TÉCNICO
LISBOA

C++ implementation of OSPFv2 extension for the support of multi-area networks with arbitrary topologies

Xavier Carneiro Gomes

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Rui Jorge Morais Tomaz Valadas

Examination Committee

Chairperson: Prof. Maria Luísa Torres Ribeiro Marques da Silva Coheur
Supervisor: Prof. Rui Jorge Morais Tomaz Valadas
Member of the Committee: Prof. João Miguel Duarte Ascenso

November 2022

Acknowledgments

First, I would also like to thank my dissertation supervisor professor Rui Valadas for his valuable guidance and support throughout the development of our work and writing of this dissertation. This dissertation would not have been possible without his help.
I would also like to thank my family and friends for their constant encouragement and support throughout the years, and for keeping pushing me forward.
Last but not least, I would like to thank my girlfriend, for her company, love and support.

To each and every one, thank you very much.

Abstract

OSPF is a popular intra-domain Internet routing protocol, and to allow for better scalability, it implements multi-area routing. However, this multi-area routing is based on a Distance Vector approach, which introduces several limitations: (i) the topology is limited to a two-level hierarchy, with a single area at the upper level, and (ii) globally optimal routing is not ensured. In this MSc Dissertation, we propose an extension to address these issues, by taking instead a Link State approach to multi-area routing in OSPF. This extension is meant for OSPFv2 and introduces new types of Link State Advertisements, used by the Area Border Routers (ABRs) to build and maintain an overlay and its respective graph, representing a logical network over the physical one, used for multi-area routing calculations. Due to these additions being exclusively used by ABRs, this extension is mostly transparent to all other types of routers. We implemented this extension using the C++ programming language, and based our work on an existing OSPFv2 implementation, written using this same language. We then tested the extended implementation, for both its correction and network convergence times. The results show that the extension operates as expected, without having a significant impact in the network convergence times.

This MSc dissertation is supported by Instituto de Telecomunicações.

Keywords

OSPF; Internet; Link State Protocols; Routing.

Resumo

O OSPF é um protocolo intradomínio de encaminhamento na Internet bastante popular e, para permitir uma melhor escalabilidade, implementa um encaminhamento multi-área. No entanto, este encaminhamento é baseado numa abordagem do tipo Distance Vector, que introduz várias restrições: (i) a topologia fica limitada a uma hierarquia de dois níveis, com uma única área no nível superior, e (ii) as melhores opções de encaminhamento não são garantidas. Nesta dissertação, propomos uma extensão para ultrapassar estes problemas, que passa por adotar uma abordagem do tipo Link State para o encaminhamento multi-área no OSPF. Esta extensão destina-se ao OSPFv2 e introduz novos tipos de Link State Advertisements (LSAs), usados pelos Area Border Routers (ABRs) para construir e manter um overlay e o grafo da sua topologia, que representa uma rede lógica sobre a rede física, usada para os cálculos efetuados para o encaminhamento multi-área. Devido a estas adições serem usadas exclusivamente por ABRs, esta extensão é praticamente transparente a todos os outros tipos de router. Implementámos esta extensão usando como linguagem de programação o C++ e baseámos o nosso trabalho numa implementação existente do OSPFv2, desenvolvida com essa mesma linguagem. Testámos também a implementação estendida, tanto para a sua correção como para os tempos convergência da rede. Os resultados mostram que a extensão funciona como esperado, sem ter um impacto significativo nos tempos de convergência da rede.

Esta dissertação foi apoiada pelo Instituto de Telecomunicações.

Palavras Chave

OSPF; Internet; Protocolos Link State; Encaminhamento.

Contents

List of Figures

Acronyms

ABR	Area Border Router
API	Application Programming Interface
AS	Autonomous System
ASBR	Autonomous System Border Router
BDR	Backup Designated Router
BGP	Border Gateway Protocol
DR	Designated Router
DV	Distance Vector
DVR	Distance Vector Routing
EGP	Exterior Gateway Protocol
IGP	Interior Gateway Protocol
IP	Internet Protocol
IS-IS	Intermediate System to Intermediate System
ISM	Interface State Machine
LSA	Link State Advertisement
LSDB	Link State Database
LSR	Link State Routing
NBMA	Non-Broadcast Multiple Access
NSM	Neighbor State Machine
OOP	Object-Oriented Programming
OS	Operating System
OSPF	Open Shortest Path First

RFC	Request for Comments
RID	Router ID
RIP	Routing Information Protocol
VL	Virtual Link
VM	Virtual Machine

1 Introduction

Contents

1.1 Motivation and Objectives

Communication over the Internet is crucial for information to be transmitted between devices. To support this communication, several routing protocols are used. There are two main groups of protocols, Interior Gateway Protocols (IGPs) and Exterior Gateway Protocols (EGPs). IGPs are further split into two subgroups of protocols, Distance Vector Routing (DVR) Protocols and Link State Routing (LSR) Protocols.

DVR protocols, such as Routing Information Protocol (RIP), rely on the transmission of messages (distance-vectors), containing (destination, cost) pairs to disseminate the routing information across the network. In this approach, routers know nothing of the network topology besides who they and their neighbors are. This approach can present several problems and is very limited when it comes to scaling of network.

LSR protocols, such as Open Shortest Path First (OSPF) or Intermediate System to Intermediate System (IS-IS), provide each router with a complete view of the network. This approach prevents the issues and limitations of DVR protocols. One very known and used protocol in Internet routing is OSPF, presenting two versions, OSPFv2 (for IPv4) and OSPFv3 (for IPv6). To provide better scalability, OSPF networks can be divided into areas, to reduce the amount of data that the routers must store, and improve its scalability. This raises some problems, as currently, OSPF's multi-area routing uses a DVR approach. OSPF's multi-area routing approach might prevent OSPF networks from achieving globally optimal routing, and it limits the network topologies, as these areas are limited to a two-level hierarchy, with a single upper-level area (the backbone).

For this MSc Dissertation, we proposed and implemented an extension to OSPFv2 to address the existing issues present in OSPF's multi-area routing, ensuring globally optimal routing, allowing the use of arbitrary multi-area topologies, and also addressing some of the current restrictions that can be found in current implementations. This solution was implemented using the C++ language and based on an already existing OSPFv2 implementation, by John T. Moy, as specified in RFC 2328 [2] and thoroughly described in its majority in [3]. Since we will be using C++, we also aimed to maintain the overall network convergence times from the base version of this implementation. This extension has been previously developed for OSPFv3, and this work can be found in [5].

1.2 Contributions

We proposed and developed an extension for OSPFv2, using the C++ programming language, allowing the protocol to configure arbitrary multi-area topologies, while overcoming some restrictions present in current implementations. We used John T. Moy's OSPFv2 implementation [3] as the base for our development. We also tested and measured convergence times for this extended implementation. From the results obtained we confirmed its correct operation and potential improvement over current OSPF implementations. The convergence times noted were not significant when faced with the restrictions and limitations that were overcome.

1.3 Structure of the document

The next chapters of this Dissertation are organized as follows. In Chapter 2 we introduce two approaches to routing, DVR and LSR, and we provide more detailed insights over a known LSR protocol, OSPF. In Chapter 3 we introduce and describe some key aspects of the implementation we will use as the base for our work. In chapter 4, we give an overview of the solution proposed and developed for this work. In chapter 5, we describe the changes and additions we implemented over the base implementation to accommodate our extension. In chapter 6 we describe the tests applied over our solution, presenting and discussing the obtained results. Finally we make our final conclusions in chapter 7.

2
Related Work

Contents

2.1 Computer Networking

In routing, we can find three main types of network elements, the end devices, or hosts, the switching equipment, and the communication media. These elements connect to form what we know as a computer network, enabling routing and data transmission. In the end devices, such as laptops or smartphones, are where the information originates and where it is delivered. Switching equipment is what performs the routing function, keeping routing tables describing how each destination can be reached. Finally, the communication media provides a means of connection between switching equipment and/or end devices, for instance via optical fiber or some form of wireless connection.

Alongside these elements, we also have routing protocols, to compute the best routes between the network elements, both switching equipment and end devices, building routing tables in the switching devices, and describing the routes to take for each destination. These routing protocols run on the switching equipment and operate by exchanging control messages between themselves and neighboring devices, excluding the hosts. Firstly, we can divide the routing protocols into two categories, *inter-domain* (EGPs), and *intra-domain* (IGPs). As can be inferred from the names, these types of protocols are meant to route information between several domains and within one, respectively. This report focuses on intra-domain routing protocols and inter-domain protocols will not be addressed.

2.2 Intra-domain Routing

There are two main different approaches to intra-domain routing, i.e., routing within a single domain. These two categories are DVR and LSR.

2.2.1 Distance Vector Routing Protocols

DVR protocols aim to implement a distributed and asynchronous version of the *Bellman-Ford algorithm*, where each router sends to its neighbors its calculated cost to every network destination. Each router determines these costs using two components, the cost advertised by their neighbors to the destinations and the cost associated with the outgoing interface connecting the router to its neighbor. After computing these costs, the routers will select the next-hop router providing the shortest path cost to each destination, updating their routing tables and advertising its computed costs to their neighbors, via a broadcasted message, the Distance Vector (DV), carrying *(destination, cost)* pairs. If a failure is detected by a router regarding one of its neighbors, the cost the router attributes to it will be set to infinity (∞).

This is the way the RIP [6] operates, in a general way. For each destination, it will store only the information received from the best neighbor(s), the one(s) who can provide the shortest path cost for

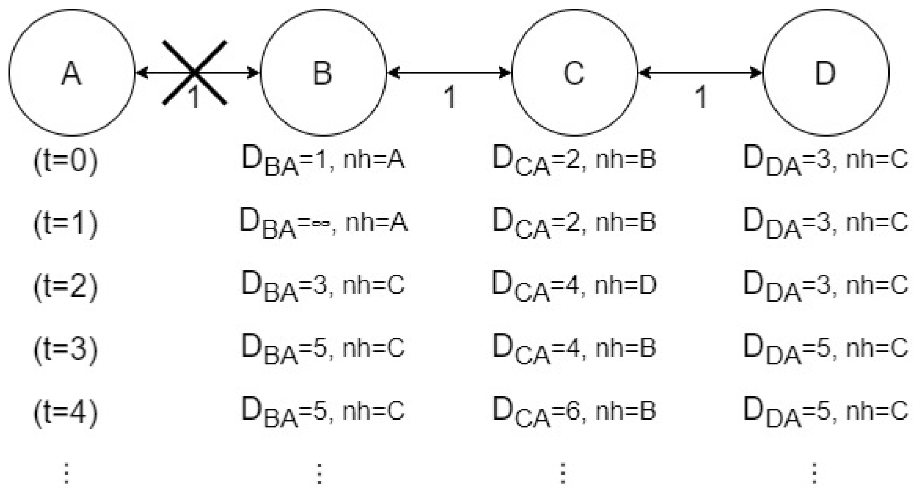


Figure 2.1: Count-to-infinity problem
Adapted from [1]

that destination. If receiving a better cost to one of its already routed destinations, a router will update its own cost to that same destination. Furthermore, if the neighbor that provided that new cost was not the current next-hop router, it will be set in the router to be. In RIP, the cost to a destination usually represents the number of hops needed to reach it. In RIP, DV are periodically transmitted using a 30 second interval.

One of the biggest known issue with DVR approaches is the Count-to-infinity problem. To illustrate this problem, consider figure 2.1, where D_{iA} represents the distance (in hops) from router *i* to router A and *nh* the next hop router in the direction of A. Let's also consider the network to be stable at t=0. By then, B registers a cost of 1 hop to reach A, C a cost of 2, and D a cost of 3, with the expected next hop routers, A for B, B for C, and C for D, as they should. Now let's assume that right after t=0 the connection between A and B fails. Realizing the failure, at t=1, B will send a cost of ∞ to C, while simultaneously C sends its stored distance of 2 to B and D. In the next iteration, at t=2, B will advertise a cost of 3 (cost advertised by C plus B's cost to it), and C will increase its cost to 4 since it now believes it must take the route offered by D to reach A. In the next iteration (t=3), because C was advertising a cost of 4, both B and D will update their costs to 5, as both of them believe C to be their next-hop router to A. This cycle then goes on indefinitely if nothing is done regarding the issue.

There are several ways to mitigate this problem. For example, in RIP infinity is "redefined" to 16 hops. A destination with a cost of 16 will be considered unreachable and will therefore be eliminated from the routers' routing tables. This introduces serious topological restrictions, as RIP cannot support paths with more than 15 hops. Also, if an error occurs, the time interval until the network converges to a stable state could be significant. Other approaches are possible, such as *triggered updates* or *split horizon*, allowing for faster convergence times after a failure occurs.

2.2.2 Link State Routing Protocols

Contrarily to DVR, where each router only stores its information regarding the network destinations, in LSR protocols, each router maintains a complete view over the network, building their routing tables from this routing information. Each router maintains a Link State Database (LSDB), in which they store the routing information needed to build their routing tables, updating it as the protocol is run and the network changes. There are three types of information to consider:

- **Topological information**, describing the routers, the links that connect them, and the costs assigned to their interfaces.
- **Addressing information**, describing the routable destinations and their connection to routers and their links. This type of information can be further distinguished as *domain-internal* or *domain-external*.
- **Link information**, describing the addresses used to transport packets across neighbors in a link. This information is local to each router and is not maintained in the LSDB.

Also, the LSDBs at all routers must be kept up to date, meaning there should exist synchronization among them. To that end, several synchronization mechanisms are used. These are described from Sections 2.3.6 to 2.3.11.

With an increase of a network's dimension, the resulting LSDB size will also increase. This enlargement of the LSDB size might not be supported by all routers. To cope with this limitation, LSR protocols structure the networks in several smaller areas, where each area has its LSDB. Routers only need to store the LSDB of the area they belong to. To communicate between areas, a special type of router is used, the Area Border Router (ABR). These routers are placed in the border between at least two areas, belonging to all the ones they directly attach to and storing all the associated LSDBs. For communication to be possible across different areas, information of the destinations inside each area is advertised to other areas, through the ABRs. To exchange this kind of information, ABRs run an inter-area routing protocol, besides the intra-area one, to gather the information that needs to be injected within the areas they directly attach to.

These are some general descriptions of what exists and is implemented in LSR protocols. Two main protocols, very similar to one another, are the OSPF and the IS-IS. In the next sections, we will go deeper into how OSPF works, and the features it has implemented. As this work focuses only on the IPv4 version of OSPF, only this version of the protocol will be discussed in detail. IS-IS will not be discussed, but its features are described and compared to those of OSPF in [1].

2.3 The Open Shortest Path First protocol

OSPF is currently one of the standards of LSR, with two versions available, **OSPFv2**, for IPv4, defined in Request for Comments (RFC) 2328 [2], and **OSPFv3**, for IPv6, defined in RFC 5340 [7]. OSPF implements the main aspects described in the previous section, with some limitations to its multi-area routing, and precisely the motivator for this work.

2.3.1 The structure of the OSPF routing domain

As we stated, OSPF's routing domains are usually structured into more than one area. But this separation into areas has its limitations, as it is restricted to a two-level hierarchy, with one area on the upper level, called the *backbone*, and all the others on the lower level, where all inter-area communication needs to cross the *backbone* area. Multi-area networks are then designated as *hierarchical networks*. Also, a distance vector approach is taken for multi-area routing, and thus the issues verified for DVR protocols are also present.

Within an OSPF routing domain, routers can be of three types, either an Autonomous System Border Router (ASBR), an ABR, or an *area-internal router*. ASBRs establish connections to external routing domains, via an EGP, such as the Border Gateway Protocol (BGP). ABRs connect the *backbone* area to lower-level areas, and exchange information regarding the areas they connect to with neighboring ABRs. Area-internal routers simply represent the routers internal to each area.

A simple example of the structure of the OSPF routing domain is presented in figure 2.2, where R1 represents an ASBR, R3, R4, and R8 represent ABRs and the remaining routers represent area-internal routers.

2.3.2 Link types and element identifiers

There are five types of links, *point-to-point*, *point-to-multipoint*, Non-Broadcast Multiple Access (NBMA), *broadcast*, and *virtual* links. For this work, we will mainly focus on *point-to-point* and *broadcast* links, also designated as *shared* links. Point-to-point links are the most common, directly connecting two routers. Shared links can be seen as smaller networks, with the capability of interconnecting several devices, such as hosts and routers. Shared links can be divided into two types. If only one router attaches to the link, it is a *stub shared link*. If there are two or more routers attached, it is a *transit shared link*. This distinction is also illustrated in figure 2.2. For each transit shared link in the network, there must be an elected Designated Router (DR), and if possible, a Backup Designated Router (BDR). We will designate the transit shared links simply as shared links.

In the network, both routers and links need to be properly identified. Routers are identified by the Router ID (RID), unique within the routing domain it belongs to. Unlike in OSPFv3, in OSPFv2, the

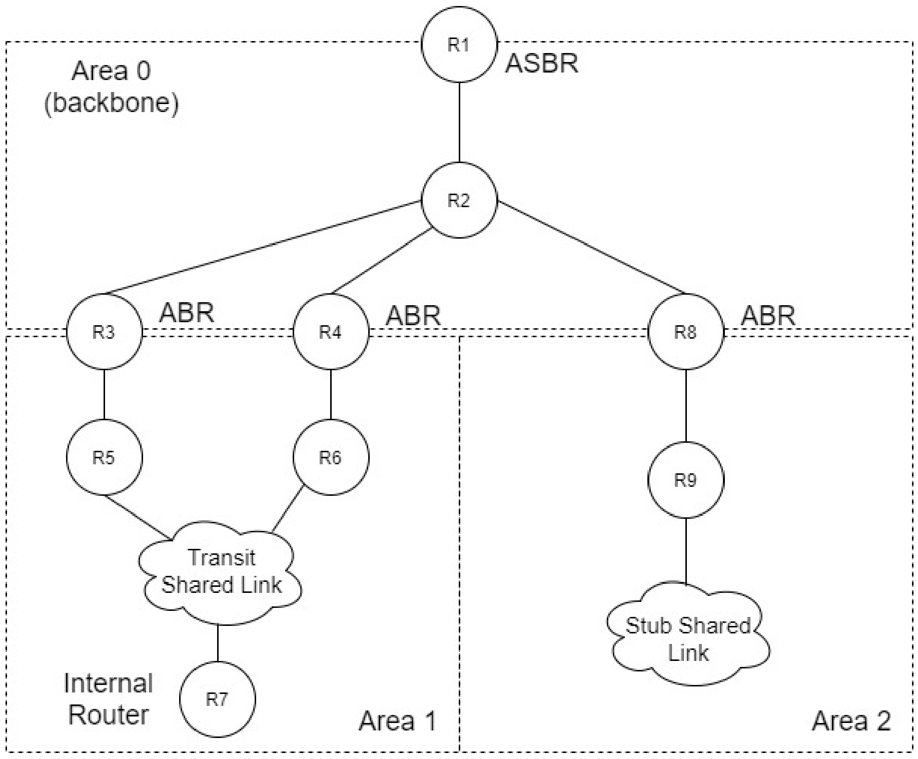


Figure 2.2: Structure of the OSPF routing domain

configuration of this RID is not mandatory, and if it is not done, the address of the interface with the highest address in the router will be used as RID. Links are identified through the *Hello protocol* (see Section 2.3.7). *Point-to-point* links are represented through the *RID of the router's neighbor* on that link, in combination with the *outgoing interface* attached to that same link, to distinguish among parallel links. *Shared* links, in OSPFv2, are identified by the address of the *DR interface* (the DR's interface connecting to the link, not the DR's RID). In OSPFv3, the shared links are identified through a combination of the *DR's RID* with a *local tag* generated by them.

2.3.3 The LSDB elements

In OSPF, the LSDB elements are called Link State Advertisements (LSAs). LSAs carry the addressing and topological information needed to correctly build the network view at each router. LSAs can be flooded independently, but in OSPFv2 some of them do not separate topological and addressing information. Figure 2.3 shows a summary of the structure of the OSPFv2 LSAs. In this figure, the repeatable fields/groups of fields are indicated with darker rectangles.

All LSAs share a common header, with only slight differences from OSPFv2 to OSPFv3. The LSA is uniquely identified by three of the fields in the header, the *Advertising Router*, the *LS type*, and the *Link State ID*. Also, three other fields define the LSA's freshness, which can be useful to resolve some

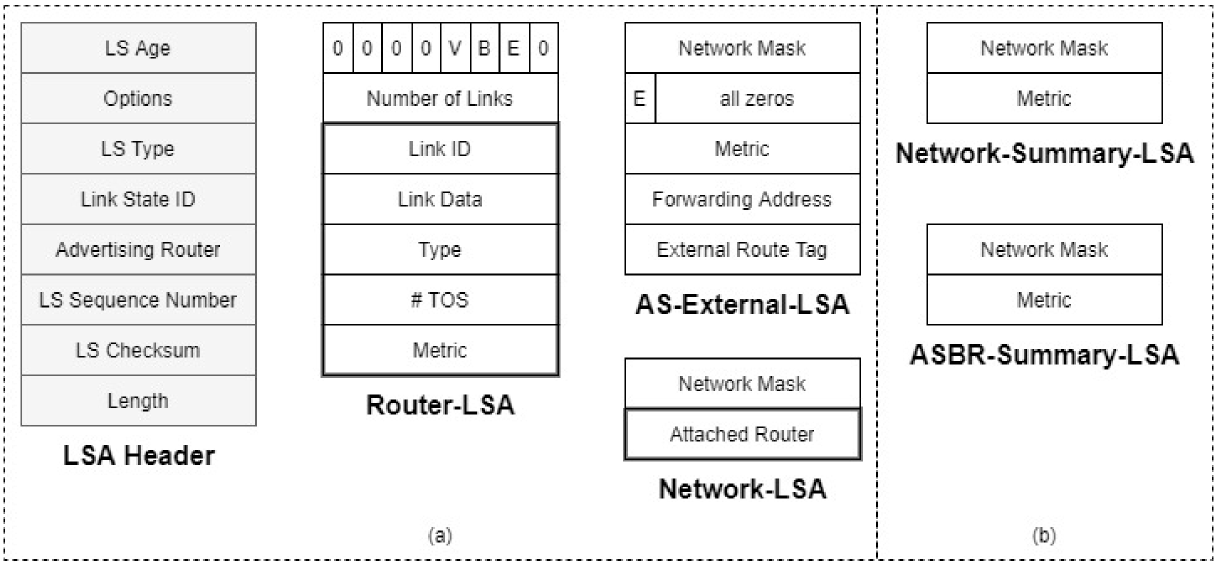


Figure 2.3: Structure of the OSPFv2 LSAs. (a) single-area routing LSAs and (b) additions for multi-area routing Adapted from [1]

conflicts, as we will describe in upcoming sections. These fields are the *LS Sequence Number*, the *LS Age*, and the *LS Checksum*.

The LSA types we will discuss are the ones present in OSPFv2, namely the *Router-LSA*, *Network-LSA*, and *AS-External-LSA*, for single-area routing, and *Network-Summary-LSA* and *ASBR-Summary-LSA* for multi-area routing, in Section 2.3.12. Some differences exist between these LSA versions in OSPFv2 and OSPFv3, besides the addition of two more LSA types in OSPFv3, the *Intra-Area-Prefix-LSA*, and *Link-LSA*, providing a better separation of addressing and topological information by the LSAs. These differences can be seen in detail in chapter 5.7 of [1].

2.3.4 Router-LSAs and Network-LSAs

Each router advertises one **Router-LSA** per area they are attached to. Router-LSAs are used to describe the *router*, its *outgoing interfaces*, and the *addresses assigned to either itself or its interfaces*. The originating router is identified through the *Advertising Router* field of the LSA header. Also, the Router-LSA is uniquely identified within the LSDB through a combination of the *Advertising Router* with the *LS Type*.

The Router-LSA contains a *Number of Links* field, indicating the number of link descriptions carried in that LSA. Also, a repeatable set of fields is used to describe the router interfaces, where each set represents a *link description*. Each link description is characterized by four fields, *Type* (type of link), *Link ID*, *Link Data*, and *Metric*. Depending on the type of element being described, the fields may contain different kinds of information.

The **Network-LSA**, contains a *Network Mask* field, followed by a repeatable number of *Attached Router* fields. Network-LSAs are used to describe the *transit shared links* (see Section 2.3.2) of the network. The *Network Mask* field contains the mask of the prefix advertised by the link, and all the attached routers are identified through their RIDs in the Attached Router fields. The link is identified by its *IPv4 address*, in the *Link State ID* field of the header, along with its network mask. For a shared link, its Network-LSA is originated by the link's elected DR.

With the information carried by the Router-LSAs and Network-LSAs, it is possible to describe the whole intra-domain area topology, including routers, point-to-point links, and shared links. Also, all the prefixes from within the domain become known, namely, the prefixes assigned to the routers themselves, to point-to-point links, to transit shared links, and to stub shared links. This is true for OSPFv2, but in OSPFv3, addressing information is instead described by Intra-Area-Prefix-LSAs. The topological information is described the same way in both versions of the protocol, with small changes in the LSAs structures and link descriptions.

2.3.5 The AS-External-LSA

In OSPF, ASBRs, inject *domain-external* information into the domain they belong to. Despite their name, these routers can serve as *domain border routers* and not only as *Autonomous System (AS) border routers*. To advertise domain-external prefixes into their domain, ASBRs use *AS-External-LSAs*, generated by themselves.

In OSPFv2, the external address is described by its *prefix address*, carried in the *Link State ID* field, and the *prefix network mask*, carried in the *Network Mask* field. Each AS-External-LSA can only carry information about one external prefix, being a new one originated for each prefix. ASBRs can also use these LSAs to inject *default routes* into their AS (by advertising the 0.0.0.0/0 address).

AS-External-LSAs carry a special *E-bit* flag, defining how routers receiving the LSA should calculate their costs to the external prefix. The *Metric* field carries the external cost to the advertised prefix, and the E-bit can either be set to 0 or 1. With the E-bit set to 0, the routers consider the route to the external prefix to be an *E1 external route*, with its cost being calculated as the sum of the external path cost of Metric with the internal path cost from the router to the ASBR. Otherwise (E-bit set to 1), the route is considered an *E2 external route*, where the cost is considered to simply be the external path cost advertised. This bit can be used to influence the choice of ASBR to external routes by the routers.

2.3.6 Control Packets

We now leave the single-area networks' part of the LSDB structure to begin with the *synchronization mechanisms* present in OSPF. Before addressing the mechanisms, we need to understand the *control* *packets* used by OSPF. These are almost identical for OSPFv2 and OSPFv3, with slight differences in the headers and in one of the packets' structures. The OSPF packets are encapsulated in Internet Protocol (IP) packets, using IP(v4/v6) 89, relative to OSPF.

There are five types of control packets:

- *HELLO* packets, supporting the *Hello protocol*;
- *LS UPDATE* packets, used to *disseminate LSAs* across the network, resulting in possible updates to the LSDBs of the routers;
- *LS ACKNOWLEDGMENT* (**LS ACK**) packets, used to *acknowledge the reception* of an LSA at the router;
- *LS REQUEST* (**LS REQ**) packets, used to *request one or more specific LSAs* from another router;
- *DB DESCRIPTION* (**DB DESC**) packets, used by a router to *advertise a summary of its LSDB* at the current state.

Just like with the LSAs, all the control packets share the same structure for the packet header. The packet type is identified by the *Type* field of the packet header. A summary of the structure of these control packets (the OSPFv2 version) can be seen in figure 2.4.

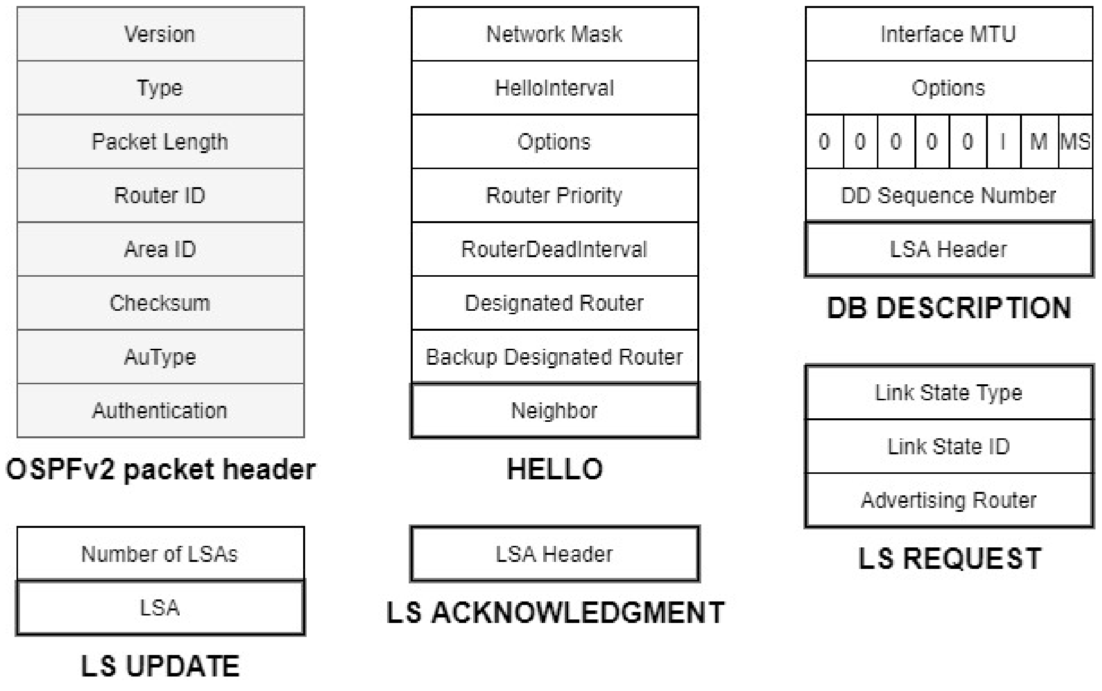


Figure 2.4: OSPFv2 control packets structure
Adapted from [1]

There are two ways of transmitting packets in shared links, either *unicast* the packets or using one of the OSPF *multicast addresses*. These addresses are the **AllSPFRouters**, targeting all the routers in a link, or the **AllDRouters**, targeting only the DR and the BDR (if there is one) of the shared link.

Finally, some rules should be met for arriving packets to be accepted at the routers. In OSPFv2, there are rules for accepting IPv4 packets at OSPF processes and for accepting OSPF-specific packets. This set of rules can be found in Section 6.1.4 of [1] and are designated as *interface acceptance rules*.

2.3.7 Hello protocol

One of the synchronization mechanisms used in OSPF is the *Hello protocol*, used by the routers to connect with their neighboring routers, as well as to maintain those connections, allowing for the detection of possible failures in neighbors.

The Hello protocol uses **HELLO** packets. These packets include the RID of the sending router (RID field in the header) and the RIDs of its neighbors (repeatable *Neighbor* field). As each interface in OSPF needs to be assigned to one and only one area, this information is also carried in the packets (*Area ID* in the header), to determine if an adjacency between routers is possible (the interfaces must belong to the same area). There is a slight difference between the OSPFv2 and OSPFv3 HELLO packets, namely the replacement of the *Network Mask* field (in OSPFv2) with the *Interface ID* (in OSPFv3).

Routers periodically broadcast HELLO packets to their neighbors. This period value is indicated in the *HelloInterval* field, with a default value of 10 seconds. The relationships are maintained by verifying that neighbors keep sending packets. If a neighbor stops sending HELLO packets, it is considered dead if a time interval, advertised in the *RouterDeadInterval* (default value 40 seconds) field, is reached without receiving a packet from the neighbor.

The Hello protocol determines if two neighbors can become adjacent, as they need to be for their connection to be part of the network map. For two neighbors to be adjacent, the HELLO packets exchanged between them must verify the *interface acceptance rules*, referred to in the previous section, and the communication between the neighbors must be *bidirectional*. A router determines this is true if it finds its RID in the *HELLO packets* sent by the neighbor.

The Hello protocol works according to a *state machine*, the Neighbor State Machine (NSM). Each router interface has its instance of the state machine. Figure 2.5 shows a summary of the NSM. The NSM is fully described in Chapter 10 of [2]. This state machine is divided as the figure suggests. The Hello protocol uses the fraction represented in figure 2.5.a, which is described in both Chapter 10.3 of [2] and in Section 6.2.4 of [1].

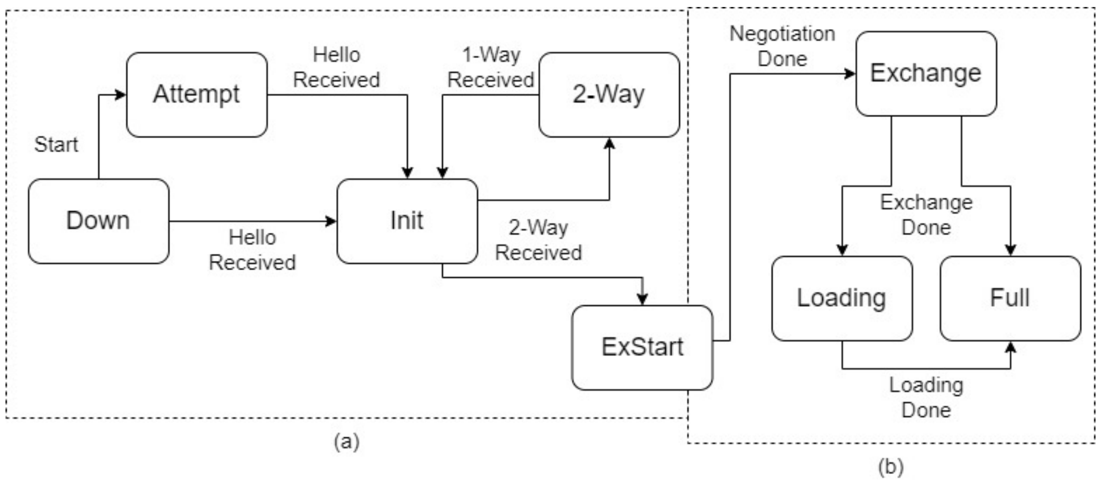


Figure 2.5: The Neighbor State Machine (a) as used by the Hello protocol (b) and by the Initial LSDB synchronization
Adapted from [2]

2.3.8 Designated Router and Backup Designated Router election

The Hello protocol is also used to elect a shared link's DR and BDR. The BDR replaces the DR in case of failure. As the Hello protocol supports this election process, HELLO packets carry, besides the *Neighbor* field(s):

- The *Designated Router* field, carrying the link's DR RID;
- The *Backup Designated Router* field, carrying the link's BDR RID;
- The *Router Priority* field, carrying the advertising router's configured priority value.

The DR and BDR are determined by the sending router's local election process outcome. The identification of the shared links is addressed in Section 2.3.2.

The election process is part of the Interface State Machine (ISM). Each router interface runs its instance of the state machine. The ISM is described in Section 9 of [2]. Figure 2.6 shows a summary of the ISM.

The ISM is described both in Chapter 9.3 of [2] and in Section 6.3.2.1 of [1]. The interfaces start in the **Down** state when switched off. For shared links, they move into the **Waiting** state when switched on. The interface leaves this state to run the election algorithm when it either establishes a bidirectional connection with an interface claiming to be the DR or BDR of the link or when a *Wait Timer* (set to 40 seconds by default) expires. After running the algorithm, the interface will end up in one of three states:

- **DR**, if the interface is the link's DR;
- **Backup**, if the interface is the link's BDR;

- **DR Other**, if the interface is neither the link's DR or BDR.

When an interface is elected DR or BDR, it won't abandon that role until it leaves the link.

The election process must be run again when a change occurs in the link. This change can be one of four:

- Attaching a new interface to the link;
- An interface abandoning the link;
- A neighbor starting to declare itself as DR or BDR;
- A neighbor changing its Router Priority.

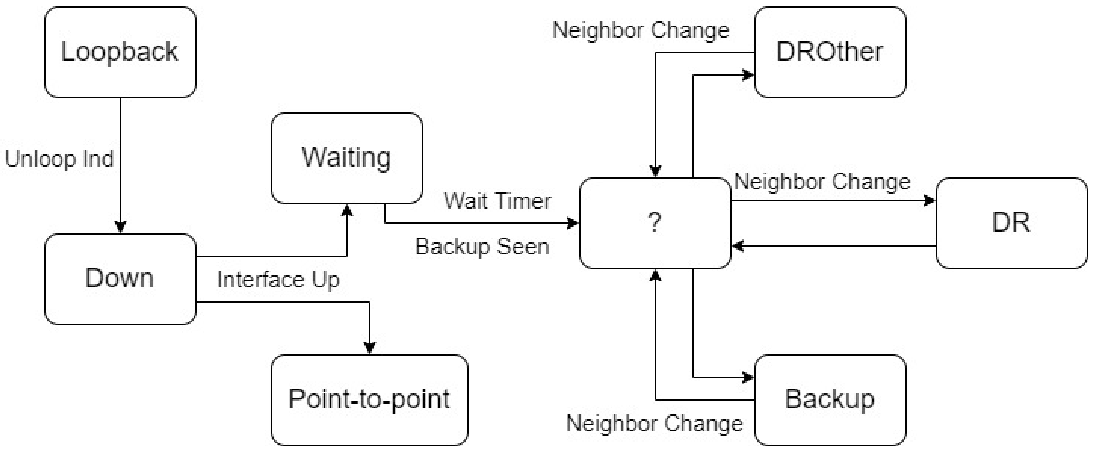


Figure 2.6: The Interface State Machine
Adapted from [2]

The election algorithm is run locally for each interface attaching to a shared link, and it favours the already elected DRs and BDRs. The election algorithm is fully described throughout Section 6.3.2 of [1].

When a new DR is elected, it must update the shared link identifier, and originate a new Network-LSA to replace the outdated one describing that shared link. Routers attached to the link must also update their Router-LSAs due to the same change in the link ID.

2.3.9 Flooding Procedure

OSPF uses a *controlled and reliable flooding procedure* to disseminate information, in the form of LSAs, across the network. When an LSA is originated at a router, it transmits that LSA on all of its interfaces. If a router receives a new or fresher instance of an existing LSA at an interface, it will, in turn, retransmit that LSA on all of its interfaces, excluding the one where it was received, and install it in its LSDB. Otherwise, the LSA is simply discarded. Whether the router accepts an incoming LSA or not, the receiving router must acknowledge its reception back to the sender.

The LSAs and their acknowledgments are carried in **LS UPDATE** and **LS ACK** packets, respectively. Both these packets can carry information regarding multiple LSAs. While this is true, each LSA needs an individual acknowledgment, and not for the LS UPDATE packet it came in. Every LSA that a router transmits is placed in a list, the *Link state retransmission list*, keeping all the LSAs whose reception is yet to be acknowledged, and retransmitting them according to a timer, the *RxmtInterval* (default time is 5 seconds), until their reception is acknowledged.

To flood an LSA into a shared link, a router will first send it to the link's DR, by using the **AllDRouters** multicast address. The DR will retransmit the LSA to all the other routers, using the **AllSPFRouters** multicast address. Then all the routers must acknowledge the reception of the LSA to the DR, again using the **AllDRouters** address. If the DR fails, the BDR will retransmit the packets. If retransmission of LSAs is needed, it will be done individually (unicasted) for each router that didn't acknowledge the reception in time.

There are optimizations to minimize the number of LS ACK transmissions, namely the *delayed acknowledgment* and the *implicit acknowledgment* mechanisms.

2.3.10 Updating procedure

The arrival of routing packets at routers is usually associated with either the *insertion*, *update*, or *deletion* of LSAs at the LSDB. These actions may or may not be executed, depending on the packets' *freshness* and on a defined *validity period*. Furthermore, the action to take might vary depending on whether the router is the *LSA's originating router*. As was described in Section 2.3.3, the LSA header contains three fields used to determine the LSA's freshness, the *LS Sequence Number* (**LS SN**), the *LS Checksum*, and the *LS Age*. When originating a new instance of an existing LSA, the router will increment that LSA's LS SN by one and reset its age.

A set of rules used to determine which instance of an LSA is fresher can be found in detail in Section 6.5.2 of [1].

For *deleting information* from the LSDB, coming in the form of LSAs, OSPF the **premature aging** mechanism. When a router seeks to delete one of its *originated LSAs* from the LSDB, it will update that LSA's *Age* field to the *MaxAge* value (3600 seconds/1 hour) and *flood* it. When receiving the LSA, a router will compare it to its stored instance, if there is one. Given the set of rules we referred to, unless the stored instance has a higher LS SN (or equal LS SN and higher LS Checksum) nor has it reached its maximum age, the received LSA instance will be considered fresher than its counterpart and will replace the old instance in the LSDB. Because this new LSA has reached its maximum age value, it will then be deleted from the LSDB.

When receiving an LSA, different actions can be carried out, depending on the freshness of the received and stored LSAs and on whether the received LSA was originated by the router receiving it (self-originated). These actions are described and can be found in Section 6.5.4 of [1].

LSAs must be refreshed before their age reaches their maximum value. For that, the LSA's originating router periodically generates a new instance of the LSA with an increment to the **LS SN** field. By default, this is done every 30 minutes (*LSRefreshTime*). If an LSA timer expires, reaching the maximum age value, the non-originating routers will remove the LSA from their LSDBs, while if it happens to its originating router, a delete indication is issued for that LSA.

2.3.11 Initial LSDB synchronization

When a router joins a new link, it needs to synchronize its LSDB with all *point-to-point* neighbors, or with the *DR and BDR* in the case of shared links. We will designate this process as the *Initial LSDB Synchronization*. This process uses almost all of the control packets, the *DB DESC*, *LS REQ*, *LS UPDATE*, and *LS ACK*. This process, just like the Hello protocol (see Section 2.3.7), follows the NSM. However, this process follows the part of the state machine indicated in figure 2.5.b. This part of the state machine is described in Section 6.6.1 of [1] and again Chapter 10.3 of [2]. This process is only necessary if two neighbors need to become fully adjacent. In Section 2.3.7, we have seen the conditions for two routers to become adjacent. Besides this, to become fully adjacent, two neighbors must synchronize their LSDBs.

This process consists of two main sub-processes, the *Database description process* and the *Loading process*. This whole process, along with a complete example, is fully described and can be found throughout Section 6.6.1 of [1].

After establishing an *adjacency relationship*, two routers start the LSDB synchronization process by establishing a *master-slave* relationship, where the master conducts the process. The whole process is protected by a *Stop-and-Wait* protocol, also controlled by the master. The master is the router with the highest RID.

The Database description process consists of the two routers establishing their master-slave relationship and then using DB DESC packets to describe the summarized contents of their LSDBs. In this process, the headers of all LSAs present at each router LSAs are put into a local list, the *Database summary list* and sent to the neighboring router. These LSAs are removed from this list when their reception is (implicitly) acknowledged. DB DESC packets are exchanged until this list is empty in both routers, signalling the end of the Database exchange process.

During the Database description process, if any of the routers receives an LSA header of fresher instances or new LSAs, it will place the corresponding **LSA ID (its three fields)** in a list called *Link state request list*. If by the end of the Database description this list is not empty, routers will request the LSAs within the list to be flooded, using **LS REQ** packets. The neighbor will respond by flooding the requested LSAs in **LS UPDATE** packets. The received LSAs are then removed from the list of requests. This goes on until the list is empty on both routers. When this happens, the process ends, and the neighbors become fully adjacent.

2.3.12 Hierarchical Networks

Currently, OSPF multi-area structures are restricted to a *two-level hierarchy*, with a single area at the upper level, the **backbone** area. The lower-level areas must directly attach to the backbone and direct communication between lower-level areas is not allowed, meaning it must cross the backbone. For multi-area routing, a DVR approach is taken, which can be limiting to the networks in some aspects, as discussed in Section 2.2.1.

In OSPF, areas are identified by the **Area ID**, a 32-bit number written in dotted notation. Each router interface is associated with exactly one area, and in the case that multiple interfaces are assigned to the same link, they must all belong to the same area.

To communicate across areas, the ABR, introduced in Section 2.2.2, is used. Each ABR must have at least one interface directly attached to the backbone and another one to a lower-level area, belonging to all the areas they directly attach to. ABRs store the LSDB of *every area* they directly attach to. Routers can be identified as ABRs or ASBRs through their *B-bit* and *E-bit* flags, respectively, contained in the Router-LSAs.

