TEKNILLINEN KORKEAKOULU
TEKNISKA HÖGSKOLAN
HELSINKI UNIVERSITY OF TECHNOLOGY

Yin Wang

Adding Multi-Class Routing into the Differentiated Services Architecture

Master's thesis submitted in partial fulfilment of the requirements for the degree of Master of Science in Engineering

Espoo, Finland, November 6, 2004

Supervisor Professor Raimo Kantola

Instructor Ph.D Peng Zhang

HELSINKI UNIVERSITY OF TECHNOLOGY ABSTRACT OF MASTER'S THESIS

Department of Computer Science and Engineering

Author:	Yin Wang
Title:	Adding Multi-Class Routing into the Differentiated Services Architecture
Date:	November 2004
Number of Pages:	89
Department:	Department of Computer Science and Engineering
Professorship:	S-38 Network Laboratory
Supervisor:	Prof. Raimo Kantola
Instructor:	Ph.D Peng Zhang

In this thesis, we introduce the multi-class routing (MCR) scheme into the differentiated services (DiffServ) architecture in order to alleviate the inter-class effects [2] and optimize the network traffic in the DiffServ network.

The thesis begins with the overview of some background knowledge, including the DiffServ network, OSPF protocol and some routing algorithms. We describe basic concepts and significant principles.

The MCR scheme, which is introduced into the traditional DiffServ architecture, forms our MCR DiffServ architecture. The MCR DiffServ architecture focuses on the intra-DS domain. Different MCR approaches that include static approach and dynamic approaches can be used for the MCR DiffServ system. Compared to a traditional DiffServ system, our MCR DiffServ system adds the MCR manager module and class-based route selection block. The implementation of the MCR manager closely depends on the routing protocols.

We extend the OSPF protocol to support the MCR capability, and design the software architecture for the OSPF with MCR extensions. Based on the Zebra software, our OSPF code, which supports the static MCR, is successfully implemented. It can provide two independent routing tables created by the SP algorithm and the BSP algorithm. Several tests for our OSPF code are used to demonstrate the work of the OSPF with MCR extensions.

As a result, our OSPF with MCR extensions can achieve the basic functions of the MCR manager for the MCR DiffServ system. The static MCR approach is implemented for the MCR DiffServ system.

Keywords: MCR, multiple routing tables, DiffServ, OSPF, MCR DiffServ

TEKNILLINEN KORKEAKOULU **DIPLOMITYÖN TIIVISTELMÄ**

Tietotekniikan osasto

Tekijä:	Yin Wang
Työn nimi:	Moniluokkareititys Eriytettyjen Palvelujen Arkkitehtuurissa
Päivämäärä:	Marraskuu 2004
Sivumäärä:	89
Osasto:	Tietotekniikan osasto
Professuuri:	S-38 Tietoverkkotekniikka
Työn valvoja:	Prof. Raimo Kantola
Työn ohjaaja:	Ph.D Peng Zhang

Tässä diplomityössä esitetään moniluokkareititysmenetelmä (multi-class routing, MCR) eriytettyjen palvelujen (Diffserv) arkkitehtuuriin. Tällä menetelmällä pyritään välttämään luokkien välistä vaikutusta ja optimoimaan verkon liikennettä DiffServ-verkossa.

Työ alkaa katsauksella taustatietoihin, jossa kuvataan DiffServ-verkko, OSPF protokolla ja eräät reititysalgoritmit. Lisäksi kuvataan suuri määrä peruskäsitteitä ja keskeisiä periaatteita.

Moniluokkareititysmenetelmä, joka on otettu käyttöön perinteisessä DiffServ-rkkitehtuurissa, muodostaa MCR-DiffServ-arkkitehtuurimme perustan. MCR-DiffServ-arkkitehtuuri keskittyy intra-DS alueeseen. MCR-DiffServ-järjestelmässä voidaan käyttää staattista ja erilaisia dynaamisia lähestymistapoja moniluokkareititykseen. Perinteiseen DiffServ-järjestelmään verrattuna, meidän MCR-DiffServ-järjestelmämme sisältää myös MCR-hallintamoduulin ja luokkapohjaisen reittivalintalohkon. MCR-hallintamoduulin toteutus on läheisesti riippuvainen reititysprotokollista.

OSPF-protokolla laajennetaan tukemaan moniluokkareititystä, ja laajennettua OSPF: ä varten suunnitellaan ohjelmistoarkkitehtuuri. Staattista moniluokkareititystä tukeva OSPF -ohjelmamme toteutetaan onnistuneesti Zebra-ohjelmiston pohjalta. Reitityksessä voidaan käyttää kahta riippumatonta reititystaulua: toinen muodostettu SP-algoritmilla ja toinen BSP-algoritmilla. Useiden testien avulla todennetaan toteutuksemme toimivuus.

Lopputuloksena, moniluokkareititykselläl laajennettu OSPF -toteutuksemme voi suorittaa moniluokkareitityshallinnan perustehtävät MCR-DiffServ-järjestelmässä. Toteutuksessamme käytetään staattista moniluokkareititysjärjestelmää.

Avainsanat: MCR, useita reititystauluja, DiffServ, OSPF, MCR DiffServ

Acknowledgments

This thesis was done in the Networking Laboratory of the Helsinki University of Technology. It is a part of the IRoNet Research Project, which is one of the NETS Spearhead Projects.

I have made my best efforts to achieve this. However, I clearly understand that I could not have completed this thesis without the help and support from my senior, friends and family.

I sincerely thank my supervisor, Professor Raimo Kantola, who is the leader of the NETS Spearhead Project, for giving me an opportunity, providing his brilliant ideas and supporting my work.

I would like to give my sincerely appreciations to my instructor, Dr Peng Zhang, who helps me a lot both on my study and work.

Finally, I would like to give my thanks to my parents, my sister and my brothers in China for their full support, to my friends in Finland and to colleagues in the Networking Laboratory for their help. Especially, I sincerely thank Lis.Sc Nicklas Beijar who kindly helps me translate the Finnish abstract.

