with SDN/OpenFlow. In *2015 7th International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–6, July 2015.

[35] J. Rückert, J. Blendin, R. Hark, and D. Hausheer. Dynsdm: Dynamic and flexible software-defined multicast for isp environments. In *IEEE International Conference on Network and Service Management (CNSM)*, November 2015.

[36] J. Rueckert, J. Blendin, R. Hark, and D. Hausheer. Flexible, Efficient, and Scalable Software-Defined Over-the-Top Multicast for ISP Environments With DynSdm. 13(4):754–767, Dec 2016.

[37] D. A. Schult. Exploring network structure, dynamics, and function using networkx. In *In Proceedings of the 7th Python in Science Conference (SciPy*, pages 11–15, 2008.

[38] S. Shalev-Shwartz and S. Ben-David. *Understanding Machine Learning: From Theory to Algorithms.* Cambridge University Press, New York, NY, USA, 2014.

[39] The P4.org API Working Group. P4 Runtime Specification. `https://s3-us-west-2.amazonaws.com/p4runtime/docs/v1.0.0/P4Runtime-Spec.html`, 2019. Accessed: 2019-03-03.

[40] K. Varda. Protocol buffers: Google's data interchange format. Technical report, Google, jun 2008.

[41] I. Wijnands, E. C. Rosen, A. Dolganow, T. Przygienda, and S. Aldrin. Multicast Using Bit Index Explicit Replication (BIER). RFC 8279, Nov. 2017.

[42] I. Wijnands, E. C. Rosen, A. Dolganow, J. Tantsura, S. Aldrin, and I. Meilik. Encapsulation for Bit Index Explicit Replication (BIER) in MPLS and Non-MPLS Networks. RFC 8296, Jan. 2018.

List of Figures

List of Tables

List of Listings