The *area-internal* routing information of each area is described as for *single-area networks*, using the same LSAs and mechanisms. For multi-area routing, ABRs must originate as many Router-LSAs as the number of areas they directly attach to.

The *area-external* routing information is carried as DVs, to reach the ABRs that will inject that information into the areas they attach to. The area-external prefixes are advertised in **Network-Summary-LSAs**. These LSAs are flooded with *area scope* and communicate the prefixes from one area to neighboring ABRs. The ABRs then flood that information into other areas, reaching their neighboring ABRs, disseminating the information throughout the network. The advertised prefixes are defined by their IPv4 address, carried in the *Link State ID* (in the header), and the *Network Mask* field. The *Metric* field carries the intra-area shortest path cost from the advertising ABR to the prefix it advertises. Each Network-Summary-LSA can carry information regarding only one prefix. OSPFv3 takes a similar approach to this but uses a different type of LSA, the **Inter-Area-Prefix-LSA**.

For *domain-external* information, besides the **AS-External-LSAs**, introduced in Section 2.3.5, which is flooded with a *domain scope* and describes the external prefixes being advertised, **ASBR-Summary-LSAs** (or the **Inter-Area-Router-LSA** in OSPFv3) are used. This LSA is originated by ABRs, and it describes the ASBRs existing in the areas they directly attach to. These LSAs are disseminated as DVs and flooded with *area scope*. They carry the RID of the ASBR they mean to advertise. The ASBR-Summary-LSA only features one specific field, the *Metric*, representing the intra-area shortest path cost from the advertising ABR to the ASBR.

As stated in Section 2.2.2, one of the main purposes for these areas is to reduce the amount of information that each router must store. To further improve this aspect, and as specified in Section 6.3 of [8], a special type of lower-level area can be used, the *Stub Areas*. These Stub areas are meant to include routers with limited resources, such as router memory. They have more restrictions than to normal areas, since AS-External-LSAs are not flooded into these Stub areas. Furthermore, the flooding of Summary-LSAs is optional. With the lack of support for this type of LSAs, the Stub Areas cannot support the configuration of Virtual Links or contain ASBRs and must be configured in the edges of the OSPF domain. External routing can then be done exclusively via the default routes injected by the areas ABRs. This can drastically reduce the resource usage in routers, and so improving the area's scalability, with the possible trade-off of not being able to calculate the optimal routes to every area-external destination.

2.3.13 Limitations over the current multi-area networks

Besides the hierarchy-imposed restrictions, which are overcome in OSPF with the use of the *virtual links*, there are two main restrictions on the way the DVs are advertised in multi-area routing that may *prevent optimal routing*. Currently:

- It is not possible to inject DVs into an area when they advertise paths to an *area-internal* destination (from an external source)
  - ABRs don't advertise routes to destinations within the areas they are injecting information into
- It is not possible to inject DVs into an area when these advertise paths to *external prefixes* that cross that same area (ABRs don't consider area-internal paths when advertising their costs to external prefixes)
  - ABRs don't advertise routes crossing the area they are advertising the information into

Apart from these, another feature that can hinder optimal routing is the *preference of area-internal routes* over area-external ones. These multi-area-related issues are the motivators for this work, and our solution focuses on overcoming them. To better illustrate some of these current restrictions over current OSPF multi-area routing, we leave in Section 2.3.14 an experiment performed by us and as described in Section 11.2 of [1].

2.3.14 Experiment on the current OSPF multi-area routing restrictions

For this experiment, we used the network topology of figure 2.7, where each router is a Docker container running the base implementation described in Chapter 3. This topology has 4 routers and 2 areas. R1 and R2 are routers internal to Area 1, while R3 and R4 are ABRs connected to both Area 0 and Area 1, with a link connecting them through Area 0. Four subnets exist, 222.222.10.0/24, 222.222.20.0/24, 222.222.30.0/24 and 222.222.40.0/24. All of these belong to Area 1, except for the last one, which belongs to Area 0. The routers are connected as we see in the figure, and apart from the eth0 interface of R2, presenting a cost of 100, all router interfaces are configured with a cost of 10.

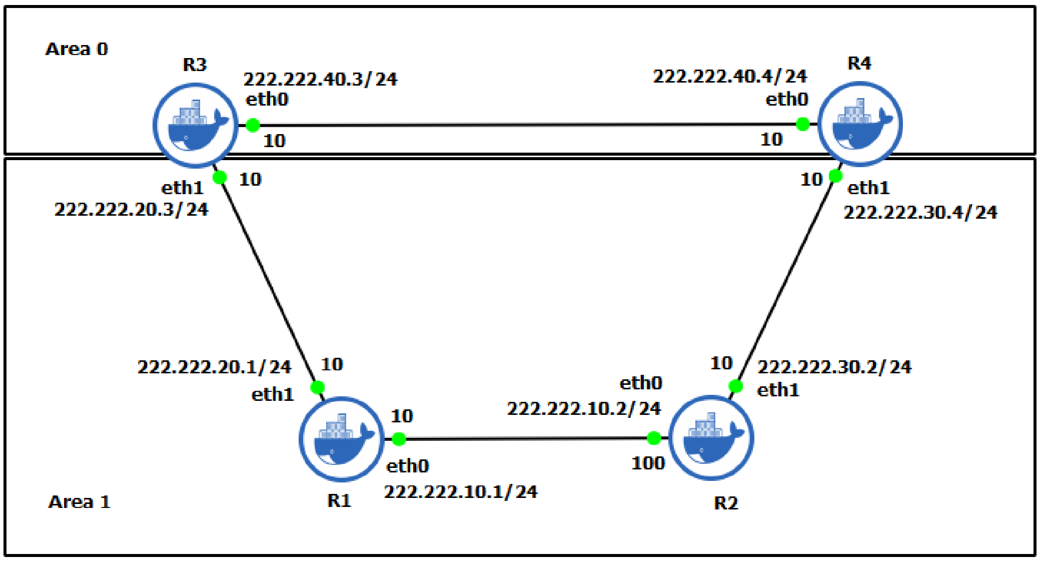
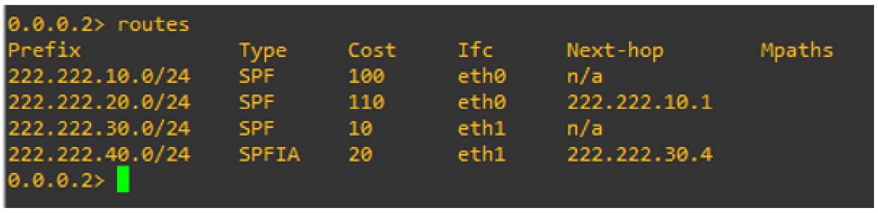


Figure 2.7: Network topology used to test multi-area routing restrictions

Under normal operation, following the illustrated network configuration, we obtain the routing tables of R2 and R4 and Network-Summary-LSAs of R3 and R4, from area 0 and relative to subnet 222.222.20.0 (subnet 20 from now on), shown in figure 2.8. From figure 2.8.a we can see that R2 is not taking the shortest path to subnet 20. As we can see the cost to the destination is 120, through R1, and the real shortest path cost is 30, through R4. R3 and R4 cannot advertise their Network-Summary-LSAs relative to Area 1 into the area, even if it would provide a better option for R2 to reach subnet 20.

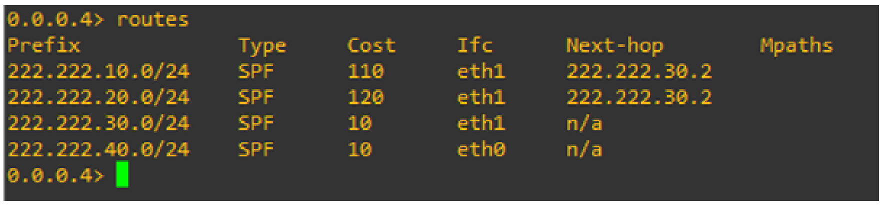
We can see the contents of the Network-Summary-LSAs from R4 and R3, respectively, advertised in Area 0 relative to subnet 20 in figures 2.8.c and 2.8.d. These LSAs are stored in the LSDBs of R3 and R4. From this information, R4 knows it could provide a path to subnet 20 with a cost of 20. Finally, we will look at figure 2.8.b, showing the routing table of R4. We know from the network topology that R4 can reach subnet 20 through R3 or R2. From R4's routing table we can see that it chose to go through Area 1, presenting a cost of 120, instead of 20 which could be achieved by crossing Area 0 instead. This is a result of R4 learning that it can reach subnet 20 directly through its area, and it the current implementation, it will prefer intra-area routes over inter-area ones, even if these provide lower path costs. This happens because, when building the routing table, the router will first try to use intra-area paths, and only if that is not possible will it try inter-area paths.

(a)

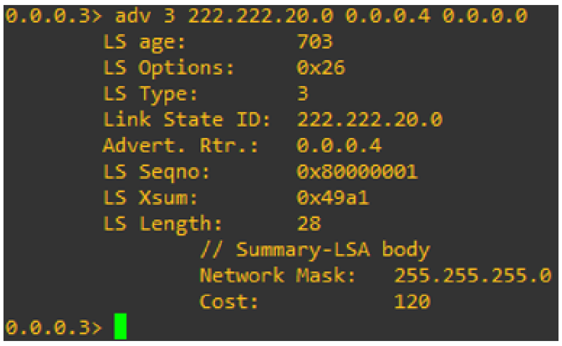


```
0.0.0.2> routes
Prefix            Type     Cost     Ifc      Next-hop         Mpaths
222.222.10.0/24   SPF      100      eth0     n/a
222.222.20.0/24   SPF      110      eth0     222.222.10.1
222.222.30.0/24   SPF      10       eth1     n/a
222.222.40.0/24   SPFIA    20       eth1     222.222.30.4
0.0.0.2>
```

(b)

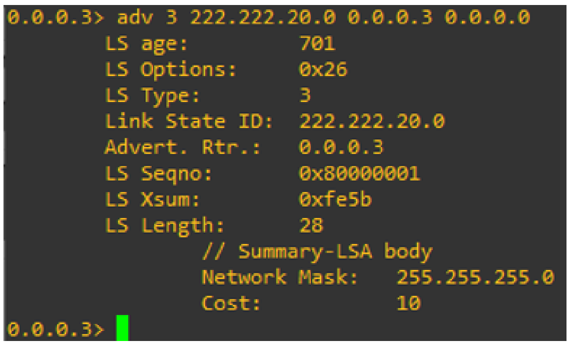


```
0.0.0.4> routes
Prefix            Type     Cost     Ifc      Next-hop         Mpaths
222.222.10.0/24   SPF      110      eth1     222.222.30.2
222.222.20.0/24   SPF      120      eth1     222.222.30.2
222.222.30.0/24   SPF      10       eth1     n/a
222.222.40.0/24   SPF      10       eth0     n/a
0.0.0.4>
```



```
0.0.0.3> adv 3 222.222.20.0 0.0.0.4 0.0.0.0
        LS age:          703
        LS Options:      0x26
        LS Type:         3
        Link State ID:   222.222.20.0
        Advert. Rtr.:    0.0.0.4
        LS Seqno:        0x80000001
        LS Xsum:         0x49a1
        LS Length:       28
                // Summary-LSA body
                Network Mask:    255.255.255.0
                Cost:            120
0.0.0.3>
```

(c)



```
0.0.0.3> adv 3 222.222.20.0 0.0.0.3 0.0.0.0
        LS age:          701
        LS Options:      0x26
        LS Type:         3
        Link State ID:   222.222.20.0
        Advert. Rtr.:    0.0.0.3
        LS Seqno:        0x80000001
        LS Xsum:         0xfe5b
        LS Length:       28
                // Summary-LSA body
                Network Mask:    255.255.255.0
                Cost:            10
0.0.0.3>
```

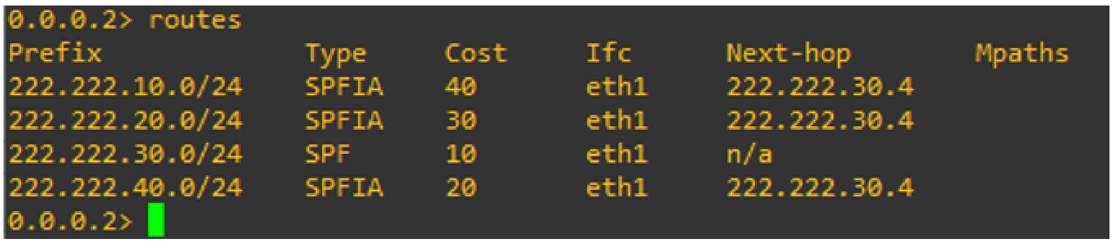
(d)

Figure 2.8: Restrictions on the shortest path selection - (a) Routing table of R2, (b) Routing table of R4, (c) Network-Summary-LSA of R4 (area 0) relative to subnet 222.222.20.0 and (d) Network-Summary-LSA of R3 (area 0) relative to subnet 222.222.20.0

If we were to shutdown interface eth0 of R2, then both R2 and R4 wouldn't be able to reach subnet 20 through area 1 anymore, and only then would they consider the area-external path to reach it. Figure 2.9 shows the routing table of R2 (figure 2.9.a), the routing table of R4 (figure 2.9.b) and the Network-Summary-LSA generated by R4 for area 1, relative to subnet 20 (figure 2.9).

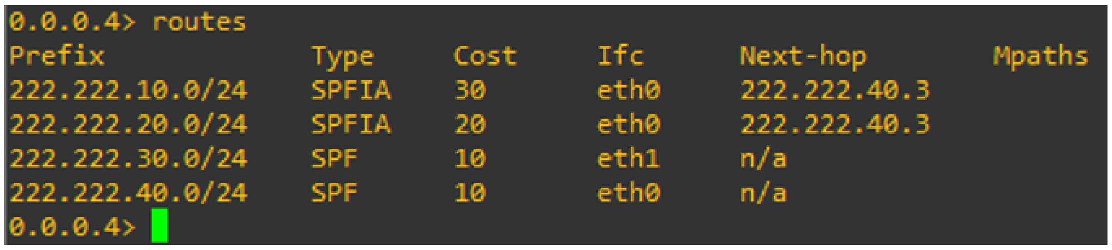
R2 and R4 are now reaching subnet 20 through area 0. But this is only happening because area 1 is partitioned and the intra-area path is unavailable. This change makes it so that R1 and R2 are essentially considered to be in different areas. R4 cannot advertise the Network-Summary-LSA relative to its intra-area path to subnet 20. So, it will instead advertise the Network-Summary-LSA calculated from the one injected into area 0 by R3. From this LSA, generated by R4 and injected into the area partition where R2 is, R2 is also able to reach the subnet.

(a)



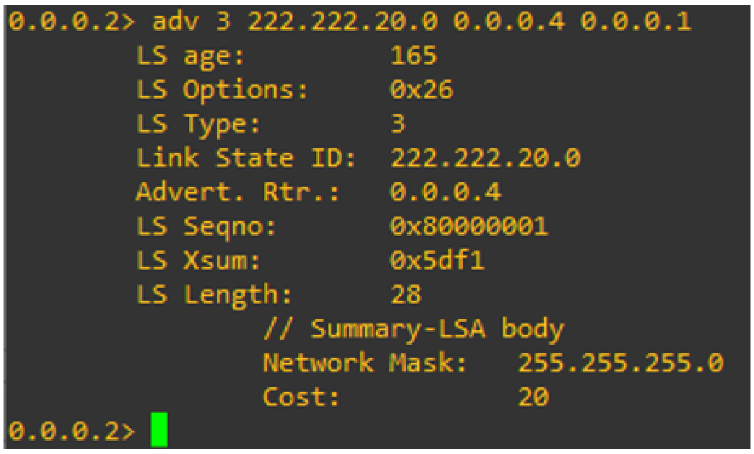
```
0.0.0.2> routes
Prefix              Type     Cost     Ifc      Next-hop         Mpaths
222.222.10.0/24     SPFIA    40       eth1     222.222.30.4
222.222.20.0/24     SPFIA    30       eth1     222.222.30.4
222.222.30.0/24     SPF      10       eth1     n/a
222.222.40.0/24     SPFIA    20       eth1     222.222.30.4
0.0.0.2>
```

(b)



```
0.0.0.4> routes
Prefix              Type     Cost     Ifc      Next-hop         Mpaths
222.222.10.0/24     SPFIA    30       eth0     222.222.40.3
222.222.20.0/24     SPFIA    20       eth0     222.222.40.3
222.222.30.0/24     SPF      10       eth1     n/a
222.222.40.0/24     SPF      10       eth0     n/a
0.0.0.4>
```

(c)



```
0.0.0.2> adv 3 222.222.20.0 0.0.0.4 0.0.0.1
        LS age:          165
        LS Options:      0x26
        LS Type:         3
        Link State ID:   222.222.20.0
        Advert. Rtr.:    0.0.0.4
        LS Seqno:        0x80000001
        LS Xsum:         0x5df1
        LS Length:       28
                // Summary-LSA body
                Network Mask:    255.255.255.0
                Cost:            20
0.0.0.2>
```

Figure 2.9: Restrictions on the shortest path selection, after shutting down eth0 for R2 - (a) Routing table of R2, (b) Routing table of R4, (c) Network-Summary-LSA of R2 (area 1) relative to subnet 222.222.20.0

3 The Base Implementation

Contents

This work has as its base a C++ OSPFv2 implementation, based on John Moy's implementation, according to RFC 2328 [2] and as described in [3]. This code is an updated version of the implementation described in [3], and we will be using its version 2.16, already supporting the use of *Opaque-LSAs*, which will prove useful for our solution. The code is written using the **C++** coding language and uses an Object-Oriented Programming (OOP) approach. RFC 2328 [2] is fully supported by this implementation, meaning the OSPF description given throughout Chapter 2 holds. In this chapter, we cover some aspects of the base implementation, although not all of them. More complete descriptions can be found in [3].

3.1 Software Architecture

3.1.1 Data Flow

The data flow of this base implementation follows the representation presented in figure 3.1. The initialization code serves to create a system interface that will perform basic lower-level operations, such as sending protocol packets or changing the machine's routing table entries. This code section also reads the OSPF configuration after creating the top-level OSPF object (see Section 3.1.2). Through an OSPF Application Programming Interface (API), this initialization section and the code's main loop communicate with the OSPF instance.

The main loop of this implementation serves five main purposes:

- Receiving input in the form of (re)configuration;
- Keeping track of the elapsed time, as well as calling the OSPF's timing function;
- Reading arriving OSPF packets;
- Detecting changes in the states of the interfaces;
- Receiving input in the form of monitoring requests.

3.1.1.A Packet Flow

The OSPF packets received by the main loop are passed down to the OSPF code via the **OSPF::rxpkt()** method. The packet type is then demultiplexed and forwarded to the corresponding function to process it. The protocol must maintain a synchronized LSDB across the network. For that, adjacencies are built and maintained between neighboring routers. Receiving or stopping to receive *HELLO* packets from a neighbor will cause the creation or destruction of an adjacency, respectively. Receiving *DB DESC* and *LS REQ* packets will progress an adjacency through the states of the NSM, in order to

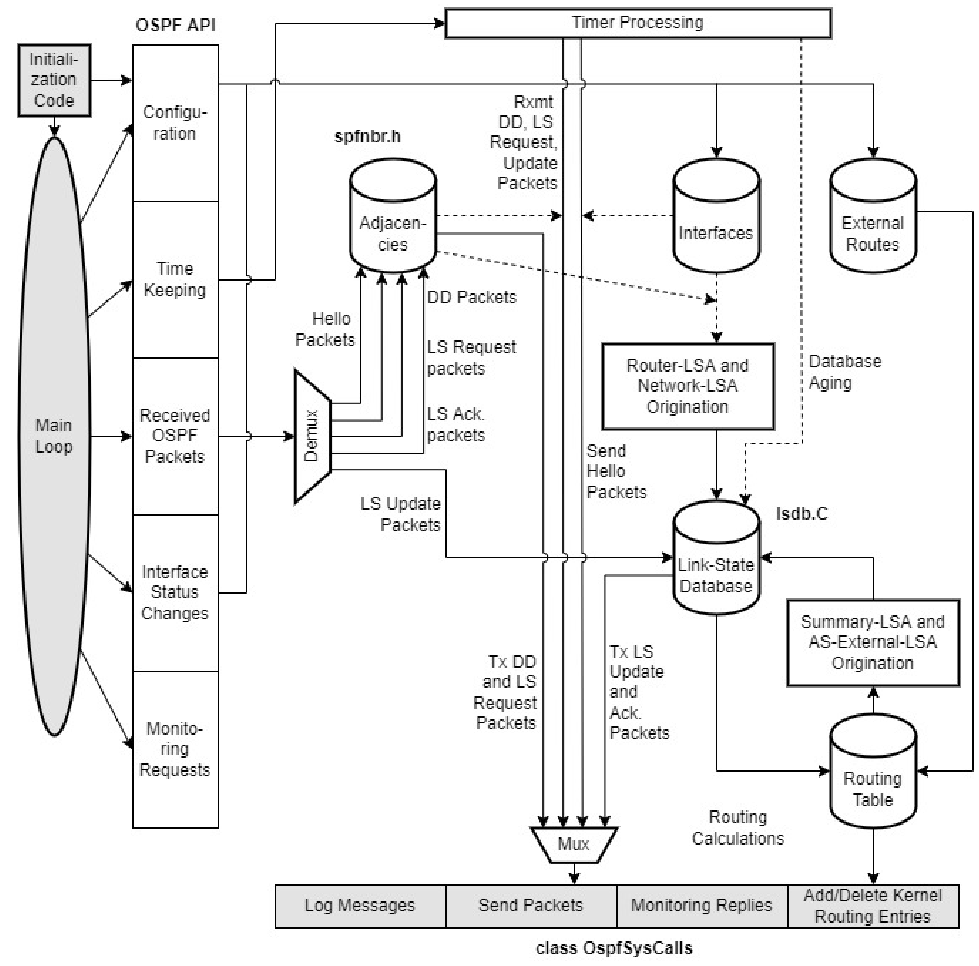


Figure 3.1: Data flow in the base implementation
Adapted from [3]

achieve synchronization between the neighbors LSDBs (see Section 2.3.11). After this synchronization is achieved, it is maintained through the flooding of *LS UPDATE* packets, as we described in Section 2.3.9. Besides the periodic *HELLO* transmissions and *LS UPDATE* transmissions whenever a router originates a new LSA, packets are mostly transmitted as a response to packets received at the router. An example of this is the acknowledgment of received LSAs through the *LS ACK* packets or flooding of requested LSAs with the *LS UPDATE* packets.

3.1.1.B LSA Flow

As described in Section 2.3.3, the LSDB stores the LSAs collected at each router. Some aspects of this implementation's LSDB are discussed in Section 3.2. LSAs can be installed in the LSDB either when they are received from neighbors or when they are originated at the router. Based on its current active interfaces and established adjacencies, a router will (re)originate its Router-LSA. Similarly, for each link in which the router is the DR, it will also generate the corresponding Network-LSA. Furthermore, based on routing calculations results, the routers might also generate Network/ASBR-Summary-LSAs and AS-External-LSAs. LSAs are continually aged, for the protocol to know when to refresh or delete them, for instance. When a new LSA is successfully installed in the LSDB, it is flooded with its respective scope. Also, to keep the routing table up-to-date, the appropriate routing calculation is done based on the newly installed LSAs.

3.1.1.C Timers

This implementation also relies highly on timers, for periodic transmissions (of *HELLO* packets) or needed retransmissions, for packets not yet acknowledged, for example. These timers are also important for the LSA aging within the LSDBs. To not overload the routers, timers can also limit the rate on some of the mechanisms, such as the importing of AS-External-LSAs into the network. Timers can also implement some of the optimizations referred in Section 2.3.9, such as the *delayed acknowledgment*. Finally, some of the mechanisms described for the state machines introduced might rely on timed events, as we have seen, for example, in Section 2.3.8 for the *Interface State Machine*.

3.1.2 Major Data Structures

As the major data structures present in this implementation, we highlight four classes. Figure 3.2 summarizes the organization of these major data structures. The first of them is the **OSPF** class. This single class is pointed at by the **ospf** global variable and serves not only as the starting point to all other data structures but also as the bridge for the platform-specific code to access the OSPF application, as was said in the previous section, by calling the **OSPF::rxpkt()** method. There are many data items present in this and other classes, and these will not be described unless necessary. Most of these, with detailed descriptions, can be found throughout [3].

As a main data structure to store groups of similar elements, this implementation uses AVL trees, represented by the **AVLtree** class. These are balanced binary trees, in which its elements use up to two indexes to identify themselves and keep the tree balanced upon an item addition or removal. The tree elements are represented by the **AVLitem** class, and all other classes that need to keep a organized structure must inherit from it. There are operations defined to add or remove, given the pointer to the item, and iterate or find a specific item, given one or both its indexes for this last case. This is highly used to keep lists of all types of LSAs, and depending on each specific LSA characteristics, they might use one or more attributes as indexes in the tree.

Next, we highlight the **SpfArea** class. Each instance of this class is used to represent an *OSPF Area*, such as the ones described throughout Chapter 2. It's possible to scan through the areas to which the router is attached to with the **AreaIterator** class. Because most of the mechanisms used in OSPF are used within each area, this class maintains some of the basic OSPF data. But especially, the **SpfArea** maintains most of the LSDB for each area, keeping separate trees (using the **AVLTree** class) for each type of LSA.

Another important structure is the **SpfIfc** class. As we have seen, several types of interfaces can be used in OSPF (see Section 2.3.2). Each of these types is represented as a different class, all inheriting from **SpfIfc**. **SpfIfc** branches out into three classes:

- **DRIfc** (Interface types that need to elect DRs), branching further down into:
  - **BroadcastIfc** (broadcast/shared interfaces)
  - **NBMAIfc** (NBMA interfaces)
- **PPIfc** (point-to-point interfaces), also serving as the base class for:
  - **VLIfc** (virtual links)
- **P2mPIfc** (point-to-multipoint interfaces)

The interfaces of the router running the program can be scanned using the **IfcIterator** class. It can either scan all the router's interfaces or only those attaching to a specific area. As we can see from figure 3.2, each area can have several associated interfaces. However, each interface must belong to exactly one area, as stated in Section 2.3.12.

Finally, we have the **SpfNbr** class. This class represents the neighboring routers connected to the router running the program through its interfaces. The neighbors are attached to the router through a specific interface. If an interface has more than one neighbor (e.g., a broadcast interface), separate **SpfNbr** instances must be created, one for each neighbor. The neighbors associated with an interface can be scanned with the **NbrIterator** class. Most of the elements contained in this class are concerned with the synchronization mechanisms between neighbors.

3.2 The Link-State Database

3.2.1 The LSDB Fundamentals

As we have seen in Section 2.3.3, LSDBs store information in the form of LSAs. In this implementation, LSAs have two types of representation, the *internal representation*, for when they are installed in the LSDB, and the *network representation*, as described in figure 2.3, used to transmit the LSAs across the network. There is a class to represent each different type of LSA. However, all these classes inherit from a base class **LSA** (which inherits from the **AVLitem** class, for storage purposes). The main **LSA** class can be split into two intermediate classes, depending on the role of the LSA in the routing

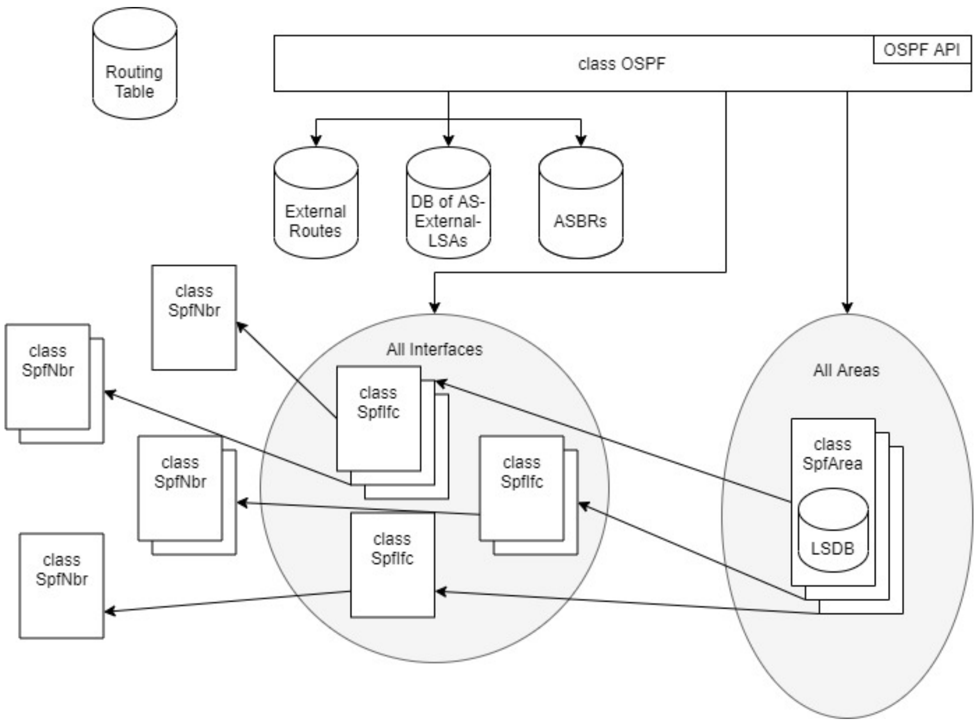


Figure 3.2: Summary of the organization of the major data structures in the implementation
Adapted from [3]

calculations. These classes refer to the internal representation of the LSAs. The first is the **TNode** class, the base class for the **rtrLSA** and **netLSA** classes, representing the *Router-LSAs* and *Network-LSAs*, respectively. The **TNode** class represents the elements that will be used as the topology nodes, when building its graph. The second one is the **rteLSA**, where the **summLSA** (*Network-Summary-LSAs*), **asbrLSA** (*ASBR-Summary-LSAs*), and **ASextLSA** (*AS-External-LSAs*) fall into. Besides storing the LSA header fields, the **LSA** class also stores several other types of information regarding the LSA, such as the area to which it belongs (**LSA::lsa_ap**), and provides methods such as **LSA::cmp_instance()**, allowing to compare a newly received instance with its stored counterpart to determine which one is fresher. A complete description of the fields present in this class can be found in Section 6.1 of [3].

Since each area has its own LSDB, it makes sense that each LSA type having area scope is stored in that area's **SpfArea** class. This list contains all the LSAs already described (**rtrLSA**, **netLSA**, **summLSA**, and **asbrLSA**) except the **ASextLSA** (as it has AS scope). This last type of LSA is instead kept directly in the OSPF class.

3.2.2 Database Operations

Several operations can be executed over the databases, which we also highlighted. We start with the **LSA *OSPF::FindLSA** (or **LSA *OSPF::myLSA** for self-originated LSAs), used to find a specific LSA.

Next, we have the **OSPF::AddLSA** function, used to add an LSA to the LSDB. LSAs are not stored in the same format in which they are received, so an **OSPF::ParseLSA** function is used to convert the new LSA to the desired format, easier to use in the routing calculations (see Section 3.5). This function then calls a type-specific *parse* function for that LSA type. Another important function is the **OSPF::DeleteLSA**, used to delete an LSA from the LSDB. As the LSAs need to be parsed when added to the LSDB, they must also be unparsed when removed, using the **OSPF::UnParseLSA**. This function also calls a type-specific *unparse* function, depending on the LSA type. Finally, to restore the network format of the LSAs from the internal format, to transmit LSAs to neighbors, or to compare LSA instances, for example, an **LShdr *OSPF::BuildLSA** function can be used. This function also resorts to type-specific *build* functions. The inner work of each of these functions is described in detail in Section 6.2 of [3].

3.3 Originating LSAs

There are several reasons why we might want to originate or re-originate an LSA. To cope with local changes, replace old self-originated LSAs on the LSDB, or keep the ones already there from being removed (LSA refreshing). This implementation considers these the three triggers that will cause LSAs to be originated (or re-originated).

This process starts by calling a type-specific function to originate an LSA. After this, we must find out where is the copy of the originated LSA in the LSDB, if there is one, resorting to the **OSPF::myLSA** function. Then the **OSPF::ospf_get_seqno** function is called, to determine the next sequence number for that LSA. This function can result in a postponed origination if it determines that the LSA cannot be originated yet (e.g., because of a minimum guard time over the origination of successive LSAs). If the LSA origination is possible, its contents are then created, both the header and the body. If after this creation, the body of the LSA is empty, then it is flushed instead of being (re)originated. Otherwise, an **OSPF::lsa_reorig** function is called, indicating the end of the process and, if the new LSA instance presents changes from the old one stored (or if it is forced, via a flag in the function's parameters), it will be updated in the LSDB and flooded. The detailed functionalities of the **OSPF::ospf_get_seqno** and **OSPF::lsa_reorig** can be found in Section 7.1 of [3].

The previous process is the general operation of the LSA origination process. However, each LSA type may present specific particularities, due to different fields or scopes. An example of this is the *Router-LSA*, where a different one needs to be originated for each area to which the router attaches (its origination function is defined in the **SpfArea** class), and the *Network-LSA*, which a router might need to originate for each interface it is using to connect to a shared link (its origination function is defined within the **SpfIfc** class).

Some LSAs cannot be generated when the origination function is called, because the minimum interval between consecutive LSAs has not yet been met (guard time). These are scheduled by the function to be originated when possible. When this interval is met, the **OSPF::deferred_lsas** function is called to originate all the LSAs in this situation.

The mechanisms described for *premature aging*, to *delete information*, or for *refreshing LSAs* are also supported by the implementation, and can be found in Section 7 of [3].

3.4 Area Routing

Area Routing is implemented in the code in a way that follows the description given in Section 2.3.12. For that reason, it suffers from the same constraints described in Section 2.3.13. In this section we leave a number of short notes on the main mechanisms in play when it comes to Area Routing.

Summary-LSAs are core elements to implementing Area Routing. They are advertised by ABRs into the areas they directly attach to, to disseminate the routing information across the AS. In this current implementation, ABRs must advertise these LSAs into the backbone area, for it to advertise them into all other lower-level areas. These LSAs are installed either when received (with the **SpfNbr::recv_update** method) or originated (with the **OSPF::sl_orig** method). When that happens, in case the LSA is changed, a full calculation (**OSPF::full_calculation**) must be performed by the ABR in order to correctly (re)originate the correspondent Summary-LSA (should the routes change) to be advertised into the areas in which they were not generated.

All area border routers must connect to the backbone area (Area ID 0.0.0.0), either directly or through a sequence of configured Virtual Links (VLs). This is done so that all inter-area communication cross the backbone, even if it is between two adjacent lower-level areas. These areas where VLs are configured so that they carry traffic neither originated from nor destined to those areas are called *transit areas*. The ABR endpoints in these transit areas then set their V bit in their Router-LSA. Also, stub areas can be configured to perform ass described in Section 2.3.12.

These mechanisms are present in the current base implementation, and a more complete and technical description can be found in Section 10.2 of [3].

3.5 Routing Calculations

In this implementation, when adding new or updated LSAs to the LSDB, routing calculations are triggered. As for new LSAs calculations will always be triggered. For LSA updates, the **LSA::cmp_contents** method is called, to differentiate actual updates from simple refreshments. Then, when an update or new addition is detected, the appropriate routing calculation is either performed or scheduled with the

OSPF::rtsched method, passing the LSA that is being updated. Based on this LSA type, the calculation type might vary, with The most common cases being installing new or updated Router-LSAs, Network-LSAs and both Network and ASBR-Summary-LSAs.

If a Router-LSA or Network-LSA are passed on to the scheduling method, a complete calculation will be scheduled, by setting the **OSPF::full_sched** flag to *true*. As for the Summary-LSAs, as well as for the AS-External-LSAs, since these refer to a single routing table entry each, it is possible to perform more specific calculations, exclusively over the affected entries, and possibly over the entries referred to by those LSA previous instances.

The full calculation, or the intra-AS calculation, is scheduled rather than directly performed, in order to limit the frequency of this heavier calculation, by default to a maximum of one run per second. It is divided into several steps, and each router will perform it with the information stored in its LSDB and considering itself the root of the calculation. The main steps of this calculation are the following:

1. A Dijkstra's Shortest Path First algorithm (described in detail in Section 4.8 of [8]) is run for all the areas to which the router attaches to, with the algorithm Nodes being the LSAs of inheriting the **TNode** class (see Section 3.2.1). This is used to calculate all of the routers' shortest intra-area routes;

2. After the intra-area routes calculation is finished, a routine, **OSPF::update_brs**, is used to check if the best paths to any of the ABRs has changed from previous calculations;

3. The router's IP routing table is iterated through (by the **OSPF::rt_scan** method), and a set of operations is performed over each one:

   - Old intra-area routes not updated by this run's Dijkstra calculation are deleted;
   - Inter-area routes are calculated for network segments present in distant areas (to which the router is not directly attached);
   - The system's kernel routing table is updated to contemplate the changes noted in this calculation process;
   - Network-Summary-LSAs referring to the changed routes that the router should (re)originate are updated.

4. The best routes to each ASBR are calculated, based on all the calculations done so far and the existing ASBR-Summary-LSAs present in the LSDB. If the chosen route to any ASBR is altered, the routes to the AS-external destinations are scheduled to be recalculated as well.

A full and detailed description of the Routing Calculations present in this implementation can be found throughout Section 11 of [3].

4

Solution overview

Contents

4.1 The ABR Overlay

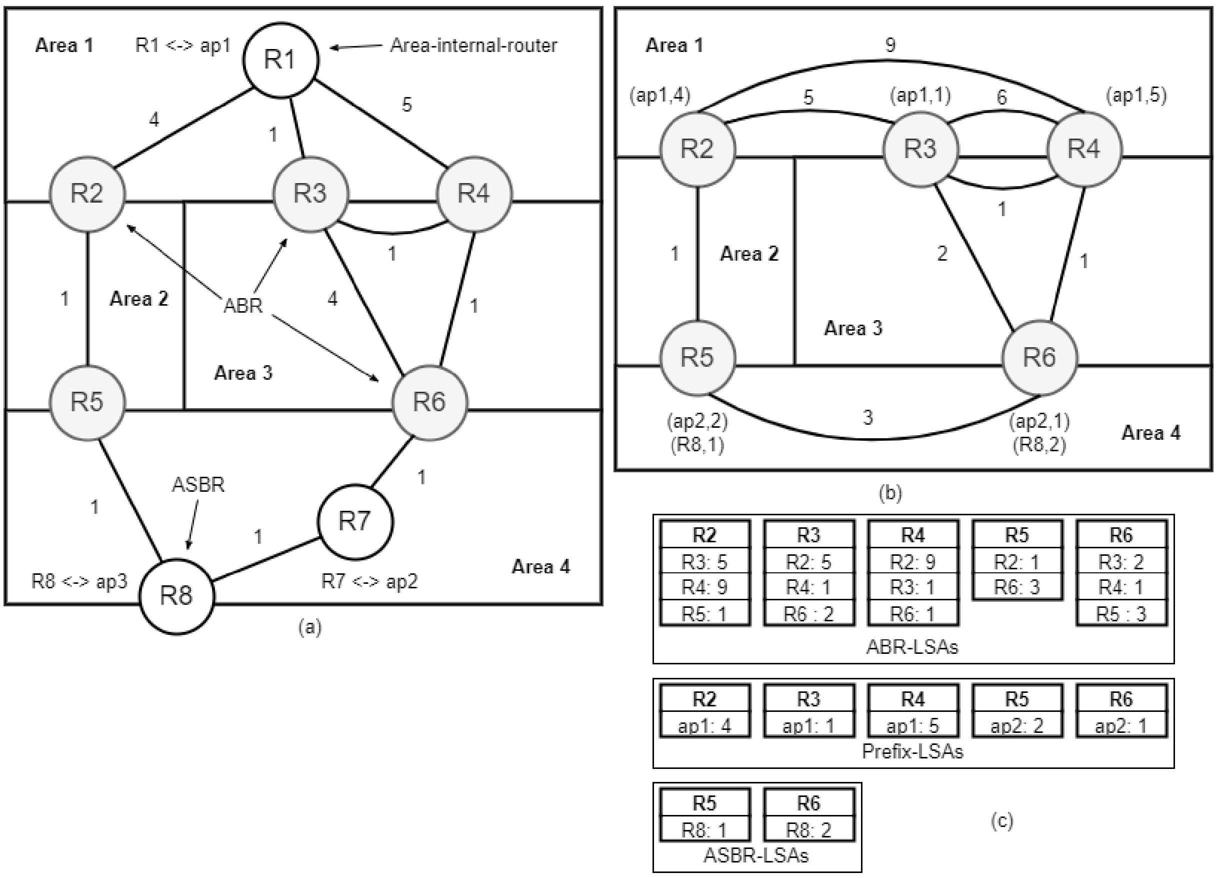
In this report, we propose the implementation of an OSPFv2 extension to support multi-area networks with arbitrary topologies to overcome the restrictions and sub-optimal routing, as described in Section 2.3.12. To achieve this, we must change the routing protocol being run by ABRs for inter-area communication, from a DVR approach to an LSR approach, as described in Section 3 of [1] and in [4].

The ABRs form a *routing overlay* over the physical network, to run the inter-area routing protocol, representing an ABR-only network, in which the ABRs will exchange the inter-area routing information. To do this, ABRs need to become overlay neighbors, i.e., each of them must have at least one interface attached to the same area, and not necessarily be directly connected. This exchange is done through the flooding mechanisms, so no new synchronization mechanisms are used to this end. So, the ABRs can store and advertise (via their common LSDBs) their intra-area costs to their ABR neighbors. The area-external prefixes can then be injected into the areas resorting to the already existing LSAs. To serve as an example of the ABR overlay and the Overlay LSAs, we have figure 4.1.

Figure 4.1.a shows a network composed of 8 routers and structured in 4 areas. Routers R2 to R6 present ABRs, R8 an ASBR, and R1 and R7 area-internal routers. Three address prefixes need to be advertised, ap1, assigned to R1, ap2, assigned to R7, and ap3, injected by R8 into the AS. Next to the links, we can see their associated costs. Figure 4.1.b shows the overlay graph corresponding with the network of figure 4.1.a. Here, the nodes represent the ABRs and the links represent the shortest intra-area path between neighboring ABRs (not necessarily direct connections) with its associated costs also displayed next to them. Furthermore, the ABRs have (destination, cost) pairs labelled next to them, representing the prefixes or ASBRs and the intra-area shortest path cost to them, available at the areas they directly attach to. Finally, figure 4.1.c shows the Overlay LSAs and their contents, used to completely describe the overlay graph shown.

As we said, this extension aims to implement a link-state multi-area routing approach. For this to happen, the ABRs must have a complete view of the overlay, building a graph just like what is done in OSPF for each area. First, each ABR has to flood its local view (who their ABR overlay neighbors are and their intra-area shortest path cost to them) of the overlay to all other ABRs (with domain scope flooding). To flood this information, the existing flooding mechanism is used, more specifically, the mechanism used to flood the AS-External-LSAs, since the same flooding scope must be used. With the views from each ABR stored, the ABRs can build an overlay LSDB, much like the area LSDBs introduced, and with the same goal, to represent the overlay topology and its routable prefixes. To describe the overlay LSDB, three new types of LSA must be introduced. Figure 4.2 shows the summary of the structure of these new LSAs.

The **ABR-LSA** is used to describe the topology of the ABR overlay. This LSA describes the originating ABR (in the Advertising Router field from the header) and it's neighbor ABRs (in the Neighbor



ABR-LSAs

R2	R3	R4	R5	R6
R3: 5	R2: 5	R2: 9	R2: 1	R3: 2
R4: 9	R4: 1	R3: 1	R6: 3	R4: 1
R5: 1	R6 : 2	R6: 1		R5 : 3

Prefix-LSAs

R2	R3	R4	R5	R6
ap1: 4	ap1: 1	ap1: 5	ap2: 2	ap2: 1

ASBR-LSAs

R5	R6
R8: 1	R8: 2

Figure 4.1: (a) An example of a multi-area network, (b) its corresponding ABR overlay and (c) LSAs describing the overlay
Adapted from [4]

RID field), as well as the intra-area shortest path cost to them (in the Metric field). These costs are obtained through the area's LSDB. The combination of a Neighbor RID and Cost can be repeated for each neighbor in the same LSA.