Yin Wang

Espoo, November 2004

Table of Contents

Figures

Tables

Acronyms

ABR	Area Border Router
AS	Autonomous System
ASBR	AS Boundary Router
BA	Behavior Aggregate
BE	Best Effort
BGP	Border Gateway Protocol
BSP	Bandwidth-inversion Shortest Path
DiffServ	Differentiated Services
DSCP	Differentiated Services Codepoint
DS domain	Differentiated Services domain
EBSP	Enhanced Bandwidth-inversion Shortest Path
EF	Expedited Forwarding
ERT	External Routing Table
ICMP	Internet Control Message Protocol
LSA	Link State Advertisement
LSDB	Link-State Database
MCR	Multi-Class Routing
MF	Multifield
NRT	Network Routing Table
OSPF	Open Shortest Path First
OSPFv2	Open Shortest Path First version 2
PHB	Per-Hop Behavior
QoS	Quality of Service
RIP	Routing Information Protocol
RRT	Router Routing Table
SP	Shortest Path with the hop count metric
SPF	Shortest Path First
TOS	Type of Service
TE	Traffic Engineering
WSP	Widest-Shortest Path

1. Introduction

With the fast development of network technologies, the number of users dramatically increases so that the network traffic quickly increases. In today's open environments with distributed control, the rapid growth of using complicated network services and applications further contribute to great expansion of the network traffic. When the network traffic is heavy, this can negatively influence different network services or applications to some extent because in the Internet packets are handled in the first-in, first-served way. There is no guarantee that each packet will be successfully delivered through the Internet. More and more new network applications, especially real time applications, have appeared. Diverse network services and applications have different requirements, such as bandwidth and delay, etc. In the best-effort (BE) IP network environment, however, all applications can not work well because of variable queuing delays, network congestion and packet losses. Thus, quality of service (QoS) is needed to effectively meet the requirements and reasonably allocate network resources.

The DiffServ model has been suggested as a scalable solution to implement the QoS in current IP networks [1]. In the DiffServ model, the traffic is divided into various traffic classes with specific priorities. Best-effort (BE) class, which is the main traffic in the networks, has the lowest priority. Other classes are called QoS classes. The traffic class with high priority always acquires more reliable network services than the traffic class with low priority by means of a packet scheduling mechanism. The queueing and dropping behaviours of packets are controlled and managed in each DiffServ node. However, the DiffServ model is originally designed to be decoupled from the IP routing mechanism, meaning that in each DiffServ node, the shortest path routing algorithm is applied to form a single routing table to forward packets. Regardless of the schedule of packets, all packets to the same destination will be delivered through the same route. When the network load is very heavy, the QoS class will produce negative influence on the BE class because packets of the BE class are more likely to be dropped or delayed than those of other classes. This problem is called inter-class effect [2].

Even though several hop-by-hop QoS based routing algorithms have gradually appeared to replace the traditional shortest path (Dijkstra's) algorithm in order to alleviate the inter-class effect and reasonably distribute the traffic through the network, QoS for all network applications cannot be satisfied at the same time. As we know, different classes could have distinct traffic characteristics and diverse QoS requirements. The purpose of each

hop-by-hop QoS based routing algorithm is to calculate and optimize feasible routes that meet specific QoS requirements of an application. A QoS based routing algorithm that can satisfy QoS requirements of all applications does not exist. For example, the Bandwidth-inversion Shortest Path (BSP) algorithm could be suitable for the traffic that needs to achieve high throughput, but it cannot provide the best service for traffic that has a strict requirement in packet delay [3]. Thus, the QoS routing algorithms only provide a limited solution to QoS.

Optimising the network traffic, especially the BE class traffic, based on a single routing table is hard and inefficient to carry out. A physical backup connection link often exists between two routers in the network, especially in a backbone network. Moreover, the backup connection links have usually high costs so that all the traffic normally traverses through non-backup connection links. As the non-backup connection link is broken down, the backup connection link will be used to forward packets after recalculating the routing table. That is to say, the routing table in each router normally records one route to one destination if equal-cost routes to the same destination are not maintained. When network congestion occurs in one non-backup connection link, the BE class traffic has to be delivered through the same link as the QoS classes even if a second path through the physical backup connection link exists. Therefore, a single routing table on each network node becomes a barrier to fairly distribute all network traffic over the network.

To address the above problems, there is a need to use multiple routing tables and multiple hop-by-hop QoS based routing algorithms in the DiffServ model. We are proposing a routing scheme based on the DiffServ, called the Multi-class Routing (MCR) DiffServ architecture. In this thesis, we discuss different MCR approaches that can be used for the MCR DiffServ architecture. We design OSPF with MCR extensions. Then, the static MCR approach is implemented based on Zebra OSPF software.

The rest of this thesis is organized as follows. In Chapter 2, we first review background knowledge about technologies, including QoS DiffServ, OSPF and different routing algorithms. Then, the MCR functions and the whole architecture of the MCR DiffServ system are discussed in Chapter 3. Next, the MCR extensions to the OSPF protocol are designed in Chapter 4. In Chapter 5, we describe how to implement the MCR extensions based on the Zebra software. After that, in Chapter 6, we present testing results of the MCR extensions and analyze the feasibility of the MCR DiffServ. Lastly, conclusions and further work are described in Chapter 7.

2. Background

In this chapter, the background knowledge about technologies, which are related to the QoS, is described. Those technologies include the DiffServ model, OSPF protocol and several routing algorithms.

2.1 DiffServ

In the DiffServ model, the network traffic is divided into a small number of classes, including one BE class and a few QoS classes. The goal of the DiffServ is to provide and allocate network services based on classes.

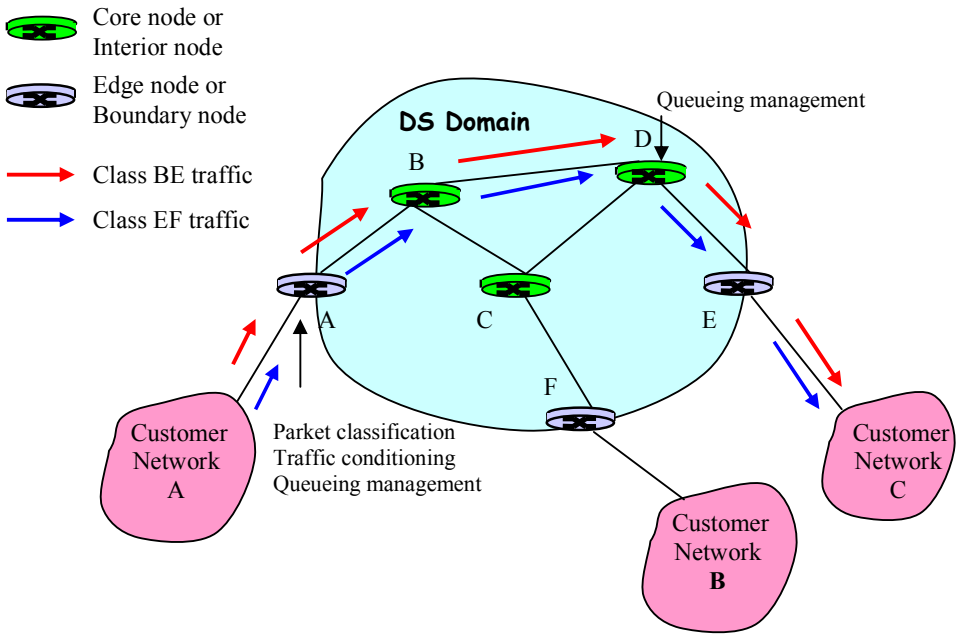

Figure 2.1: An example of a DiffServ network

There are two types of DiffServ nodes in a DiffServ network, which is shown in Figure 2.1. The nodes located at the boundary of the DiffServ network are called boundary nodes or edge nodes, whereas the nodes inside the DiffServ network are called interior nodes or core nodes.