The **Prefix-LSA** is used to describe the address prefixes available at each area. Each ABR originates the LSA associated with the prefixes available at the areas they directly attach to. This LSA contains a prefix advertised inside an area (Subnet Address and network Mask fields) and the intra-area shortest path cost from the advertising ABR to it (Metric field). This information can be obtained from the area's Router-LSAs and Network-LSAs.

The **ASBR-LSA** is used to describe the existing ASBRs in an area. As the external prefixes are already advertised by the AS-external-LSAs, there is no need to repeat this information. Thus, the ASBR-LSA is used by the ABRs to describe the ASBRs existing inside the areas they directly attach to (through their RID in the Destination RID field), and their intra-area shortest path cost to them (Metric field).

For each prefix or ASBR being advertised, a new Prefix-LSA or ASBR-LSA must be originated, re-

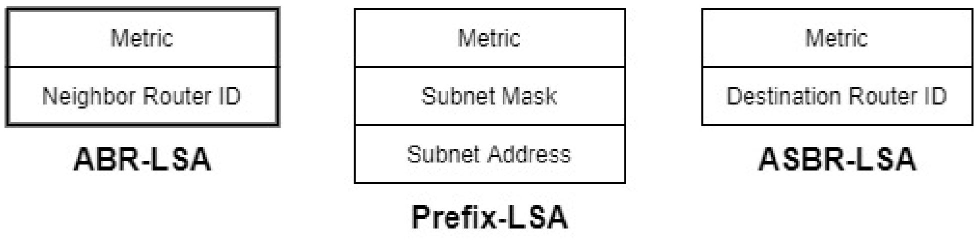


Figure 4.2: Structure of the OSPFv2 Overlay LSAs.
Adapted from [4]

spectively. The three new LSAs introduced will be flooded with domain scope. Just like for area-internal routing, if the ABRs detect changes regarding the overlay that require an update to the advertised LSAs, they must originate new instances of these Overlay LSAs and flood them.

As the information is spread across the network and stored into the overlay LSDB, the ABRs must build and maintain a graph of the ABR overlay, with the nodes representing the ABRs and the links their shortest intra-area path to each other, with the associated costs. With this graph built, the ABRs must then compute their costs to the routable addresses advertised, to inject this information into the areas they directly attach to. This, just like what happens internally to the areas, is done using Dijkstra's algorithm.

The information is then injected into the areas using the already existing LSA types used for those purposes, in this case, the Network-Summary-LSAs for area-external prefixes and ASBR-Summary-LSAs for information regarding the ASBRs. Other routers will not need to change the way they process the information for this approach to work, this extension is transparent to all routers but ABRs.

4.2 Opaque-LSAs

These new LSAs can be introduced into the protocol by using **Opaque-LSAs**. The Opaque-LSA aims to provide better extensibility to the protocol. This type of LSA can be used to support new LSA types, such as the ones described. The Opaque-LSA is fully described in [9]. The Opaque-LSA uses the same header structure, in terms of the packet space it requires, as regular LSAs. The only difference is the way the *Link State ID* bits are used. For other LSAs, this field alone uses 4 bytes (32 bits). The Opaque-LSA uses these 32 bits differently, splitting them into two new fields, the *Opaque Type* (8 bits) and *Opaque ID* (24 bits). To illustrate difference we compare in figure 4.3 the header structure of regular and Opaque LSAs. When flooding the Opaque-LSAs, this difference is not noted as it is not represented in the implementation's code, and they can be flooded as any other LSA, as long as their LSA type is accepted by the routers.

The current version of our base implementation already supports the use of Opaque-LSAs. The class representing the Opaque-LSA, the **opqLSA** class derives from the base LSA class, this meaning that the header structure as well as some basic operations is similar enough to share this same super-class.

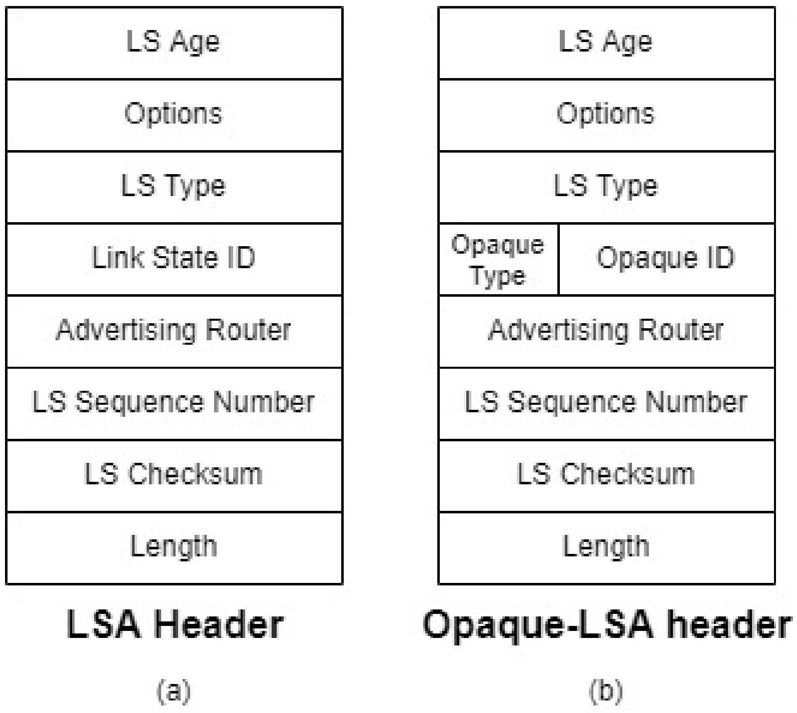


Figure 4.3: Structure of the (a) General LSAs and (b) Opaque-LSAs

The body of the Opaque-LSA is stored under the form of bytes, without any particular structure and with an arbitrary length. It is then up to us to configure our own Opaque Types and how they should be stored and used. Generalized functions to originate (**OSPF::opq_orig**), parse (**opqLSA::parse**), and unparse (**opqLSA::unparse**) these LSAs are already part of the base implementation.

5 Implementation

Contents

In this chapter we address the implementation of the extension proposed in this dissertation and described in Chapter 4. This extension has been developed using the C++ coding language, over the base implementation briefly described throughout Chapter 3 and fully detailed in [3]. Another key aspect, that made this direct integration of the extension into the original version possible, was the fact that this implementation already supported the use of Opaque-LSAs, which is not addressed in [3] nor present in earlier versions of the source code. We will leave, in Appendix A, a list summarizing all the files/functions/classes that were added or changed to accommodate our extension.

The changes made to the base implementation can be divided into several main steps, these being:

1. Including the new LSAs in the protocol's implementation;
2. Defining the operations that must be done over these new Overlay LSAs by the ABRs;
3. Performing the full Overlay routing calculation:
   - Using the Overlay LSAs information to build and maintain an ABR Overlay graph;
   - Using the resulting graph together with the Overlay LSAs information to calculate all the routes within an AS;
4. Injecting the routing information resulting from the calculation into the areas to which the ABRs directly attach, reaching the area-internal routers.

In the next sections we will describe how each of these main steps were accomplished. Furthermore, we will not only describe the supporting mechanisms created and put to use to better accommodate the main features but also provide the necessary context from the base version of this implementation whenever deemed necessary.

Although we started by defining the operations to accommodate all three Overlay LSAs in our implementation, we have decided, since we would only test this extension within a single AS, to focus our development in both the ABR-LSA and the Prefix-LSA, since the ASBR-LSA requirements are very similar to those of the Prefix-LSA. With this in mind, the next Sections will mainly refer to the ABR-LSA and Prefix-LSA, with the ASBR-LSA still being mentioned when supported in the changes made.

5.1 Inclusion of the Overlay LSAs

The first step in implementing this extension is to include everything necessary for the current implementation to support the existence of the Overlay LSAs. As we state in Section 4.2, this is done with the help of the already supported Opaque-LSAs.

We started by defining the Opaque Type for these LSAs. According to Section 9 of [9], four OSPF extensions already use Opaque-LSAs, and values up to 4 for this Opaque Type are already assigned.

Considering that, and even though not all these are depicted in the base version of our code, we opted to assign the first available values to our extension, with an Opaque Type of 5 to ABR-LSAs, 6 to Prefix-LSAs and 7 to ASBR-LSAs. With this simple assignment, the new types can be recognized as valid Opaque-LSA types.

We then defined the structure for each of the LSAs' bodies, by adding them as structs within the **lshdr.h** file, and following the structure presented in figure 4.2, in which:

- The ABR-LSA uses a repeatable set of two fields, the **Metric** and the **Neighbor Router ID**;
- The Prefix-LSA uses a single set of three fields, the **Metric**, the **Subnet Mask** and the **Subnet Address**;
- The ASBR-LSA uses a single set of two fields, the **Metric** and the **Destination Router ID**.

5.2 Operations over the Overlay LSAs

After including the Overlay-LSAs in our implementation, we must define how and when these LSAs will be created and processed. Here we include the origination of the LSAs, as well as the parsing done when being installed into the database and the unparsing done before deleting these LSAs. Within the context of this work, we consider *origination* the LSA building and subsequent flooding by its advertising router. *Parsing* means to process the information contained in the LSA (received or self-generated) into the form in which we will need it for routing calculations. *Unparsing* an LSA is the reverse of the parsing procedure, undoing the processing done.

All matters related to the origination of both Summary-LSAs and Overlay LSAs and to the processing of the Overlay-LSA specific parts of Opaque-LSAs were made unavailable to area-internal routers, since these should be exclusively handled by ABRs from now on. The only feature kept available to area-internal routers is the ability to process received Summary-LSAs, which ABRs no longer do, since all of their inter-area routing will be based on the new Overlay LSAs.

5.2.1 Originating the Overlay-LSAs

Before we address the origination of the LSAs, we will introduce the structures that will support it. With the Prefix-LSAs, we use the **INrte** class, representing an IP routing table entry that should be installed into the Operating System (OS)'s routing table. If we do not explicitly indicate otherwise, we will be referring to *routing table* as the one stored within the OSPF implementation. Similarly to the Prefix-LSAs, the ASBR-LSAs use the **ASBRrte** class, representing the routing table entries to AS-internal ASBRs. Both the **INrte** and **ASBRrte** classes inherit from the **RTE** class, the super class for all types of routes (e.g., routes to destinations and routes to routers). This class contains the elements common to all types of routing table entries, such as the cost, type of route or the set of intra-area next-hops to the destination. Finally, the ABR-LSA uses the newly defined **ABRNbr** class, where we store the information relative to ABRs connected to the same areas as the ABR running the implementation, i.e., the router's ABR neighbors. This class includes the ABR neighbor's Router ID, the intra-area cost to reach it, and links itself (by storing a pointer to the correspondent object) to its corresponding Router-LSA and to the area in which it was advertised, used to differentiate alternative paths (across different areas) to the same ABR.

5.2.1.A Gathering the LSA required information

The creation and value attribution for these **ABRNbr** instances is done in two separate occasions. Figure 5.1 shows two simple and summarized diagrams, illustrating what is done in these two occasions.

The first occasion (figure 5.1.a) is during the parsing of a Router-LSA as it is installed into the database. If (i) our router is an ABR and (ii) if the received LSA comes from an ABR (B bit is set), they are ABR neighbors. If no object of the **ABRNbr** class is linked to a previous instance of this Router-LSA, it will be created and linked to it, by storing its pointer. Otherwise, if this link is already created, the **ABRNbr** object will be reset if an error is detected with its link to the previous instance of the received LSA. This is done within the **rtrLSA::parse** routine.

The second occasion (figure 5.1.b) is during the intra-area Dijkstra calculation. By then, all the required fields in the **ABRNbr** are set but the cost, as it is only set during the Dijkstra calculation. During the calculation (see Section 3.5), the TNode (see Section 3.2.1) inheriting objects (Router-LSAs and Network-LSAs) within each attaching area are processed to create the intra-area topology graph. Then, when processing an ABR during that calculation, if that ABR's Router-LSA is linked with an ABRNbr object, we check if the ABRNbr stored cost differs from the recently calculated intra-area cost to the router. If so, we update that cost with the new intra-area cost and schedule the (re)origination of our router's ABR-LSA. This is done within the **OSPF::dijkstra** routine.

Also during the complete intra-area calculation, routes with changes signal this, via a **RTE::changed** flag. After the Dijkstra calculation is complete and all intra-area costs have been assigned to their routes, the **OSPF::rt_scan** routine is called to iterate over all routes, to check which ones present changes from previous calculations. Here, we must decide whether to originate the route's corresponding Prefix-LSA. This is done by verifying if (i) our router is an ABR, (ii) the route is reachable within one of our directly attached areas (via a intra-area path), and (iii) by confirming that we already advertised our first ABR-LSA into the AS. This last step indicates if there are more ABRs within the network. There is no need to use Overlay LSAs if only one ABR exists. If that is the case, we would directly originate the Network-Summary-LSAs for the destinations into our attached areas, instead of their Prefix-LSAs.

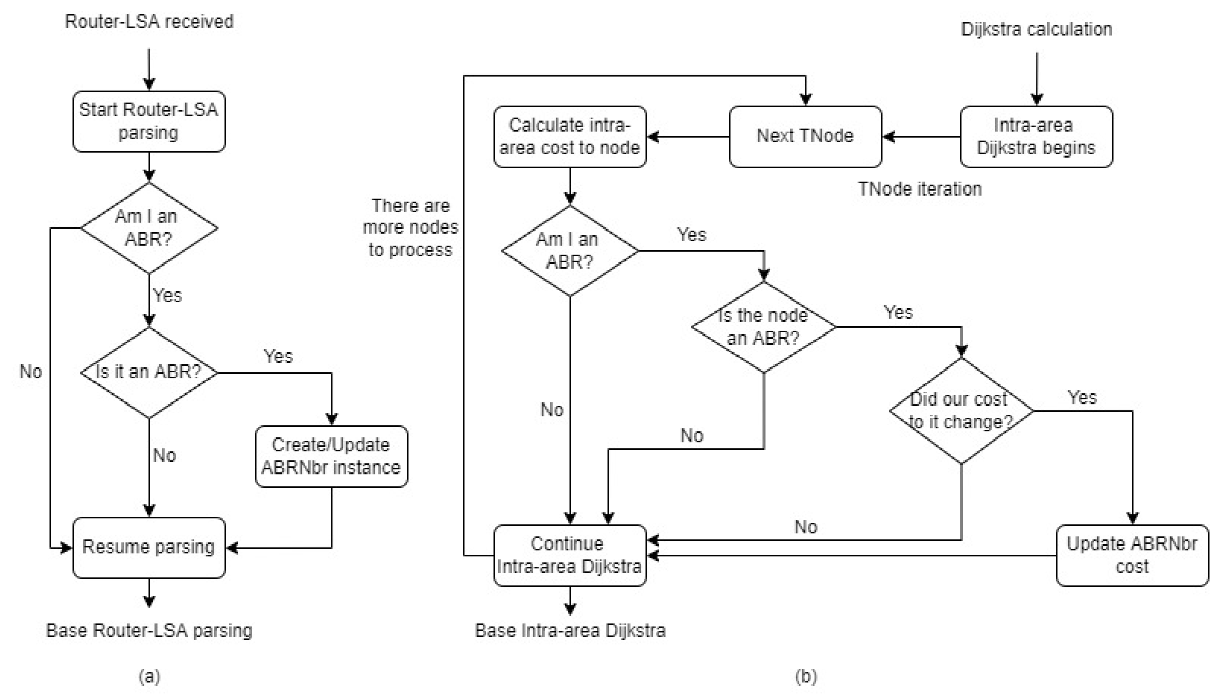


Figure 5.1: Simple representation of how ABRNbr objects are (a) created/reset and (b) assigned a cost

A final routine is then called within the full intra-area calculation, the **OSPF::update_asbrs** routine. This routine will call the **ASBRrte::run_calculation** routine for all the currently known ASBR routing table entries. This routine will then (re)calculate the best path to each ASBR. If available, the best intra-area path will always be stored (path and cost), even if it is not used for routing. If we find an intra-area path, by the end of the routine we as for the Prefix-LSAs. Granted (i) we are an ABR and (ii) changes are detected from previous calculations, (iii) if our first ABR-LSA has been advertised, we advertise the ASBR's corresponding ASBR-LSA. Otherwise, we directly advertise its ASBR-Summary-LSA.

We found the need to separate the storage of the existing intra-area paths and costs from the ones that will be used in forwarding. This extension requires that differentiation to properly generate the Overlay LSAs, even if the router opts for an inter-area route to a destination. We did this by changing the **RTE::new_intra** routine, where we associate a new intra-area path and cost to a given destination. If no changes were applied, this routine would simply replace the current path with an intra-area path, if an inter-area route was being used.

When reaching the decision point on whether to advertise the Prefix-LSAs/Network-Summary-LSAs and the ASBR-LSA/ASBR-Summary-LSAs, a flag is set in their correspondent routing table entries, **RTE::adv_overlay**, indicating these entries are ready to be advertised to the ABR Overlay. This is done as the Overlay LSAs might not be generated at that time, and instead marked to be generated whenever possible.

5.2.1.B Building the Overlay LSAs

ABRs will most certainly originate several Prefix-LSAs and ASBR-LSAs. For this reason, we must properly use the LS ID field to differentiate LSAs of these types advertised by the same ABR. Opaque-LSAs are differentiated from each other through two elements, (i) their advertising router's ID and (ii) the LS ID field, which in this case depicts the combination of Opaque Type and Opaque ID. Figure 5.2 shows the identifying elements of the Overlay LSAs. Since each ABR will only originate one ABR-LSA, we only need to consider the Opaque Type, the one chosen by us to define the ABR-LSA, when building it. This is not true for the other two LSAs. We must also decide how the Opaque ID portion of the field will differentiate the LSAs among each other. As only 24 bits remain available after the Opaque Type is defined, we decided to implement a simple unique ID (each router assigns locally a unique ID to each existing routing table entry), that would not be assigned to the Overlay LSAs but to the routing table entries, as a new parameter of the **INrte** class, named **INrte::uid**. This unique ID increments by one with each new routing table entry added by the router. Although this could eventually present issues, by limiting the number of possible destinations that would be covered by this unique ID, these issues were not addressed in this work.

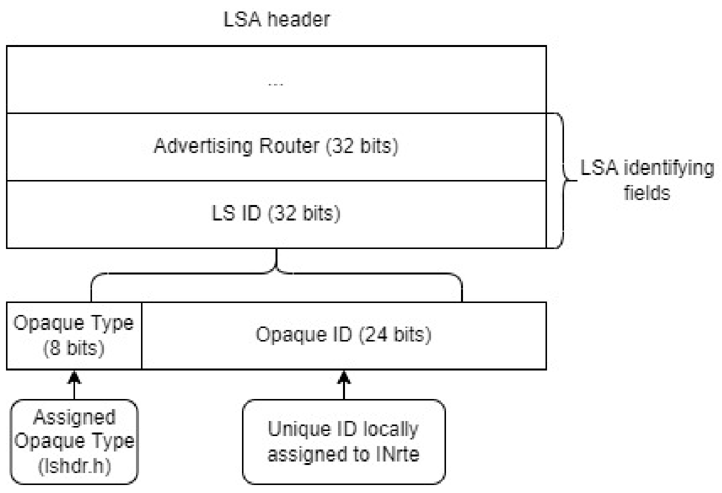

LSA header
...
Advertising Router (32 bits)
LS ID (32 bits)
LSA identifying fields
Opaque Type (8 bits)
Opaque ID (24 bits)
Assigned Opaque Type (lshdr.h)
Unique ID locally assigned to INrte

Figure 5.2: Summary of the identifying elements of Overlay LSAs

The process of building the body of Overlay LSAs is simple, since the necessary verifications are done before calling the routines to originate the LSAs. The construction of the Prefix-LSAs and ASBR-LSAs is the most straightforward. This consists of (i) defining what the LS ID field will be (the concatenation of the Opaque Type defined for the LSA with the unique ID assigned to the routing table entry described) and (ii) building the fixed length LSA body, by assigning a value to each of the fields, taken from the corresponding routing table entry.

Building the ABR-LSA is not as simple, not only because the body length is not fixed but also because some ABR neighbors might be reachable through multiple areas. Building the ABR-LSA body consists in only adding the best path to each ABR neighbor, where each possible path is represented by an object of the **ABRNbr** class. To help with this, we opted to include a flag (**ABRNbr::use_in_lsa**) that indicates whether that instance should be considered in the LSA. When building the ABR-LSA body, we keep track of how many neighbors are considered in the LSA. We start by clearing the **use_in_lsa** flag in all the ABRNbr objects, so that old instances aren't incorrectly considered. We then iterate over all the ABRNbr objects, performing the following set of steps for each object to determine whether to consider it for the LSA body:

1. If a cost is not assigned, we skip to the next instance, and repeat this step for the next ABRNbr object (this is not likely but can occur if a new ABR neighbor discovery causes the scheduling of the ABR-LSA origination and a new Router-LSA arrives from a neighboring ABR before a new intra-area calculation takes place and after the ABR-LSA origination process begins);

2. If the cost is assigned, we check if another object referring to this ABR is already being considered in the LSA body; If not, we move to step 3, otherwise we go to step 4;

3. Since no other objects referring to this ABR are being considered yet, we set the **use_in_lsa** flag to *true*, increase the counter of the number of neighbors considered by one and move on to the next object.

4. If there is an object for the same ABR already being considered, we compare the costs assigned to these objects; the object presenting the lower cost will have its **use_in_lsa** flag set to *true* and the other one will have it set to *false*; If there is a draw, the new object is used; we then repeat the cycle for the next object, if any exists.

After deciding which objects of the ABRNbr class will be used to build the LSA, we iterate over them, adding their Router ID and cost to the LSA body. We then calculate the body length by multiplying the number of added neighbors by the size each of them uses in the body. After building and sending out the LSA, we signal that our first ABR-LSA has been flooded to the network, by setting a global flag, **OSPF::first_abrLSA_sent**, to true. This flag is used in other areas of the code to determine if this event has occurred. This serves mainly to determine whether we should advertise the Prefix-LSAs and ASBR-LSAs right away, when their needed information has been determined (see Section 5.2.1.A). If this flag had not yet been set to true, we also schedule the origination of all Prefix-LSAs and ASBR-LSAs that are ready to be advertised.

The (re)origination of the ABR-LSAs is scheduled when updates to our ABR neighbors are detected, these being (i) a change in the shortest intra-area path cost to it, (ii) the recognition of a new ABR neighbor, or (iii) when an ABR neighbor is deemed unreachable within the area in which they are neighbors.

For all the three Overlay LSAs, a general routine, **OSPF::opq_orig**, is called after building the LSA body, to build and advertise the Opaque-LSA. Among others, this routine takes as input parameters the *LS Type*, to determine the flooding scope of the LSA, the *LS ID*, to correctly identify it, and the *LSA body* and its corresponding **body length**, to correctly store the LSA body as part of the Opaque-LSA.

5.2.2 Parsing the Overlay-LSAs

When installing a new LSA into the database, either received or generated, the protocol will parse that LSA. When an Opaque-LSA is received or generated, the **OSPF::ParseLSA** routine is called at the end of the process of installing the LSA into the database (via the **OSPF::AddLSA** routine). This **ParseLSA** routine will call the Opaque-LSA specific parse routine, **opqLSA::parse**. To parse the Overlay LSAs, we verify (i) that our router is an ABR and (ii) that the Opaque Type corresponds to one of the assigned types for the Overlay LSAs. Meeting these two conditions, the **opqLSA::parse** routine forwards the LSA to the newly created **opqLSA::parse_overlay_lsa** routine, to parse the LSA based on its Opaque Type.

All three Overlay LSAs use new classes created to store the relevant information needed for the Overlay routing calculations. These are the **overlayAbrLSA**, **overlayPrefixLSA** and **overlayAsbrLSA**, for the ABR-LSA, Prefix-LSA and ASBR-LSA, respectively. These classes keep a pointer to the Opaque-LSA that generated them, since it was not possible to directly inherit these classes from the **opqLSA** class. Objects of these classes are stored in global structures, the **AVLtrees** (see Section 3.1.2), with a separate tree existing for each class. Within this structure, the class objects need a way to differentiate themselves, mainly for searching for a specific item within the tree. Each class uses different indexes for this. **ABR-LSAs** use the *advertising router*, **Prefix-LSAs** use a combination of the *advertising router* with the advertised *subnet address*, and the **ASBR-LSA** uses the combination of the *advertising router* with the *router ID* of the advertised ASBR.

Inside the **parse_overlay_lsa** routine, we verify the Opaque Type, and from there go into the LSA specific portion of the routine. Inside the specific sub-routine, we verify, through the identifying attributes of their corresponding classes, if an object of that LSA class is already stored. If so, we use the stored object for the next steps. Otherwise, we create a new object of that class. We then store a pointer to this object in the Opaque-LSA class object being parsed, to link the Opaque-LSA object to the specific Overlay-LSA class object. This is the first step, common to all Overlay LSAs. Beyond that, the parsing routine differs greatly between the ABR-LSA and both the Prefix-LSA and ASBR-LSA. As this routine follows the same steps for both Prefix-LSAs and ASBR-LSAs, only differing in the structures addressed, we will only describe it for the ABR-LSA and Prefix-LSA. Figure 5.3 shows an overview of the parsing of the ABR-LSA and Prefix-LSA.

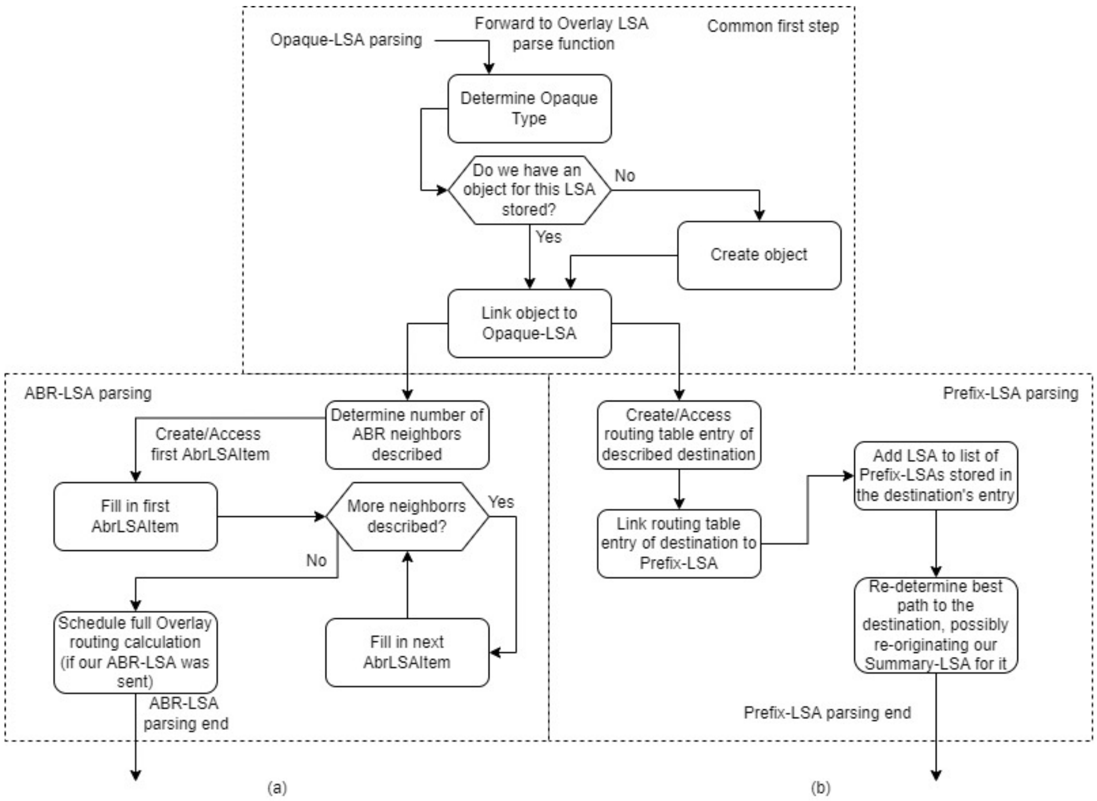

Opaque-LSA parsing
Forward to Overlay LSA parse function
Common first step
Determine Opaque Type
Do we have an object for this LSA stored?
No
Yes
Create object
Link object to Opaque-LSA
ABR-LSA parsing
Determine number of ABR neighbors described
Create/Access first AbrLSAItem
Fill in first AbrLSAItem
More neighborrs described?
Yes
No
Schedule full Overlay routing calculation (if our ABR-LSA was sent)
Fill in next AbrLSAItem
ABR-LSA parsing end
Prefix-LSA parsing
Create/Access routing table entry of described destination
Add LSA to list of Prefix-LSAs stored in the destination's entry
Link routing table entry of destination to Prefix-LSA
Re-determine best path to the destination, possibly re-originating our Summary-LSA for it
Prefix-LSA parsing end
(a)
(b)

Figure 5.3: Parsing of the (a) ABR-LSA and (b) Prefix-LSA

5.2.2.A ABR-LSA parsing

For the ABR-LSA, after the common first step, we calculate and store the number of neighbors being described, by dividing the length of the LSA body by the size that each described neighbor uses. We then proceed to locally store the descriptions of each neighbor advertised in the LSA.

For this, we have a simple class, the **AbrLSAItem**, of which each object stores the description of one neighbor ABR and links itself, through pointers, to the next and previous neighbors described in the LSA, if they exist, creating a linked list. Besides links to list elements, each object of this class stores the Router ID and intra-area cost to the described neighbor ABR. Each ABR-LSA will have an independent list formed with objects of this class. This is done by storing a pointer to the first item of the list in the **overlayAbrLSA** object.

If the current overlayAbrLSA object does not yet have an item linked to it, we create and link the first **AbrLSAItem** to it, assigning it the values for the *cost* and *router ID* of the first ABR neighbor described in the LSA being parsed. If it has a list element linked to it from previous parsing routines, we instead update that first list item's stored information. This process is then repeated (using the first AbrLSAItem as the root instead of the overlayAbrLSA) for following ABR neighbor descriptions, if any, and until no more ABR neighbors are described in the LSA. Finally, if our own ABR-LSA has already been advertised to the AS, we schedule a full overlay calculation. Otherwise, the full calculation will be scheduled when our ABR-LSA is generated.

5.2.2.B Prefix-LSA parsing

For the Prefix-LSA, after the first common step, we access the advertised LSA body to store it and to either create and link or simply link (if already created) the corresponding routing table entry to that overlayPrefixLSA instance.

Each existing IP routing table entry (**INrte** class, and one per destination) stores a linked list of all the Prefix-LSAs, received from different ABRs, referring to itself, to later iterate through all the possible Prefix choices for that destination. As we process the new/updated Prefix-LSA, we link the current overlayPrefixLSA instance to its corresponding routing table entry's list of Prefix-LSAs.

Finally, if (i) our first ABR-LSA has been advertised and (ii) our first full overlay routing calculation has been performed, we (re)compare all the Prefix-LSAs linked to the destination at hand, re-originating the Network-Summary-LSA for that destination if a new and better path is chosen.

5.2.3 Unparsing the Overlay-LSAs

There are some cases in which we might want to delete our generated Overlay LSAs. If all of an ABR's neighbors become unreachable, it will delete (age prematurely) its ABR-LSA from the database and reset the **OSPF::first_abrLSA_sent** flag, changing its behaviour to as if it is the only ABR in the network. As for Prefix-LSAs and ASBR-LSAs, these will be prematurely aged when an intra-area route to a destination is no longer available to the advertising router.

When we delete an LSA from the database, we must unparse it before fully deleting it from the database. In the case of Overlay LSAs, unparsing essentially means to remove any pointers between objects (e.g., the pointer to the Prefix-LSA stored within the routing table entry) and, if necessary, re-determine the best possible path to the corresponding destination, for Prefix-LSAs. The **UnParseLSA** routine is called before an LSA deletion, from which the LSA specific unparse routine is called. Similarly to the **opqLSA::parse**, a reverse **opqLSA::unparse** routine redirects the LSA to a newly created **opqLSA::unparse_overlay_lsa** routine.

This new routine operates like its opposite, filtering the LSA based on its Opaque Type before entering the LSA specific processing. The unparsing of overlayAbrLSA objects clears the fields, such as the number of neighbors and their respective information, and breaks the link between that object and its corresponding Opaque-LSA.

The unparsing of Prefix-LSAs and ASBR-LSAs is again very similar, and only the Prefix-LSA is addressed here. When we unparse a Prefix-LSA, we remove it from the list of overlayPrefixLSA objects stored in its corresponding routing table entry. We must also be aware that, if the Prefix-LSA being deleted is the one currently being considered for the destination's Network-Summary-LSA, we must re-determine which of the remaining Prefix-LSAs should replace it. If the Prefix-LSA being deleted was in fact the one our ABR used to generate the Network-Summary-LSA for that destination, we recalculate what Prefix-LSA should be considered to advertise the corresponding Network-Summary-LSA, possibly re-originating that LSA, after removing the Prefix-LSA from the list in the routing table entry. Finally, we clear the links between the Prefix-LSA and its corresponding routing table entry and between the Prefix-LSA and the Opaque-LSA from which it originated.

5.3 The Full Overlay Routing Calculation

With most of the processing already done for Overlay LSAs, it becomes significantly easier to implement the remaining processes relative to the full overlay routing calculation. This calculation is scheduled before it is performed, similarly to the intra-area full calculation. To trigger both these calculations, a global timer set to trigger every second is run to find if the calculations are scheduled. Only ABRs can access the full overlay calculation routine, and this same routine is limited (like the intra-area full calculation) to a rate of one run per second, imposed by the timing function from which it is called. It would have been possible to implement an event-driven alternative to this, performing the calculation as soon as possible, but we opted for this option, to aggregate possible consecutive triggers for the calculation within a short time. An event-driven alternative could slow the protocol down, as this can be a heavy operation, since it processes all the destinations and ABRs existing in the network. The full overlay calculation is composed of three main routines:

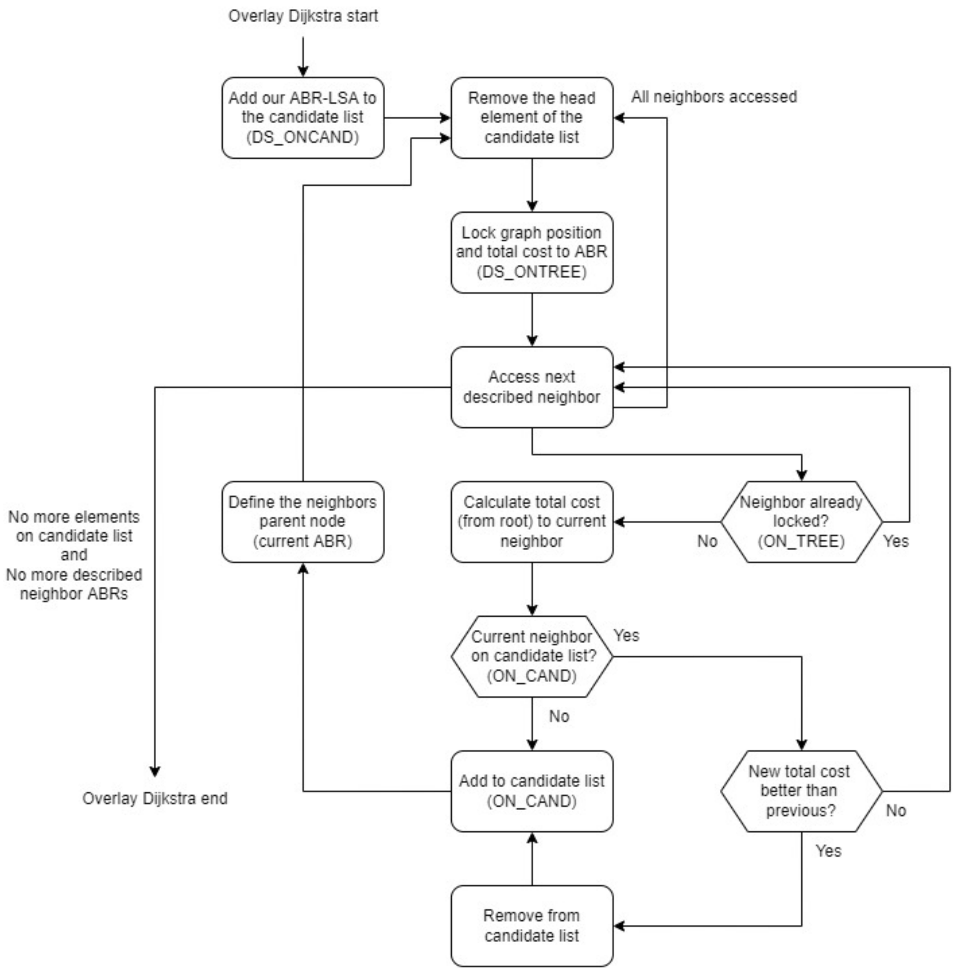
- The Overlay Dijkstra calculation, to build the ABR Overlay Graph;
- A simple routine to determine the next hop ABRs to reach each ABR included in the overlay;
- The scan of Prefix-LSAs and ASBR-LSAs generated for each destination, to determine the globally optimal path to it.

5.3.1 Building the Overlay Graph

To build The ABR overlay graph, we implemented a Dijkstra Shortest Path First algorithm, similar to what is done for the intra-area routing calculations. As nodes, this algorithm uses the stored **overlayAbrLSA** objects, each representing a different ABR.

This Dijkstra algorithm uses two structures, the **PriQ** class, or Priority Queue, and the **PriQElt** class, or Priority Queue Element. The **overlayAbrLSA** class inherits from the latter. The queue elements have one or more (in this case, one suffices) costs assigned (designated **PriQElt::cost0**), which in this case depicts the current cost to an ABR. When adding an element to the queue, it will be placed so that the element with the lowest cost is placed at the head of the queue. We call this priority queue the candidate list. There are three possible states for each ABR, DS_UNINIT (meaning it has not yet been added to the candidate list), DS_ONCAND (the element is placed in the candidate list) and DS_ONTREE (the element has been placed in the overlay graph and does not require further processing). This state is stored inside a variable called **t_state**, as part of the overlayAbrLSA class.

Figure 5.4 shows a representation of how the Overlay Dijkstra algorithm proceeds.



Overlay Dijkstra start
Add our ABR-LSA to the candidate list (DS_ONCAND)
Remove the head element of the candidate list
All neighbors accessed
Lock graph position and total cost to ABR (DS_ONTREE)
Access next described neighbor
No more elements on candidate list and No more described neighbor ABRs
Define the neighbors parent node (current ABR)
Calculate total cost (from root) to current neighbor
Neighbor already locked? (ON_TREE)
No
Yes
Current neighbor on candidate list? (ON_CAND)
Yes
No
Overlay Dijkstra end
Add to candidate list (ON_CAND)
New total cost better than previous?
No
Yes
Remove from candidate list

Figure 5.4: Graphic representation of the Overlay Dijkstra algorithm

We begin the Dijkstra calculation by adding our own ABR-LSA to the candidate list, changing its state to DS_ONCAND and assigning it a cost0 of 0, as it is the root for the calculation. All other ABR-LSAs remain in the DS_UNINIT state. We then begin iterating through the candidate list until no elements remain. Each iteration, we remove the head element from the queue and process its information:

1. Before processing the current ABR-LSA, we change its state to DS_ONTREE, locking its place in

the overlay graph, and assign it the cost (in a **overlayAbrLSA::cost** variable) it currently holds in cost0. This represents the total cost to reach this ABR;

2. We then iterate through the ABR neighbors described in the LSA, stored as AbrLSAItem objects (see Section 5.2.2.A). For each neighbor, and until none remain:

   (a) We access the corresponding overlayAbrLSA object of the current neighbor;

   (b) If this neighbor is already in the DS_ONTREE state, we continue to the next one;

   (c) Otherwise, we assign a total cost, resulting from the sum of the cost assigned to the current ABR-LSA (cost0 in step 1) with the cost advertised in the LSA for this neighbor;

   (d) If the current neighbor is already placed in the candidate list (DS_ONCAND state), we must determine if this new cost is better than the one it already had assigned to it. If so, we remove the neighbor from the candidate list, to later be re-added. Otherwise, we continue to the next described neighbor;

   (e) If the neighbor being processed is still in the DS_UNINIT state, or if its new cost is better than a previous one, it will be (re)added to the candidate list, assigning its cost0 variable the cost calculated in step 2.c and changing its state to DS_ONCAND.

   (f) When adding a neighbor to the candidate list, we also keep track of which ABR is the parent node of the added neighbor (which ABR added it to the list), which will be the current ABR-LSA, whose neighbors are currently being described. This information is stored in the **overlayAbrLSA::t_parent** variable, and is used to determine our next hop ABR to each ABR.

With the Overlay Dijkstra algorithm complete, the overlay graph is built. We then go on to define our next hop ABRs to each of the ABRs described in the overlay graph. This is done to determine the physical next hop that must be taken to each destination. This is determined from the next physical hop to the next hop ABR, which is determined during the intra-area full calculation. Although we will store the Overlay information, we can still only fully decide on the paths that are taken inside the areas we attach to, since ABRs do not have information about the intra-area topologies of other areas.

The process of setting the next hop ABRs to each ABR is simple, as for each ABR-LSA stored, we simply trace back the parent nodes up to the ABR which has our router as its parent. Essentially, this process informs the ABRs which of their own ABR neighbors should be used as a gateway to reach other ABRs.

5.3.2 Calculating the Overlay Routes

As the final step to complete the full Overlay calculation, we scan all the destinations advertised in Prefix-LSAs and ASBR-LSAs by ABRs across the AS. We will only describe the process of scanning the destinations advertised in Prefix-LSAs, since the process is almost identical for ASBRs, only differing in the type of routing table entry and LSA used.

For this we call the **OSPF::prefix_scan** routine. This routine iterates through the routing table entries for all stored destinations. For each routing table entry, we call the **OSPF::adv_best_prefix** routine. In this routine, we determine which Prefix-LSA, from those advertising this destination, is the best option for our ABR to originate the corresponding Summary-LSA from.

- First, we verify if the destination has Prefix-LSAs linked to it, through the **INrte::prefixes** field of the IP routing table entry class, storing a list of all the Prefix-LSAs currently advertising it. If no Prefix-LSAs are associated with this destination yet, we skip to the next entry;
- Otherwise, we keep track of the best total cost advertised for the destination, as well as which Prefix-LSA (and ABR) is advertising it. For this, we iterate through all the Prefix-LSAs linked to the current routing table entry. For each one and until none are left:
  - We access the overlayAbrLSA instance of the ABR advertising the current Prefix;
  - We calculate the total cost to the destination, as the sum of the cost to the ABR (calculated during the Overlay Dijkstra algorithm) with the cost advertised in the Prefix-LSA;
  - If this cost is better than the current best cost, we update the current best cost, the Prefix-LSA that originated it, and the ABR advertising it;
- After verifying all the Prefix-LSAs, we will update our own chosen route to the destination and advertise its corresponding Network-Summary-LSA if at least one suitable candidate was found and if (at least one of the following):
  - The destination was not yet advertised in a Network-Summary-LSA;
  - The Prefix-LSA used to advertise the Network-Summary-LSA has changed;
  - The cost advertised in the previous Network-Summary-LSA differs (not necessarily for the best) from the newly determined best cost;

To update our own route to the destination we simply update the routing table entry, in which we (i) update the cost, with the new best cost, (ii) the type of route, depending on whether we choose our own Prefix-LSA to advertise the destination or the cost matches our intra-area cost, and (iii) the path taken. The path to the destination is either the intra-area path to the destination or the path to the next hop ABR to reach that destination, depending on the type of route used. This update is done inside a routine created for that purpose, the **OSPF::update_path_overlay**, receiving as input the routing table entry, the ABR that originated the chosen Prefix-LSA and the best cost determined for that destination.