Edge nodes are responsible for packet classification and traffic conditioning. All incoming traffic is rightly mapped to different forwarding classes at the edge node. Then, the traffic must be handled by the traffic conditioning before the packets enter the DiffServ network. According to users' traffic profiles or contract, the edge node exactly

meters users' traffic, precisely marks a corresponding Differentiated Services Codepoint (DS codepoint) into the IP header of each packet and sends those packets into the DiffServ network. When a user's practical traffic exceeds the limitation of its traffic profile, the traffic conditioning must carry out the functions of dropping or shaping to the user's traffic.

Core nodes are in charge of providing different traffic classes with variable forwarding treatments. Core nodes allocate network resources to distinct traffic classes by means of Per-Hop Behaviors (PHBs) that depict the forwarding treatments. One of PHBs is selected at each core node in terms of the DS codepoint set in every packet's IP header by a certain edge node.

In the DiffServ network, edge nodes perform different functions from core nodes. Only edge nodes keep track of per-class states to execute traffic conditioning, whereas core nodes always deal with the network traffic as an aggregate. Traffic policing is performed at the edge of the DiffServ network, and the class-based forwarding inside the DiffServ network.

In the DiffServ network, all traffic from one source to the same destination traverses through the same path because a single routing table exists in each node. In Figure 2.1, the BE class traffic and EF class traffic use the same path between the customer network A and the customer network C.

2.1.1 DS Domain

A DS domain is composed of a set of contiguous DiffServ nodes, including edge nodes and core nodes. Those DiffServ nodes are controlled and managed by the same administration. For example, an AS of an ISP can be configured as a DS domain. Within a DS domain, the interpretation of the DS codepoints is uniform so that all the traffic with the same DS codepoint is provided the same forwarding treatment by DiffServ nodes.

2.1.2 DS Codepoint (DSCP)

DS codepoint (DSCP) is a label, which is used to classify the network traffic in the DiffServ network [4]. The DS codepoint must be set to each packet before packets enter the DiffServ network. It is placed in the Type of Service (TOS) field of the IP header. The DS codepoint occupies the leftmost 6 bits of the TOS field to encode the forwarding treatments. The rest of bits in the TOS field are currently unused. Figure 2.2 shows the

format of the DS codepoint in the TOS field. The current standard DS codepoint allocation is defined in RFC2474.

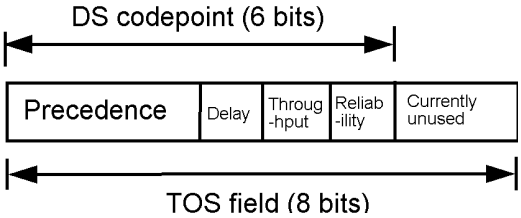

Figure 2.2: DS codepoint in the TOS field

2.1.3 Per-Hop Behaviors (PHBs)

Per-Hop behavior (PHB) describes the forwarding treatment in DiffServ nodes, including edge nodes and core nodes. Each PHB is represented by a DS codepoint. PHBs are often implemented by buffer management and queueing. Within a DS domain, all packets with the same DS codepoint share the same forwarding treatment.

2.1.4 Traffic Conditioning

Traffic conditioning is a more sophisticated mechanism. Only edge nodes need to perform the traffic conditioning to their incoming traffic. Figure 2.3 illustrates different elements of the traffic conditioning [4].

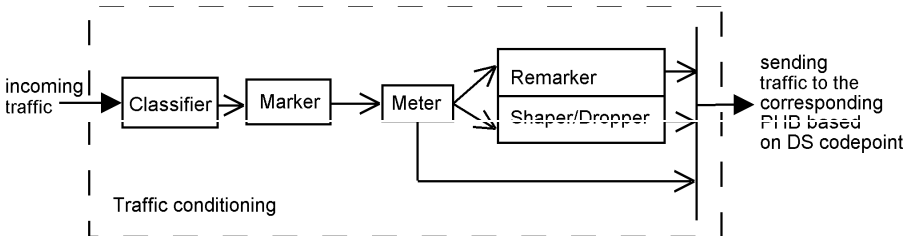

Figure 2.3: Traffic conditioning elements

Traffic conditioning functions consist of five elements:

- Classifier classifies the incoming traffic into diverse classes in terms of some pre-defined rules. There are two kinds of classifiers, known as behavior aggregate (BA) classifiers and multifield (MF) classifiers. BA classifiers separate the traffic based on DS codepoints, whereas MF classifiers divide the traffic based on a combination of one or more fields in the IP header, such as source IP address and source port. In the traffic

conditioning mechanism, the MF classifier is employed to perform the function of packet classification.

- Marker/Remarker places a particular DS codepoint into the TOS field of each classified IP packet. Sometimes the remarker may change the DS codepoint of some packets as needed.
- Meter measures the traffic from customers based on their traffic profiles. If the traffic does not exceed the profile, the traffic can be sent to the forwarding treatment module.
- Shaper delays packets from a certain customer in order to meet the traffic rate specified by the customer's traffic profile if the current traffic from the customer exceeds the profile.
- Dropper drops packets that are out-of-profile.