Finally, we will address the installation of the determined paths into the OS's routing table. This is always done at the end of full intra-area and Overlay routing calculations, for all routes presenting changes. Occasionally, when only one specific route is processed, its need for an OS update is checked. It is done by calling a system function and passing the relevant parameters: (i) the subnet address and (ii) mask, (iii) the cost assigned to it, and (iv) the path taken to reach it, this last one being limited to the intra-area portion of the path.

5.4 Injecting the routing information into the attached areas

If the conditions have been met to update the path from the ABR to a destination based on the selected Prefix-LSA for it, described in Section 5.3.2, before exiting the **OSPF::adv_best_prefix** routine, we also generate the Network-Summary-LSA relative to that destination. The origination of Network-Summary-LSAs has been subject to a few changes, relatively to the base implementation. A couple of verifications prevented the LSAs from being advertised inside the areas containing the destinations, which have also been removed. These were (i) the comparison of the router's area with the destination area and (ii) the verification of the path to the destination. If the router's area was the same as the destination's, or if the remaining path to the destination was fully contained within the destination area, the LSA would not be advertised into that area.

The Network-Summary-LSA is then originated and advertised into the areas to which the ABRs directly attach to. Upon receiving these LSAs, ABRs not process them for any inter-area route calculations. The area-internal routers, on the other hand, may only process the Network-Summary-LSAs, as they already did, and not the Overlay LSAs.

Area-internal routers must now consider that inter-area approaches for all routes, and for that we defined that instead of resorting to the Network-Summary-LSAs only when an intra-area path is not available, it will always be done, replacing the intra-area path if it presents a better cost to the destination, independently of the router's or the destination's areas. To accommodate this, slight changes were required to the inter-area path calculation for each destination (represented by the **INrte::run_inter_area** routine). These consist of (i) preventing the function from exiting if we already had an intra-area route available to the destination and (ii) to compare the calculated inter-area route with the possibly existing intra-area route, in order to always choose the globally best route.

Essentially, these all the main differences from the base version that affect the area-internal routers. This means that although it is almost there, this extension (at least for this specific implementation) is not completely transparent to area-internal routers, as they see slight changes to their end of the implementation.

6 Evaluation

Contents

To prove the correct operation of this extension, we submitted the developed implementation to a number of test cases, with a predictable outcome, and with a topology complex enough to cover all the aspects we aim to address. This chapter will be divided in two main sections, one for functional tests to our extension, and another for convergence time tests, in which we will determine and compare our solution convergence times when reacting to changes to those of the base version used for this implementation (see Chapter 3). As we did not develop a full implementation of the OSPFv2 protocol, but only the proposed extension over an existing implementation, for this work we will assume the base version of the code to be correctly implemented and functional. This refers to all the pre-existing mechanisms present in the code and described throughout Chapters 2 and 3, e.g., the Hello Protocol, or the Initial Database Synchronization.

Our extended version of the code, as well as its base version, was deployed in Docker [10] containers, and these containers were run inside a GNS3 [11] Virtual Machine (VM) instance. We rely solely on these containers to represent the routers in our tests, since we could not achieve interoperability with the Cisco Router Images available in GNS3. Docker containers use a static interface configuration, made available by GNS3. Appendix B contains a set of instructions on how to install and configure the Docker containers with the implementations used. We also rely on a monitoring tool accompanying the base implementation as well as on Wireshark [12] packet captures to gather our results.

For our functional tests, we used the topology shown in figure 6.1. This topology is composed of three areas, Area 1 (with an Area ID of 0.0.0.1), Area 2 (Area ID 0.0.0.2) and Area 3 (Area ID 0.0.0.3). There is a total of seven routers, each one running in a separate Docker container. Each router is identified by its unique Router ID, assigned in the form 0.0.0.x and from now on designated as Rx (for simplicity), with x being the router number assigned to them in GNS3, as show in the figure. R1, R5 and R7 are area-internal routers, belonging to Areas 1, 2 and 3, respectively. R2, R3, R4 and R6 are all ABRs. R2 attaches to Areas 1 and 2, R3 and R4 both attach to Areas 1 and 3, and R6 attaches to Areas 2 and 3. There is a total of 9 subnets configured. Subnets 11.0.0.0/24, 12.0.0.0/24 and 13.0.0.0/24 belong to Area 1, subnets 21.0.0.0/24 and 22.0.0.0/24 belong to Area 2 and subnets 30.0.0.0/24, 32.0.0.0/24, 33.0.0.0/24 and 34.0.0.0/24 belong to Area 3. By default, all outgoing interfaces have been assigned a cost of 10. Any change to these costs for our tests will be indicated. The connections between the routers are as presented in the figure, with the interface addresses being formed as the subnet address to which they connect, ending with the router number.

In these tests, we purposely avoided the use of a Backbone area (Area ID 0.0.0.0), and created a cycle in the network. Such cycle will most likely cause problems in a plain OSPF implementation. For our tests, we will not cover the use of ASBRs or ASBR-LSAs, as the ASBR-LSAs function is extremely similar to that of the Prefix-LSAs. For that reason, we shall refer to the Network-Summary-LSAs simply as Summary-LSAs throughout this chapter.

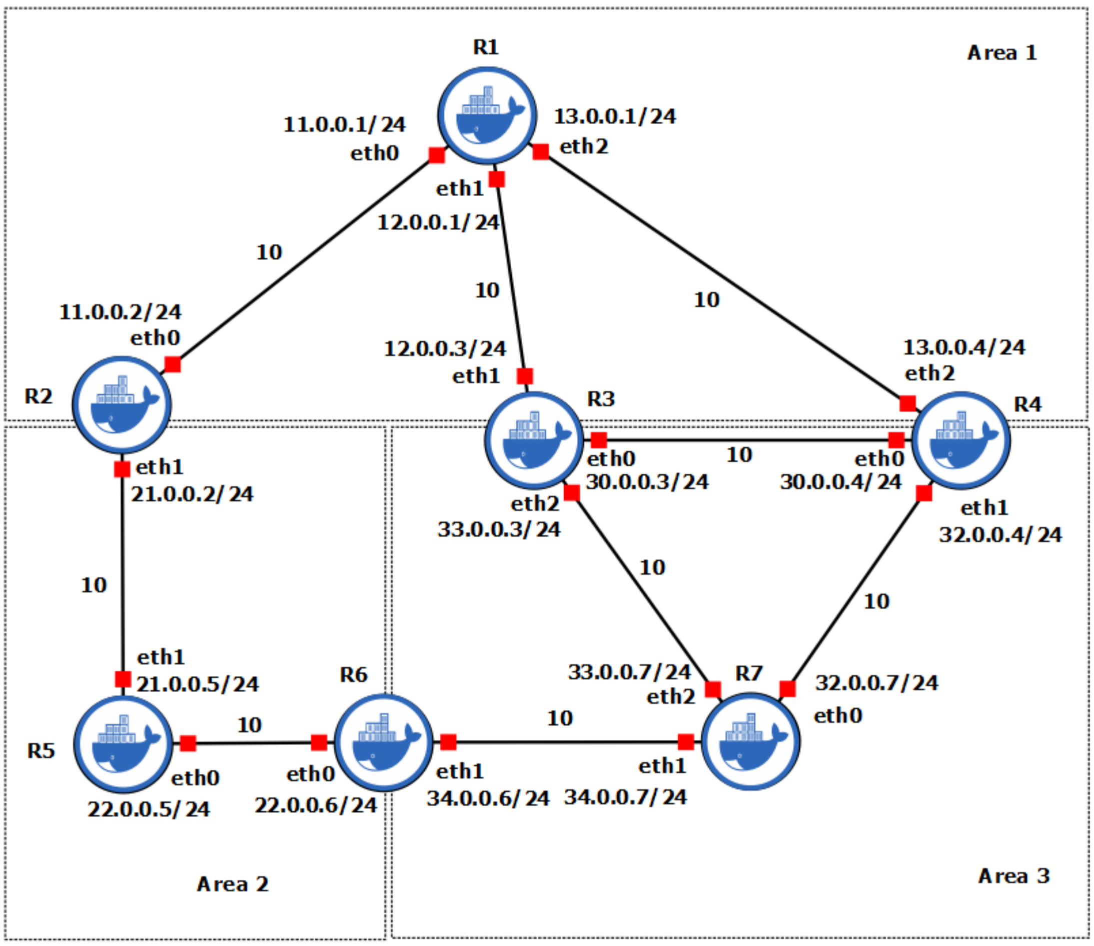
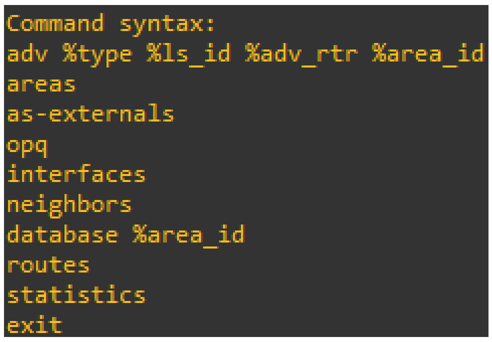


Figure 6.1: Multi-area cyclical network topology

6.1 Functional tests

This section includes tests to prove the correct operation of our developed extension to the base version of the protocol implementation. First, we will show the database portion containing the originated Overlay LSAs for the given topology and how the Summary-LSAs are now being advertised into the areas where the destination they advertise is contained. Then, we will demonstrate that the current OSPF limitations have been overcome. Finally, we will leave a set of three tests, to illustrate the reaction to changes in the topology, by both ABRs and area-internal routers. In here we will include changes to costs, the removal of a link between two routers and the removal of an ABR.

To support our tests, and provide visual results, we used data gathered from a monitoring tool included with the base implementation (**ospfd_mon**), which communicates directly with the running protocol to obtain information, and Wireshark packet captures. To see the representation of the Overlay LSAs, simple additions were made to the monitoring tool so that these LSAs would present a structured body, instead of a string of bytes. A list of the available commands within this tool is shown in figure 6.2.



```
Command syntax:
adv %type %ls_id %adv_rtr %area_id
areas
as-externals
opq
interfaces
neighbors
database %area_id
routes
statistics
exit
```

Figure 6.2: Summary of the commands available in the ospfd_mon monitoring tool

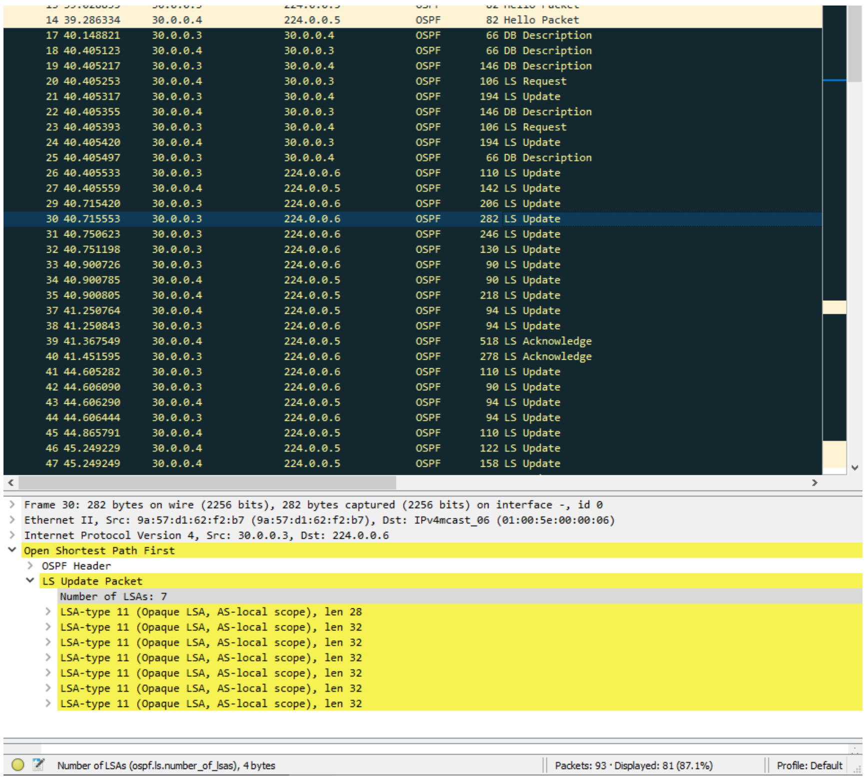
From these commands we highlight three, which we used the most, (i) the *routes* command, that prints out the router's current routing table, (ii) the *opq* command, included by us, to print out the database portion containing the AS-scoped-LSAs (in this case, only the Overlay LSAs) and (iii) the *adv* command, used to print out the contents of a specific LSA, by receiving as additional inputs the LSA's Type, LS ID, Advertising router ID and Area ID. In future figures shown with information gathered from this monitoring tool, the used command will be included in the top of the figure. The parameters of the *adv* command in these figures are the ones described and by that same order.

6.1.1 Changes to Summary-LSA reachability and Overlay LSAs

In this first test, we show how the Summary-LSAs are now advertised into the areas in which the advertised destination is contained, and also the correct origination and flooding of the Overlay LSAs, as well as the contents of their LSA bodies. We performed a single Wireshark capture, over the link that represents sub-network 30.0.0.0/24.

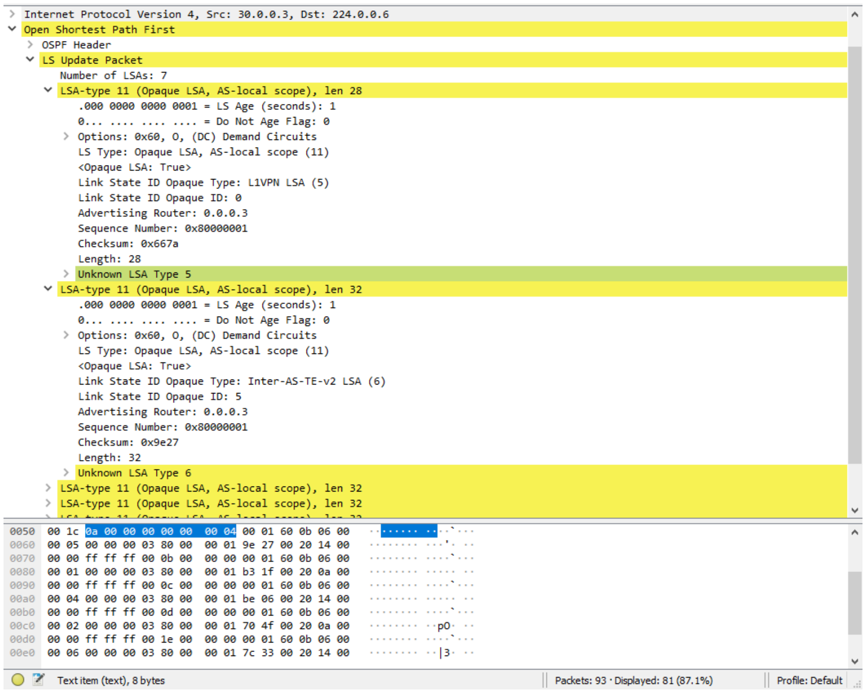
We start by examining the Wireshark packet capture obtained, presented in part in Figure 6.3. This capture was performed in subnet 30.0.0.0/24, but it could have been performed in any other location, since it shows the AS-scoped Overlay LSAs. In figure 6.3.a we see the moment when the first set of Overlay LSAs were originated and flooded to the AS. Because Wireshark does not recognize the Opaque Type described in the LSA header, it signals the packet, more specifically the Opaque Type, as unknown. From analyzing the packet headers presented in figure 6.3.b, we can see that this packet refers to the origination of the ABR-LSA (Opaque Type 5, seen immediately after the LSA header) and several Prefix-LSAs (Opaque Type 6) from R3.

We now move on to the information gathered with the **ospfd_mon** monitoring tool. We start by showing, in figure 6.4 the resulting routing table of routers R1 (6.4.a), R3 (6.4.b), R5 (6.4.c) and R7 (6.4.d). The routes with the SPF Type refer to intra-area paths, while the ones of Type SPFIA refer to the inter-area paths. By analyzing the information contained in the routing tables alongside the provided topology in figure 6.1, we can see that all the presented routes are using the shortest available paths to



```
14 39.286334   30.0.0.4   224.0.0.5   OSPF    82 Hello Packet
17 40.148821   30.0.0.3   30.0.0.4    OSPF    66 DB Description
18 40.405123   30.0.0.4   30.0.0.3    OSPF    66 DB Description
19 40.405217   30.0.0.3   30.0.0.4    OSPF   146 DB Description
20 40.405253   30.0.0.4   30.0.0.3    OSPF   106 LS Request
21 40.405317   30.0.0.3   30.0.0.4    OSPF   194 LS Update
22 40.405355   30.0.0.4   30.0.0.3    OSPF   146 DB Description
23 40.405393   30.0.0.3   30.0.0.4    OSPF   106 LS Request
24 40.405420   30.0.0.4   30.0.0.3    OSPF   194 LS Update
25 40.405497   30.0.0.3   30.0.0.4    OSPF    66 DB Description
26 40.405533   30.0.0.3   224.0.0.6   OSPF   110 LS Update
27 40.405559   30.0.0.4   224.0.0.5   OSPF   142 LS Update
29 40.715420   30.0.0.3   224.0.0.6   OSPF   206 LS Update
30 40.715553   30.0.0.3   224.0.0.6   OSPF   282 LS Update
31 40.750623   30.0.0.3   224.0.0.6   OSPF   246 LS Update
32 40.751198   30.0.0.3   224.0.0.6   OSPF   130 LS Update
33 40.900726   30.0.0.3   224.0.0.6   OSPF    90 LS Update
34 40.900785   30.0.0.4   224.0.0.5   OSPF    90 LS Update
35 40.900805   30.0.0.4   224.0.0.5   OSPF   218 LS Update
37 41.250764   30.0.0.4   224.0.0.5   OSPF    94 LS Update
38 41.250843   30.0.0.3   224.0.0.6   OSPF    94 LS Update
39 41.367549   30.0.0.4   224.0.0.5   OSPF   518 LS Acknowledge
40 41.451595   30.0.0.3   224.0.0.6   OSPF   278 LS Acknowledge
41 44.605282   30.0.0.3   224.0.0.6   OSPF   110 LS Update
42 44.606090   30.0.0.3   224.0.0.6   OSPF    90 LS Update
43 44.606290   30.0.0.4   224.0.0.5   OSPF    94 LS Update
44 44.606444   30.0.0.3   224.0.0.6   OSPF    94 LS Update
45 44.865791   30.0.0.4   224.0.0.5   OSPF   110 LS Update
46 45.249229   30.0.0.4   224.0.0.5   OSPF   122 LS Update
47 45.249249   30.0.0.4   224.0.0.5   OSPF   158 LS Update

> Frame 30: 282 bytes on wire (2256 bits), 282 bytes captured (2256 bits) on interface -, id 0
> Ethernet II, Src: 9a:57:d1:62:f2:b7 (9a:57:d1:62:f2:b7), Dst: IPv4mcast_06 (01:00:5e:00:00:06)
> Internet Protocol Version 4, Src: 30.0.0.3, Dst: 224.0.0.6
v Open Shortest Path First
  > OSPF Header
  v LS Update Packet
      Number of LSAs: 7
    > LSA-type 11 (Opaque LSA, AS-local scope), len 28
    > LSA-type 11 (Opaque LSA, AS-local scope), len 32
    > LSA-type 11 (Opaque LSA, AS-local scope), len 32
    > LSA-type 11 (Opaque LSA, AS-local scope), len 32
    > LSA-type 11 (Opaque LSA, AS-local scope), len 32
    > LSA-type 11 (Opaque LSA, AS-local scope), len 32
    > LSA-type 11 (Opaque LSA, AS-local scope), len 32

Number of LSAs (ospf.ls.number_of_lsas), 4 bytes    Packets: 93 · Displayed: 81 (87.1%)    Profile: Default
```

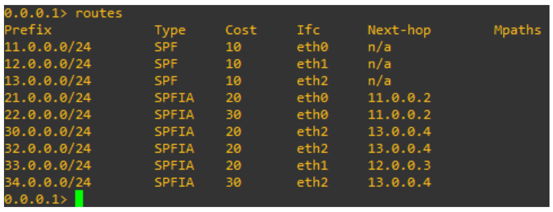
(a)



```
> Internet Protocol Version 4, Src: 30.0.0.3, Dst: 224.0.0.6
v Open Shortest Path First
  > OSPF Header
  v LS Update Packet
      Number of LSAs: 7
    v LSA-type 11 (Opaque LSA, AS-local scope), len 28
        .000 0000 0000 0001 = LS Age (seconds): 1
        0... .... .... .... = Do Not Age Flag: 0
      > Options: 0x60, O, (DC) Demand Circuits
        LS Type: Opaque LSA, AS-local scope (11)
        <Opaque LSA: True>
        Link State ID Opaque Type: L1VPN LSA (5)
        Link State ID Opaque ID: 0
        Advertising Router: 0.0.0.3
        Sequence Number: 0x80000001
        Checksum: 0x667a
        Length: 28
      > Unknown LSA Type 5
    v LSA-type 11 (Opaque LSA, AS-local scope), len 32
        .000 0000 0000 0001 = LS Age (seconds): 1
        0... .... .... .... = Do Not Age Flag: 0
      > Options: 0x60, O, (DC) Demand Circuits
        LS Type: Opaque LSA, AS-local scope (11)
        <Opaque LSA: True>
        Link State ID Opaque Type: Inter-AS-TE-v2 LSA (6)
        Link State ID Opaque ID: 5
        Advertising Router: 0.0.0.3
        Sequence Number: 0x80000001
        Checksum: 0x9e27
        Length: 32
      > Unknown LSA Type 6
    > LSA-type 11 (Opaque LSA, AS-local scope), len 32
    > LSA-type 11 (Opaque LSA, AS-local scope), len 32

0050  00 1c 0a 00 00 00 00 00  00 04 00 01 60 0b 06 00   ········ ····`···
0060  00 05 00 00 00 03 80 00  00 01 9e 27 00 20 14 00   ········ ···'· ··
0070  00 00 ff ff ff 00 0b 00  00 00 00 01 60 0b 06 00   ········ ····`···
0080  00 01 00 00 00 03 80 00  00 01 b3 1f 00 20 0a 00   ········ ····· ··
0090  00 00 ff ff ff 00 0c 00  00 00 00 01 60 0b 06 00   ········ ····`···
00a0  00 04 00 00 00 03 80 00  00 01 be 06 00 20 14 00   ········ ····· ··
00b0  00 00 ff ff ff 00 0d 00  00 00 00 01 60 0b 06 00   ········ ····`···
00c0  00 02 00 00 00 03 80 00  00 01 70 4f 00 20 0a 00   ········ ··pO· ··
00d0  00 00 ff ff ff 00 1e 00  00 00 00 01 60 0b 06 00   ········ ····`···
00e0  00 06 00 00 00 03 80 00  00 01 7c 33 00 20 14 00   ········ ··|3· ··

Text item (text), 8 bytes    Packets: 93 · Displayed: 81 (87.1%)    Profile: Default
```

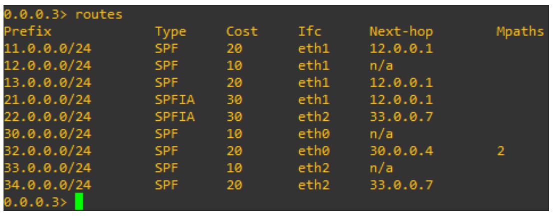
(b)

Figure 6.3: Wireshark packet capture for the first functionality test



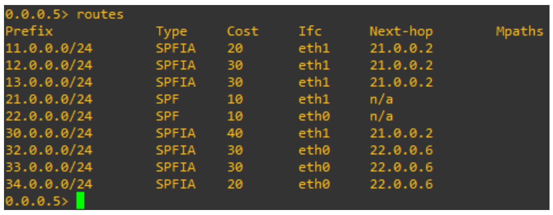
```
0.0.0.1> routes
Prefix              Type     Cost     Ifc      Next-hop          Mpaths
11.0.0.0/24         SPF      10       eth0     n/a
12.0.0.0/24         SPF      10       eth1     n/a
13.0.0.0/24         SPF      10       eth2     n/a
21.0.0.0/24         SPFIA    20       eth0     11.0.0.2
22.0.0.0/24         SPFIA    30       eth0     11.0.0.2
30.0.0.0/24         SPFIA    20       eth2     13.0.0.4
32.0.0.0/24         SPFIA    20       eth2     13.0.0.4
33.0.0.0/24         SPFIA    20       eth1     12.0.0.3
34.0.0.0/24         SPFIA    30       eth2     13.0.0.4
0.0.0.1>
```

(a)



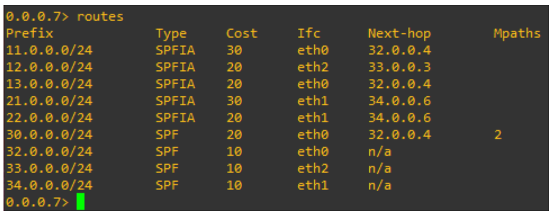
```
0.0.0.3> routes
Prefix              Type     Cost     Ifc      Next-hop          Mpaths
11.0.0.0/24         SPF      20       eth1     12.0.0.1
12.0.0.0/24         SPF      10       eth1     n/a
13.0.0.0/24         SPF      20       eth1     12.0.0.1
21.0.0.0/24         SPFIA    30       eth1     12.0.0.1
22.0.0.0/24         SPFIA    30       eth2     33.0.0.7
30.0.0.0/24         SPF      10       eth0     n/a
32.0.0.0/24         SPF      20       eth0     30.0.0.4          2
33.0.0.0/24         SPF      10       eth2     n/a
34.0.0.0/24         SPF      20       eth2     33.0.0.7
0.0.0.3>
```

(b)



```
0.0.0.5> routes
Prefix              Type     Cost     Ifc      Next-hop          Mpaths
11.0.0.0/24         SPFIA    20       eth1     21.0.0.2
12.0.0.0/24         SPFIA    30       eth1     21.0.0.2
13.0.0.0/24         SPFIA    30       eth1     21.0.0.2
21.0.0.0/24         SPF      10       eth1     n/a
22.0.0.0/24         SPF      10       eth0     n/a
30.0.0.0/24         SPFIA    40       eth1     21.0.0.2
32.0.0.0/24         SPFIA    30       eth0     22.0.0.6
33.0.0.0/24         SPFIA    30       eth0     22.0.0.6
34.0.0.0/24         SPFIA    20       eth0     22.0.0.6
0.0.0.5>
```

(c)



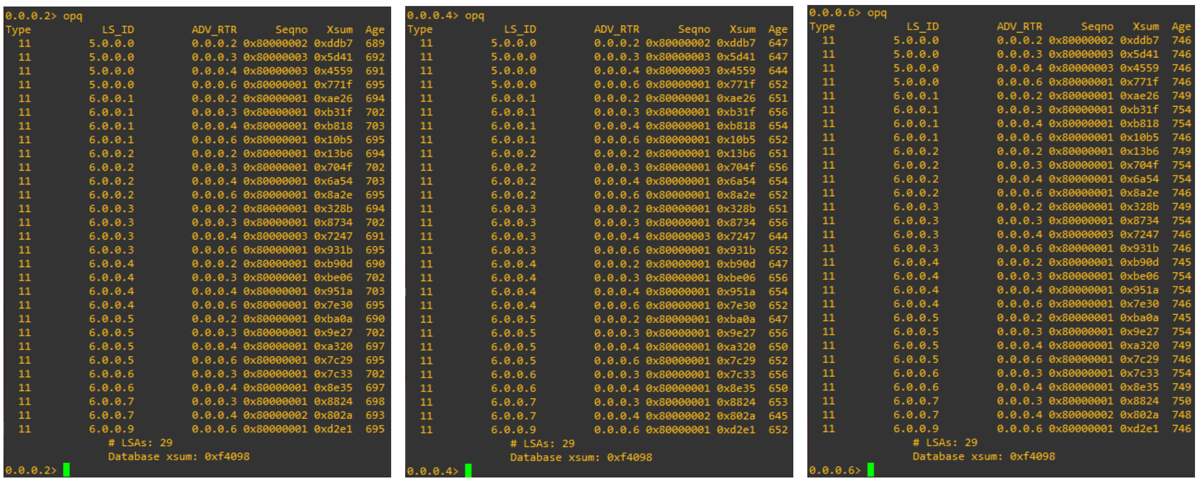
```
0.0.0.7> routes
Prefix              Type     Cost     Ifc      Next-hop          Mpaths
11.0.0.0/24         SPFIA    30       eth0     32.0.0.4
12.0.0.0/24         SPFIA    20       eth2     33.0.0.3
13.0.0.0/24         SPFIA    20       eth0     32.0.0.4
21.0.0.0/24         SPFIA    30       eth1     34.0.0.6
22.0.0.0/24         SPFIA    20       eth1     34.0.0.6
30.0.0.0/24         SPF      20       eth0     32.0.0.4          2
32.0.0.0/24         SPF      10       eth0     n/a
33.0.0.0/24         SPF      10       eth2     n/a
34.0.0.0/24         SPF      10       eth1     n/a
0.0.0.7>
```

(d)

Figure 6.4: Routing tables for routers (a) R1, (b) R3, (c) R5 and (d) R7

the destinations. As an example we can look at the entry presented in R3 to both subnets 21.0.0.0/24 and 22.0.0.0/24. These subnets are contained in Area 2, not directly reachable by R3, and the router takes different paths to each of the routes, using R1 and R7 as next hops, respectively, to these destinations, and presenting a cost of 30 to reach both of them. We did not yet provide the network with better inter-area paths to destinations than their possible intra-area alternatives. However, in our next test, we will change interface costs in a way that some inter-area paths are preferable to the intra-area path to that same destination.

We proceed to show the contents of the portion of the LSDB containing the Overlay LSAs. To confirm that these contents are consistent across the whole network, we have figure 6.5. In this figure we see the global portion of the LSDB, not attached to any specific area, storing the Type 11 (AS-scoped) Opaque-LSAs (in which the Overlay LSAs are advertised), seen by routers R2 (6.5.a), R4 (6.5.b) and R6 (6.5.c). By analyzing the contents of the databases, namely the LSAs' Sequence Numbers, which present the same values for the same LSAs in all routers, we can confirm that the information is consistent among the ABRs. In this database we have one ABR-LSA originated by each of the ABRs, as well as Prefix-LSAs originated by each router equal to the number of destinations reachable by them through intra-area paths. This high volume of Overlay LSAs is expected, as each ABR should advertise exactly one ABR-LSA and one Prefix-LSA for each prefix reachable through an intra-area route. For this topology, this means seven Prefix-LSAs generated by R3, another seven by R4, for all destinations present in Areas 1 and 3, five Prefix-LSAs generated by R2, for the destinations in Areas 1 and 2, and six Prefix-LSAs by R6, for the destinations in Areas 2 and 3. For each ABR we count one more, for its ABR-LSA. This leads



```
0.0.0.2> opq
Type          LS_ID          ADV_RTR       Seqno    Xsum  Age
  11        5.0.0.0          0.0.0.2 0x80000002 0xddb7  689
  11        5.0.0.0          0.0.0.3 0x80000003 0x5d41  692
  11        5.0.0.0          0.0.0.4 0x80000003 0x4559  691
  11        5.0.0.0          0.0.0.6 0x80000001 0x771f  695
  11        6.0.0.1          0.0.0.2 0x80000001 0xae26  694
  11        6.0.0.1          0.0.0.3 0x80000001 0xb31f  702
  11        6.0.0.1          0.0.0.4 0x80000001 0xb818  703
  11        6.0.0.1          0.0.0.6 0x80000001 0x10b5  695
  11        6.0.0.2          0.0.0.2 0x80000001 0x13b6  694
  11        6.0.0.2          0.0.0.3 0x80000001 0x704f  702
  11        6.0.0.2          0.0.0.4 0x80000001 0x6a54  703
  11        6.0.0.2          0.0.0.6 0x80000001 0x8a2e  695
  11        6.0.0.3          0.0.0.2 0x80000001 0x328b  694
  11        6.0.0.3          0.0.0.3 0x80000001 0x8734  702
  11        6.0.0.3          0.0.0.4 0x80000003 0x7247  691
  11        6.0.0.3          0.0.0.6 0x80000001 0x931b  695
  11        6.0.0.4          0.0.0.2 0x80000001 0xb90d  690
  11        6.0.0.4          0.0.0.3 0x80000001 0xbe06  702
  11        6.0.0.4          0.0.0.4 0x80000001 0x951a  703
  11        6.0.0.4          0.0.0.6 0x80000001 0x7e30  695
  11        6.0.0.5          0.0.0.2 0x80000001 0xba0a  690
  11        6.0.0.5          0.0.0.3 0x80000001 0x9e27  702
  11        6.0.0.5          0.0.0.4 0x80000001 0xa320  697
  11        6.0.0.5          0.0.0.6 0x80000001 0x7c29  695
  11        6.0.0.6          0.0.0.3 0x80000001 0x7c33  702
  11        6.0.0.6          0.0.0.4 0x80000001 0x8e35  697
  11        6.0.0.7          0.0.0.3 0x80000001 0x8824  698
  11        6.0.0.7          0.0.0.4 0x80000002 0x802a  693
  11        6.0.0.9          0.0.0.6 0x80000001 0xd2e1  695
               # LSAs: 29
               Database xsum: 0xf4098
0.0.0.2>
```

(a)

```
0.0.0.4> opq
Type          LS_ID          ADV_RTR       Seqno    Xsum  Age
  11        5.0.0.0          0.0.0.2 0x80000002 0xddb7  647
  11        5.0.0.0          0.0.0.3 0x80000003 0x5d41  647
  11        5.0.0.0          0.0.0.4 0x80000003 0x4559  644
  11        5.0.0.0          0.0.0.6 0x80000001 0x771f  652
  11        6.0.0.1          0.0.0.2 0x80000001 0xae26  651
  11        6.0.0.1          0.0.0.3 0x80000001 0xb31f  656
  11        6.0.0.1          0.0.0.4 0x80000001 0xb818  654
  11        6.0.0.1          0.0.0.6 0x80000001 0x10b5  652
  11        6.0.0.2          0.0.0.2 0x80000001 0x13b6  651
  11        6.0.0.2          0.0.0.3 0x80000001 0x704f  656
  11        6.0.0.2          0.0.0.4 0x80000001 0x6a54  654
  11        6.0.0.2          0.0.0.6 0x80000001 0x8a2e  652
  11        6.0.0.3          0.0.0.2 0x80000001 0x328b  651
  11        6.0.0.3          0.0.0.3 0x80000001 0x8734  656
  11        6.0.0.3          0.0.0.4 0x80000003 0x7247  644
  11        6.0.0.3          0.0.0.6 0x80000001 0x931b  652
  11        6.0.0.4          0.0.0.2 0x80000001 0xb90d  647
  11        6.0.0.4          0.0.0.3 0x80000001 0xbe06  656
  11        6.0.0.4          0.0.0.4 0x80000001 0x951a  654
  11        6.0.0.4          0.0.0.6 0x80000001 0x7e30  652
  11        6.0.0.5          0.0.0.2 0x80000001 0xba0a  647
  11        6.0.0.5          0.0.0.3 0x80000001 0x9e27  656
  11        6.0.0.5          0.0.0.4 0x80000001 0xa320  650
  11        6.0.0.5          0.0.0.6 0x80000001 0x7c29  652
  11        6.0.0.6          0.0.0.3 0x80000001 0x7c33  656
  11        6.0.0.6          0.0.0.4 0x80000001 0x8e35  650
  11        6.0.0.7          0.0.0.3 0x80000001 0x8824  653
  11        6.0.0.7          0.0.0.4 0x80000002 0x802a  645
  11        6.0.0.9          0.0.0.6 0x80000001 0xd2e1  652
               # LSAs: 29
               Database xsum: 0xf4098
0.0.0.4>
```

(b)

```
0.0.0.6> opq
Type          LS_ID          ADV_RTR       Seqno    Xsum  Age
  11        5.0.0.0          0.0.0.2 0x80000002 0xddb7  746
  11        5.0.0.0          0.0.0.3 0x80000003 0x5d41  746
  11        5.0.0.0          0.0.0.4 0x80000003 0x4559  746
  11        5.0.0.0          0.0.0.6 0x80000001 0x771f  746
  11        6.0.0.1          0.0.0.2 0x80000001 0xae26  749
  11        6.0.0.1          0.0.0.3 0x80000001 0xb31f  754
  11        6.0.0.1          0.0.0.4 0x80000001 0xb818  754
  11        6.0.0.1          0.0.0.6 0x80000001 0x10b5  746
  11        6.0.0.2          0.0.0.2 0x80000001 0x13b6  749
  11        6.0.0.2          0.0.0.3 0x80000001 0x704f  754
  11        6.0.0.2          0.0.0.4 0x80000001 0x6a54  754
  11        6.0.0.2          0.0.0.6 0x80000001 0x8a2e  746
  11        6.0.0.3          0.0.0.2 0x80000001 0x328b  749
  11        6.0.0.3          0.0.0.3 0x80000001 0x8734  754
  11        6.0.0.3          0.0.0.4 0x80000003 0x7247  746
  11        6.0.0.3          0.0.0.6 0x80000001 0x931b  746
  11        6.0.0.4          0.0.0.2 0x80000001 0xb90d  745
  11        6.0.0.4          0.0.0.3 0x80000001 0xbe06  754
  11        6.0.0.4          0.0.0.4 0x80000001 0x951a  754
  11        6.0.0.4          0.0.0.6 0x80000001 0x7e30  746
  11        6.0.0.5          0.0.0.2 0x80000001 0xba0a  745
  11        6.0.0.5          0.0.0.3 0x80000001 0x9e27  754
  11        6.0.0.5          0.0.0.4 0x80000001 0xa320  749
  11        6.0.0.5          0.0.0.6 0x80000001 0x7c29  746
  11        6.0.0.6          0.0.0.3 0x80000001 0x7c33  754
  11        6.0.0.6          0.0.0.4 0x80000001 0x8e35  749
  11        6.0.0.7          0.0.0.3 0x80000001 0x8824  750
  11        6.0.0.7          0.0.0.4 0x80000002 0x802a  748
  11        6.0.0.9          0.0.0.6 0x80000001 0xd2e1  746
               # LSAs: 29
               Database xsum: 0xf4098
0.0.0.6>
```

(c)

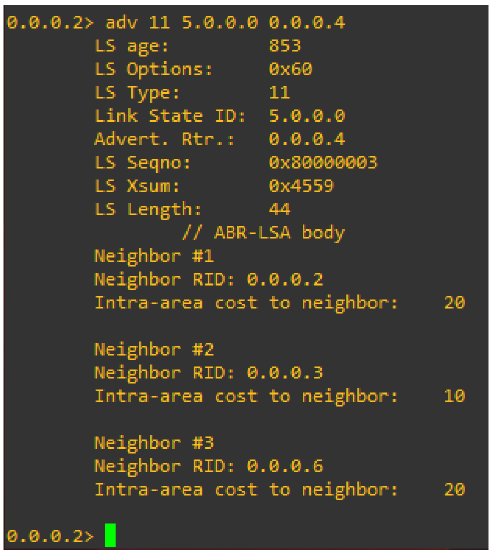
Figure 6.5: Overlay LSAs stored in routers (a) R2 (b) R4 and (c) R6

to a total count of twenty nine Overlay LSAs. We can confirm through the *ADV_RTR* column present in the figure that each ABR generated the number of Overlay LSAs they were supposed to.

We also show in figure 6.6 two of the Overlay LSAs, as seen within the monitoring tool. In this case, we see, in figure 6.6.a, the ABR-LSA originated by R4, seen by R2, showing each of R4's ABR neighbors and the intra-area cost to reach each of them. In figure 6.6.b, we can see the Prefix-LSA, generated by R2 and seen by R6, for the subnet 11.0.0.0/24, clearly showing its subnet address, mask and the intra-area cost to reach it from the originating router.

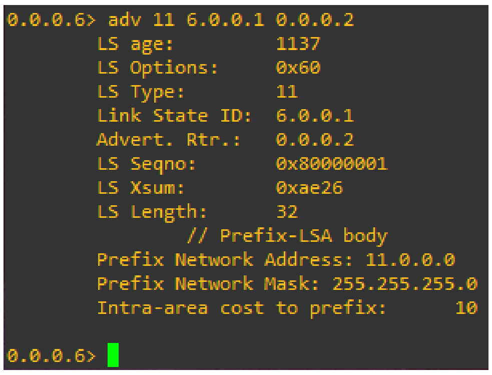
To confirm that Summary-LSAs are now being advertised into the areas to which the destinations belong, we have figure 6.7. Figure 6.7.a shows the database for Area 1, stored in R1, while figure 6.7.b shows the database for Area 2, stored in R5. From analyzing these, we can see that the Summary-LSAs for the destinations present within those areas are represented in their respective databases. In fact, both databases have a number of Summary-LSAs for each destination equal to the number of ABRs attaching to the area they belong to, i.e., each ABR attached to these areas generates one Summary-LSA for each of the network destinations (In this case 9 Summary-LSAs generated per ABR). This shows that all Summary-LSAs are being originated and advertised by the ABRs, providing the necessary information to area-internal routers for them to choose the best possible route to each destination, as we would expect and wish from the extended version of the implementation. In the base implementation, only the prefixes not available in the destination areas would be advertised.