2.1.5 DiffServ Reference Implementation Model

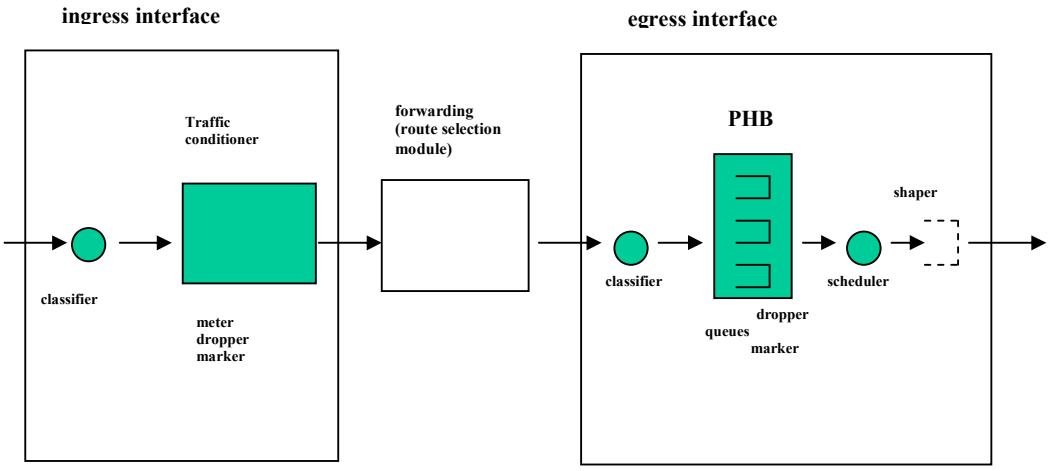

Figure 2.4: A typical DiffServ node implementation model

Fundamental functions of a DiffServ node can be divided into two modules, including traffic conditioning module at the ingress interface and PHB module at the egress interface [16]. Two modules are necessarily implemented at the interfaces in an edge node, whereas interfaces of a core node are enough to possess the PHB module. A typical implementation model of the DiffServ node is shown in Figure 2.4.

At the ingress interface of edge nodes all incoming traffic should be first separated by a MF classifier and sent to a traffic conditioner, which is responsible for metering users'

traffic, marking the corresponding DS codepoint and shaping or dropping the out-of-profile traffic.

To schedule the packet forwarding, the PHB module, which includes a BA classifier and queuing technologies, is implemented at the egress interface of both edge nodes and core nodes. Especially, the queuing technologies play an important role in the process of the packet delivery. A queuing discipline, which realizes different PHBs, consists of two essential elements: packet scheduler and buffer management. When packets reach the correct egress interface, the BA classifier classifies them to select one actual PHB based on their specific DS codepoints. Then, they are sent to the correct queue.

In the DiffServ implementation model, the process of selecting routes is still based on the IP destination address in each packet instead of DS codepoints. A normal packet forwarding mechanism is used in the DiffServ implementation model.

2.2 OSPF

Open Shortest Path First (OSPF) is a link-state routing protocol. It is based on the SPF algorithm. OSPF version 2 (OSPFv2) [5] is widely supported and used in the current Internet.

The OSPF protocol is a dynamic routing protocol. It can recalculate new routes after the procedure of the convergence when any change is detected. The procedure of the convergence is involved with the traffic of the routing information, the flooding procedure and the routing calculation.

Within an Autonomous System (AS) that is an OSPF routing domain, all routers maintain an identical link-state database describing the topology of the AS. The link-state database is composed of diverse types of link-state advertisements (LSAs). The OSPF protocol can rapidly detect any change in the AS topology, exactly synchronize all routers' link-state databases by means of the procedure of flooding and successfully recalculate new routes through performing the same routing algorithm.

The same routing algorithm is used in all routers within the AS. The routing algorithm is responsible for deploying the link state information stored in the link-state database to build up a shortest-path tree with the router as the root. In a routing table, the route to each destination in the AS can be formed in terms of the shortest-path tree.

The OSPF protocol supports a hierarchical network topology. The OSPF routing domain can be divided into several groups, which consist of one or more contiguous networks and hosts. Such a group is called an area. In a multi-area topology, the routing information traffic is reduced, but the inter-area routing calculation is needed.

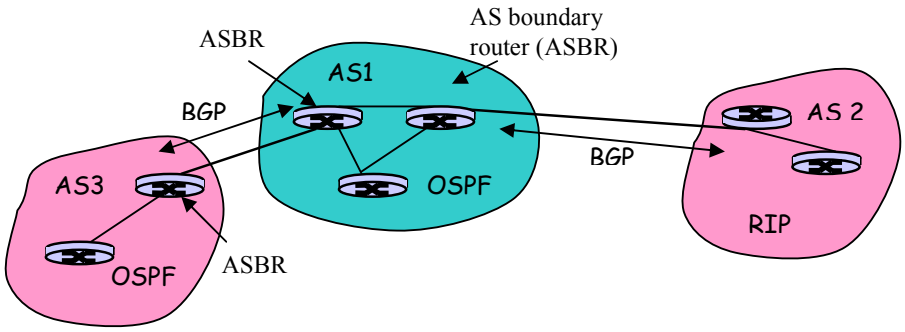

Figure 2.5: A multi-AS topology

The external routing information created by other routing protocols, such as BGP and RIP, can be advertised throughout the whole AS. An AS boundary router (ASBR), which exchanges routing information with routers belonging to other ASs, is responsible for mapping the external routing information into the corresponding OSPF routing information. The corresponding routes can be installed into the routing table after the external routing calculation is performed. A multi-AS topology is shown in Figure 2.5.

2.2.1 OSPF Area and Area Border Router

The OSPF protocol allows one or more areas within an OSPF routing domain to construct a hierarchical topology, which is shown in Figure 2.6. The hierarchical topology, which is viewed as a two-level structure, consists of one backbone area and several non-backbone areas. The backbone area, which is the special OSPF area 0, distributes link-state advertisements among non-backbone areas.

The routes within an area are only determined by the area itself because the topology of the area is always hidden from other areas of the Autonomous System. Each area maintains its own link-state database. That is to say, routers belonging to the same area have an identical link-state database. The link-state routing algorithm is applied to the link-state database to create the intra-area routes.