We finally show, in figure 6.8, a small number of examples of the various Summary-LSAs originated by the ABRs. These Summary-LSAs are being advertised with the correct information, this being the globally shortest path cost to the destinations advertised. In figure 6.8.a, we see the Summary-LSA



```
0.0.0.2> adv 11 5.0.0.0 0.0.0.4
        LS age:         853
        LS Options:     0x60
        LS Type:        11
        Link State ID:  5.0.0.0
        Advert. Rtr.:   0.0.0.4
        LS Seqno:       0x80000003
        LS Xsum:        0x4559
        LS Length:      44
                // ABR-LSA body
        Neighbor #1
        Neighbor RID: 0.0.0.2
        Intra-area cost to neighbor:    20

        Neighbor #2
        Neighbor RID: 0.0.0.3
        Intra-area cost to neighbor:    10

        Neighbor #3
        Neighbor RID: 0.0.0.6
        Intra-area cost to neighbor:    20

0.0.0.2>
```

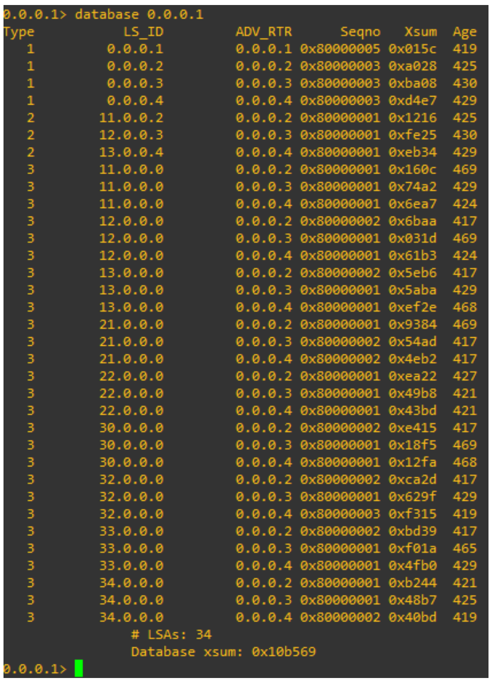
(a)



```
0.0.0.6> adv 11 6.0.0.1 0.0.0.2
        LS age:         1137
        LS Options:     0x60
        LS Type:        11
        Link State ID:  6.0.0.1
        Advert. Rtr.:   0.0.0.2
        LS Seqno:       0x80000001
        LS Xsum:        0xae26
        LS Length:      32
                // Prefix-LSA body
        Prefix Network Address: 11.0.0.0
        Prefix Network Mask: 255.255.255.0
        Intra-area cost to prefix:      10

0.0.0.6>
```

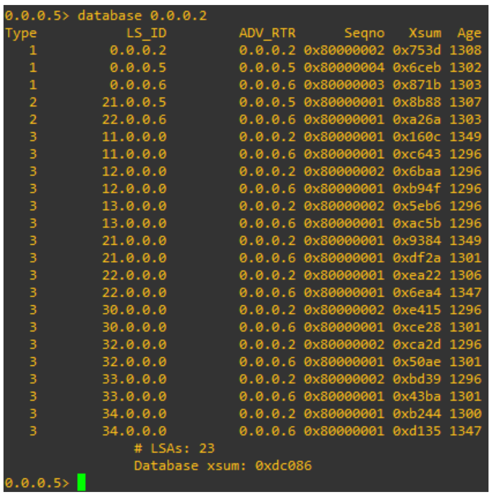
(b)

Figure 6.6: Representation of the bodies of (a) an ABR-LSA and (b) a Prefix-LSA



```
0.0.0.1> database 0.0.0.1
Type          LS_ID          ADV_RTR      Seqno    Xsum  Age
   1        0.0.0.1          0.0.0.1 0x80000005 0x015c  419
   1        0.0.0.2          0.0.0.2 0x80000003 0xa028  425
   1        0.0.0.3          0.0.0.3 0x80000003 0xba08  430
   1        0.0.0.4          0.0.0.4 0x80000003 0xd4e7  429
   2       11.0.0.2          0.0.0.2 0x80000001 0x1216  425
   2       12.0.0.3          0.0.0.3 0x80000001 0xfe25  430
   2       13.0.0.4          0.0.0.4 0x80000001 0xeb34  429
   3       11.0.0.0          0.0.0.2 0x80000001 0x160c  469
   3       11.0.0.0          0.0.0.3 0x80000001 0x74a2  429
   3       11.0.0.0          0.0.0.4 0x80000001 0x6ea7  424
   3       12.0.0.0          0.0.0.2 0x80000002 0x6baa  417
   3       12.0.0.0          0.0.0.3 0x80000001 0x031d  469
   3       12.0.0.0          0.0.0.4 0x80000001 0x61b3  424
   3       13.0.0.0          0.0.0.2 0x80000002 0x5eb6  417
   3       13.0.0.0          0.0.0.3 0x80000001 0x5aba  429
   3       13.0.0.0          0.0.0.4 0x80000001 0xef2e  468
   3       21.0.0.0          0.0.0.2 0x80000001 0x9384  469
   3       21.0.0.0          0.0.0.3 0x80000002 0x54ad  417
   3       21.0.0.0          0.0.0.4 0x80000002 0x4eb2  417
   3       22.0.0.0          0.0.0.2 0x80000001 0xea22  427
   3       22.0.0.0          0.0.0.3 0x80000001 0x49b8  421
   3       22.0.0.0          0.0.0.4 0x80000001 0x43bd  421
   3       30.0.0.0          0.0.0.2 0x80000002 0xe415  417
   3       30.0.0.0          0.0.0.3 0x80000001 0x18f5  469
   3       30.0.0.0          0.0.0.4 0x80000001 0x12fa  468
   3       32.0.0.0          0.0.0.2 0x80000002 0xca2d  417
   3       32.0.0.0          0.0.0.3 0x80000001 0x629f  429
   3       32.0.0.0          0.0.0.4 0x80000003 0xf315  419
   3       33.0.0.0          0.0.0.2 0x80000002 0xbd39  417
   3       33.0.0.0          0.0.0.3 0x80000001 0xf01a  465
   3       33.0.0.0          0.0.0.4 0x80000001 0x4fb0  429
   3       34.0.0.0          0.0.0.2 0x80000001 0xb244  421
   3       34.0.0.0          0.0.0.3 0x80000001 0x48b7  425
   3       34.0.0.0          0.0.0.4 0x80000002 0x40bd  419
                 # LSAs: 34
                 Database xsum: 0x10b569
0.0.0.1>
```

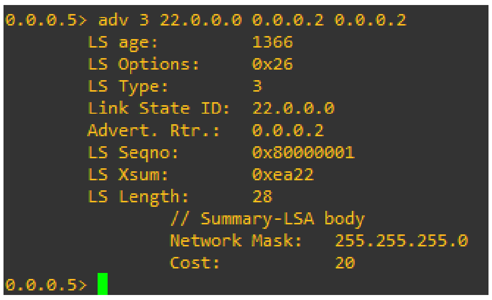
(a)



```
0.0.0.5> database 0.0.0.2
Type          LS_ID          ADV_RTR      Seqno    Xsum  Age
   1        0.0.0.2          0.0.0.2 0x80000002 0x753d 1308
   1        0.0.0.5          0.0.0.5 0x80000004 0x6ceb 1302
   1        0.0.0.6          0.0.0.6 0x80000003 0x871b 1303
   2       21.0.0.5          0.0.0.5 0x80000001 0x8b88 1307
   2       22.0.0.6          0.0.0.6 0x80000001 0xa26a 1303
   3       11.0.0.0          0.0.0.2 0x80000001 0x160c 1349
   3       11.0.0.0          0.0.0.6 0x80000001 0xc643 1296
   3       12.0.0.0          0.0.0.2 0x80000002 0x6baa 1296
   3       12.0.0.0          0.0.0.6 0x80000001 0xb94f 1296
   3       13.0.0.0          0.0.0.2 0x80000002 0x5eb6 1296
   3       13.0.0.0          0.0.0.6 0x80000001 0xac5b 1296
   3       21.0.0.0          0.0.0.2 0x80000001 0x9384 1349
   3       21.0.0.0          0.0.0.6 0x80000001 0xdf2a 1301
   3       22.0.0.0          0.0.0.2 0x80000001 0xea22 1306
   3       22.0.0.0          0.0.0.6 0x80000001 0x6ea4 1347
   3       30.0.0.0          0.0.0.2 0x80000002 0xe415 1296
   3       30.0.0.0          0.0.0.6 0x80000001 0xce28 1301
   3       32.0.0.0          0.0.0.2 0x80000002 0xca2d 1296
   3       32.0.0.0          0.0.0.6 0x80000001 0x50ae 1301
   3       33.0.0.0          0.0.0.2 0x80000002 0xbd39 1296
   3       33.0.0.0          0.0.0.6 0x80000001 0x43ba 1301
   3       34.0.0.0          0.0.0.2 0x80000001 0xb244 1300
   3       34.0.0.0          0.0.0.6 0x80000001 0xd135 1347
                 # LSAs: 23
                 Database xsum: 0xdc086
0.0.0.5>
```

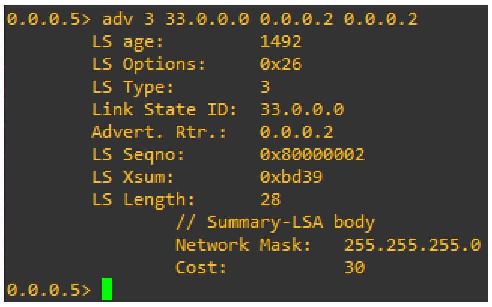
(b)

Figure 6.7: Database stored in routers (a) R1 and (b) R5



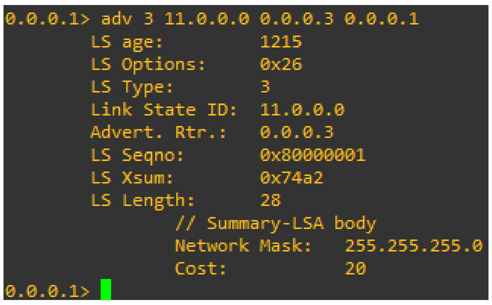
```
0.0.0.5> adv 3 22.0.0.0 0.0.0.2 0.0.0.2
        LS age:         1366
        LS Options:     0x26
        LS Type:        3
        Link State ID:  22.0.0.0
        Advert. Rtr.:   0.0.0.2
        LS Seqno:       0x80000001
        LS Xsum:        0xea22
        LS Length:      28
                // Summary-LSA body
                Network Mask:   255.255.255.0
                Cost:           20
0.0.0.5>
```

(a) Summary-LSA generated by R2 for subnet 22.0.0.0/24 (seen by R5)



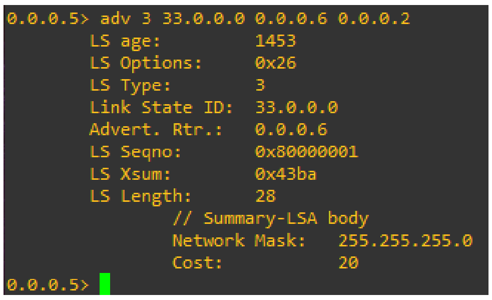
```
0.0.0.5> adv 3 33.0.0.0 0.0.0.2 0.0.0.2
        LS age:         1492
        LS Options:     0x26
        LS Type:        3
        Link State ID:  33.0.0.0
        Advert. Rtr.:   0.0.0.2
        LS Seqno:       0x80000002
        LS Xsum:        0xbd39
        LS Length:      28
                // Summary-LSA body
                Network Mask:   255.255.255.0
                Cost:           30
0.0.0.5>
```

(b) Summary-LSA generated by R2 for subnet 33.0.0.0/24 (seen by R5)



```
0.0.0.1> adv 3 11.0.0.0 0.0.0.3 0.0.0.1
        LS age:         1215
        LS Options:     0x26
        LS Type:        3
        Link State ID:  11.0.0.0
        Advert. Rtr.:   0.0.0.3
        LS Seqno:       0x80000001
        LS Xsum:        0x74a2
        LS Length:      28
                // Summary-LSA body
                Network Mask:   255.255.255.0
                Cost:           20
0.0.0.1>
```

(c) Summary-LSA generated by R3 for subnet 11.0.0.0/24 (seen by R1)



```
0.0.0.5> adv 3 33.0.0.0 0.0.0.6 0.0.0.2
        LS age:         1453
        LS Options:     0x26
        LS Type:        3
        Link State ID:  33.0.0.0
        Advert. Rtr.:   0.0.0.6
        LS Seqno:       0x80000001
        LS Xsum:        0x43ba
        LS Length:      28
                // Summary-LSA body
                Network Mask:   255.255.255.0
                Cost:           20
0.0.0.5>
```

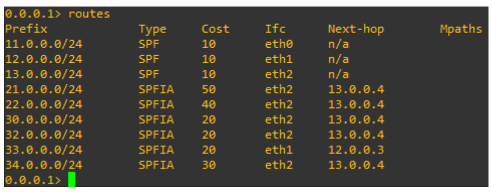
(d) Summary-LSA generated by R6 for subnet 33.0.0.0/24 (seen by R5)

Figure 6.8: Examples of some Summary-LSAs originated by various ABRs

originated by R2 for subnet 22.0.0.0/24 and in figure 6.8.b, the Summary-LSA originated by R2 for subnet 33.0.0.0/24. From the topology (figure 6.1) we can see that R2 here advertises an inter-area path cost to this subnet, using the shortest path possible, through R1 and R3, which is the only possible route to this destination presenting a cost of 30. Any alternative path would present a cost of at least 40 to reach the same destination. We also see, in figure 6.8.c, the Summary-LSA originated by R3 for subnet 11.0.0.0/24, and in figure 6.8.d, the Summary-LSA originated by R6 for subnet 33.0.0.0/24.

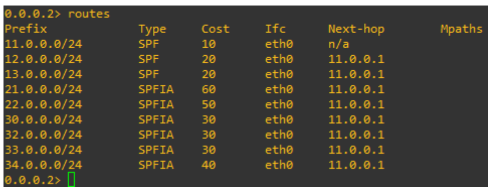
6.1.2 Globally optimal routing choice

For this next test, we used the same topology of figure 6.1, with some changes to the interface costs presented. We will assign a cost of 100 to R2's eth1, rather than 10, thus making it better to access the destinations contained in Area 2 from R6, independently of which router we are calculating the route from. This means, for example, that R2 should take inter-area paths to the destinations within Area 2, despite having intra-area paths available to reach those destinations. Also, we verified how R2 advertises different costs in it's originated Summary-LSA and Prefix-LSA for the same destination, with the best possible cost in the former and the best intra-area cost in the latter.



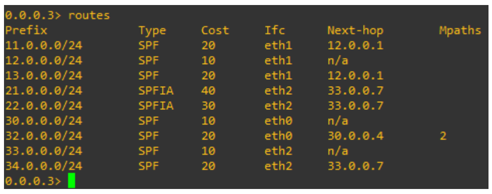
```
0.0.0.1> routes
Prefix          Type    Cost    Ifc     Next-hop        Mpaths
11.0.0.0/24     SPF     10      eth0    n/a
12.0.0.0/24     SPF     10      eth1    n/a
13.0.0.0/24     SPF     10      eth2    n/a
21.0.0.0/24     SPFIA   50      eth2    13.0.0.4
22.0.0.0/24     SPFIA   40      eth2    13.0.0.4
30.0.0.0/24     SPFIA   20      eth2    13.0.0.4
32.0.0.0/24     SPFIA   20      eth2    13.0.0.4
33.0.0.0/24     SPFIA   20      eth1    12.0.0.3
34.0.0.0/24     SPFIA   30      eth2    13.0.0.4
0.0.0.1>
```

(a)



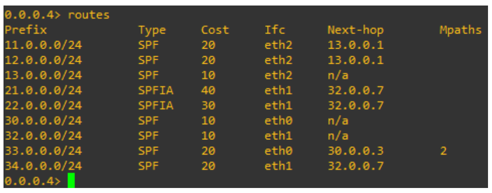
```
0.0.0.2> routes
Prefix          Type    Cost    Ifc     Next-hop        Mpaths
11.0.0.0/24     SPF     10      eth0    n/a
12.0.0.0/24     SPF     20      eth0    11.0.0.1
13.0.0.0/24     SPF     20      eth0    11.0.0.1
21.0.0.0/24     SPFIA   60      eth0    11.0.0.1
22.0.0.0/24     SPFIA   50      eth0    11.0.0.1
30.0.0.0/24     SPFIA   30      eth0    11.0.0.1
32.0.0.0/24     SPFIA   30      eth0    11.0.0.1
33.0.0.0/24     SPFIA   30      eth0    11.0.0.1
34.0.0.0/24     SPFIA   40      eth0    11.0.0.1
0.0.0.2>
```

(b)



```
0.0.0.3> routes
Prefix          Type    Cost    Ifc     Next-hop        Mpaths
11.0.0.0/24     SPF     20      eth1    12.0.0.1
12.0.0.0/24     SPF     10      eth1    n/a
13.0.0.0/24     SPF     20      eth1    12.0.0.1
21.0.0.0/24     SPFIA   40      eth2    33.0.0.7
22.0.0.0/24     SPFIA   30      eth2    33.0.0.7
30.0.0.0/24     SPF     10      eth0    n/a
32.0.0.0/24     SPF     20      eth0    30.0.0.4        2
33.0.0.0/24     SPF     10      eth2    n/a
34.0.0.0/24     SPF     20      eth2    33.0.0.7
0.0.0.3>
```

(c)



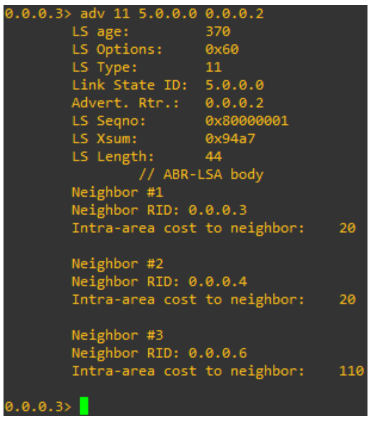
```
0.0.0.4> routes
Prefix          Type    Cost    Ifc     Next-hop        Mpaths
11.0.0.0/24     SPF     20      eth2    13.0.0.1
12.0.0.0/24     SPF     20      eth2    13.0.0.1
13.0.0.0/24     SPF     10      eth2    n/a
21.0.0.0/24     SPFIA   40      eth1    32.0.0.7
22.0.0.0/24     SPFIA   30      eth1    32.0.0.7
30.0.0.0/24     SPF     10      eth0    n/a
32.0.0.0/24     SPF     10      eth1    n/a
33.0.0.0/24     SPF     20      eth0    30.0.0.3        2
34.0.0.0/24     SPF     20      eth1    32.0.0.7
0.0.0.4>
```

(d)

Figure 6.9: Routing tables of routers (a) R1, (b) R2, (c) R3 and (d) R4

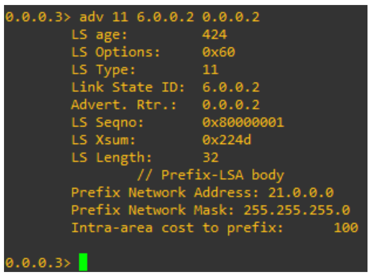
We start by verifying the routing tables of some routers. In this case we look into the routers affected by this change, routers R1, R2, R3 and R4, which would have at least one route to a destination passing R2. These routing tables are shown in figure 6.9, R1's in figure 6.9.a, R2's in figure 6.9.b, R3's in figure 6.9.c and R4's in figure 6.9. We can see, by comparing R1's and R3's routing tables obtained in the previous test, presented in figures 6.4.a and 6.4.b, with this new ones, presented in figures 6.9.a and 6.9.c, respectively, that both present changes in the routes to at least one of the destinations. This is the case as paths to these destinations were being directed through R2, which now advertises a cost of 100 and 110 to subnets 21.0.0.0/24 and 22.0.0.0/24, respectively. From examining the routing tables obtained for R2, present in figure 6.9.b, none of its advertised paths exit through its eth1 interface, choosing the better inter-area paths instead to reach all destinations. A good example of this can be seen in the routing table of R2. Without this change in cost, its cost to reach subnets 21.0.0.0/24 and 22.0.0.0/24 would be, respectively, 10 and 20. With this change to R2's interface, we can see these routes are being advertised with costs of 60 and 50, both as an inter-area route, and having R1 as its next hop. In the base implementation, the worse intra-area paths with costs of 100 and 110 would still be chosen over these inter-area paths.

We also take a look at the Overlay LSAs and Summary-LSAs originated by R2. These are shown in figures 6.10 and 6.11, respectively. In figure 6.10.a, showing R2's ABR-LSA, we can see its advertised cost to R6 to be of 110, its intra-area cost to reach it, even though it can reach that same router with a cost of 40, by taking the path advertised for subnet 34.0.0.0/24, which is in fact what is done. Similarly, R2 advertises costs of 100 and 110 in its Prefix-LSAs referring to subnets 21.0.0.0/24 and 22.0.0.0/24, illustrated in figures 6.10.b and 6.10.c, respectively. We can see the choice made by the router by



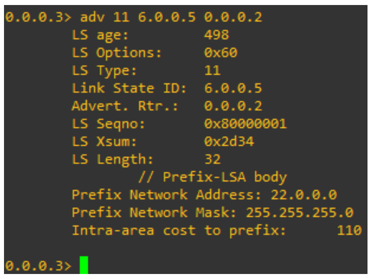
```
0.0.0.3> adv 11 5.0.0.0 0.0.0.2
        LS age:         370
        LS Options:     0x60
        LS Type:        11
        Link State ID:  5.0.0.0
        Advert. Rtr.:   0.0.0.2
        LS Seqno:       0x80000001
        LS Xsum:        0x94a7
        LS Length:      44
                // ABR-LSA body
        Neighbor #1
        Neighbor RID: 0.0.0.3
        Intra-area cost to neighbor:    20

        Neighbor #2
        Neighbor RID: 0.0.0.4
        Intra-area cost to neighbor:    20

        Neighbor #3
        Neighbor RID: 0.0.0.6
        Intra-area cost to neighbor:    110

0.0.0.3> 
```

(a)



```
0.0.0.3> adv 11 6.0.0.2 0.0.0.2
        LS age:         424
        LS Options:     0x60
        LS Type:        11
        Link State ID:  6.0.0.2
        Advert. Rtr.:   0.0.0.2
        LS Seqno:       0x80000001
        LS Xsum:        0x224d
        LS Length:      32
                // Prefix-LSA body
        Prefix Network Address: 21.0.0.0
        Prefix Network Mask: 255.255.255.0
        Intra-area cost to prefix:      100

0.0.0.3> 
```

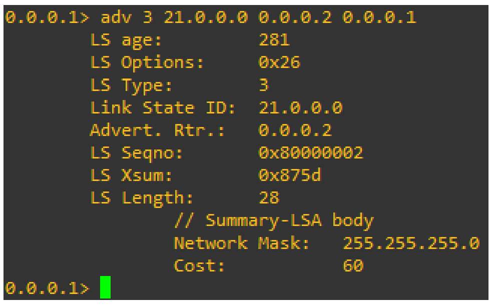
(b)



```
0.0.0.3> adv 11 6.0.0.5 0.0.0.2
        LS age:         498
        LS Options:     0x60
        LS Type:        11
        Link State ID:  6.0.0.5
        Advert. Rtr.:   0.0.0.2
        LS Seqno:       0x80000001
        LS Xsum:        0x2d34
        LS Length:      32
                // Prefix-LSA body
        Prefix Network Address: 22.0.0.0
        Prefix Network Mask: 255.255.255.0
        Intra-area cost to prefix:      110

0.0.0.3> 
```

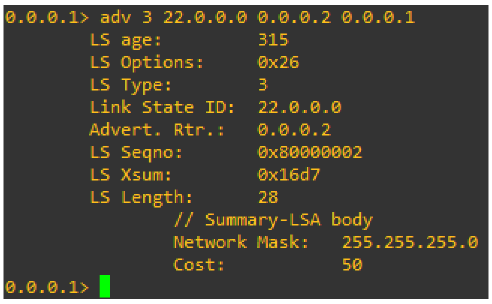
(c)

Figure 6.10: Overlay LSAs originated by R2; (a) ABR-LSA, (b) Prefix-LSA for subnet 21.0.0.0/24 and (c) Prefix-LSA for subnet 22.0.0.0/24



```
0.0.0.1> adv 3 21.0.0.0 0.0.0.2 0.0.0.1
        LS age:         281
        LS Options:     0x26
        LS Type:        3
        Link State ID:  21.0.0.0
        Advert. Rtr.:   0.0.0.2
        LS Seqno:       0x80000002
        LS Xsum:        0x875d
        LS Length:      28
                // Summary-LSA body
                Network Mask:   255.255.255.0
                Cost:           60
0.0.0.1> 
```

(a)



```
0.0.0.1> adv 3 22.0.0.0 0.0.0.2 0.0.0.1
        LS age:         315
        LS Options:     0x26
        LS Type:        3
        Link State ID:  22.0.0.0
        Advert. Rtr.:   0.0.0.2
        LS Seqno:       0x80000002
        LS Xsum:        0x16d7
        LS Length:      28
                // Summary-LSA body
                Network Mask:   255.255.255.0
                Cost:           50
0.0.0.1> 
```

(b)

Figure 6.11: Summary-LSAs originated by R2 for (a) subnet 21.0.0.0/24 and (b) 22.0.0.0/24

contrasting this Prefix-LSA advertisement with the Summary-LSAs advertised by R2 for those subnets, which present costs of 60 and 50, as shown in figures 6.11.a and 6.11.b, respectively. The advertised costs for these subnets correspond to those calculated and shown in the router's own routing table. This further proves the capacity of ABRs to always advertise the shortest path cost to a destination in their Summary-LSAs, despite of the route type used.

6.1.3 Reaction to changes: Changes to costs

For this test we begin by analyzing the capacity of our extended implementation to react to changes in interface costs within the network. A Wireshark capture was performed between routers R6 and R7, to capture the packets that would be sent as a reaction to the change applied. First, we see how it adapts to a change in the cost assigned to a specific interface. As a starting point we will be using the topology from figure 6.1 as described. Then, after the network converges, we change the interface cost of R6's interface eth1, from 10 to 100. After the network finishes converging, we change the same interface cost to its initial value, ending up with the starting topology. We should see a change in some of the routing choices made by certain routers, in this case R2 and R5, not only with the first change, but with the second change as well, which should revert the changes made with the first change.

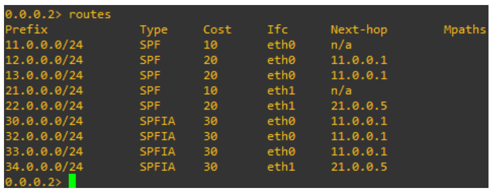
This test can then be separated into three phases:

1. The network converged previously to the cost change;
2. The cost change was applied and the network converged (R6's eth1 from 10 to 100);
3. The cost change was reversed and the network converged to its initial state (R6's eth1 from 100 to 10).

For each phase, we will analyze a certain amount of information, to evaluate the state of the network at that point in time. Since we changed the cost of the entry point from R6 into Area 3, it is logical that the routers mostly affected by this change would be R6 itself, R2 and R5, which would be using R6 as the gateway to at least some subnets in Area 3. We will start by analyzing the routing tables of routers R2 and R5, R6's ABR-LSA, and both the Prefix-LSA and Summary-LSA originated by R6 for a subnet contained in Area 3. This information can be seen in figures 6.12.a to 6.12.e, in the order described. We can see that with all interfaces having a cost of 10, R2 uses R6 to reach subnet 34.0.0.0/24, R5 uses it to reach all but one of the destinations in Area 3, R6 advertises intra-area costs of 20 to all its neighbor ABRs, and both the Prefix-LSA and Summary-LSA, referring to subnet 33.0.0.0/24, advertise the same cost of 20, reflecting the choice of intra-area routing from R6 to this destination.

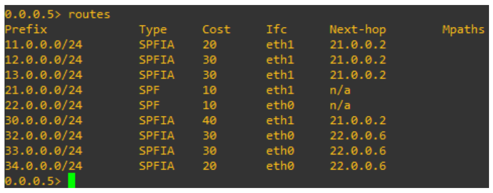
We then changed the cost of R6's eth1 interface, from 10 to 100, entering the second phase of this test. We were able to see, via our Wireshark capture, the packets resulting from the reactions to these changes, present in Area 3. This includes all Overlay LSA updates, and Summary-LSA updates within Area 3. Note that a change to a Summary-LSA by R2 was not captured here. In figure 6.13 we see the summary of the packets captured, including all the LS Update packets flooded into Area 3 as a reaction to the cost change. Inside the four LS Update packets advertised by R6 is included the updated versions of the router's Router-LSA, ABR-LSA, all the Prefix-LSAs referring to the subnets contained within Area 3 and all the Summary-LSAs originated for the destination for which R6 previously used the changed interface as the outgoing interface.

Besides this capture we gathered the same information as before, alongside the OS's routing table for router R5. This is the only case in which this was done, to prove the correct updating of the OS routing table from the OSPF stored one through the comparison of the two tables. Figure 6.14 then presents, in this order, from figure 6.14.a to 6.14.f, the routing table for R2, the routing table for R5, the kernel routing table for R5, R6's ABR-LSA, and the Prefix-LSA and Summary-LSA generated by R6 for subnet 33.0.0.0/24. We see a slight change with R2's subnet 34.0.0.0/24 entry and bigger changes to R5's



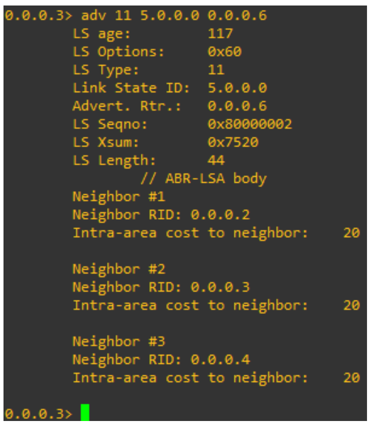
```
0.0.0.2> routes
Prefix              Type    Cost    Ifc     Next-hop        Mpaths
11.0.0.0/24         SPF     10      eth0    n/a
12.0.0.0/24         SPF     20      eth0    11.0.0.1
13.0.0.0/24         SPF     20      eth0    11.0.0.1
21.0.0.0/24         SPF     10      eth1    n/a
22.0.0.0/24         SPF     20      eth1    21.0.0.5
30.0.0.0/24         SPFIA   30      eth0    11.0.0.1
32.0.0.0/24         SPFIA   30      eth0    11.0.0.1
33.0.0.0/24         SPFIA   30      eth0    11.0.0.1
34.0.0.0/24         SPFIA   30      eth1    21.0.0.5
0.0.0.2>
```

(a) Routing table of R2



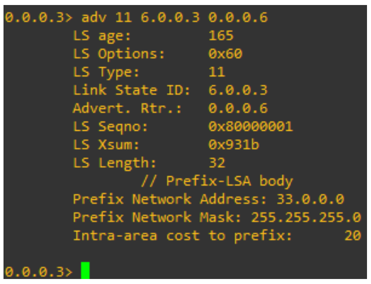
```
0.0.0.5> routes
Prefix              Type    Cost    Ifc     Next-hop        Mpaths
11.0.0.0/24         SPFIA   20      eth1    21.0.0.2
12.0.0.0/24         SPFIA   30      eth1    21.0.0.2
13.0.0.0/24         SPFIA   30      eth1    21.0.0.2
21.0.0.0/24         SPF     10      eth1    n/a
22.0.0.0/24         SPF     10      eth0    n/a
30.0.0.0/24         SPFIA   40      eth1    21.0.0.2
32.0.0.0/24         SPFIA   30      eth0    22.0.0.6
33.0.0.0/24         SPFIA   30      eth0    22.0.0.6
34.0.0.0/24         SPFIA   20      eth0    22.0.0.6
0.0.0.5>
```

(b) Routing table of R5



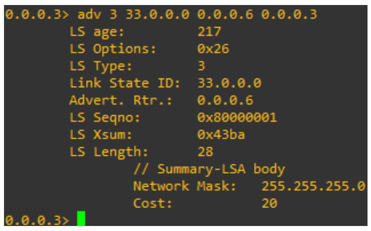
```
0.0.0.3> adv 11 5.0.0.0 0.0.0.6
        LS age:         117
        LS Options:     0x60
        LS Type:        11
        Link State ID:  5.0.0.0
        Advert. Rtr.:   0.0.0.6
        LS Seqno:       0x80000002
        LS Xsum:        0x7520
        LS Length:      44
                // ABR-LSA body
        Neighbor #1
        Neighbor RID: 0.0.0.2
        Intra-area cost to neighbor:    20

        Neighbor #2
        Neighbor RID: 0.0.0.3
        Intra-area cost to neighbor:    20

        Neighbor #3
        Neighbor RID: 0.0.0.4
        Intra-area cost to neighbor:    20

0.0.0.3>
```

(c) ABR-LSA of R6 (seen by R5)



```
0.0.0.3> adv 11 6.0.0.3 0.0.0.6
        LS age:         165
        LS Options:     0x60
        LS Type:        11
        Link State ID:  6.0.0.3
        Advert. Rtr.:   0.0.0.6
        LS Seqno:       0x80000001
        LS Xsum:        0x931b
        LS Length:      32
                // Prefix-LSA body
        Prefix Network Address: 33.0.0.0
        Prefix Network Mask: 255.255.255.0
        Intra-area cost to prefix:      20

0.0.0.3>
```

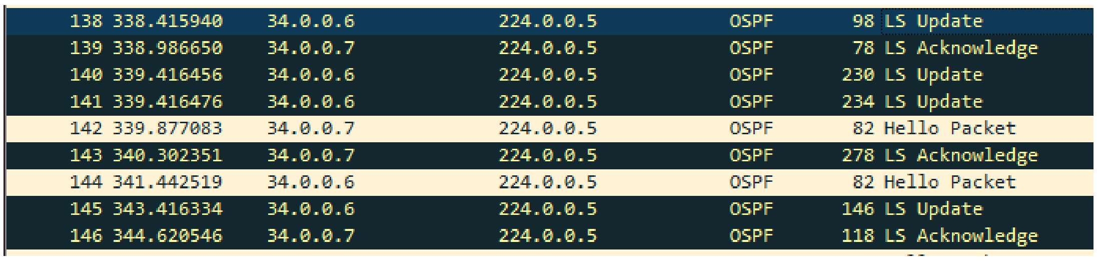
(d) Prefix-LSA generated by R6 for subnet 33.0.0.0/24 (seen by R3)



```
0.0.0.3> adv 3 33.0.0.0 0.0.0.6 0.0.0.3
        LS age:         217
        LS Options:     0x26
        LS Type:        3
        Link State ID:  33.0.0.0
        Advert. Rtr.:   0.0.0.6
        LS Seqno:       0x80000001
        LS Xsum:        0x43ba
        LS Length:      28
                // Summary-LSA body
                Network Mask:   255.255.255.0
                Cost:           20
0.0.0.3>
```

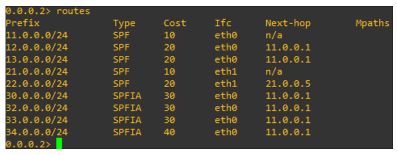
(e) Summary-LSA generated by R6 for subnet 33.0.0.0/24 (seen by R3)

Figure 6.12: State of the network during the first phase (before the cost change)



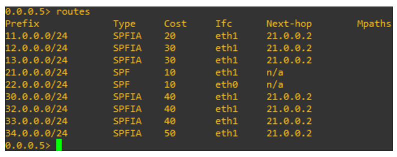
```
138 338.415940   34.0.0.6    224.0.0.5   OSPF    98 LS Update
139 338.986650   34.0.0.7    224.0.0.5   OSPF    78 LS Acknowledge
140 339.416456   34.0.0.6    224.0.0.5   OSPF   230 LS Update
141 339.416476   34.0.0.6    224.0.0.5   OSPF   234 LS Update
142 339.877083   34.0.0.7    224.0.0.5   OSPF    82 Hello Packet
143 340.302351   34.0.0.7    224.0.0.5   OSPF   278 LS Acknowledge
144 341.442519   34.0.0.6    224.0.0.5   OSPF    82 Hello Packet
145 343.416334   34.0.0.6    224.0.0.5   OSPF   146 LS Update
146 344.620546   34.0.0.7    224.0.0.5   OSPF   118 LS Acknowledge
```

Figure 6.13: Summary of the packet capture portion referring to the first change applied



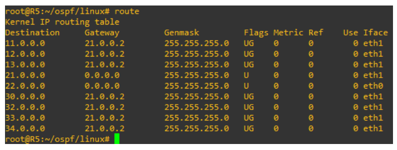
```
0.0.0.2> routes
Prefix          Type    Cost    Ifc     Next-hop        Mpaths
11.0.0.0/24     SPF     10      eth0    n/a
12.0.0.0/24     SPF     20      eth0    11.0.0.1
13.0.0.0/24     SPF     20      eth0    11.0.0.1
21.0.0.0/24     SPF     10      eth1    n/a
22.0.0.0/24     SPF     20      eth1    21.0.0.5
30.0.0.0/24     SPFIA   30      eth0    11.0.0.1
32.0.0.0/24     SPFIA   30      eth0    11.0.0.1
33.0.0.0/24     SPFIA   30      eth0    11.0.0.1
34.0.0.0/24     SPFIA   40      eth0    11.0.0.1
0.0.0.2>
```

(a) Routing table of R2



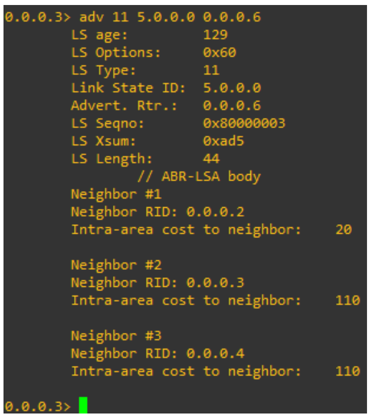
```
0.0.0.5> routes
Prefix          Type    Cost    Ifc     Next-hop        Mpaths
11.0.0.0/24     SPFIA   20      eth1    21.0.0.2
12.0.0.0/24     SPFIA   30      eth1    21.0.0.2
13.0.0.0/24     SPFIA   30      eth1    21.0.0.2
21.0.0.0/24     SPF     10      eth1    n/a
22.0.0.0/24     SPF     10      eth0    n/a
30.0.0.0/24     SPFIA   40      eth1    21.0.0.2
32.0.0.0/24     SPFIA   40      eth1    21.0.0.2
33.0.0.0/24     SPFIA   40      eth1    21.0.0.2
34.0.0.0/24     SPFIA   50      eth1    21.0.0.2
0.0.0.5>
```

(b) Routing table of R5



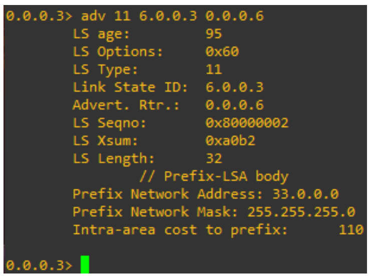
```
root@R5:~/ospf/linux# route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
11.0.0.0        21.0.0.2        255.255.255.0   UG    0      0        0 eth1
12.0.0.0        21.0.0.2        255.255.255.0   UG    0      0        0 eth1
13.0.0.0        21.0.0.2        255.255.255.0   UG    0      0        0 eth1
21.0.0.0        0.0.0.0         255.255.255.0   U     0      0        0 eth1
22.0.0.0        0.0.0.0         255.255.255.0   U     0      0        0 eth0
30.0.0.0        21.0.0.2        255.255.255.0   UG    0      0        0 eth1
32.0.0.0        21.0.0.2        255.255.255.0   UG    0      0        0 eth1
33.0.0.0        21.0.0.2        255.255.255.0   UG    0      0        0 eth1
34.0.0.0        21.0.0.2        255.255.255.0   UG    0      0        0 eth1
root@R5:~/ospf/linux#
```

(c) OS Routing table of R5



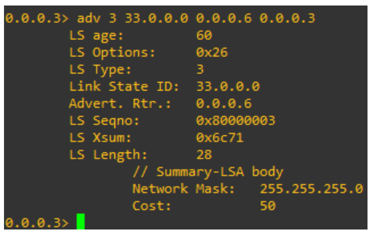
```
0.0.0.3> adv 11 5.0.0.0 0.0.0.6
        LS age:         129
        LS Options:     0x60
        LS Type:        11
        Link State ID:  5.0.0.0
        Advert. Rtr.:   0.0.0.6
        LS Seqno:       0x80000003
        LS Xsum:        0xad5
        LS Length:      44
                // ABR-LSA body
        Neighbor #1
        Neighbor RID: 0.0.0.2
        Intra-area cost to neighbor:    20

        Neighbor #2
        Neighbor RID: 0.0.0.3
        Intra-area cost to neighbor:    110

        Neighbor #3
        Neighbor RID: 0.0.0.4
        Intra-area cost to neighbor:    110

0.0.0.3>
```

(d) ABR-LSA of R6 (seen by R3)



```
0.0.0.3> adv 11 6.0.0.3 0.0.0.6
        LS age:         95
        LS Options:     0x60
        LS Type:        11
        Link State ID:  6.0.0.3
        Advert. Rtr.:   0.0.0.6
        LS Seqno:       0x80000002
        LS Xsum:        0xa0b2
        LS Length:      32
                // Prefix-LSA body
        Prefix Network Address: 33.0.0.0
        Prefix Network Mask: 255.255.255.0
        Intra-area cost to prefix:      110
0.0.0.3>
```

(e) Prefix-LSA generated by R6 for subnet 33.0.0.0/24 (seen by R3)



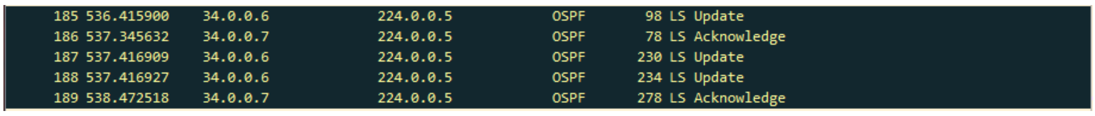
```
0.0.0.3> adv 3 33.0.0.0 0.0.0.6 0.0.0.3
        LS age:         60
        LS Options:     0x26
        LS Type:        3
        Link State ID:  33.0.0.0
        Advert. Rtr.:   0.0.0.6
        LS Seqno:       0x80000003
        LS Xsum:        0x6c71
        LS Length:      28
                // Summary-LSA body
                Network Mask:   255.255.255.0
                Cost:           50
0.0.0.3>
```

(f) Summary-LSA generated by R6 for subnet 33.0.0.0/24 (seen by R3)

Figure 6.14: State of the network during the second phase (after the cost change)

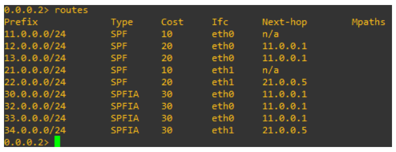
routing table, which no longer uses R6 as the next hop to any destinations. We can also see changes to the three LSAs generated by R6. Its ABR-LSA now advertises a cost of 110 to R3 and R4 (from 20), its Prefix-LSA now presents a cost of 110 (from 20) to subnet 33.0.0.0/24 and the Summary-LSA for that same destination advertises a cost of 50 (from 20).

Finally we move on to the third phase, by reverting the change on R6's interface. As we would expect, the outcome of this change was the restoring of the state of the network back to how it was presented in figure 6.12. Figure 6.15 shows the portion of the packet capture corresponding to the reaction to this last change. It contains the same type of information as that of the previous capture portion. Also figure 6.16 shows the information gathered after this change, in the same order as that indicated for figure 6.12. We can see the restoration of the routing table of the routers, as well as of the information contained in the advertised LSAs. With this, we prove that this extended implementation will adapt itself, not only to better routing options, but also to negative changes, without issue, overcoming the count-to-infinity problem (see Section 2.2.1).