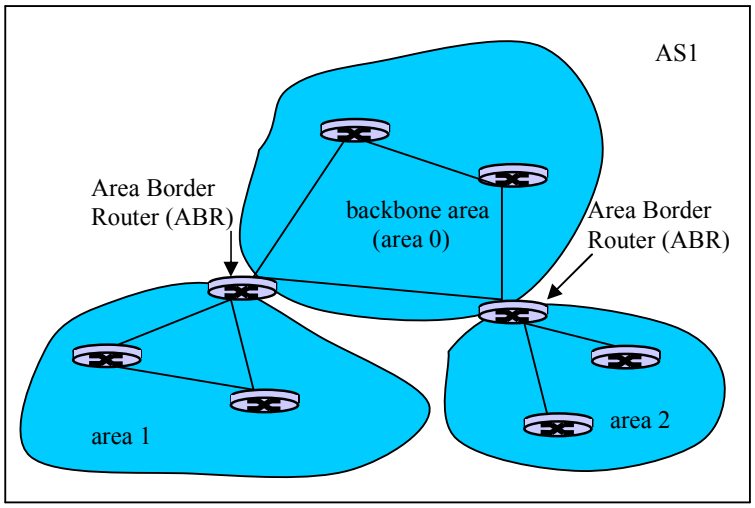

Figure 2.6: A hierarchical topology with multiple areas

A router belonging to more than one area is called an area border router (ABR). That means the ABR is attached to multiple areas. The ABR possesses a separate link-state database for each attached area, and it runs the routing algorithm to the link-state database of each attached area in parallel. According to the intra-area routes in the routing table, the ABR is in charge of summarizing the routes and creating the Summary-LSAs used to calculate the inter-area routes.

2.2.2 OSPF Sub-protocols

The OSPF protocol consists of three sub-protocols: Hello protocol, Database Exchange protocol and Flooding protocol. The Hello protocol is used to discover the neighbouring routers, establish and maintain the relationship with them. Hello packets are periodically sent out on all interfaces to neighbors in order to ensure that communication between two neighbors is bidirectional. The Database Exchange protocol is used to synchronize the link-state database between two neighbors during the course of building up the adjacency. The Flooding protocol is used to distribute the routing information throughout either an area or an OSPF routing domain.

To negotiate the optional capability between two neighboring routers, the OSPF protocol uses the Options field in the OSPF Hello packets and Database Description packets. The Options field is still present in all LSAs (see Figure 2.8). During the process of establishing the adjacency, the Options field in the Hello packets allows a router to reject another router if a certain capability does not match. Figure 2.7 shows the format of the

Options field. Now, five bits in the Options octet are defined for OSPFv2. The specific bit that represents TOS support does not exist in the OSPFv2.

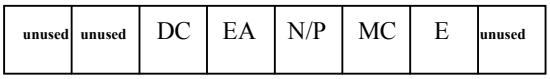

Figure 2.7: The Options field

The Flooding protocol defines a reliable and bidirectional procedure. The flooding procedure uses Link State Update packets to carry LSAs. Link State Acknowledge packets carrying the acknowledgement of each LSA implement the reliability of the flooding procedure. In OSPFv2, only AS-external-LSAs (type 5) need to be flooded throughout the entire OSPF routing domain. Other types of the LSAs, for example Router-LSAs, are flooded within a specific area.

2.2.3 OSPF link State Advertisements

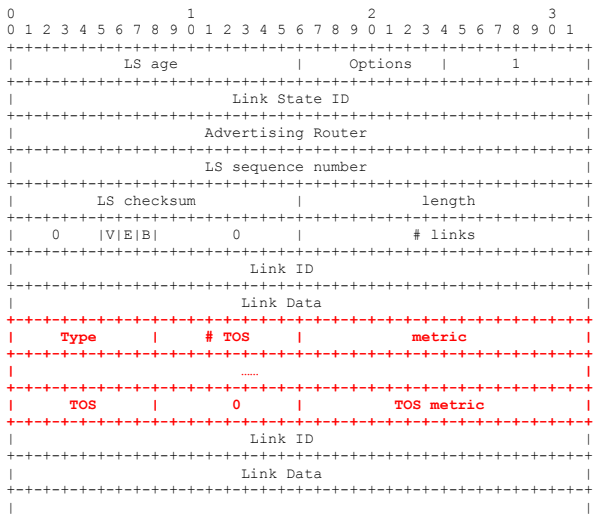

Figure 2.8: The Router-LSA format of the OSPF RFC1583

The OSPF protocol defines five types of link state advertisements (LSAs). Every distinct LSA type is used to implement different functions. A Router-LSA provides the information about connections between two routers or between one router and one network. A Network-LSA describes the information about all routers, which connect to the same broadcast network. A Summary-LSA originated by the ABR offers the summary information about one area's routes. An AS-external-LSA originated by the ASBR

describes the external routing information that will be flooded throughout an entire OSPF routing domain.

To support five service types specified in the IP datagram, the previous version of the OSPF protocol [6] originally supports the Type of Service (TOS) based routing. That means that more than one metric can be set into the OSPF LSAs. Figure 2.8 illustrates the format of the Router-LSA supporting the TOS. In the Router-LSA format the #TOS field represents the number of TOS metrics. The TOS field is set to a codepoint to represent a certain TOS. The metric field is set to the normal metric of the link, whereas the TOS metric field provides the TOS cost of the link. As a matter of fact, TOS based routing is not supported in the OSPFv2 [5]. That means that only one metric is present in all LSAs of OSPFv2.

2.2.4 Calculation of OSPF Routing Table

Based on a multi-area network topology, the whole routing table calculation of the OSPF protocol consists of the intra-area routing calculation, the inter-area routing calculation and the external routing calculation.

In a router, the intra-area routing calculation applies the SPF algorithm (details see next section 2.3) to all Router-LSAs and Network-LSAs belonging to the same area to build up the shortest path tree with the router as the root. According to the shortest path tree, the related intra-area routes are installed into the routing table. The router performs the SPF algorithm for each attached area.

Examining each Summary-LSA, the inter-area routing calculation uses the distance-vector approach to calculate the inter-area routes [7]. The approach to create the inter-area routes is to sum up the cost of the route to an ABR and the metric set in the Summary-LSA originated by the ABR. To avoid inter-area routing loops, the OSPF protocol adopts a split-horizon mechanism to the Summary-LSAs' creation and the inter-area routing calculation. It permits ABRs to advertise only Summary-LSAs related to the intra-area routes into the backbone area. Moreover, only Summary-LSAs stored in the backbone area's link-state database are examined during the routing calculation in each ABR.

The external routing calculation is performed based on all AS-external-LSAs in the link-state database. The approach to create the external routes is to sum up the cost of the route to an ASBR and the metric set in the AS-external-LSA originated by the ASBR. The

approach is similar to the inter-area routing calculation. However, the split-horizon mechanism is not needed in the external routing calculation because each AS-external-LSA is flooded throughout the entire routing domain.

The OSPF protocol supports incremental update calculations for inter-area routing calculation and external routing calculation. Receiving a new Summary-LSA or AS-external-LSA, a router does not recalculate the whole routing table because the route to the ABR or ASBR that originates the LSA could already exist in the routing table. In such a situation, the intra-area routing calculation is not necessary.

In short, the SPF algorithm is really performed during the intra-area routing calculation. However, both the inter-area routing calculation and the external routing calculation deploy the distance-vector approach.

2.2.5 OSPF Convergence

The OSPF protocol provides a fast, loop-less convergence [8]. OSPF convergence with normal settings takes the time, the range of which is from 30 seconds to 40 seconds [17][18]. The practical settings of the OSPF protocol can be extremely less than the normal settings [18]. The OSPF convergence consists of two phases [8]. The first phase is that the new routing information updates throughout the area or the OSPF routing domain by means of the flooding procedure. Please note that this phase includes the process of failure detection. The second phase is that the new shortest-path tree is calculated and the routing table is updated. During the procedure of the OSPF convergence, different LSAs are essential elements transmitted by the flooding procedure and examined by the routing calculations. Thus, the number of LSAs has an impact on the OSPF convergence.

A reasonable network topology plays an important role in the OSPF convergence. There are two reasons for this. One reason is that it can decrease the number of LSAs. Another reason is that the size of the network topology can negatively affect the speed of the LSAs' flooding and the SPF algorithm calculation. The OSPF protocol adopts a multi-area topology to reduce the LSA traffic and the size of the areas.

Route summarization mechanism can further contribute to reduce the number of Summary-LSAs. The goal of the route summarization is to aggregate several routes into one single link-state advertisement. A reasonable network topology can facilitate the

route summarization mechanism to aggregate addresses. Route summarization is normally done by the ABRs.

Incremental updates for inter-area routing calculation and external routing calculation can further reduce the time of the SPF calculation. The reason for saving the calculation time is that incremental updates can avoid the unnecessary intra-area routing calculation.

In short, the OSPF protocol provides some mechanisms to speed up the procedure of the convergence so that it can be used in a large network.

2.2.6 Extensions to OSPF

When the OSPF protocol supports one or more new functions, it may need to be extended in some aspects. As we know, the OSPF originally provides five types of LSAs. Several new types of LSAs have been defined to support special functions in the OSPF protocol. For example, a Type-10 LSA is used to support traffic engineering (TE) in the OSPF protocol [9]. The TE in the OSPF protocol is implemented as an independent module.

The extensions to OSPF could influence on some original modules in the OSPF. For example, if the format of the existing LSAs is changed, the routing calculation needs to be correspondingly modified. But, the structure of the link-state database does not need to be altered because an LSA is stored as a whole in the link-state database.

The extensions to OSPF could affect the OSPF convergence. Sometimes the extensions may increase new types of link state advertisements for the OSPF protocol so that the flooding traffic increases in the flooding procedure. The time of the SPF algorithm could be increased since there is a need to check those new LSAs.

In brief, the extensions to the OSPF protocol should produce a minimal impact on the original OSPF protocol, including the OSPF convergence. The new function implemented in the OSPF should be viewed as an independent module.

2.3 Link-state Routing Algorithms

The OSPF protocol originally supports the Dijkstra's algorithm as the routing algorithm. However, it can support several QoS routing algorithms after the OSPF protocol is reasonably extended [10].

2.3.1 Dijkstra's Algorithm

The Dijkstra's algorithm, called the SPF algorithm, computes the shortest path based on the weight function. It is responsible for building up a shortest path tree such that the path to each destination has the minimum summary value of the weight function among all feasible paths.

The following pseudocode describes the SPF algorithm:

```
Algorithm ShortestPath(G, v):
Input:   A shortest path tree G
         A root vertex v of G
Output: D[u] is used for each vertex u of G
        D[u] is the hop count of a shortest path from v to u in G.

initialize D[v] ← 0 and D[u] ∞ ← +∞ for each vertex v ≠ u
        let Q be a priority queue that contains all of the vertices of G using the value of
        D[u] as keys. The D[u] with the smallest value is listed on top of Q
while Q ≠ ∅ do {pull u into the cloud C}
      u ← Q.removeMininumElement()
      for each vertex z adjacent to u such that z is in Q do
            /* perform the relaxation operation on edge (u, z) */
            if z is not in the cloud C then
                  if D[u] + weight (u,z) < D[z] then
                     D[z] = D[u] + weight (u,z)
            else
                  if D[u] + weight (u, z) < D[z] then
                     D[z] ← D[u] + weight (u, z)
            change the key value of z in Q to D[z]
return the D[u] of each vertex u.
```

Cloud C: the algorithm keeps track of the set of vertices for which the distance has been computed, called the Cloud C

Weight function:
- Dijkstra's algorithm: weight (u,z) = distance between u and z
- SP algorithm: weight (u,z) = hop count
- BSP algorithm: weight (u,z) = 1/bandwidth

The Dijkstra's algorithm can form a shortest path tree started from a root vertex v. It computes the distance D[u] for each vertex u. D[u] represents the distance from the root vertex v to vertex u. In the algorithm, the cloud C is defined to keep track of the collection of vertices, the distance to which has been computed. A priority queue Q is used to list a set of D[u] based on their values. The vertex u with the smallest D[u] is always listed at the top of the queue Q. The vertex u with the smallest D[u] is removed from the queue Q to determine new vertices into the cloud C and update the distance of vertices that has adjacency with the vertex u. The algorithm is responsible for updating the value of D[u] if a shorter path from v to u is found during the calculation. When the

queue Q is empty, D[u], which is the smallest distance for each vertex, has been computed.

2.3.2 SP Algorithm and BSP Algorithm

Both SP algorithm and BSP algorithm are essentially Dijkstra's algorithms. However, the weight function of the SP algorithm is different from that of the BSP algorithm. Two algorithms can use the pseudocode of Dijkstra's algorithm with their own weight function. Only one cost for each destination is obtained after the algorithm calculation is performed.

The SP algorithm computes a shortest path with the hop count metric. It is responsible for building up a shortest path tree such that the path to each destination has the minimum hop count among all feasible paths [11]. The SP algorithm supports equal-cost paths. If several feasible paths with the same hop count exist, one of them is randomly selected for use. The weight function is defined as

$$weight\ (vi,vj) = hop\ count$$

and

$$weight\ (p(v1, v2,...vn)) = \sum_{i=1,j=2}^{n} weigth(vi,vj)$$

The BSP algorithm is basically a shortest-path algorithm, the weight function of which is the sum of the inversed bandwidths. It belongs to the QoS routing algorithms. It can meet the QoS requirement on maximum throughput because the path with the maximum bandwidth is always selected. The weight function is defined as

$$weight\ (vi, vj) = \frac{1}{bandwith(vi,vj)}$$

and

$$weight\ (P(v1,v2,...vn)) = \sum_{i=1,j=2}^{n} weigth(vi,vj)\,.$$

The path determined by the SP algorithm is really a shortest path, whereas the path selected by the BSP algorithm could not be a shortest path. From the view of the hop count, sometimes the BSP algorithm could select a longer path because the longer path has the widest bandwidth.