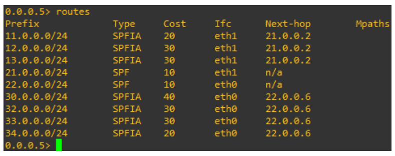
```
185 536.415900    34.0.0.6          224.0.0.5         OSPF        98 LS Update
186 537.345632    34.0.0.7          224.0.0.5         OSPF        78 LS Acknowledge
187 537.416909    34.0.0.6          224.0.0.5         OSPF       230 LS Update
188 537.416927    34.0.0.6          224.0.0.5         OSPF       234 LS Update
189 538.472518    34.0.0.7          224.0.0.5         OSPF       278 LS Acknowledge
```

Figure 6.15: Summary of the packet capture portion referring to the second change applied



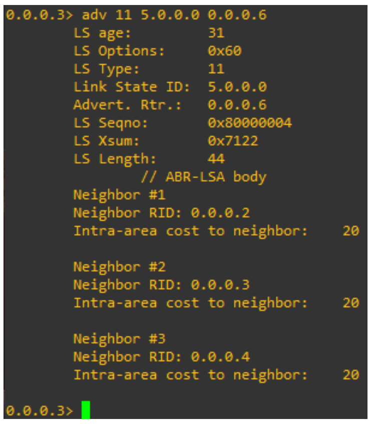
```
0.0.0.2> routes
Prefix              Type     Cost     Ifc      Next-hop        Mpaths
11.0.0.0/24         SPF      10       eth0     n/a
12.0.0.0/24         SPF      20       eth0     11.0.0.1
13.0.0.0/24         SPF      20       eth0     11.0.0.1
21.0.0.0/24         SPF      10       eth1     n/a
22.0.0.0/24         SPF      20       eth1     21.0.0.5
30.0.0.0/24         SPFIA    30       eth0     11.0.0.1
32.0.0.0/24         SPFIA    30       eth0     11.0.0.1
33.0.0.0/24         SPFIA    30       eth0     11.0.0.1
34.0.0.0/24         SPFIA    30       eth1     21.0.0.5
0.0.0.2>
```

(a) Routing table of R2



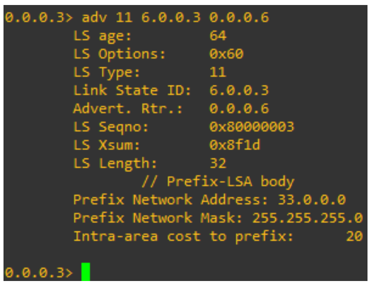
```
0.0.0.5> routes
Prefix              Type     Cost     Ifc      Next-hop        Mpaths
11.0.0.0/24         SPFIA    20       eth1     21.0.0.2
12.0.0.0/24         SPFIA    30       eth1     21.0.0.2
13.0.0.0/24         SPFIA    30       eth1     21.0.0.2
21.0.0.0/24         SPF      10       eth1     n/a
22.0.0.0/24         SPF      10       eth0     n/a
30.0.0.0/24         SPFIA    40       eth0     22.0.0.6
32.0.0.0/24         SPFIA    30       eth0     22.0.0.6
33.0.0.0/24         SPFIA    30       eth0     22.0.0.6
34.0.0.0/24         SPFIA    20       eth0     22.0.0.6
0.0.0.5>
```

(b) Routing table of R5



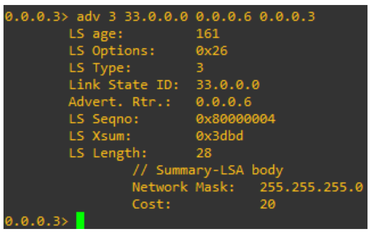
```
0.0.0.3> adv 11 5.0.0.0 0.0.0.6
        LS age:          31
        LS Options:      0x60
        LS Type:         11
        Link State ID:   5.0.0.0
        Advert. Rtr.:    0.0.0.6
        LS Seqno:        0x80000004
        LS Xsum:         0x7122
        LS Length:       44
                // ABR-LSA body
        Neighbor #1
        Neighbor RID: 0.0.0.2
        Intra-area cost to neighbor:    20

        Neighbor #2
        Neighbor RID: 0.0.0.3
        Intra-area cost to neighbor:    20

        Neighbor #3
        Neighbor RID: 0.0.0.4
        Intra-area cost to neighbor:    20
0.0.0.3>
```

(c) ABR-LSA of R6 (seen by R3)



```
0.0.0.3> adv 11 6.0.0.3 0.0.0.6
        LS age:          64
        LS Options:      0x60
        LS Type:         11
        Link State ID:   6.0.0.3
        Advert. Rtr.:    0.0.0.6
        LS Seqno:        0x80000003
        LS Xsum:         0x8f1d
        LS Length:       32
                // Prefix-LSA body
        Prefix Network Address: 33.0.0.0
        Prefix Network Mask: 255.255.255.0
        Intra-area cost to prefix:      20

0.0.0.3>
```

(d) Prefix-LSA generated by R6 for subnet 33.0.0.0/24 (seen by R3)



```
0.0.0.3> adv 3 33.0.0.0 0.0.0.6 0.0.0.3
        LS age:          161
        LS Options:      0x26
        LS Type:         3
        Link State ID:   33.0.0.0
        Advert. Rtr.:    0.0.0.6
        LS Seqno:        0x80000004
        LS Xsum:         0x3dbd
        LS Length:       28
                // Summary-LSA body
                Network Mask:    255.255.255.0
                Cost:            20

0.0.0.3>
```

(e) Summary-LSA generated by R6 for subnet 33.0.0.0/24 (seen by R3)

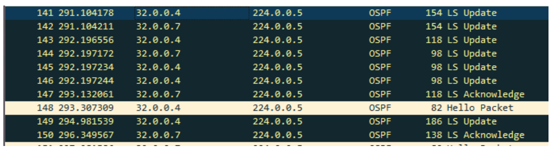
Figure 6.16: State of the network during the third and last phase (after the second cost change, from 100 back to 10)

6.1.4 Reaction to changes: Changes to links between routers

For the next test, we will not change any interface cost, but instead remove the link between two routers, completely cutting off paths that would otherwise cross that link. For this test, start out with the topology of figure 6.1, as described. A Wireshark packet capture was performed between routers R4 and R7, on subnet 32.0.0.0/24. After the network converged, we removed the link between R6 and R7 (first change), preventing all routes from crossing subnet 34.0.0.0/24. Note that this subnet is still available at both routers R6 and R7, as their interfaces were kept running. Then, after the network converged to a new state, we re-attached the link between these routers to see how the network reacted (second change). The outcome of this test should be very similar to that of the previous test, with R5 being the router most affected by this change. In this test we will see a difference in the nature of the change detection, now done through the Hello Protocol (see Section 2.3.7). The expected result for this test is an evolution by R5 similar to that of the previous test case, with the exception that the route taken for subnet 34.0.0.0/24 remains the same. Also is the need to update the ABR-LSAs of ABRs attached to area 3, since intra-area paths will cease to exist between R6 and both R3 and R4. Furthermore, we should see the premature aging of R6's Prefix-LSAs originated for all but one destination in Area 3 (subnet 34.0.0.0/24), alongside the update of some Summary-LSAs, similarly to the previous test case.

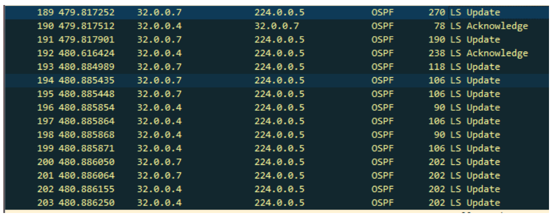
Just like the previous case, this test follows three simple steps. However, to show a different perspective over similar results we will present values over the steps for the same information, instead of all the information after each step taken.

We start by examining the disturbances captured in Wireshark. The summary of this disturbances is shown in figure 6.17. We examined these captures and seen that after the first change was detected (figure 6.17.a), R7 updated its Router-LSA, by removing R6 related information from it, and deleted its Network-LSA for subnet 34.0.0.0/24. After that, ABR-LSA updates come from both R3 and R4, also removing R6 from the advertised ABR neighbors. Finally, another ABR-LSA update appears, this time from R6, alongside the premature aging for three of its generated Prefix-LSAs, for subnets 30.0.0.0/24, 32.0.0.0/24 and 33.0.0.0/24. In figures 6.18.a and 6.18.b, we show a more detailed view of the contents of packets 142 and 149, summarized in figure 6.17.a. Packet 142 shows R7's updated Router-LSA (LS Age value reset) and the premature aging of the Network-LSA of subnet 34.0.0.0/24 (LS Age value of 3600). Packet 149, although not recognized by Wireshark, shows the update of R6's ABR-LSA (Opaque Type 5) and premature aging of its three Prefix-LSAs (Opaque Type 6), from which we show one. The second change, as we can see from figure 6.17.b, presents a higher number of packets generated. By the order of the received packets, these were, in the first place, the updated Summary-LSAs from R6, generated after the first change, which were probably held in a retransmission list, waiting for an acknowledgment. After that we see the update of both R6's and R7's Router-LSAs, as well as the update of the Network-LSA relative to subnet 34.0.0.0/24, shown in figure 6.19. After that, the update of the



```
141 291.104178   32.0.0.4      224.0.0.5      OSPF    154 LS Update
142 291.104211   32.0.0.7      224.0.0.5      OSPF    154 LS Update
143 292.196556   32.0.0.4      224.0.0.5      OSPF    118 LS Update
144 292.197172   32.0.0.7      224.0.0.5      OSPF     98 LS Update
145 292.197234   32.0.0.4      224.0.0.5      OSPF     98 LS Update
146 292.197244   32.0.0.4      224.0.0.5      OSPF     98 LS Update
147 293.132061   32.0.0.7      224.0.0.5      OSPF    118 LS Acknowledge
148 293.307309   32.0.0.4      224.0.0.5      OSPF     82 Hello Packet
149 294.981539   32.0.0.4      224.0.0.5      OSPF    186 LS Update
150 296.349567   32.0.0.7      224.0.0.5      OSPF    138 LS Acknowledge
```

(a)



```
189 479.817252   32.0.0.7      224.0.0.5      OSPF    270 LS Update
190 479.817512   32.0.0.4      32.0.0.7       OSPF     78 LS Acknowledge
191 479.817901   32.0.0.7      224.0.0.5      OSPF    190 LS Update
192 480.616424   32.0.0.4      224.0.0.5      OSPF    238 LS Acknowledge
193 480.884989   32.0.0.7      224.0.0.5      OSPF    118 LS Update
194 480.885435   32.0.0.7      224.0.0.5      OSPF    106 LS Update
195 480.885448   32.0.0.7      224.0.0.5      OSPF    106 LS Update
196 480.885854   32.0.0.4      224.0.0.5      OSPF     90 LS Update
197 480.885864   32.0.0.4      224.0.0.5      OSPF    106 LS Update
198 480.885868   32.0.0.4      224.0.0.5      OSPF     90 LS Update
199 480.885871   32.0.0.4      224.0.0.5      OSPF    106 LS Update
200 480.886050   32.0.0.7      224.0.0.5      OSPF    202 LS Update
201 480.886064   32.0.0.7      224.0.0.5      OSPF    202 LS Update
202 480.886155   32.0.0.4      224.0.0.5      OSPF    202 LS Update
203 480.886250   32.0.0.4      224.0.0.5      OSPF    202 LS Update
```

(b)

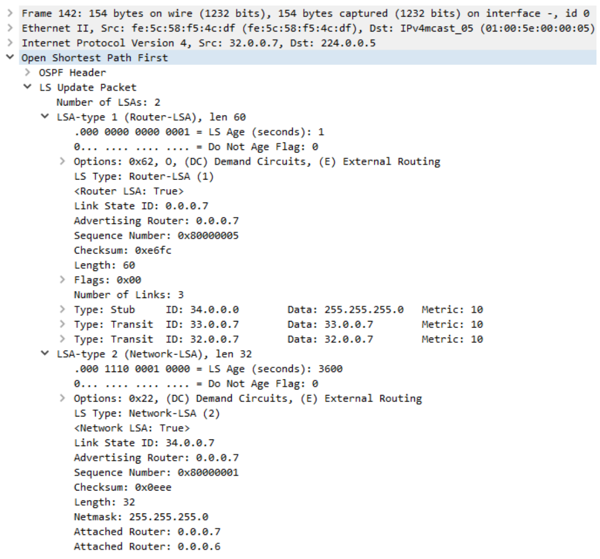
Figure 6.17: Summary of the portions of the packet capture including the packets generated after the (a) first and (b) second changes applied

ABR-LSAs of R3, R4 and R6, to their initial state, were flooded followed by the origination of three new Prefix-LSAs that would replace the prematurely aged ones by R6. Finally, R6 floods into Area 3 the Summary-LSAs of all destinations whose chosen routes were affected by the first change.

We now proceed to the findings from within the monitoring tool. For the next presented figures, please consider that sub-figures **a** will refer to the state before the changes, sub-figures **b** will refer to the state after reacting to the first change, and sub-figures **c** will refer to the final state, after the network converges after the second change.

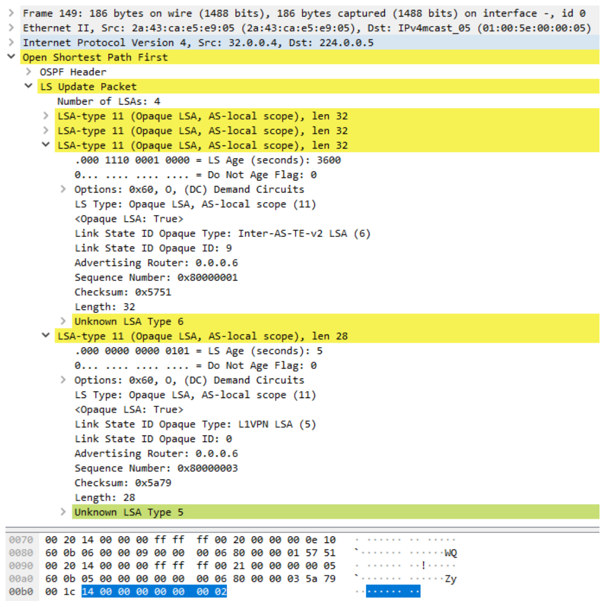
We follow up with the evolution of the global portion of the database, storing the Overlay LSAs. This evolution is presented in figure 6.20. By comparing figure 6.20.a to figure 6.20.b, we can see there are three Prefix-LSAs missing, all three generated by R6 in the first figure (for subnets 30.0.0.0/24, 32.0.0.0/24 and 33.0.0.0/24). We also see three updated ABR-LSAs, originated by R3, R4 and R6, as we can see by comparing the sequence number of these ABR-LSAs, and which present a value for their age much lower than the other LSAs in the second figure. These are the changes applied over the ABR Overlay in reaction to the link removal. Finally, in figure 6.20.c, we can see the ABR-LSAs being updated again, with the same verification, as well as three new Prefix-LSAs originated by R6, as we can see by searching the lowest aged Prefix-LSAs, and presenting a sequence number of 1 and a LS Age value lower than the remaining LSAs.

On this same note, we now see the evolution of both R4's and R6's ABR-LSAs, to give an example of their correct update. These ABR-LSAs evolution is present in figures 6.21 and 6.22, respectively. Here, we can see that after the first change was noticed, R4 removed R6 from its list of ABR neighbors, adding it back when the link between R6 and R7 was reestablished. The same is true for R6, which only left R2 described in the LSA, restoring its Area 3 neighbor ABRs after the second change was applied. We also show, in figure 6.23, the evolution of one of R6' generated Summary-LSAs, in this case the one referring to subnet 33.0.0.0/24. We can see the first advertised cost of 20 changing to 50 after the link removal, cost of its inter-area path. After the link reattachment between routers R6 and R7, we can see this cost going back to the previous 20, of the now available intra-area path.



```
> Frame 142: 154 bytes on wire (1232 bits), 154 bytes captured (1232 bits) on interface -, id 0
> Ethernet II, Src: fe:5c:58:f5:4c:df (fe:5c:58:f5:4c:df), Dst: IPv4mcast_05 (01:00:5e:00:00:05)
> Internet Protocol Version 4, Src: 32.0.0.7, Dst: 224.0.0.5
v Open Shortest Path First
  > OSPF Header
  v LS Update Packet
      Number of LSAs: 2
    v LSA-type 1 (Router-LSA), len 60
        .000 0000 0000 0001 = LS Age (seconds): 1
        0... .... .... .... = Do Not Age Flag: 0
      > Options: 0x62, O, (DC) Demand Circuits, (E) External Routing
        LS Type: Router-LSA (1)
        <Router LSA: True>
        Link State ID: 0.0.0.7
        Advertising Router: 0.0.0.7
        Sequence Number: 0x80000005
        Checksum: 0xe6fc
        Length: 60
      > Flags: 0x00
        Number of Links: 3
      > Type: Stub     ID: 34.0.0.0       Data: 255.255.255.0   Metric: 10
      > Type: Transit  ID: 33.0.0.7       Data: 33.0.0.7        Metric: 10
      > Type: Transit  ID: 32.0.0.7       Data: 32.0.0.7        Metric: 10
    v LSA-type 2 (Network-LSA), len 32
        .000 1110 0001 0000 = LS Age (seconds): 3600
        0... .... .... .... = Do Not Age Flag: 0
      > Options: 0x22, (DC) Demand Circuits, (E) External Routing
        LS Type: Network-LSA (2)
        <Network LSA: True>
        Link State ID: 34.0.0.7
        Advertising Router: 0.0.0.7
        Sequence Number: 0x80000001
        Checksum: 0x0eee
        Length: 32
        Netmask: 255.255.255.0
        Attached Router: 0.0.0.7
        Attached Router: 0.0.0.6
```

(a)



```
> Frame 149: 186 bytes on wire (1488 bits), 186 bytes captured (1488 bits) on interface -, id 0
> Ethernet II, Src: 2a:43:ca:e5:e9:05 (2a:43:ca:e5:e9:05), Dst: IPv4mcast_05 (01:00:5e:00:00:05)
> Internet Protocol Version 4, Src: 32.0.0.4, Dst: 224.0.0.5
v Open Shortest Path First
  > OSPF Header
  v LS Update Packet
      Number of LSAs: 4
    > LSA-type 11 (Opaque LSA, AS-local scope), len 32
    > LSA-type 11 (Opaque LSA, AS-local scope), len 32
    v LSA-type 11 (Opaque LSA, AS-local scope), len 32
        .000 1110 0001 0000 = LS Age (seconds): 3600
        0... .... .... .... = Do Not Age Flag: 0
      > Options: 0x60, O, (DC) Demand Circuits
        LS Type: Opaque LSA, AS-local scope (11)
        <Opaque LSA: True>
        Link State ID Opaque Type: Inter-AS-TE-v2 LSA (6)
        Link State ID Opaque ID: 9
        Advertising Router: 0.0.0.6
        Sequence Number: 0x80000001
        Checksum: 0x5751
        Length: 32
      > Unknown LSA Type 6
    v LSA-type 11 (Opaque LSA, AS-local scope), len 28
        .000 0000 0000 0101 = LS Age (seconds): 5
        0... .... .... .... = Do Not Age Flag: 0
      > Options: 0x60, O, (DC) Demand Circuits
        LS Type: Opaque LSA, AS-local scope (11)
        <Opaque LSA: True>
        Link State ID Opaque Type: L1VPN LSA (5)
        Link State ID Opaque ID: 0
        Advertising Router: 0.0.0.6
        Sequence Number: 0x80000003
        Checksum: 0x5a79
        Length: 28
      > Unknown LSA Type 5

0070  00 20 14 00 00 00 ff ff  ff 00 20 00 00 00 0e 10   · ······ ·· ·····
0080  60 0b 06 00 00 09 00 00  00 06 80 00 00 01 57 51   `······· ······WQ
0090  00 20 14 00 00 00 ff ff  ff 00 21 00 00 00 00 05   · ······ ··!·····
00a0  60 0b 05 00 00 00 00 00  00 06 80 00 00 03 5a 79   `······· ······Zy
00b0  00 1c 14 00 00 00 00 00  00 02                     ········ ··
```

(b)

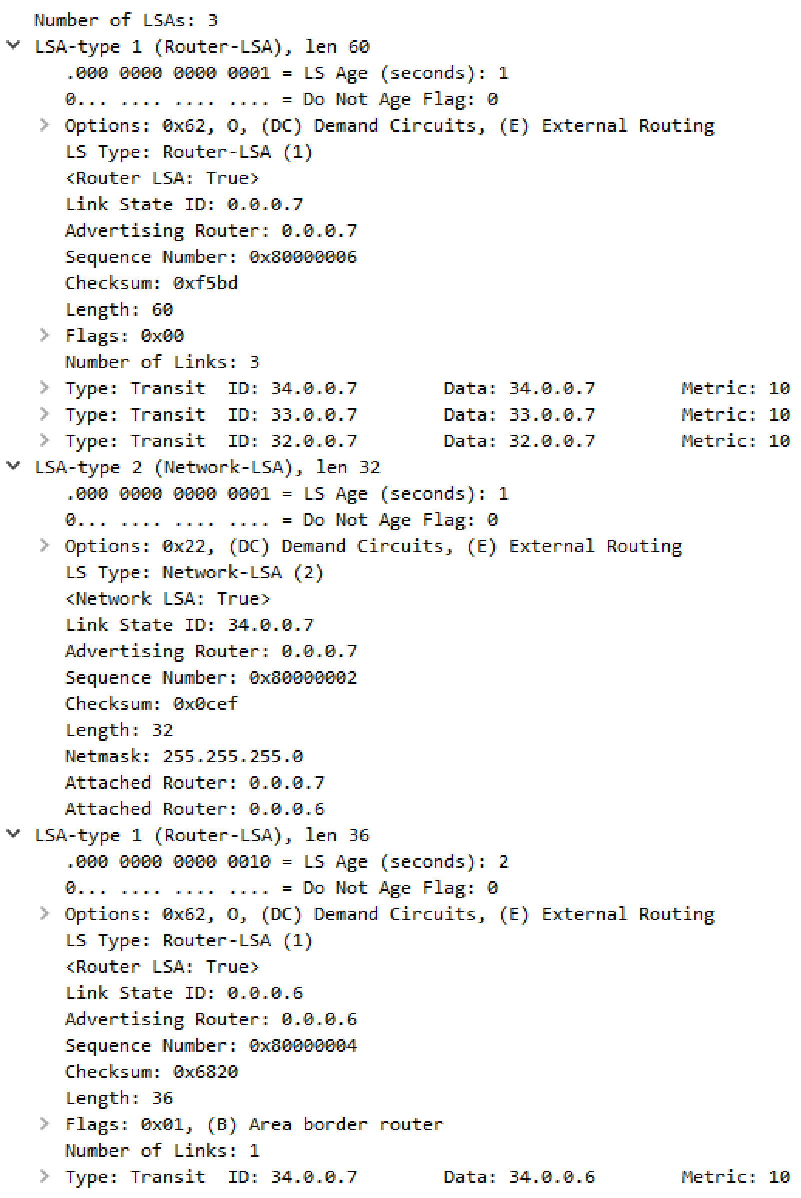
Figure 6.18: Detailed views of packets (a) 142 and (b) 149

Finally, we can look into some of the routing tables resulting from these changes. Figures 6.24, 6.25 and 6.26 show the routing table evolution of routers R4, R5 and R6, respectively. We can verify the reaction to the changes in any of these routing tables. In the case of R4, we can see that after the first change, the only route using R7 as its next hop, other that the route to subnet 34.0.0.0/24, changed its presented cost from 30 to 40 and next-hop router to R1, with this change being reversed after the last step. Similar changes occur at the other routers routing tables (e.g., R5's route to subnet 33.0.0.0/24, with the cost changing from 30 to 40, or R6's route to subnet 30.0.0.0/24, with the cost changing from 30 to 50).

6.1.5 Reaction to changes: Changes to ABR presence

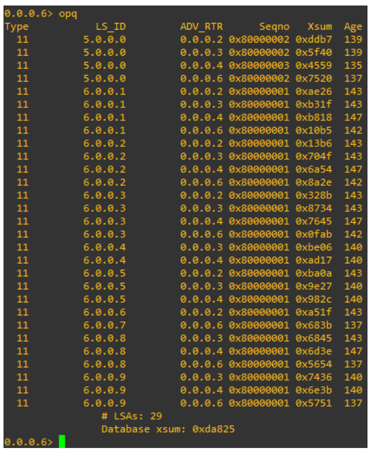
For this last functionality test, we will be deleting (simulating a crash) one of the ABRs from the network, adding a new instance of that router to replace it after the network converges from its removal. The ABR we will remove is R6. We performed a Wireshark packet capture between routers R4 and R7 until the ABR is removed from the network. Before launching the new R6 instance, we performed a new capture between routers R5 and R6, to detect the initial synchronization process.

The initial synchronization process between ABRs is the only main feature we could not completely achieve in this implementation. In the current version of the extended implementation, area-internal routers store the Overlay LSAs, and perform the usual Database Description process with the ABRs,



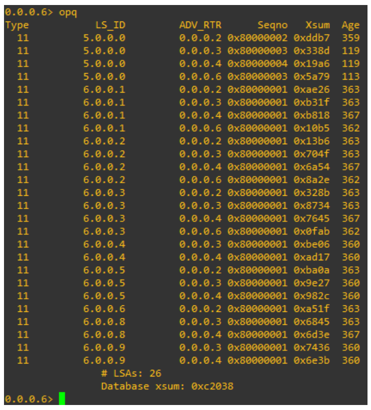
```
  Number of LSAs: 3
v LSA-type 1 (Router-LSA), len 60
    .000 0000 0000 0001 = LS Age (seconds): 1
    0... .... .... .... = Do Not Age Flag: 0
  > Options: 0x62, O, (DC) Demand Circuits, (E) External Routing
    LS Type: Router-LSA (1)
    <Router LSA: True>
    Link State ID: 0.0.0.7
    Advertising Router: 0.0.0.7
    Sequence Number: 0x80000006
    Checksum: 0xf5bd
    Length: 60
  > Flags: 0x00
    Number of Links: 3
  > Type: Transit  ID: 34.0.0.7        Data: 34.0.0.7        Metric: 10
  > Type: Transit  ID: 33.0.0.7        Data: 33.0.0.7        Metric: 10
  > Type: Transit  ID: 32.0.0.7        Data: 32.0.0.7        Metric: 10
v LSA-type 2 (Network-LSA), len 32
    .000 0000 0000 0001 = LS Age (seconds): 1
    0... .... .... .... = Do Not Age Flag: 0
  > Options: 0x22, (DC) Demand Circuits, (E) External Routing
    LS Type: Network-LSA (2)
    <Network LSA: True>
    Link State ID: 34.0.0.7
    Advertising Router: 0.0.0.7
    Sequence Number: 0x80000002
    Checksum: 0x0cef
    Length: 32
    Netmask: 255.255.255.0
    Attached Router: 0.0.0.7
    Attached Router: 0.0.0.6
v LSA-type 1 (Router-LSA), len 36
    .000 0000 0000 0010 = LS Age (seconds): 2
    0... .... .... .... = Do Not Age Flag: 0
  > Options: 0x62, O, (DC) Demand Circuits, (E) External Routing
    LS Type: Router-LSA (1)
    <Router LSA: True>
    Link State ID: 0.0.0.6
    Advertising Router: 0.0.0.6
    Sequence Number: 0x80000004
    Checksum: 0x6820
    Length: 36
  > Flags: 0x01, (B) Area border router
    Number of Links: 1
  > Type: Transit  ID: 34.0.0.7        Data: 34.0.0.6        Metric: 10
```

Figure 6.19: Detailed view of packet 191



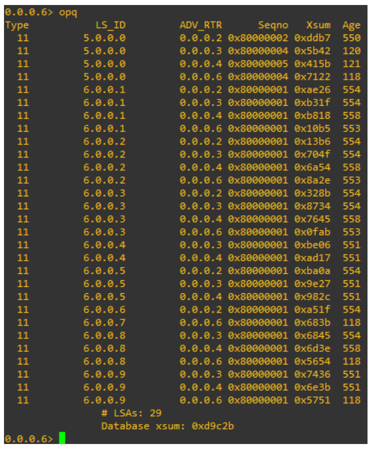
```
0.0.0.6> opq
Type             LS_ID          ADV_RTR      Seqno    Xsum   Age
  11           5.0.0.0          0.0.0.2 0x80000002 0xddb7  139
  11           5.0.0.0          0.0.0.3 0x80000002 0x5f40  139
  11           5.0.0.0          0.0.0.4 0x80000003 0x4559  135
  11           5.0.0.0          0.0.0.6 0x80000002 0x7520  137
  11           6.0.0.1          0.0.0.2 0x80000001 0xae26  143
  11           6.0.0.1          0.0.0.3 0x80000001 0xb31f  143
  11           6.0.0.1          0.0.0.4 0x80000001 0xb818  147
  11           6.0.0.1          0.0.0.6 0x80000001 0x10b5  142
  11           6.0.0.2          0.0.0.2 0x80000001 0x13b6  143
  11           6.0.0.2          0.0.0.3 0x80000001 0x704f  143
  11           6.0.0.2          0.0.0.4 0x80000001 0x6a54  147
  11           6.0.0.2          0.0.0.6 0x80000001 0x8a2e  142
  11           6.0.0.3          0.0.0.2 0x80000001 0x328b  143
  11           6.0.0.3          0.0.0.3 0x80000001 0x8734  143
  11           6.0.0.3          0.0.0.4 0x80000001 0x7645  147
  11           6.0.0.3          0.0.0.6 0x80000001 0x0fab  142
  11           6.0.0.4          0.0.0.3 0x80000001 0xbe06  140
  11           6.0.0.4          0.0.0.4 0x80000001 0xad17  140
  11           6.0.0.5          0.0.0.2 0x80000001 0xba0a  143
  11           6.0.0.5          0.0.0.3 0x80000001 0x9e27  140
  11           6.0.0.5          0.0.0.4 0x80000001 0x982c  140
  11           6.0.0.6          0.0.0.2 0x80000001 0xa51f  143
  11           6.0.0.7          0.0.0.6 0x80000001 0x683b  137
  11           6.0.0.8          0.0.0.3 0x80000001 0x6845  143
  11           6.0.0.8          0.0.0.4 0x80000001 0x6d3e  147
  11           6.0.0.8          0.0.0.6 0x80000001 0x5654  137
  11           6.0.0.9          0.0.0.3 0x80000001 0x7436  140
  11           6.0.0.9          0.0.0.4 0x80000001 0x6e3b  140
  11           6.0.0.9          0.0.0.6 0x80000001 0x5751  137
                # LSAs: 29
                Database xsum: 0xda825
0.0.0.6>
```

(a) Opaque-LSAs before link removal



```
0.0.0.6> opq
Type             LS_ID          ADV_RTR      Seqno    Xsum   Age
  11           5.0.0.0          0.0.0.2 0x80000002 0xddb7  359
  11           5.0.0.0          0.0.0.3 0x80000003 0x338d  119
  11           5.0.0.0          0.0.0.4 0x80000004 0x19a6  119
  11           5.0.0.0          0.0.0.6 0x80000003 0x5a79  113
  11           6.0.0.1          0.0.0.2 0x80000001 0xae26  363
  11           6.0.0.1          0.0.0.3 0x80000001 0xb31f  363
  11           6.0.0.1          0.0.0.4 0x80000001 0xb818  367
  11           6.0.0.1          0.0.0.6 0x80000001 0x10b5  362
  11           6.0.0.2          0.0.0.2 0x80000001 0x13b6  363
  11           6.0.0.2          0.0.0.3 0x80000001 0x704f  363
  11           6.0.0.2          0.0.0.4 0x80000001 0x6a54  367
  11           6.0.0.2          0.0.0.6 0x80000001 0x8a2e  362
  11           6.0.0.3          0.0.0.2 0x80000001 0x328b  363
  11           6.0.0.3          0.0.0.3 0x80000001 0x8734  363
  11           6.0.0.3          0.0.0.4 0x80000001 0x7645  367
  11           6.0.0.3          0.0.0.6 0x80000001 0x0fab  362
  11           6.0.0.4          0.0.0.3 0x80000001 0xbe06  360
  11           6.0.0.4          0.0.0.4 0x80000001 0xad17  360
  11           6.0.0.5          0.0.0.2 0x80000001 0xba0a  363
  11           6.0.0.5          0.0.0.3 0x80000001 0x9e27  360
  11           6.0.0.5          0.0.0.4 0x80000001 0x982c  360
  11           6.0.0.6          0.0.0.2 0x80000001 0xa51f  363
  11           6.0.0.8          0.0.0.3 0x80000001 0x6845  363
  11           6.0.0.8          0.0.0.4 0x80000001 0x6d3e  367
  11           6.0.0.9          0.0.0.3 0x80000001 0x7436  360
  11           6.0.0.9          0.0.0.4 0x80000001 0x6e3b  360
                # LSAs: 26
                Database xsum: 0xc2038
0.0.0.6>
```

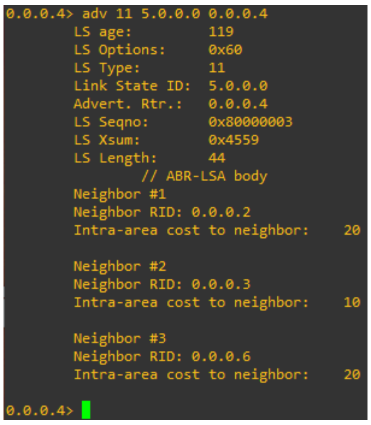
(b) Opaque-LSAs after link removal



```
0.0.0.6> opq
Type             LS_ID          ADV_RTR      Seqno    Xsum   Age
  11           5.0.0.0          0.0.0.2 0x80000002 0xddb7  550
  11           5.0.0.0          0.0.0.3 0x80000004 0x5b42  120
  11           5.0.0.0          0.0.0.4 0x80000005 0x415b  121
  11           5.0.0.0          0.0.0.6 0x80000004 0x7122  118
  11           6.0.0.1          0.0.0.2 0x80000001 0xae26  554
  11           6.0.0.1          0.0.0.3 0x80000001 0xb31f  554
  11           6.0.0.1          0.0.0.4 0x80000001 0xb818  558
  11           6.0.0.1          0.0.0.6 0x80000001 0x10b5  553
  11           6.0.0.2          0.0.0.2 0x80000001 0x13b6  554
  11           6.0.0.2          0.0.0.3 0x80000001 0x704f  554
  11           6.0.0.2          0.0.0.4 0x80000001 0x6a54  558
  11           6.0.0.2          0.0.0.6 0x80000001 0x8a2e  553
  11           6.0.0.3          0.0.0.2 0x80000001 0x328b  554
  11           6.0.0.3          0.0.0.3 0x80000001 0x8734  554
  11           6.0.0.3          0.0.0.4 0x80000001 0x7645  558
  11           6.0.0.3          0.0.0.6 0x80000001 0x0fab  553
  11           6.0.0.4          0.0.0.3 0x80000001 0xbe06  551
  11           6.0.0.4          0.0.0.4 0x80000001 0xad17  551
  11           6.0.0.5          0.0.0.2 0x80000001 0xba0a  554
  11           6.0.0.5          0.0.0.3 0x80000001 0x9e27  551
  11           6.0.0.5          0.0.0.4 0x80000001 0x982c  551
  11           6.0.0.6          0.0.0.2 0x80000001 0xa51f  554
  11           6.0.0.7          0.0.0.6 0x80000001 0x683b  118
  11           6.0.0.8          0.0.0.3 0x80000001 0x6845  554
  11           6.0.0.8          0.0.0.4 0x80000001 0x6d3e  558
  11           6.0.0.8          0.0.0.6 0x80000001 0x5654  118
  11           6.0.0.9          0.0.0.3 0x80000001 0x7436  551
  11           6.0.0.9          0.0.0.4 0x80000001 0x6e3b  551
  11           6.0.0.9          0.0.0.6 0x80000001 0x5751  118
                # LSAs: 29
                Database xsum: 0xd9c2b
0.0.0.6>
```

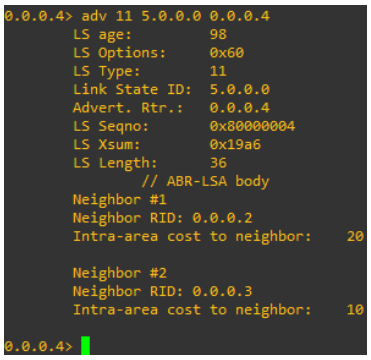
(c) Opaque-LSAs after reattaching the link

Figure 6.20: Evolution of the global portion of the database



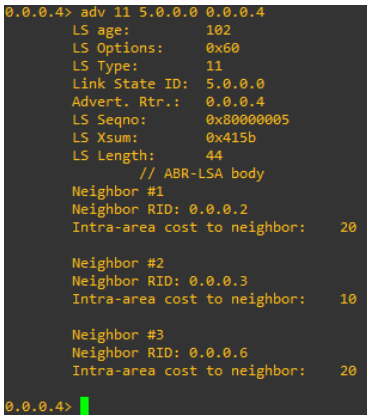
```
0.0.0.4> adv 11 5.0.0.0 0.0.0.4
        LS age:         119
        LS Options:     0x60
        LS Type:        11
        Link State ID:  5.0.0.0
        Advert. Rtr.:   0.0.0.4
        LS Seqno:       0x80000003
        LS Xsum:        0x4559
        LS Length:      44
                // ABR-LSA body
        Neighbor #1
        Neighbor RID: 0.0.0.2
        Intra-area cost to neighbor:    20

        Neighbor #2
        Neighbor RID: 0.0.0.3
        Intra-area cost to neighbor:    10

        Neighbor #3
        Neighbor RID: 0.0.0.6
        Intra-area cost to neighbor:    20

0.0.0.4>
```

(a) Before link removal



```
0.0.0.4> adv 11 5.0.0.0 0.0.0.4
        LS age:         98
        LS Options:     0x60
        LS Type:        11
        Link State ID:  5.0.0.0
        Advert. Rtr.:   0.0.0.4
        LS Seqno:       0x80000004
        LS Xsum:        0x19a6
        LS Length:      36
                // ABR-LSA body
        Neighbor #1
        Neighbor RID: 0.0.0.2
        Intra-area cost to neighbor:    20

        Neighbor #2
        Neighbor RID: 0.0.0.3
        Intra-area cost to neighbor:    10

0.0.0.4>
```

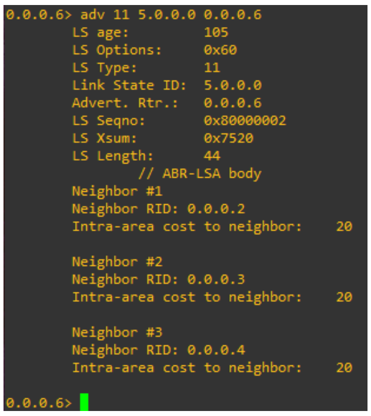
(b) After link removal



```
0.0.0.4> adv 11 5.0.0.0 0.0.0.4
        LS age:         102
        LS Options:     0x60
        LS Type:        11
        Link State ID:  5.0.0.0
        Advert. Rtr.:   0.0.0.4
        LS Seqno:       0x80000005
        LS Xsum:        0x415b
        LS Length:      44
                // ABR-LSA body
        Neighbor #1
        Neighbor RID: 0.0.0.2
        Intra-area cost to neighbor:    20

        Neighbor #2
        Neighbor RID: 0.0.0.3
        Intra-area cost to neighbor:    10

        Neighbor #3
        Neighbor RID: 0.0.0.6
        Intra-area cost to neighbor:    20

0.0.0.4>
```

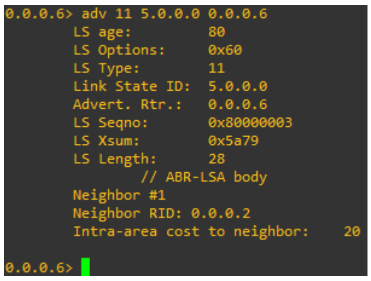
(c) After reattaching the link

Figure 6.21: Evolution of R4's ABR-LSA



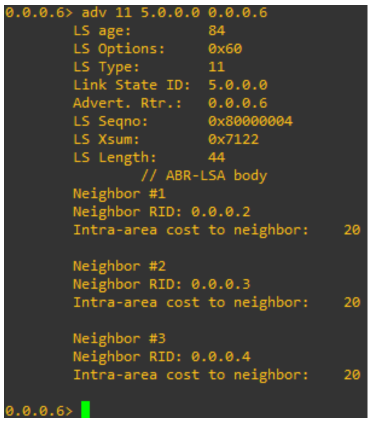
```
0.0.0.6> adv 11 5.0.0.0 0.0.0.6
        LS age:         105
        LS Options:     0x60
        LS Type:        11
        Link State ID:  5.0.0.0
        Advert. Rtr.:   0.0.0.6
        LS Seqno:       0x80000002
        LS Xsum:        0x7520
        LS Length:      44
                // ABR-LSA body
        Neighbor #1
        Neighbor RID: 0.0.0.2
        Intra-area cost to neighbor:    20

        Neighbor #2
        Neighbor RID: 0.0.0.3
        Intra-area cost to neighbor:    20

        Neighbor #3
        Neighbor RID: 0.0.0.4
        Intra-area cost to neighbor:    20

0.0.0.6>
```

(a) Before link removal



```
0.0.0.6> adv 11 5.0.0.0 0.0.0.6
        LS age:         80
        LS Options:     0x60
        LS Type:        11
        Link State ID:  5.0.0.0
        Advert. Rtr.:   0.0.0.6
        LS Seqno:       0x80000003
        LS Xsum:        0x5a79
        LS Length:      28
                // ABR-LSA body
        Neighbor #1
        Neighbor RID: 0.0.0.2
        Intra-area cost to neighbor:    20

0.0.0.6>
```

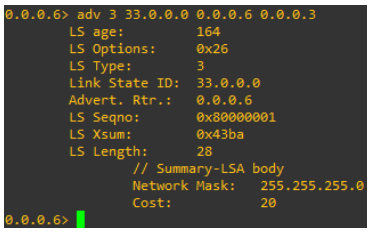
(b) After link removal



```
0.0.0.6> adv 11 5.0.0.0 0.0.0.6
        LS age:         84
        LS Options:     0x60
        LS Type:        11
        Link State ID:  5.0.0.0
        Advert. Rtr.:   0.0.0.6
        LS Seqno:       0x80000004
        LS Xsum:        0x7122
        LS Length:      44
                // ABR-LSA body
        Neighbor #1
        Neighbor RID: 0.0.0.2
        Intra-area cost to neighbor:    20

        Neighbor #2
        Neighbor RID: 0.0.0.3
        Intra-area cost to neighbor:    20

        Neighbor #3
        Neighbor RID: 0.0.0.4
        Intra-area cost to neighbor:    20

0.0.0.6>
```

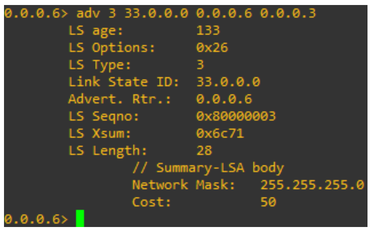
(c) After reattaching the link

Figure 6.22: Evolution of R6's ABR-LSA



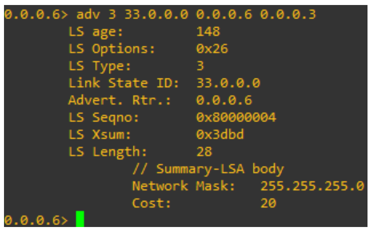
```
0.0.0.6> adv 3 33.0.0.0 0.0.0.6 0.0.0.3
        LS age:         164
        LS Options:     0x26
        LS Type:        3
        Link State ID:  33.0.0.0
        Advert. Rtr.:   0.0.0.6
        LS Seqno:       0x80000001
        LS Xsum:        0x43ba
        LS Length:      28
                // Summary-LSA body
                Network Mask:   255.255.255.0
                Cost:           20
0.0.0.6>
```

(a) Before link removal



```
0.0.0.6> adv 3 33.0.0.0 0.0.0.6 0.0.0.3
        LS age:         133
        LS Options:     0x26
        LS Type:        3
        Link State ID:  33.0.0.0
        Advert. Rtr.:   0.0.0.6
        LS Seqno:       0x80000003
        LS Xsum:        0x6c71
        LS Length:      28
                // Summary-LSA body
                Network Mask:   255.255.255.0
                Cost:           50
0.0.0.6>
```

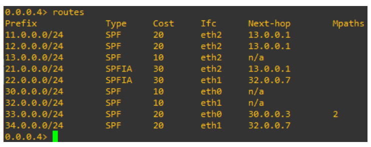
(b) After link removal



```
0.0.0.6> adv 3 33.0.0.0 0.0.0.6 0.0.0.3
        LS age:         148
        LS Options:     0x26
        LS Type:        3
        Link State ID:  33.0.0.0
        Advert. Rtr.:   0.0.0.6
        LS Seqno:       0x80000004
        LS Xsum:        0x3dbd
        LS Length:      28
                // Summary-LSA body
                Network Mask:   255.255.255.0
                Cost:           20
0.0.0.6>
```

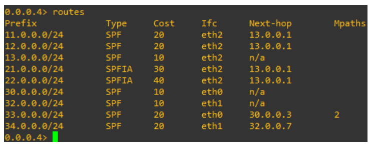
(c) After reattaching the link

Figure 6.23: Evolution of one of R6's originated Summary-LSAs (generated for subnet 33.0.0.0/24)



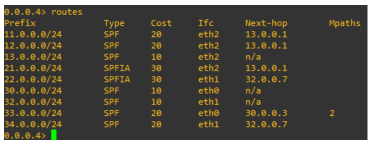
```
0.0.0.4> routes
Prefix          Type    Cost    Ifc     Next-hop        Mpaths
11.0.0.0/24     SPF     20      eth2    13.0.0.1
12.0.0.0/24     SPF     20      eth2    13.0.0.1
13.0.0.0/24     SPF     10      eth2    n/a
21.0.0.0/24     SPFIA   30      eth2    13.0.0.1
22.0.0.0/24     SPFIA   30      eth1    32.0.0.7
30.0.0.0/24     SPF     10      eth0    n/a
32.0.0.0/24     SPF     10      eth1    n/a
33.0.0.0/24     SPF     20      eth0    30.0.0.3        2
34.0.0.0/24     SPF     20      eth1    32.0.0.7
0.0.0.4>
```

(a) Before link removal



```
0.0.0.4> routes
Prefix          Type    Cost    Ifc     Next-hop        Mpaths
11.0.0.0/24     SPF     20      eth2    13.0.0.1
12.0.0.0/24     SPF     20      eth2    13.0.0.1
13.0.0.0/24     SPF     10      eth2    n/a
21.0.0.0/24     SPFIA   30      eth2    13.0.0.1
22.0.0.0/24     SPFIA   40      eth2    13.0.0.1
30.0.0.0/24     SPF     10      eth0    n/a
32.0.0.0/24     SPF     10      eth1    n/a
33.0.0.0/24     SPF     20      eth0    30.0.0.3        2
34.0.0.0/24     SPF     20      eth1    32.0.0.7
0.0.0.4>
```