The BSP algorithm can achieve better saturate bandwidth that is the maximum value of the bandwidth than the SP algorithm. However, as network topology changes, the performance of the BSP algorithm could be decreased sharply. For example, the path

selected by the BSP algorithm becomes longer so that more network nodes and more traffic share the same path. In this situation, the saturate bandwidth is quickly dropped [12].

2.3.3 Other Hop-by-Hop Routing Algorithms

WSP algorithm calculates a shortest path tree that the path to each destination is the minimum hop count among all feasible paths. If several such paths exist, the one with the maximum bandwidth is used for the destination. If several paths with the same bandwidth exist, one of them is randomly selected. To calculate the shortest tree, the WSP algorithm simultaneously needs both the hop count metric and the bandwidth. Two costs as a pair for each destination are obtained after performing the WSP algorithm [10].

EBSP algorithm is the extension to the BSP algorithm. The EBSP algorithm introduces a "penalty" factor θ into the weight function of the BSP algorithm. The factor θ is likely to be exponential with regard to the hop count in order to avoid a path becoming too long.

2.4 Summary

Nowadays the OSPF protocol is an important routing protocol used in the Internet. It supports a multi-area topology for a large network. The OSPF convergence is a fast, loop-less procedure. The OSPF protocol possesses the ability to extend to support new functions. The SP algorithm can be used in the intra-area routing calculation of the OSPF protocol as the SPF routing algorithm, whereas the BSP algorithm can be deployed as the QoS routing algorithm. They can meet the different QoS requirements in the routing calculation. The distance-vector approach is used for the inter-area routing calculation and the external routing calculation. The OSPF protocol can provide a single routing table for the route selection module in the DiffServ architecture. It determines which path the traffic should select before the traffic is handled by the queueing technology on the specific egress interface. Thus, the OSPF protocol can provide the routing table for both the Internet and the QoS DiffServ network.

3. Multi-class Routing (MCR) Based on DiffServ Scheme

In this chapter, we will discuss the multi-class routing (MCR) scheme for the DiffServ network. The multi-class routing (MCR) based on the DiffServ architecture is used to reasonably optimize the network traffic and sharply alleviate the inter-class effect to the BE class traffic. The principle of the MCR is described in detail. We define the architecture of the MCR DiffServ system.

3.1 Related Work

A lot of research work about QoS routing has been done. Some work focus on the QoS routing algorithms, which can satisfy a certain demand on the specific QoS requirements [12]. To improve the performance of QoS routing, the optimization of topology aggregation is discussed in Woogui's work [13]. QoS routing extensions to OSPF have been defined in an RFC specification [10]. The study of the performance of the QoS routing extensions to OSPF is presented by G. Apostolopouls [14]. The fact that the above research work is always based on a single routing table can be obviously noted. All classes in the QoS network use the same routing table. As a result, the throughput of the BE class drops more than that of other QoS classes when network congestion occurs or the network link load is heavy. Any QoS routing algorithm and the network architecture based on one routing table cannot efficiently alleviate the inter-class effect.

The class-based route selection is considered in some recent research work. Multiple routing tables are used for the class-based route selection [3] [15]. In the multi-class routing (MCR) scheme [3], multiple routing tables are respectively created by means of different routing algorithms, including the standard SPF algorithm and QoS routing algorithms. The diverse classes use the different routing tables to make the route selection.

MCR can effectively alleviate the inter-class effect [3]. Four different routing algorithms, including SP, BSP, WSP and EBSP, are studied by means of simulation. Different routing algorithms are responsible for calculating a routing table for distinct classes. For example, the EF class traffic uses the SP routing table to make the route selection while the BE class traffic deploys the BSP routing table to make the route selection. There is an assumption that the BE class aims for maximum throughput, and the EF class aims for minimum delay. According to the results of the MCR simulation, MCR can largely improve the performance of the BE class traffic in comparison with the route selection

using the same routing table. When the BSP algorithm and the SP algorithm are used for the BE class and the EF class respectively, two classes can be maximally optimized.

On the other hand, the number of the routing tables can negatively influence on the performance of the routing protocols, for example, the convergence, and the software complexity. Thus, we focus on the MCR scheme with two routing tables in this thesis. To alleviate the inter-class effect, one routing table is used for the QoS traffic, whereas another is used for the BE traffic.

3.2 Multi-class Routing scheme

The multi-class routing (MCR) scheme simultaneously provides more than one routing table computed by the various routing algorithms on each network node. In the traditional routing scheme, there is only one routing table on each network node so that both the BE traffic and the QoS traffic must use the same routing table. On the contrary, the different classes are allowed to use the distinct routing tables to make the route decision in the MCR scheme. In short, the MCR scheme can implement the class-based route selection in the IP network.

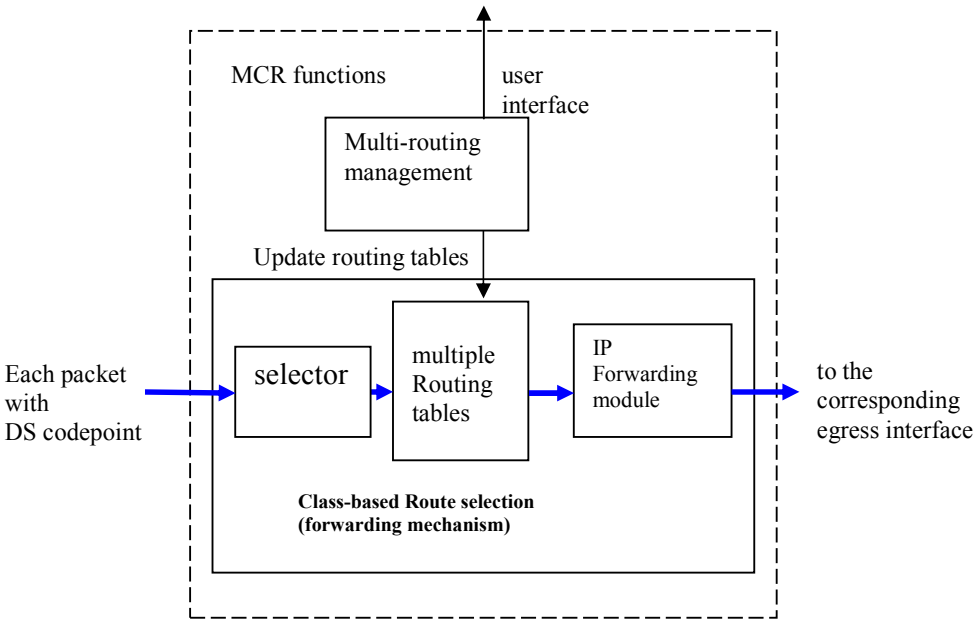

Figure 3.1: Functions of the MCR scheme

To support the MCR scheme, each network node should possess two essential functions, known as multi-routing management and class-based route selection. Figure 3.1 illustrates the functions of the MCR scheme.