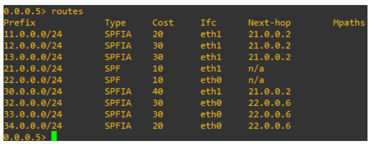
(b) After link removal



```
0.0.0.4> routes
Prefix          Type    Cost    Ifc     Next-hop        Mpaths
11.0.0.0/24     SPF     20      eth2    13.0.0.1
12.0.0.0/24     SPF     20      eth2    13.0.0.1
13.0.0.0/24     SPF     10      eth2    n/a
21.0.0.0/24     SPFIA   30      eth2    13.0.0.1
22.0.0.0/24     SPFIA   30      eth1    32.0.0.7
30.0.0.0/24     SPF     10      eth0    n/a
32.0.0.0/24     SPF     10      eth1    n/a
33.0.0.0/24     SPF     20      eth0    30.0.0.3        2
34.0.0.0/24     SPF     20      eth1    32.0.0.7
0.0.0.4>
```

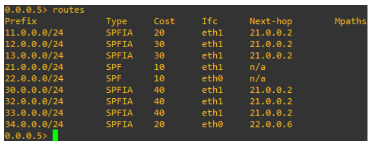
(c) After reattaching the link

Figure 6.24: Evolution of R4's routing table



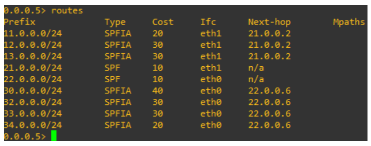
```
0.0.0.5> routes
Prefix          Type    Cost    Ifc     Next-hop        Mpaths
11.0.0.0/24     SPFIA   20      eth1    21.0.0.2
12.0.0.0/24     SPFIA   30      eth1    21.0.0.2
13.0.0.0/24     SPFIA   30      eth1    21.0.0.2
21.0.0.0/24     SPF     10      eth1    n/a
22.0.0.0/24     SPF     10      eth0    n/a
30.0.0.0/24     SPFIA   40      eth1    21.0.0.2
32.0.0.0/24     SPFIA   30      eth0    22.0.0.6
33.0.0.0/24     SPFIA   30      eth0    22.0.0.6
34.0.0.0/24     SPFIA   20      eth0    22.0.0.6
0.0.0.5>
```

(a) Before link removal



```
0.0.0.5> routes
Prefix          Type    Cost    Ifc     Next-hop        Mpaths
11.0.0.0/24     SPFIA   20      eth1    21.0.0.2
12.0.0.0/24     SPFIA   30      eth1    21.0.0.2
13.0.0.0/24     SPFIA   30      eth1    21.0.0.2
21.0.0.0/24     SPF     10      eth1    n/a
22.0.0.0/24     SPF     10      eth0    n/a
30.0.0.0/24     SPFIA   40      eth1    21.0.0.2
32.0.0.0/24     SPFIA   40      eth1    21.0.0.2
33.0.0.0/24     SPFIA   40      eth1    21.0.0.2
34.0.0.0/24     SPFIA   20      eth0    22.0.0.6
0.0.0.5>
```

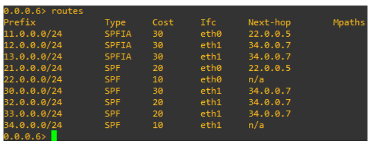
(b) After link removal



```
0.0.0.5> routes
Prefix          Type    Cost    Ifc     Next-hop        Mpaths
11.0.0.0/24     SPFIA   20      eth1    21.0.0.2
12.0.0.0/24     SPFIA   30      eth1    21.0.0.2
13.0.0.0/24     SPFIA   30      eth1    21.0.0.2
21.0.0.0/24     SPF     10      eth1    n/a
22.0.0.0/24     SPF     10      eth0    n/a
30.0.0.0/24     SPFIA   40      eth0    22.0.0.6
32.0.0.0/24     SPFIA   30      eth0    22.0.0.6
33.0.0.0/24     SPFIA   30      eth0    22.0.0.6
34.0.0.0/24     SPFIA   20      eth0    22.0.0.6
0.0.0.5>
```

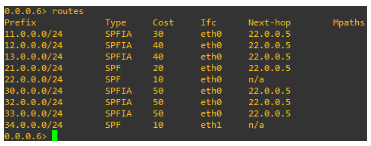
(c) After reattaching the link

Figure 6.25: Evolution of R5's routing table



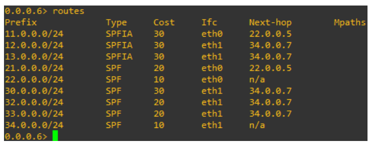
```
0.0.0.6> routes
Prefix              Type     Cost     Ifc      Next-hop        Mpaths
11.0.0.0/24         SPFIA    30       eth0     22.0.0.5
12.0.0.0/24         SPFIA    30       eth1     34.0.0.7
13.0.0.0/24         SPFIA    30       eth1     34.0.0.7
21.0.0.0/24         SPF      20       eth0     22.0.0.5
22.0.0.0/24         SPF      10       eth0     n/a
30.0.0.0/24         SPF      30       eth1     34.0.0.7
32.0.0.0/24         SPF      20       eth1     34.0.0.7
33.0.0.0/24         SPF      20       eth1     34.0.0.7
34.0.0.0/24         SPF      10       eth1     n/a
0.0.0.6>
```

(a) Before link removal



```
0.0.0.6> routes
Prefix              Type     Cost     Ifc      Next-hop        Mpaths
11.0.0.0/24         SPFIA    30       eth0     22.0.0.5
12.0.0.0/24         SPFIA    40       eth0     22.0.0.5
13.0.0.0/24         SPFIA    40       eth0     22.0.0.5
21.0.0.0/24         SPF      20       eth0     22.0.0.5
22.0.0.0/24         SPF      10       eth0     n/a
30.0.0.0/24         SPFIA    50       eth0     22.0.0.5
32.0.0.0/24         SPFIA    50       eth0     22.0.0.5
33.0.0.0/24         SPFIA    50       eth0     22.0.0.5
34.0.0.0/24         SPF      10       eth1     n/a
0.0.0.6>
```

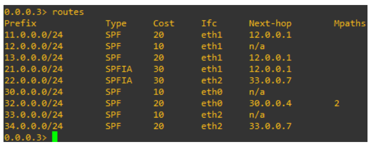
(b) After link removal



```
0.0.0.6> routes
Prefix              Type     Cost     Ifc      Next-hop        Mpaths
11.0.0.0/24         SPFIA    30       eth0     22.0.0.5
12.0.0.0/24         SPFIA    30       eth1     34.0.0.7
13.0.0.0/24         SPFIA    30       eth1     34.0.0.7
21.0.0.0/24         SPF      20       eth0     22.0.0.5
22.0.0.0/24         SPF      10       eth0     n/a
30.0.0.0/24         SPF      30       eth1     34.0.0.7
32.0.0.0/24         SPF      20       eth1     34.0.0.7
33.0.0.0/24         SPF      20       eth1     34.0.0.7
34.0.0.0/24         SPF      10       eth1     n/a
0.0.0.6>
```

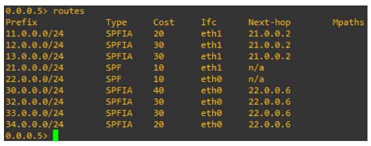
(c) After reattaching the link

Figure 6.26: Evolution of R6's routing table



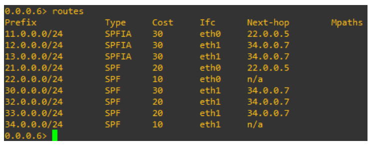
```
0.0.0.3> routes
Prefix              Type     Cost     Ifc      Next-hop        Mpaths
11.0.0.0/24         SPF      20       eth1     12.0.0.1
12.0.0.0/24         SPF      10       eth1     n/a
13.0.0.0/24         SPF      20       eth1     12.0.0.1
21.0.0.0/24         SPFIA    30       eth1     12.0.0.1
22.0.0.0/24         SPFIA    30       eth2     33.0.0.7
30.0.0.0/24         SPF      10       eth0     n/a
32.0.0.0/24         SPF      20       eth0     30.0.0.4        2
33.0.0.0/24         SPF      10       eth2     n/a
34.0.0.0/24         SPF      20       eth2     33.0.0.7
0.0.0.3>
```

(a)



```
0.0.0.5> routes
Prefix              Type     Cost     Ifc      Next-hop        Mpaths
11.0.0.0/24         SPFIA    20       eth1     21.0.0.2
12.0.0.0/24         SPFIA    30       eth1     21.0.0.2
13.0.0.0/24         SPFIA    30       eth1     21.0.0.2
21.0.0.0/24         SPF      10       eth1     n/a
22.0.0.0/24         SPF      10       eth0     n/a
30.0.0.0/24         SPFIA    40       eth0     22.0.0.6
32.0.0.0/24         SPFIA    30       eth0     22.0.0.6
33.0.0.0/24         SPFIA    30       eth0     22.0.0.6
34.0.0.0/24         SPFIA    20       eth0     22.0.0.6
0.0.0.5>
```

(b)



```
0.0.0.6> routes
Prefix              Type     Cost     Ifc      Next-hop        Mpaths
11.0.0.0/24         SPFIA    30       eth0     22.0.0.5
12.0.0.0/24         SPFIA    30       eth1     34.0.0.7
13.0.0.0/24         SPFIA    30       eth1     34.0.0.7
21.0.0.0/24         SPF      20       eth0     22.0.0.5
22.0.0.0/24         SPF      10       eth0     n/a
30.0.0.0/24         SPF      30       eth1     34.0.0.7
32.0.0.0/24         SPF      20       eth1     34.0.0.7
33.0.0.0/24         SPF      20       eth1     34.0.0.7
34.0.0.0/24         SPF      10       eth1     n/a
0.0.0.6>
```

(c)

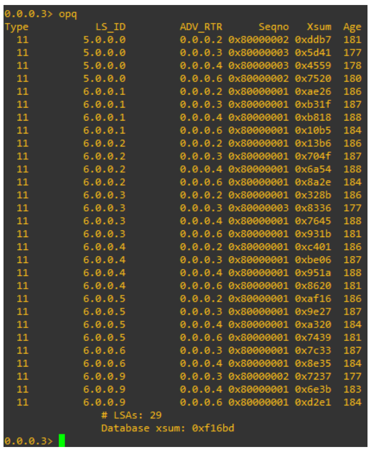
Figure 6.27: Routing tables of (a) R3, (b) R5 and (c) R6 obtained before removing R6

with the addition of sending the Overlay LSAs to the requesting ABRs. We store the Overlay LSAs in the area-internal routers, even though they never process them, not only for this reason but also due to issues that would arise within the base implementation when we tried to discard them after the flooding procedure. We decided to settle for this, since we aim, above other aspects, to guarantee a correct initial synchronization process in the ABR side, and correct convergence afterwards. In a future version of this extension, it would be ideal for area-internal routers to not store the Overlay LSAs, and a process similar to the Initial Database Synchronization to be performed between ABRs, but exclusively for Overlay LSAs.

We started the test by initializing all the routers, following the topology of figure 6.1. The initial convergence proceeded as usual. We leave figure 6.27 to show some of the information gathered with the monitoring tool. Figure 6.27.a, figure 6.27.b and figure 6.27.c show the routing tables obtained in routers R3, R5 and R6, respectively.

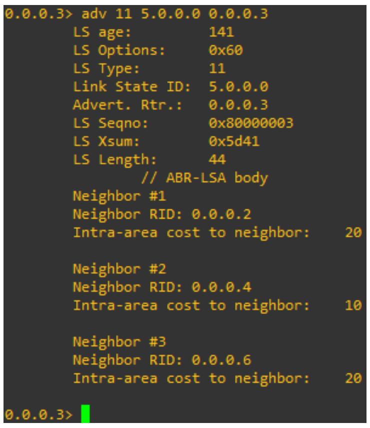
We also show, in figure 6.28.a, the initial state of the Overlay LSA portion of the database, and in figures 6.28.b and 6.28.c the initially obtained ABR-LSAs for R3 and R6, respectively.

After we finished gathering the initial information, we removed R6 from the topology. When the Hello Protocol detected that R6 could not be reached, the network reacted to this change, captured in part with the Wireshark packet capture we first set up between routers R4 and R7. The summary of the portion of this capture concerning the convergence of the network can be found in figure 6.29. The contents of this captured packets is essentially the same as those for the network reaction to a link being broken between two routers, including updates to R7's Router-LSA, to the Network-LSA for subnet 34.0.0.0/24, to R3's and R4's ABR-LSAs and to changed Summary-LSAs by R3 and R4. Figure 6.30 shows details of one of the packets captured from this change, in this case of R4 updating its ABR-LSA, as it no longer has an intra-area route available to reach R6.



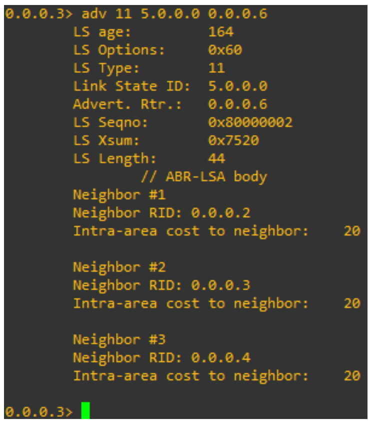
```
0.0.0.3> opq
Type         LS_ID            ADV_RTR      Seqno   Xsum  Age
  11       5.0.0.0         0.0.0.2 0x80000002 0xddb7  181
  11       5.0.0.0         0.0.0.3 0x80000003 0x5d41  177
  11       5.0.0.0         0.0.0.4 0x80000003 0x4559  178
  11       5.0.0.0         0.0.0.6 0x80000002 0x7520  180
  11       6.0.0.1         0.0.0.2 0x80000001 0xae26  186
  11       6.0.0.1         0.0.0.3 0x80000001 0xb31f  187
  11       6.0.0.1         0.0.0.4 0x80000001 0xb818  188
  11       6.0.0.1         0.0.0.6 0x80000001 0x10b5  184
  11       6.0.0.2         0.0.0.2 0x80000001 0x13b6  186
  11       6.0.0.2         0.0.0.3 0x80000001 0x704f  187
  11       6.0.0.2         0.0.0.4 0x80000001 0x6a54  188
  11       6.0.0.2         0.0.0.6 0x80000001 0x8a2e  184
  11       6.0.0.3         0.0.0.2 0x80000001 0x328b  186
  11       6.0.0.3         0.0.0.3 0x80000003 0x8336  177
  11       6.0.0.3         0.0.0.4 0x80000001 0x7645  188
  11       6.0.0.3         0.0.0.6 0x80000001 0x931b  181
  11       6.0.0.4         0.0.0.2 0x80000001 0xc401  186
  11       6.0.0.4         0.0.0.3 0x80000001 0xbe06  187
  11       6.0.0.4         0.0.0.4 0x80000001 0x951a  188
  11       6.0.0.4         0.0.0.6 0x80000001 0x8620  181
  11       6.0.0.5         0.0.0.2 0x80000001 0xaf16  186
  11       6.0.0.5         0.0.0.3 0x80000001 0x9e27  187
  11       6.0.0.5         0.0.0.4 0x80000001 0xa320  184
  11       6.0.0.5         0.0.0.6 0x80000001 0x7439  181
  11       6.0.0.6         0.0.0.3 0x80000001 0x7c33  187
  11       6.0.0.6         0.0.0.4 0x80000001 0x8e35  184
  11       6.0.0.9         0.0.0.3 0x80000002 0x7237  177
  11       6.0.0.9         0.0.0.4 0x80000001 0x6e3b  183
  11       6.0.0.9         0.0.0.6 0x80000001 0xd2e1  184
                 # LSAs: 29
                 Database xsum: 0xf16bd
0.0.0.3>
```

(a)



```
0.0.0.3> adv 11 5.0.0.0 0.0.0.3
        LS age:         141
        LS Options:     0x60
        LS Type:        11
        Link State ID:  5.0.0.0
        Advert. Rtr.:   0.0.0.3
        LS Seqno:       0x80000003
        LS Xsum:        0x5d41
        LS Length:      44
                // ABR-LSA body
        Neighbor #1
        Neighbor RID: 0.0.0.2
        Intra-area cost to neighbor:    20

        Neighbor #2
        Neighbor RID: 0.0.0.4
        Intra-area cost to neighbor:    10

        Neighbor #3
        Neighbor RID: 0.0.0.6
        Intra-area cost to neighbor:    20

0.0.0.3>
```

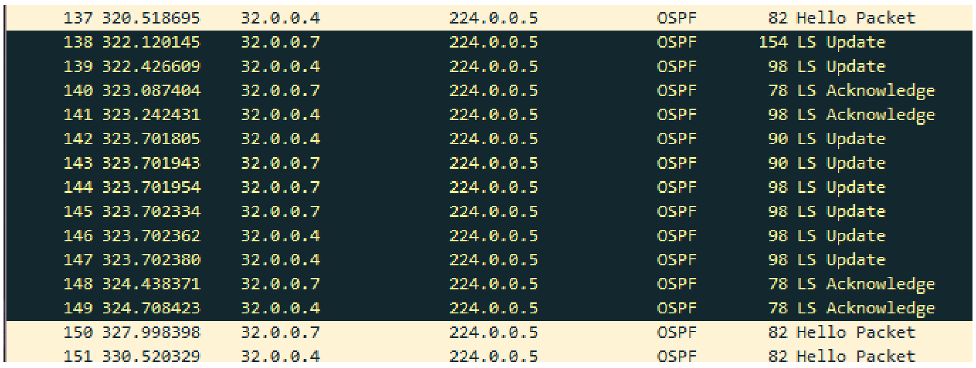
(b)



```
0.0.0.3> adv 11 5.0.0.0 0.0.0.6
        LS age:         164
        LS Options:     0x60
        LS Type:        11
        Link State ID:  5.0.0.0
        Advert. Rtr.:   0.0.0.6
        LS Seqno:       0x80000002
        LS Xsum:        0x7520
        LS Length:      44
                // ABR-LSA body
        Neighbor #1
        Neighbor RID: 0.0.0.2
        Intra-area cost to neighbor:    20

        Neighbor #2
        Neighbor RID: 0.0.0.3
        Intra-area cost to neighbor:    20

        Neighbor #3
        Neighbor RID: 0.0.0.4
        Intra-area cost to neighbor:    20

0.0.0.3>
```

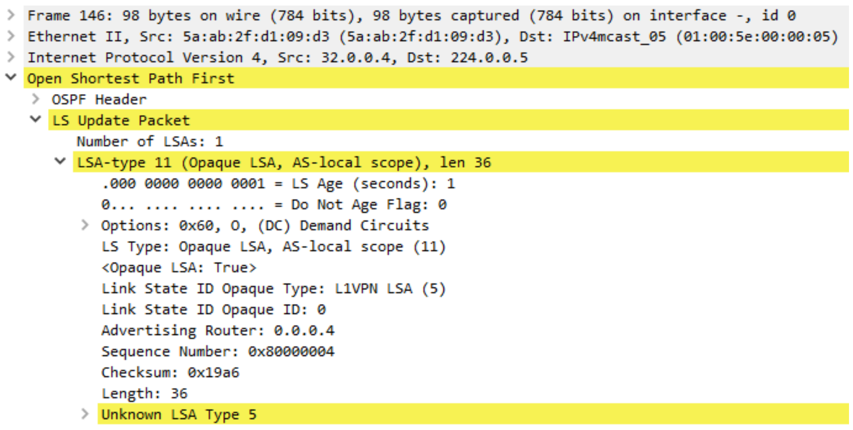
(c)

Figure 6.28: (a) Overlay database and ABR-LSAs, generated by (b) R3 and (c) R6, before removing R6



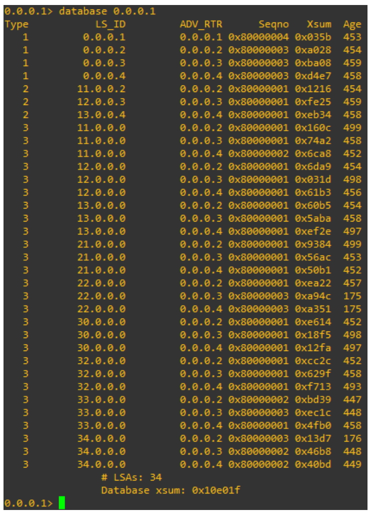
```
137 320.518695   32.0.0.4        224.0.0.5       OSPF      82 Hello Packet
138 322.120145   32.0.0.7        224.0.0.5       OSPF     154 LS Update
139 322.426609   32.0.0.4        224.0.0.5       OSPF      98 LS Update
140 323.087404   32.0.0.7        224.0.0.5       OSPF      78 LS Acknowledge
141 323.242431   32.0.0.4        224.0.0.5       OSPF      98 LS Acknowledge
142 323.701805   32.0.0.4        224.0.0.5       OSPF      90 LS Update
143 323.701943   32.0.0.7        224.0.0.5       OSPF      90 LS Update
144 323.701954   32.0.0.7        224.0.0.5       OSPF      98 LS Update
145 323.702334   32.0.0.7        224.0.0.5       OSPF      98 LS Update
146 323.702362   32.0.0.4        224.0.0.5       OSPF      98 LS Update
147 323.702380   32.0.0.4        224.0.0.5       OSPF      98 LS Update
148 324.438371   32.0.0.7        224.0.0.5       OSPF      78 LS Acknowledge
149 324.708423   32.0.0.4        224.0.0.5       OSPF      78 LS Acknowledge
150 327.998398   32.0.0.7        224.0.0.5       OSPF      82 Hello Packet
151 330.520329   32.0.0.4        224.0.0.5       OSPF      82 Hello Packet
```

Figure 6.29: Summary of the packets captured after the ABR was removed

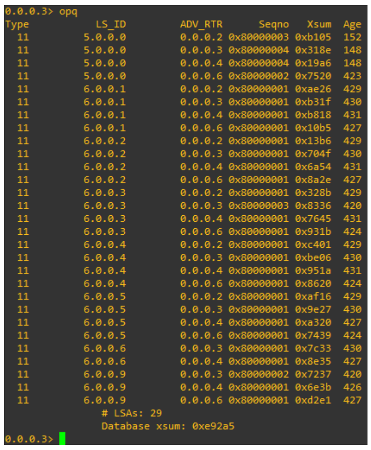


```
> Frame 146: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface -, id 0
> Ethernet II, Src: 5a:ab:2f:d1:09:d3 (5a:ab:2f:d1:09:d3), Dst: IPv4mcast_05 (01:00:5e:00:00:05)
> Internet Protocol Version 4, Src: 32.0.0.4, Dst: 224.0.0.5
v Open Shortest Path First
  > OSPF Header
  v LS Update Packet
      Number of LSAs: 1
    v LSA-type 11 (Opaque LSA, AS-local scope), len 36
        .000 0000 0000 0001 = LS Age (seconds): 1
        0... .... .... .... = Do Not Age Flag: 0
      > Options: 0x60, O, (DC) Demand Circuits
        LS Type: Opaque LSA, AS-local scope (11)
        <Opaque LSA: True>
        Link State ID Opaque Type: L1VPN LSA (5)
        Link State ID Opaque ID: 0
        Advertising Router: 0.0.0.4
        Sequence Number: 0x80000004
        Checksum: 0x19a6
        Length: 36
      > Unknown LSA Type 5
```

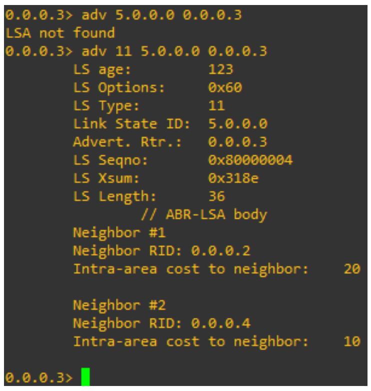
Figure 6.30: Detailed view of packet 146



```
0.0.0.1> database 0.0.0.1
Type            LS_ID           ADV_RTR      Seqno   Xsum  Age
   1          0.0.0.1           0.0.0.1 0x80000004 0x035b  453
   1          0.0.0.2           0.0.0.2 0x80000003 0xa028  454
   1          0.0.0.3           0.0.0.3 0x80000003 0xba08  459
   1          0.0.0.4           0.0.0.4 0x80000003 0xd4e7  458
   2         11.0.0.2           0.0.0.2 0x80000001 0x1216  454
   2         12.0.0.3           0.0.0.3 0x80000001 0xfe25  459
   2         13.0.0.4           0.0.0.4 0x80000001 0xeb34  458
   3         11.0.0.0           0.0.0.2 0x80000001 0x160c  499
   3         11.0.0.0           0.0.0.3 0x80000001 0x74a2  458
   3         11.0.0.0           0.0.0.4 0x80000002 0x6ca8  452
   3         12.0.0.0           0.0.0.2 0x80000001 0x6da9  454
   3         12.0.0.0           0.0.0.3 0x80000001 0x031d  498
   3         12.0.0.0           0.0.0.4 0x80000001 0x61b3  456
   3         13.0.0.0           0.0.0.2 0x80000001 0x60b5  454
   3         13.0.0.0           0.0.0.3 0x80000001 0x5aba  458
   3         13.0.0.0           0.0.0.4 0x80000001 0xef2e  497
   3         21.0.0.0           0.0.0.2 0x80000001 0x9384  499
   3         21.0.0.0           0.0.0.3 0x80000001 0x56ac  453
   3         21.0.0.0           0.0.0.4 0x80000001 0x50b1  452
   3         22.0.0.0           0.0.0.2 0x80000001 0xea22  457
   3         22.0.0.0           0.0.0.3 0x80000003 0xa94c  175
   3         22.0.0.0           0.0.0.4 0x80000003 0xa351  175
   3         30.0.0.0           0.0.0.2 0x80000001 0xe614  452
   3         30.0.0.0           0.0.0.3 0x80000001 0x18f5  498
   3         30.0.0.0           0.0.0.4 0x80000001 0x12fa  497
   3         32.0.0.0           0.0.0.2 0x80000001 0xcc2c  452
   3         32.0.0.0           0.0.0.3 0x80000001 0x629f  458
   3         32.0.0.0           0.0.0.4 0x80000001 0xf713  493
   3         33.0.0.0           0.0.0.2 0x80000002 0xbd39  447
   3         33.0.0.0           0.0.0.3 0x80000003 0xec1c  448
   3         33.0.0.0           0.0.0.4 0x80000001 0x4fb0  458
   3         34.0.0.0           0.0.0.2 0x80000003 0x13d7  176
   3         34.0.0.0           0.0.0.3 0x80000002 0x46b8  448
   3         34.0.0.0           0.0.0.4 0x80000002 0x40bd  449
               # LSAs: 34
               Database xsum: 0x10e01f
0.0.0.1>
```



```
0.0.0.3> opq
Type            LS_ID           ADV_RTR      Seqno   Xsum  Age
  11          5.0.0.0           0.0.0.2 0x80000003 0xb105  152
  11          5.0.0.0           0.0.0.3 0x80000004 0x318e  148
  11          5.0.0.0           0.0.0.4 0x80000004 0x19a6  148
  11          5.0.0.0           0.0.0.6 0x80000002 0x7520  423
  11          6.0.0.1           0.0.0.2 0x80000001 0xae26  429
  11          6.0.0.1           0.0.0.3 0x80000001 0xb31f  430
  11          6.0.0.1           0.0.0.4 0x80000001 0xb818  431
  11          6.0.0.1           0.0.0.6 0x80000001 0x10b5  427
  11          6.0.0.2           0.0.0.2 0x80000001 0x13b6  429
  11          6.0.0.2           0.0.0.3 0x80000001 0x704f  430
  11          6.0.0.2           0.0.0.4 0x80000001 0x6a54  431
  11          6.0.0.2           0.0.0.6 0x80000001 0x8a2e  427
  11          6.0.0.3           0.0.0.2 0x80000001 0x328b  429
  11          6.0.0.3           0.0.0.3 0x80000003 0x8336  420
  11          6.0.0.3           0.0.0.4 0x80000001 0x7645  431
  11          6.0.0.3           0.0.0.6 0x80000001 0x931b  424
  11          6.0.0.4           0.0.0.2 0x80000001 0xc401  429
  11          6.0.0.4           0.0.0.3 0x80000001 0xbe06  430
  11          6.0.0.4           0.0.0.4 0x80000001 0x951a  431
  11          6.0.0.4           0.0.0.6 0x80000001 0x8620  424
  11          6.0.0.5           0.0.0.2 0x80000001 0xaf16  429
  11          6.0.0.5           0.0.0.3 0x80000001 0x9e27  430
  11          6.0.0.5           0.0.0.4 0x80000001 0xa320  427
  11          6.0.0.5           0.0.0.6 0x80000001 0x7439  424
  11          6.0.0.6           0.0.0.3 0x80000001 0x7c33  430
  11          6.0.0.6           0.0.0.4 0x80000001 0x8e35  427
  11          6.0.0.9           0.0.0.3 0x80000002 0x7237  420
  11          6.0.0.9           0.0.0.4 0x80000001 0x6e3b  426
  11          6.0.0.9           0.0.0.6 0x80000001 0xd2e1  427
               # LSAs: 29
               Database xsum: 0xe92a5
0.0.0.3>
```



```
0.0.0.3> adv 5.0.0.0 0.0.0.3
LSA not found
0.0.0.3> adv 11 5.0.0.0 0.0.0.3
        LS age:         123
        LS Options:     0x60
        LS Type:        11
        Link State ID:  5.0.0.0
        Advert. Rtr.:   0.0.0.3
        LS Seqno:       0x80000004
        LS Xsum:        0x318e
        LS Length:      36
                // ABR-LSA body
        Neighbor #1
        Neighbor RID: 0.0.0.2
        Intra-area cost to neighbor:    20

        Neighbor #2
        Neighbor RID: 0.0.0.4
        Intra-area cost to neighbor:    10

0.0.0.3>
```

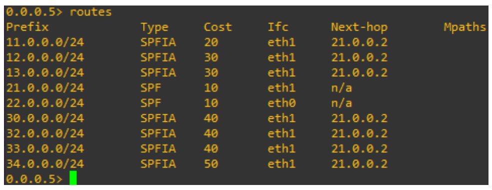
(a) (b) (c)

Figure 6.31: State of (a) Area 1's database, (b) the database Overlay portion and (c) R3's ABR-LSA after removing the ABR

With the ABR removed, we re-examined the information available through the monitoring tool. In figure 6.31, we see the updated database of Area 1 (figure 6.31.a), where the updated Summary-LSAs, one generated by each remaining ABR, have an advertised age of under 200 (for subnet 22.0.0.0/24, generated by R3 and R4, and for subnet 34.0.0.0, generated by R2). In figure 6.31.b, we see the state of the Overlay LSA portion of the database, which did not change besides the update of the remaining ABRs ABR-LSAs (entries with LS_ID 5.0.0.0). In this case this is expected, since R6 did not have the chance to delete its own LSAs before being removed. Figure 6.31.c shows the contents of one of the remaining ABR-LSAs, in this R3's, updated to exclude R6 from its ABR neighbors. The same has been done for other ABRs as well.

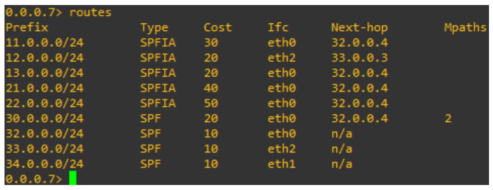
We also show the resulting routing tables of routers R5 and R7, in figures 6.32.a and 6.32.b, respectively. These are the routers where this removal made the most impact, placing them at opposite extremes of the network. While verifying these routing tables, take into consideration that, under normal circumstances, the only and highest cost route is the one from R5 to subnet 30.0.0.0/24, presenting a cost of 40. Only among these two routers, the number of routes with costs greater or equal to this increased from one to six.

To finish this test, we configured a new router to replace R6. The router has been configured exactly as R6 and placed in its place, as shown in figure 6.1. Before initializing the router, we closed the existing packet capture and performed a new one between R5 and R6, over subnet 22.0.0.0/24. We then started the router, obtaining the packet capture shown in figure 6.33. This capture describes the initial synchronization and subsequent Overlay routing calculation for R6. In this capture, R6 receives



```
0.0.0.5> routes
Prefix              Type     Cost     Ifc      Next-hop       Mpaths
11.0.0.0/24         SPFIA    20       eth1     21.0.0.2
12.0.0.0/24         SPFIA    30       eth1     21.0.0.2
13.0.0.0/24         SPFIA    30       eth1     21.0.0.2
21.0.0.0/24         SPF      10       eth1     n/a
22.0.0.0/24         SPF      10       eth0     n/a
30.0.0.0/24         SPFIA    40       eth1     21.0.0.2
32.0.0.0/24         SPFIA    40       eth1     21.0.0.2
33.0.0.0/24         SPFIA    40       eth1     21.0.0.2
34.0.0.0/24         SPFIA    50       eth1     21.0.0.2
0.0.0.5>
```

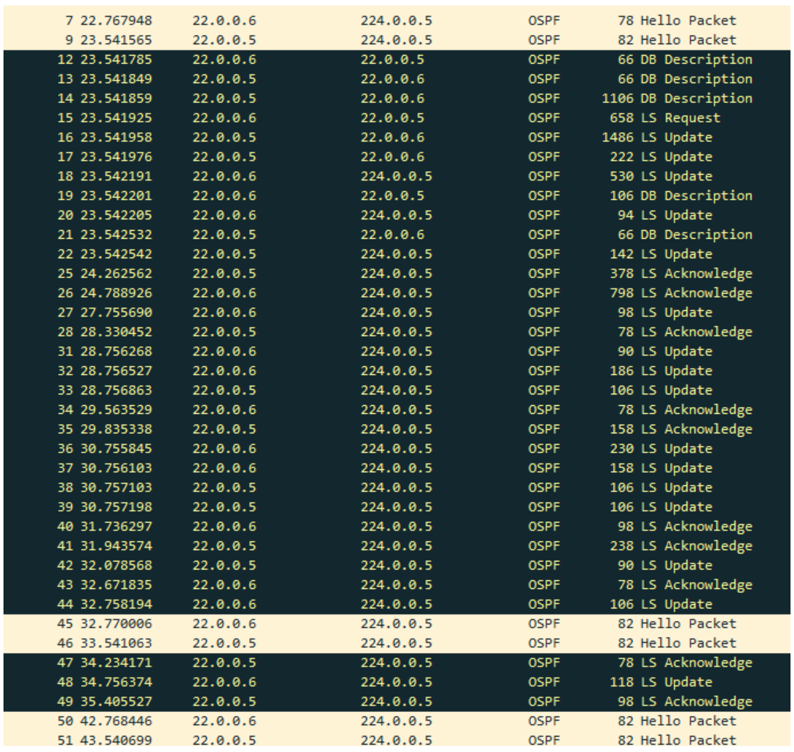
(a)



```
0.0.0.7> routes
Prefix              Type     Cost     Ifc      Next-hop       Mpaths
11.0.0.0/24         SPFIA    30       eth0     32.0.0.4
12.0.0.0/24         SPFIA    20       eth2     33.0.0.3
13.0.0.0/24         SPFIA    20       eth0     32.0.0.4
21.0.0.0/24         SPFIA    40       eth0     32.0.0.4
22.0.0.0/24         SPFIA    50       eth0     32.0.0.4
30.0.0.0/24         SPF      20       eth0     32.0.0.4       2
32.0.0.0/24         SPF      10       eth0     n/a
33.0.0.0/24         SPF      10       eth2     n/a
34.0.0.0/24         SPF      10       eth1     n/a
0.0.0.7>
```

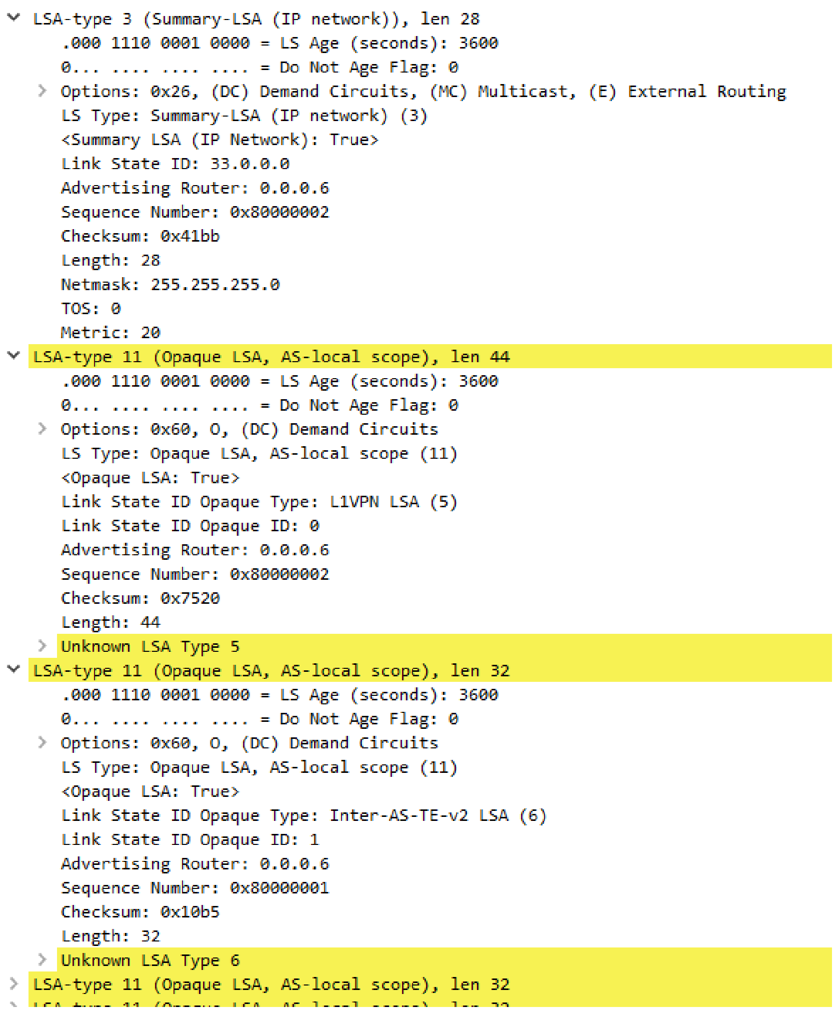
(b)

Figure 6.32: Routing tables of (a) R5 and (b) R7 after removing the ABR



```
 7 22.767948   22.0.0.6       224.0.0.5      OSPF     78 Hello Packet
 9 23.541565   22.0.0.5       224.0.0.5      OSPF     82 Hello Packet
12 23.541785   22.0.0.6       22.0.0.5       OSPF     66 DB Description
13 23.541849   22.0.0.5       22.0.0.6       OSPF     66 DB Description
14 23.541859   22.0.0.5       22.0.0.6       OSPF   1106 DB Description
15 23.541925   22.0.0.6       22.0.0.5       OSPF    658 LS Request
16 23.541958   22.0.0.5       22.0.0.6       OSPF   1486 LS Update
17 23.541976   22.0.0.5       22.0.0.6       OSPF    222 LS Update
18 23.542191   22.0.0.6       224.0.0.5      OSPF    530 LS Update
19 23.542201   22.0.0.6       22.0.0.5       OSPF    106 DB Description
20 23.542205   22.0.0.6       224.0.0.5      OSPF     94 LS Update
21 23.542532   22.0.0.5       22.0.0.6       OSPF     66 DB Description
22 23.542542   22.0.0.5       224.0.0.5      OSPF    142 LS Update
25 24.262562   22.0.0.5       224.0.0.5      OSPF    378 LS Acknowledge
26 24.788926   22.0.0.6       224.0.0.5      OSPF    798 LS Acknowledge
27 27.755690   22.0.0.6       224.0.0.5      OSPF     98 LS Update
28 28.330452   22.0.0.5       224.0.0.5      OSPF     78 LS Acknowledge
31 28.756268   22.0.0.6       224.0.0.5      OSPF     90 LS Update
32 28.756527   22.0.0.6       224.0.0.5      OSPF    186 LS Update
33 28.756863   22.0.0.5       224.0.0.5      OSPF    106 LS Update
34 29.563529   22.0.0.6       224.0.0.5      OSPF     78 LS Acknowledge
35 29.835338   22.0.0.5       224.0.0.5      OSPF    158 LS Acknowledge
36 30.755845   22.0.0.6       224.0.0.5      OSPF    230 LS Update
37 30.756103   22.0.0.6       224.0.0.5      OSPF    158 LS Update
38 30.757103   22.0.0.5       224.0.0.5      OSPF    106 LS Update
39 30.757198   22.0.0.5       224.0.0.5      OSPF    106 LS Update
40 31.736297   22.0.0.6       224.0.0.5      OSPF     98 LS Acknowledge
41 31.943574   22.0.0.5       224.0.0.5      OSPF    238 LS Acknowledge
42 32.078568   22.0.0.5       224.0.0.5      OSPF     90 LS Update
43 32.671835   22.0.0.6       224.0.0.5      OSPF     78 LS Acknowledge
44 32.758194   22.0.0.6       224.0.0.5      OSPF    106 LS Update
45 32.770006   22.0.0.6       224.0.0.5      OSPF     82 Hello Packet
46 33.541063   22.0.0.5       224.0.0.5      OSPF     82 Hello Packet
47 34.234171   22.0.0.5       224.0.0.5      OSPF     78 LS Acknowledge
48 34.756374   22.0.0.6       224.0.0.5      OSPF    118 LS Update
49 35.405527   22.0.0.5       224.0.0.5      OSPF     98 LS Acknowledge
50 42.768446   22.0.0.6       224.0.0.5      OSPF     82 Hello Packet
51 43.540699   22.0.0.5       224.0.0.5      OSPF     82 Hello Packet
```

Figure 6.33: Summary of the packets captured after R6 was added back to the network



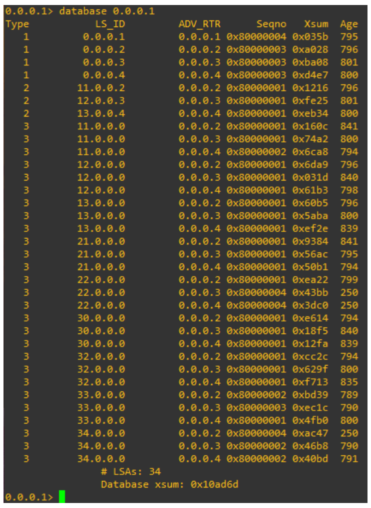
```
v LSA-type 3 (Summary-LSA (IP network)), len 28
    .000 1110 0001 0000 = LS Age (seconds): 3600
    0... .... .... .... = Do Not Age Flag: 0
  > Options: 0x26, (DC) Demand Circuits, (MC) Multicast, (E) External Routing
    LS Type: Summary-LSA (IP network) (3)
    <Summary LSA (IP Network): True>
    Link State ID: 33.0.0.0
    Advertising Router: 0.0.0.6
    Sequence Number: 0x80000002
    Checksum: 0x41bb
    Length: 28
    Netmask: 255.255.255.0
    TOS: 0
    Metric: 20
v LSA-type 11 (Opaque LSA, AS-local scope), len 44
    .000 1110 0001 0000 = LS Age (seconds): 3600
    0... .... .... .... = Do Not Age Flag: 0
  > Options: 0x60, O, (DC) Demand Circuits
    LS Type: Opaque LSA, AS-local scope (11)
    <Opaque LSA: True>
    Link State ID Opaque Type: L1VPN LSA (5)
    Link State ID Opaque ID: 0
    Advertising Router: 0.0.0.6
    Sequence Number: 0x80000002
    Checksum: 0x7520
    Length: 44
  > Unknown LSA Type 5
v LSA-type 11 (Opaque LSA, AS-local scope), len 32
    .000 1110 0001 0000 = LS Age (seconds): 3600
    0... .... .... .... = Do Not Age Flag: 0
  > Options: 0x60, O, (DC) Demand Circuits
    LS Type: Opaque LSA, AS-local scope (11)
    <Opaque LSA: True>
    Link State ID Opaque Type: Inter-AS-TE-v2 LSA (6)
    Link State ID Opaque ID: 1
    Advertising Router: 0.0.0.6
    Sequence Number: 0x80000001
    Checksum: 0x10b5
    Length: 32
  > Unknown LSA Type 6
> LSA-type 11 (Opaque LSA, AS-local scope), len 32
```

Figure 6.34: Detailed view of packet 18

all the LSAs present in the database, including the Overlay LSAs. All self-originated LSAs received are then deleted and reinstalled by R6, before performing its full Overlay routing calculations. Figure 6.34 shows one of the packets generated by R6 during the initial synchronization process, deleting some of its self-generated LSAs, before advertising the new ones.

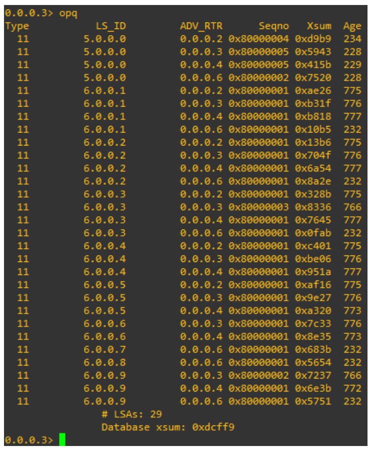
We show the updated database of Area 1 in figure 6.35.a, the Overlay LSA portion of the database in figure 6.35.b, and the reinstalled ABR-LSA of router R6 in figure 6.35.c. As for the Area 1 database, we can see that the Summary-LSAs previously updated, were updated again, reverting the previous updates done. We also see that the Overlay LSA portion of the database now contains new LSAs, all originated by R6, with a significantly lower Age attribute. We can also the restoration of the ABR-LSA for R6, containing the same information as before being removed.

We also leave, in figures 6.36.a, 6.36.b and 6.36.c the final routing tables of routers R3, R5 and R6, respectively, with stored information as the tables in figure 6.27 for these same routers.



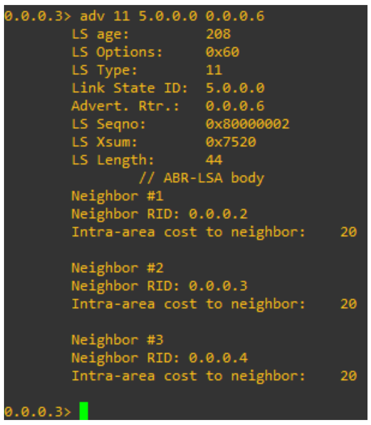
```
0.0.0.1> database 0.0.0.1
Type          LS_ID        ADV_RTR      Seqno   Xsum  Age
   1        0.0.0.1        0.0.0.1 0x80000004 0x035b  795
   1        0.0.0.2        0.0.0.2 0x80000003 0xa028  796
   1        0.0.0.3        0.0.0.3 0x80000003 0xba08  801
   1        0.0.0.4        0.0.0.4 0x80000003 0xd4e7  800
   2       11.0.0.2        0.0.0.2 0x80000001 0x1216  796
   2       12.0.0.3        0.0.0.3 0x80000001 0xfe25  801
   2       13.0.0.4        0.0.0.4 0x80000001 0xeb34  800
   3       11.0.0.0        0.0.0.2 0x80000001 0x160c  841
   3       11.0.0.0        0.0.0.3 0x80000001 0x74a2  800
   3       11.0.0.0        0.0.0.4 0x80000002 0x6ca8  794
   3       12.0.0.0        0.0.0.2 0x80000001 0x6da9  796
   3       12.0.0.0        0.0.0.3 0x80000001 0x031d  840
   3       12.0.0.0        0.0.0.4 0x80000001 0x61b3  798
   3       13.0.0.0        0.0.0.2 0x80000001 0x60b5  796
   3       13.0.0.0        0.0.0.3 0x80000001 0x5aba  800
   3       13.0.0.0        0.0.0.4 0x80000001 0xef2e  839
   3       21.0.0.0        0.0.0.2 0x80000001 0x9384  841
   3       21.0.0.0        0.0.0.3 0x80000001 0x56ac  795
   3       21.0.0.0        0.0.0.4 0x80000001 0x50b1  794
   3       22.0.0.0        0.0.0.2 0x80000001 0xea22  799
   3       22.0.0.0        0.0.0.3 0x80000004 0x43bb  250
   3       22.0.0.0        0.0.0.4 0x80000004 0x3dc0  250
   3       30.0.0.0        0.0.0.2 0x80000001 0xe614  794
   3       30.0.0.0        0.0.0.3 0x80000001 0x18f5  840
   3       30.0.0.0        0.0.0.4 0x80000001 0x12fa  839
   3       32.0.0.0        0.0.0.2 0x80000001 0xcc2c  794
   3       32.0.0.0        0.0.0.3 0x80000001 0x629f  800
   3       32.0.0.0        0.0.0.4 0x80000001 0xf713  835
   3       33.0.0.0        0.0.0.2 0x80000002 0xbd39  789
   3       33.0.0.0        0.0.0.3 0x80000003 0xec1c  790
   3       33.0.0.0        0.0.0.4 0x80000001 0x4fb0  800
   3       34.0.0.0        0.0.0.2 0x80000004 0xac47  250
   3       34.0.0.0        0.0.0.3 0x80000002 0x46b8  790
   3       34.0.0.0        0.0.0.4 0x80000002 0x40bd  791
               # LSAs: 34
               Database xsum: 0x10ad6d
0.0.0.1>
```

(a)



```
0.0.0.3> opq
Type          LS_ID        ADV_RTR      Seqno   Xsum  Age
  11        5.0.0.0        0.0.0.2 0x80000004 0xd9b9  234
  11        5.0.0.0        0.0.0.3 0x80000005 0x5943  228
  11        5.0.0.0        0.0.0.4 0x80000005 0x415b  229
  11        5.0.0.0        0.0.0.6 0x80000002 0x7520  228
  11        6.0.0.1        0.0.0.2 0x80000001 0xae26  775
  11        6.0.0.1        0.0.0.3 0x80000001 0xb31f  776
  11        6.0.0.1        0.0.0.4 0x80000001 0xb818  777
  11        6.0.0.1        0.0.0.6 0x80000001 0x10b5  232
  11        6.0.0.2        0.0.0.2 0x80000001 0x13b6  775
  11        6.0.0.2        0.0.0.3 0x80000001 0x704f  776
  11        6.0.0.2        0.0.0.4 0x80000001 0x6a54  777
  11        6.0.0.2        0.0.0.6 0x80000001 0x8a2e  232
  11        6.0.0.3        0.0.0.2 0x80000001 0x328b  775
  11        6.0.0.3        0.0.0.3 0x80000003 0x8336  766
  11        6.0.0.3        0.0.0.4 0x80000001 0x7645  777
  11        6.0.0.3        0.0.0.6 0x80000001 0x0fab  232
  11        6.0.0.4        0.0.0.2 0x80000001 0xc401  775
  11        6.0.0.4        0.0.0.3 0x80000001 0xbe06  776
  11        6.0.0.4        0.0.0.4 0x80000001 0x951a  777
  11        6.0.0.5        0.0.0.2 0x80000001 0xaf16  775
  11        6.0.0.5        0.0.0.3 0x80000001 0x9e27  776
  11        6.0.0.5        0.0.0.4 0x80000001 0xa320  773
  11        6.0.0.6        0.0.0.3 0x80000001 0x7c33  776
  11        6.0.0.6        0.0.0.4 0x80000001 0x8e35  773
  11        6.0.0.7        0.0.0.6 0x80000001 0x683b  232
  11        6.0.0.8        0.0.0.6 0x80000001 0x5654  232
  11        6.0.0.9        0.0.0.3 0x80000002 0x7237  766
  11        6.0.0.9        0.0.0.4 0x80000001 0x6e3b  772
  11        6.0.0.9        0.0.0.6 0x80000001 0x5751  232
               # LSAs: 29
               Database xsum: 0xdcff9
0.0.0.3>
```

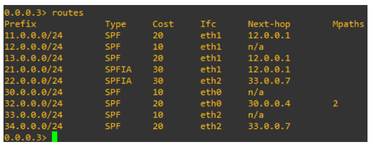
(b)



```
0.0.0.3> adv 11 5.0.0.0 0.0.0.6
        LS age:         208
        LS Options:     0x60
        LS Type:        11
        Link State ID:  5.0.0.0
        Advert. Rtr.:   0.0.0.6
        LS Seqno:       0x80000002
        LS Xsum:        0x7520
        LS Length:      44
                // ABR-LSA body
        Neighbor #1
        Neighbor RID: 0.0.0.2
        Intra-area cost to neighbor:    20

        Neighbor #2
        Neighbor RID: 0.0.0.3
        Intra-area cost to neighbor:    20

        Neighbor #3
        Neighbor RID: 0.0.0.4
        Intra-area cost to neighbor:    20
0.0.0.3>
```

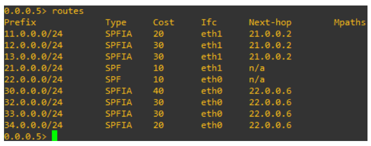
(c)

Figure 6.35: State of (a) Area 1's database, (b) the database Overlay portion and (c) R3's ABR-LSA after adding R6 back to the network



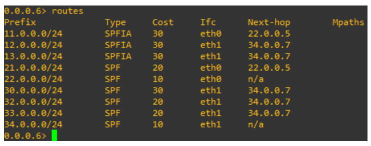
```
0.0.0.3> routes
Prefix              Type     Cost     Ifc      Next-hop        Mpaths
11.0.0.0/24         SPF      20       eth1     12.0.0.1
12.0.0.0/24         SPF      10       eth1     n/a
13.0.0.0/24         SPF      20       eth1     12.0.0.1
21.0.0.0/24         SPFIA    30       eth1     12.0.0.1
22.0.0.0/24         SPFIA    30       eth2     33.0.0.7
30.0.0.0/24         SPF      10       eth0     n/a
32.0.0.0/24         SPF      20       eth0     30.0.0.4        2
33.0.0.0/24         SPF      10       eth2     n/a
34.0.0.0/24         SPF      20       eth2     33.0.0.7
0.0.0.3>
```

(a)



```
0.0.0.5> routes
Prefix              Type     Cost     Ifc      Next-hop        Mpaths
11.0.0.0/24         SPFIA    20       eth1     21.0.0.2
12.0.0.0/24         SPFIA    30       eth1     21.0.0.2
13.0.0.0/24         SPFIA    30       eth1     21.0.0.2
21.0.0.0/24         SPF      10       eth1     n/a
22.0.0.0/24         SPF      10       eth0     n/a
30.0.0.0/24         SPFIA    40       eth0     22.0.0.6
32.0.0.0/24         SPFIA    30       eth0     22.0.0.6
33.0.0.0/24         SPFIA    30       eth0     22.0.0.6
34.0.0.0/24         SPFIA    20       eth0     22.0.0.6
0.0.0.5>
```

(b)



```
0.0.0.6> routes
Prefix              Type     Cost     Ifc      Next-hop        Mpaths
11.0.0.0/24         SPFIA    30       eth0     22.0.0.5
12.0.0.0/24         SPFIA    30       eth1     34.0.0.7
13.0.0.0/24         SPFIA    30       eth1     34.0.0.7
21.0.0.0/24         SPF      20       eth0     22.0.0.5
22.0.0.0/24         SPF      10       eth0     n/a
30.0.0.0/24         SPF      30       eth1     34.0.0.7
32.0.0.0/24         SPF      20       eth1     34.0.0.7
33.0.0.0/24         SPF      20       eth1     34.0.0.7
34.0.0.0/24         SPF      10       eth1     n/a
0.0.0.6>
```

(c)

Figure 6.36: Routing tables of (a) R3, (b) R5 and (c) R6 obtained after adding R6 back to the network

6.2 Convergence tests

To test the convergence times of our extended implementation, we will be measuring the time it takes to complete certain events. We did this manually, so we will be using a relatively small amount of samples (15 for each event). We will cover the tests already described in Section 6.1, these being:

- The time it takes the network to converge after a cold network start;
- The time it takes to converge after changing the cost of a router's outgoing interface;
- The time it takes to converge after changing an existing interface, both removal and reattachment;
- The time it takes to converge after removing or adding an ABR from the network;

These tests have been already covered, so we only address the obtained results. These tests will also be performed for the base version of this implementation whenever possible, to allow us to compare the results. We obtained these results by resorting to the log messages generated by each router, which include a timestamp. Since all the router containers are run inside the same VM instance, their clocks are synchronized. In this implementation, we can only ensure a precision of seconds (it's the maximum precision achievable with the timing function used in the implementation), which is not ideal, but suffices for us to take concrete conclusions. We will calculate the convergence times as the timestamp difference, in seconds, between an initial event and a final event indicating the convergence of the network. This final event will be, in all cases, the last update among the routers' routing tables. The initial event varies with the test case, these being:

- **Cold start** - The time of the first router being turned on;
- **Cost change** - The time of the interface change;
- **Link change** - Neighbor failure detected by the Hello protocol (link removal); Neighbor detected through the Hello packet exchange (link addition);
- **ABR removal** - First detection of the ABR down by the routers neighbors;
- **ABR addition** - Time of the router being turned on.

We will use the multi-area topology described in figure 6.1, and a two level hierarchical network illustrated in figure 6.37. This topology is divided into three areas, Area 0 (Backbone, Area ID 0.0.0.0), Area 1 (Area ID 0.0.0.1) and Area 2 (Area ID 0.0.0.2). There are nine routers in total, each one running in its own Docker container. Routers are identified through their Router ID, composed as 0.0.0.x, with x being the number assigned to the router in GNS3. Four of the routers are ABRs. These are R2 and R3 connecting Areas 0 and 1, and R4 and R5 connecting Areas 0 and 2. The remaining routers are area-internal routers. R1 belonging to Area 0, R6 and R7 to Area 1, and R8 and R9 to Area 2. Ten sub-networks were configured: subnets 11.0.0.0/24, 12.0.0.0/24, 13.0.0.0/24 and 14.0.0.0/24 in Area 0, subnets 21.0.0.0/24, 22.0.0.0/24 and 23.0.0.0/24 in Area 1 and subnets 31.0.0.0/24, 32.0.0.0/24 and 33.0.0.0/24 in Area 3. By default, all outgoing interfaces have been assigned a cost of 10.

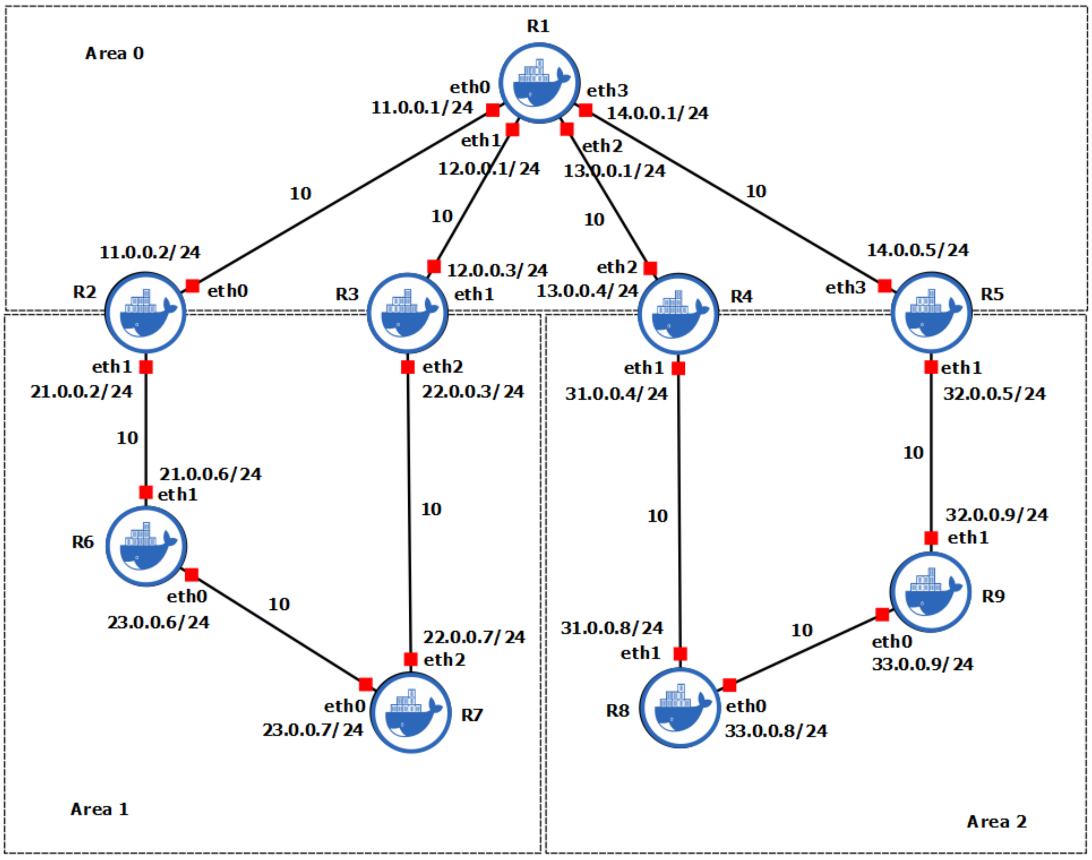
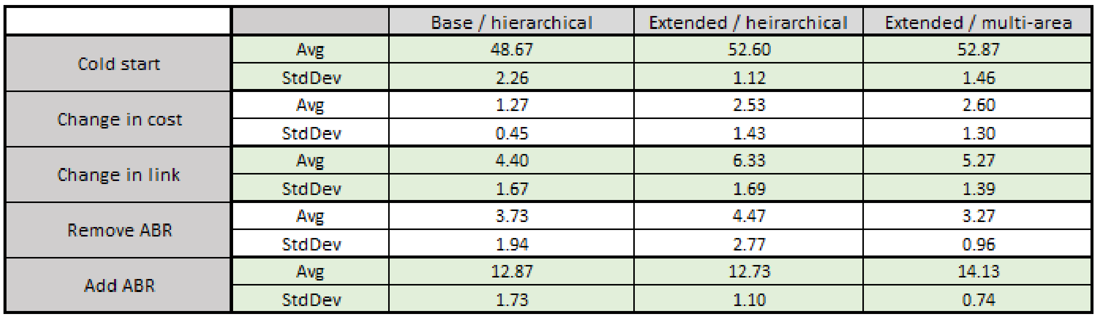


Figure 6.37: Hierarchical topology used for the convergence times testing

We performed our tests for three topology/implementation combinations: base/hierarchical, extended/hierarchical and extended/multi-area. We use the base/hierarchical combination as a control for the other combinations, since it uses the base implementation with a topology that was already possible to use. We then use the extended/hierarchical combination to assess the overhead added by our extension, as it would not really be needed when using a hierarchical topology. Finally, the extended/multi-area combination serves to verify what advantages can come from using a more arbitrary, cyclical multi-area topology, instead of the hierarchical one.

For future reference, we show in figure 6.38 a summary of the obtained times, showing the average and standard deviation for these three combinations.

We started by testing the convergence time of the network after all the existing routers are cold started. These results are illustrated in figure 6.39. Here we see that, usually, the base version has



		Base / hierarchical	Extended / heirarchical	Extended / multi-area
Cold start	Avg	48.67	52.60	52.87
	StdDev	2.26	1.12	1.46
Change in cost	Avg	1.27	2.53	2.60
	StdDev	0.45	1.43	1.30
Change in link	Avg	4.40	6.33	5.27
	StdDev	1.67	1.69	1.39
Remove ABR	Avg	3.73	4.47	3.27
	StdDev	1.94	2.77	0.96
Add ABR	Avg	12.87	12.73	14.13
	StdDev	1.73	1.10	0.74

Figure 6.38: Summary of the obtained times (in seconds) for the tested cases

shorter convergence times than the extended version, which maintain similar convergence times for the different topologies used. On average, the time taken by the base version was only four seconds less than that of the extended version. This initial convergence time includes the time taken by routers to trigger a 40 second timer that will ultimately initiate the Database synchronization process between them. These results are very positive, especially when we consider that the extended version limits the rate of the overlay calculation to one run per second, while the base version will instantly process and flood newly received Summary-LSAs. This rate limitation serves the purpose of not overloading the routers' processing capacity, since the full Overlay routing calculation can be a relatively heavy process. However, it does not influence the correction of our implementation.

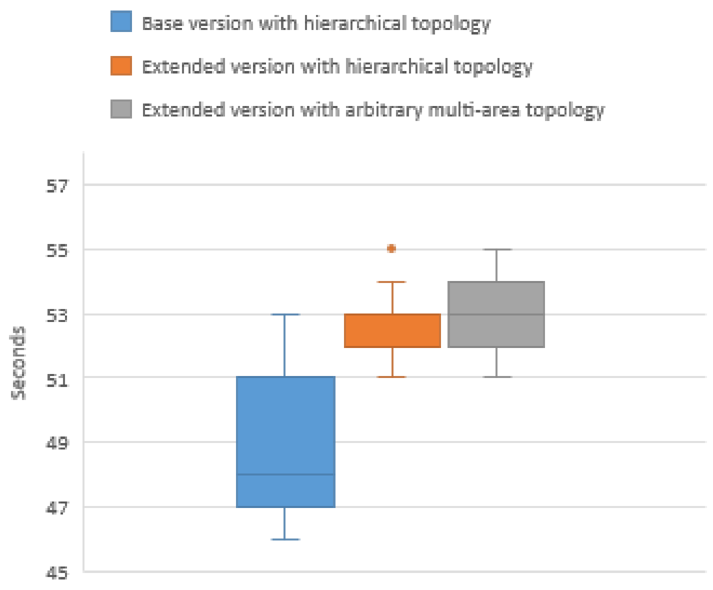


Figure 6.39: Boxplot of the results for a full network cold start

We then tested the reaction time of the network to a change in the cost of a router interface. The obtained results are shown in figure 6.40. For the hierarchical network, we changed the costs in both interfaces of R4. For the multi-area topology, we changed the costs in both interfaces of R6. For this test we only applied changes to ABR interfaces, as these would have a more similar type of reaction among all version/topology combinations. For each interface, we applied a change in the interface cost from 10 to 100, and back to 10 after that. To gather the time for each test run, we used the average from both changes applied to each interface (10 to 100 and 100 to 10). Each event is then measured as the average between (i) the time taken to converge after the interface cost is changed from 10 to 100 and (ii) the time taken to converge after changing the interface cost from 100 to 10. For this test, we noted an average convergence time of around 1.25 seconds for the base version, versus the 2.5 second average time presented by the extended version. For this particular case, there is not a significant distinction to be made about the different topologies used. These times can be mostly explained by the need that may arise, after an intra-area calculation, to re-originate some Overlay LSAs, which can lead to a full overlay routing calculation. This calculation is rate limited to one run per second. The base version however is only limited by the intra-area calculation common to both versions. This means a faster time for the base implementation, with the downside of possibly storing sub-optimal intra-area routes to destinations.

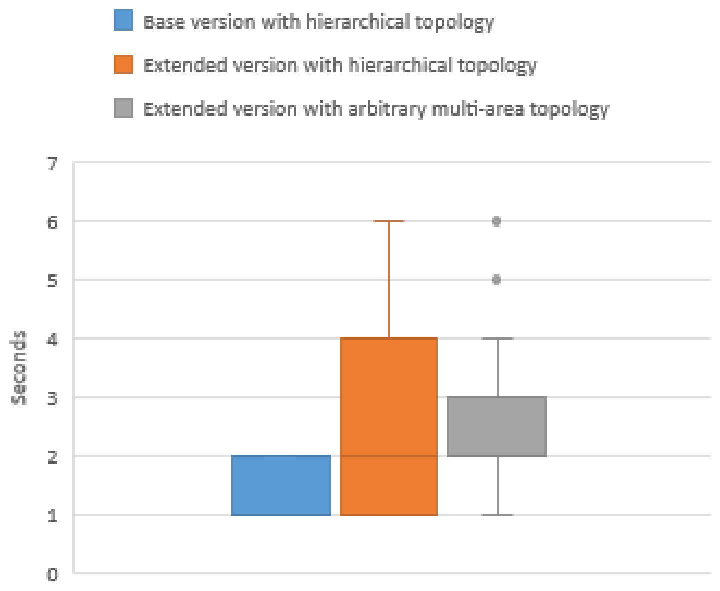


Figure 6.40: Boxplot of the results for changes in interface costs

We then tested the network reaction to the removal and subsequent addition of links between routers. In the hierarchical network, we broke and reattached the links between routers R1/R4 and R4/R8. For the multi-area topology, we did the same to the links between routers R5/R6 and R6/R7. For this test we decided to affect the links connecting an ABR to an area-internal router. This keeps the test conditions as equal as they can be for the different topologies used. Similarly to the values gathered for the cost changes, each event time was measured as the average between (i) the time taken to converge after the link is removed and (ii) the time taken to converge after the re-addition of the link. We chose not to separate these test cases because they resulting times were within the same level of magnitude. The results of this are illustrated in figure 6.41. In this case, we can see the base version presenting better values when compared with the extended version, for the same topology, but a considerable improvement from using the multi-area topology. For the hierarchical topology, the worst times were obtained for the changes made in the link belonging to the backbone, since these had the most impact over the rest of the network. We also noted that when removing the link between R1 and R4, R4 no longer had access to Area 0 and Area 1's advertised destinations, since it could no longer reach any other ABR through the backbone, even though this would be possible to do through Area 2. Although this did not impact the convergence times of the network, it reinforces the current restrictions seen in OSPF. On average, the resulting times went up by one second from the base implementation to the extended version applied over the multi-area topology, and two seconds from the base version to the extended version when used over the same topology. We did not consider the DR election process for this test, as it relies on the Hello protocol. If we did, the times gathered would increase somewhat randomly (due to the different timings used by the routers to send out Hello packets), by up to 5 seconds.

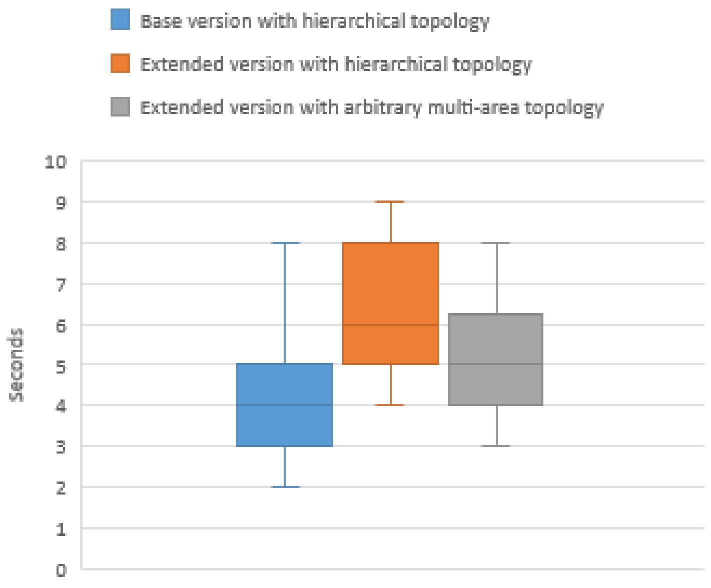


Figure 6.41: Boxplot of the results for changes in the existing interfaces connecting two routers

Finally, we tested the reaction times to the removal and addition of ABRs to the topology. We separated the gathered times for removal and addition of these ABRs, as each of these cases have its own particular aspects. For the hierarchical topology, we tested this by removing and re-adding R2, while in the multi-area topology we did the same with R6. This choice is simply made by the fact that both these routers are ABRs, following the same logic as for the other cases. The results for the removal and addition of the ABRs are represented in figures 6.42 and 6.43, respectively. For this test, we have very close average values for all test cases, with the lowest, and more consistent being those presented by the extended version applied over the multi-area topology, in the case of the ABR removal. Because the multi-area network uses a cyclical topology, when a change is detected, it will be faster for the information change to reach the remaining areas, as the information does not need to be spread down the hierarchy to the routers farthest from the change, and can instead be disseminated through all remaining possible directions. For other test cases this did not occur. This is most likely explained by the small dimension of the used networks, which will not allow this new feature to be clearly shown.

For the addition of the ABRs, the average times are again similar to one another, with the extended version over the multi-area topology having a slightly higher average value. For this initial version of the extension, the added ABR is receiving Summary-LSAs, which it no longer requires, as it no longer processes them. Depending on the size of the network used, this might lead to significantly longer times, as most of the contents of LSDBs are Summary-LSAs, as we can see from the results gathered in our functionality tests (e.g., figure 6.35.a, where all LSAs of Type 3 are Summary-LSAs). For the results obtained for the extended version, we must keep in mind that the initial ABR synchronization process is not featured as it ultimately should. However, given the overhead that is also added by including Summary-LSAs in this process for ABRs, the resulting times would not likely vary. Again The higher times presented here are a result of the high amount of information that must be exchanged between the routers upon the ABR addition, followed, in the case of the extended version, by the full Overlay routing calculations that must be (re)done by the networks ABRs.

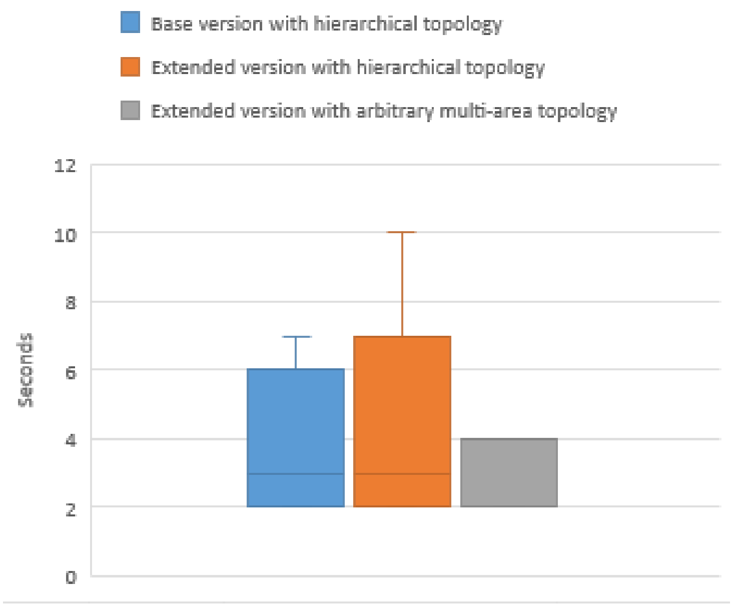


Figure 6.42: Boxplot of the results for the removal of an ABR to the network

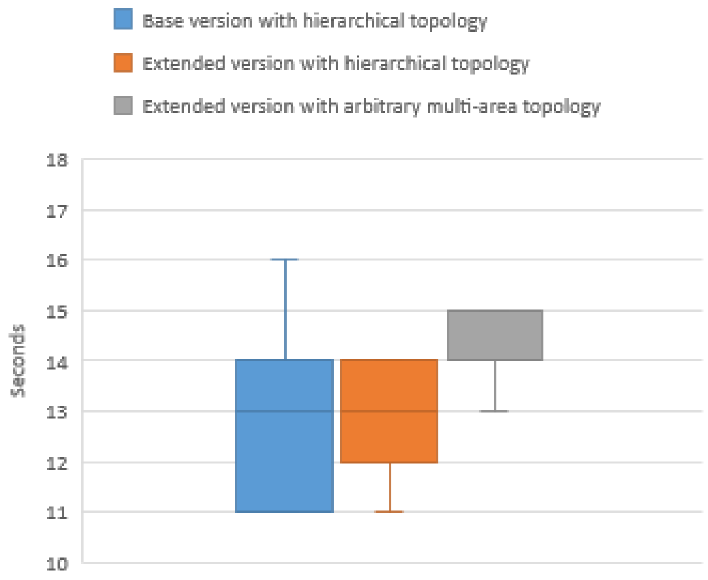


Figure 6.43: Boxplot of the results for the addition of an ABR to the network

7 Conclusion

The OSPF protocol currently uses a DVR approach to inter-area routing, which suffers from some restrictions, such as (i) the count-to-Infinity problem, inherent to this type of approach, (ii) the limitation on the multi-area network topology configuration and (iii) possibly, sub-optimal path choice to some destinations.

To overcome these restrictions, we proposed and implemented an extension, in this case over OSPFv2, not only to allow the configuration of arbitrary multi-area topologies, but also to always guarantee optimal routing across the whole network. This extension changes the inter-area approach used by OSPF, from a DVR to an LSR approach. This is done by resorting to a newly introduced ABR overlay, providing a simplified view of the whole AS over the physical network, by only including the network ABRs in it. This overlay is synchronized among all ABRs across the AS, and stores the necessary information for ABRs to reach all destinations within the network. This information is stored by the newly introduced Overlay-LSAs, and is processed and injected into the network areas through already known LSAs, making this extension only visible to ABRs.

We based our implementation on an existing OSPFv2 implementation, developed by John T. Moy. We aimed to integrate our extension into this existing implementation as simply and as effectively as possible. We successfully developed an initial and functional, yet not completely stable nor optimized, version of our extended implementation. We reached the conclusion that a final version for the integration of this extension in the base version would require an extensive and deep revision and refactoring of the code, including some of the processes used by OSPF.

Our results for this work were undoubtedly positive. We were able to prove the correct and expected operation of our proposed extension, overcoming the current restrictions faced by the protocol. Furthermore, we obtained very satisfactory convergence times for this extended implementation, not significantly differing from its base version. We have seen satisfactory convergence times, from an extended protocol that not only allows for an arbitrary network topology configuration, but for better overall destination reachability. We can then conclude there would be an advantage in using this extension, as it provides a solution to heavy restrictions without significant impact on the network convergence times.

As possible future work, a full revision of the base implementation would be ideal to perhaps restructure it, by removing the now unnecessary aspects from it, such as Virtual links or the differentiation among the backbone and other areas. Also, a revision to the data structures used widely across the base implementation could prove valuable to a better accommodation of not only this extension, but others as well. For example, with the current AVLtree/AVLitem structures, it might be troublesome for us to try to insert an AVLitem inheriting object into two or more separate AVLtree structures, as the objects might break the links from objects in one tree to form new links with objects of another tree.

Bibliography

[1] R. Valadas, *OSPF and IS-IS From Link State Routing Principles to Technologies*, 1st ed. CRC Press, 2019. [Online]. Available: https://www.crcpress.com/OSPF-and-IS-IS-From-Link-State-Routing-Principles-to-Technologies/Valadas/p/book/9781138504554

[2] J. T. Moy, “OSPF version 2,” RFC Editor, RFC 2328, 4 1998. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2328.html

[3] ——, *OSPF Complete Implementation*, 1st ed. Addison-Wesley, 2001.

[4] R. Valadas, “OSPF extension for the support of multi-area networks with arbitrary topologies,” 2017.

[5] J. Fonseca, “Ospfv3’s extension for the support of multi-area networks with arbitrary topologies,” MSc Dissertation, Instituto Superior Técnico, 12 2021. [Online]. Available: https://fenix.tecnico.ulisboa.pt/cursos/merc/dissertacao/846778572213609

[6] G. Malkin, “RIP version 2,” RFC Editor, RFC 2453, 11 1998. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2453.html

[7] R. Coltun, D. Ferguson, J. T. Moy, and A. Lindem, “Ospf for ipv6,” RFC Editor, RFC 5340, 7 2008. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5340.html

[8] J. T. Moy, *OSPF : anatomy of an Internet routing protocol*, 1st ed. Addison-Wesley, 1998.

[9] L. Berger, I. Bryskin, A. Zinin, and R. Coltun, “The OSPF Opaque LSA Option,” RFC Editor, RFC 5250, 7 2008. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5250.html

[10] ”Docker: Accelerated, Containerized Application Development”. [Online]. Available: https://www.docker.com/

[11] ”GNS3 — The software that empowers network professionals”. [Online]. Available: https://www.gns3.com/

[12] ”Wireshark · Go Deep.”. [Online]. Available: https://www.wireshark.org/

[13] X. Gomes. ”xaviercgomes99/ospf_ext”. [Online]. Available: https://hub.docker.com/r/xaviercgomes99/ospf_ext

[14] ——. ”xaviergomes99/ospf-extension-multiarea-topology”. [Online]. Available: https://github.com/xaviergomes99/ospf-extension-multiarea-topology

A

Summary of the new and changed files, functions and classes

List of files with changed or added functions/classes (with respective starting code line):

- asbrlsa.C
  - SpfArea::asbr_orig (l.69)
- asexlsa.C
  - ASBRrte::ASBRrte (l.61)
  - ASBRrte::run_calculation (l.594)
- config.C
  - OSPF::cfgStart (l.60)
- dbage.C
  - DBageTimer::action (l.101)
- lsa.C
  - LSA::LSA (l.33)
- lsa.h
  - class rtrLSA (l.286)
- lshdr.h
  - Opaque types (l.181)
  - ABRhdr (l.191)
  - Prefixhdr (l.196)
  - ASBRhdr (l.202)
- opqlsa.C
  - opqLSA::opqLSA (l.30)
  - opqLSA::parse (l.48)
  - opqLSA::unparse (l.73)
  - OSPF::opq_orig (l.115)
- opqlsa.h
  - class opqLSA (l.33)
  - class AbrLSAItem (l.73)
  - class overlayAbrLSA (l.84)
  - class overlayPrefixLSA (l.101)
  - class overlayAsbrLSA (l.114)
- ospf.C
  - OSPF::OSPF (l.54)
  - OSPF:: OSPF (l.180)
- ospf.h
  - class OSPF (l.62)
- rte.C
  - INtbl::add (l.305)
- rte.h

  - class INtbl (l.71)
  - class RTE (l.109)
  - class INrte (l.194)
  - INrte::INrte (l.235)
  - class ASBRrte (l.343)
  - class ABRrte (l.372)
- rtrlsa.C
  - rtrLSA::rtrLSA (l.32)
  - rtrLSA::rl_orig (l.234)
  - rtrLSA::parse (l.471)
  - rtrLSA::unparse (l.639)
- spfarea.C
  - SpfArea::generate_summaries (l.298)
- spfcalc.C
  - OSPF::rtsched (l.37)
  - OSPF::dijkstra (l.165)
  - RTE::RTE (l.279)
  - RTE::new_intra (l.301)
  - RTE::declare_unreachable (l.461)
  - OSPF::update_brs (l.578)
  - OSPF::rt_scan (l.625)
  - INrte::sys_install (l.727)
  - FWDrte::resolve (l.890)
  - INrte::incremental_summary (l.944)
- spfhello.C
  - SpfIfc::build_hello (l.169)
- spflog.h
  - Log code (l.99)
- spfutil.C
  - msgtext (l.290)
- summlsa.C
  - OSPF::sl_orig (l.63)
  - SpfArea::sl_cost (l.128)
  - INrte::run_inter_area (l.319)

We also added three new files to the implementation (we don't specify the functions/classes as all of them are new):

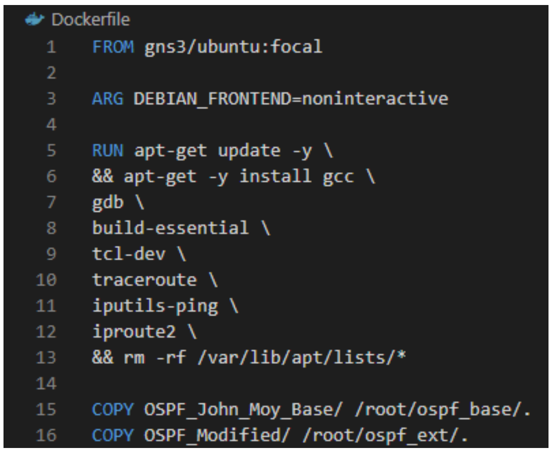
- **overlay.h** includes the ABR neighbor structure definition;
- **overlaycalc.C** includes the functions to perform the overlay routing calculations;
- **overlaylsas.C** includes the operations over the Overlay LSAs, such as origination or parsing.

B

Installation and configuration of the routers

In this appendix, we include the instructions for the installation and configuration of our OSPF routers within GNS3 VM. Both the base and extended versions were packaged inside the same Docker container. The Docker image is available at Docker Hub [13].

To create the container from which we generated the Docker image, we used the Dockerfile shown in figure B.1. First, we need to create a folder containing the source code for both the implementation versions and this Dockerfile (available in GitHub [14]). Then, we must access this folder from within a console terminal, and input the command **docker build -t *name* .**, where *name* is the name we want to give our Docker container/image.



```
FROM gns3/ubuntu:focal

ARG DEBIAN_FRONTEND=noninteractive

RUN apt-get update -y \
&& apt-get -y install gcc \
gdb \
build-essential \
tcl-dev \
traceroute \
iputils-ping \
iproute2 \
&& rm -rf /var/lib/apt/lists/*

COPY OSPF_John_Moy_Base/ /root/ospf_base/.
COPY OSPF_Modified/ /root/ospf_ext/.
```

Figure B.1: Contents of the Dockerfile used to create the Docker containers

To install our Docker image into GNS3, we have to access GNS3, and under *Edit*:

1. Access *Preferences / Docker / Docker containers*;
2. Click on **New**;
3. Select the **Run this Docker container on the GNS3 VM** option and click on **Next**;
4. Select **New image**, type the image name in the field (xaviercgomes99/ospf_ext:latest) and click on **Next**;
5. Type the desired name for the container and click on **Next**;
6. Choose the number of adapters (interfaces) the container will have available and click on **Next**;
7. We can skip the rest of the steps and click on **Next** and **Finish** until we exit the window.

The container should now be available under our list of devices , with the name chosen by us. Then, in a GNS3 project, we can use any number of containers we want to. The first time we do this, and only the first, the GNS3 VM will pull the image from Docker Hub, which may take some time. To assign IP addresses to the container's specific interfaces, we use the GNS3 configuration file, available by right-clicking the device and choosing the *Edit config* option, where we assign IP addresses and masks to the routers' interfaces.

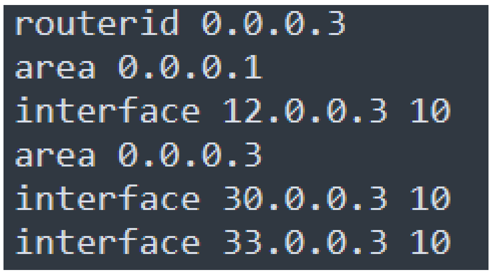
This is all that is needed to install our Docker containers. We will now finish by going through the basic steps to configure and initiate a router using this implementation. This explanation will only cover the basics of the configuration, i.e., the Router ID, Areas and interfaces. A complete description of the configuration options for this implementation can be found in section 14.1 of [3].

To configure a router we must first create a configuration file, with a full path name of **/etc/ospfd.conf**. This is the file the implementation will access when configuring the router. An example of the contents for this configuration file is shown in figure B.2, in this case, describing the configuration of an ABR.

- We start by defining the Router ID, using the line **routerid** *rid*, where *rid* is a 32 bit value, expressed in dotted-decimal notation;
- Then we will describe the areas to which the router attaches, and within those areas, the interfaces attaching the router to those areas. To describe an area, we use the line **area** *aid*, where *aid* is, using the same notation as the *rid*, the 32 bit value for the Area identifier. After writing this line, subsequent lines of configuration will refer to this area;
- For each area we can have a set of interfaces attaching to it. To configure each interface we use the parameter **interface** *address cost*, where *address* is the IPv4 address we wish to assign to the interface and *cost* is its outgoing cost.

To launch the router, we must access the directory where we have the implementation's Makefile, under ospf_*version*/linux, where *version* is either replaced with *base* or *ext*, depending on whether we want to use the base or extended version, respectively. From there we compile the program with the **make** command, and after the compilation finishes we can launch the program with the command **ospfd** (or **ospfd &** if we want to run it in the background).

Furthermore, the implementation supports dynamic reconfiguration. To change a router configuration without having to restart it, we simply change the contents of the configuration file, and use the command *kill -s USR1* ***pid***, where *pid* is replaced by the process ID of the running implementation (if we don't know this value we can use the command *ps — grep ospfd* to find it).



```
routerid 0.0.0.3
area 0.0.0.1
interface 12.0.0.3 10
area 0.0.0.3
interface 30.0.0.3 10
interface 33.0.0.3 10
```

Figure B.2: Example of the contents of the configuration file for a router