The multi-routing management is responsible for collecting the routing information, calculating the multiple routing tables, controlling the routing algorithms, optimizing the routing calculation and updating multiple routing tables. Work modes of the multi-routing management can be static or dynamic (see next section). An administrator can configure the multi-routing management by means of the user interface provided in the multi-routing management. According to characteristics of the different classes, the administrator can choose the appropriate routing algorithms to create the routing tables. The multi-routing management automatically updates the routing tables in the class-based route selection after the routing tables are recalculated.

The class-based route selection, which implements functions of the IP forwarding mechanism, is in charge of selecting one routing table in terms of the DS codepoint in each packet's header, and then making the correct route selection for each packet. Packets belonging to the same class select the same routing table because each class is represented by its unique DS codepoint in one routing domain. In the class-based route selection, a selector that is used to correctly choose one routing table from multiple routing tables is needed. Moreover, the selector works based on the DS codepoints. The normal IP forwarding module sends the packet to the corresponding egress interface after looking up the next hop for the packet from the selected routing table.

3.3 Different MCR Approaches

The multi-routing management, which is an essential MCR function, can be implemented by means of the following MCR approaches. To make MCR more efficient, there is a need to simultaneously deploy more than one approach for the multi-routing management.

3.3.1 Static MCR

Static MCR is the simplest approach to implement the multi-routing management. In a router, link weights, such as hop count and bandwidth, are static. They are only assigned by the network administrator. The administrators can make choices on which links to carry the traffic of which class. The weights of each link cannot be frequently and automatically changed such that the limited routing traffic is added into the routing domain. This approach can produce a minimal impact on convergence.

However, routes in each routing table, which is calculated by a certain QoS routing algorithm, cannot exactly reflect the network situation. When network congestion takes

place on a link, the routes traversing through the link cannot be automatically adjusted by routers.

3.3.2 Dynamic Distributed MCR with Time Based Triggering

The second MCR approach is to trigger to update a router's link state information once in time *T* by means of using a timer. When the timer expires, the router immediately takes the measurement of class weights on each link. Please note that the router measures the weights of any link only if the timer is expired. If the change of any weight has taken place, the router instantaneously creates its own link state information, and then floods it to the neighbors of the router. This approach is called dynamic distributed MCR with time based triggering. In a router, network administrators can configure a period *T* for a timer so that the router can periodically measure its links and update its link state information. Thus, the routes in each routing table can dynamically reflect the network situation since the router can automatically adjust them.

However, this approach could result in a frequent convergence process and increase packet loss since it unavoidably introduces more additional routing information into the routing domain. Moreover, a router cannot respond to any sudden big change of link weights within a time *T*.

This dynamic approach needs to take the time *T* of a timer into account. If the time of a router's timer is too short, the router could frequently update its routing tables and more routing information could be introduced. On the contrary, if the period time *T* is too long, the routing information could be decreased. But, the router could miss some significant events.

In addition, the time *T* of the timer in different routers should be used carefully since the approach could introduce the problem of timing of the LSAs from diverse routers. If the timer triggers approximately at the same timer in all routers, a peak of LSA traffic is introduced. This could result in increasing the packet loss for user traffic. On the other hand, if timer expirations in different routes are evenly spaced over time, they could make the network unstable because of the frequent routing table calculations.

3.3.3 Dynamic Distributed MCR with Significant Event Triggering

The third MCR approach is to trigger to update a router's link state information based on either time *Tmax* or a significant event. This approach is called dynamic distributed MCR

with significant event triggering. In the approach, timer *Tmax*, *Tmin* and *Twait* are used for a router. The router automatically detects significant events, for instance, a sharp increase in the bandwidth usage of a link. When a certain significant event happens in the router, the router does nothing if less than *Tmin* has passed after the previous link state information. Otherwise, the router creates its updated link state information, and then floods the link state information to its neighbors. Having sent at least one updated link state information, the router will wait for *Twait* for other link state information to arrive, and then calculate its routing tables. When the *Tmax* is expired and no significant events happen, the router measures weights of the router's links. If the change of any weight takes place, the new link state information of the router is immediately created and flooded.

This approach is similar to the dynamic distributed MCR with time based triggering. This approach could result in a frequent convergence process and increase packet loss since it unavoidably introduces more additional routing information into a routing domain. The values of the timers *Tmax, Tmin* and *Twait* need to be sincerely considered. However, compared to the dynamic distributed MCR with time based triggering, the approach can respond to sudden significant events.

3.3.4 Centralized Dynamic MCR

Centralized dynamic MCR is an approach that the whole multi-routing management is implemented in a centralized route server. The centralized route server can work based on time based triggering or significant event triggering. It has two different work modes: routing information listener and polling.

In the routing information listener mode, the centralized route server works like a normal router except that it does not create any link state information and only receives other routers' link state information. In the polling mode, the centralized route server obtains each router's link state information by means of polling. Routers can also obtain other routes' link state information. Having received all routing information, the centralized route server can separately calculate the MCR routing tables for each router. Then, it distributes the corresponding routing tables for each router. The routes, which are downloaded from the centralized route server, are stored into each router's routing tables with high priority. The routers calculate the corresponding routes for themselves based on receiving the routing information. Those routes are still stored in the routing tables with low priority. A router always uses the routes with the high priority for making the route

selection as long as the routes are not expired. When the routes with the high priority are expired or the centralized route server is broken down, the routes with the low priority will be used for making the route selection.

The advantage of the centralized dynamic MCR approach is that the server works based on a network wide view. However, convergence could be still the main problem for the centralized dynamic MCR approach. The approach is a kind of dynamic MCR approach. It cannot avoid the increasing routing information. In addition, to normally communicate with routers, the centralized dynamic MCR needs to define new functions and mechanisms.

3.4 MCR DiffServ

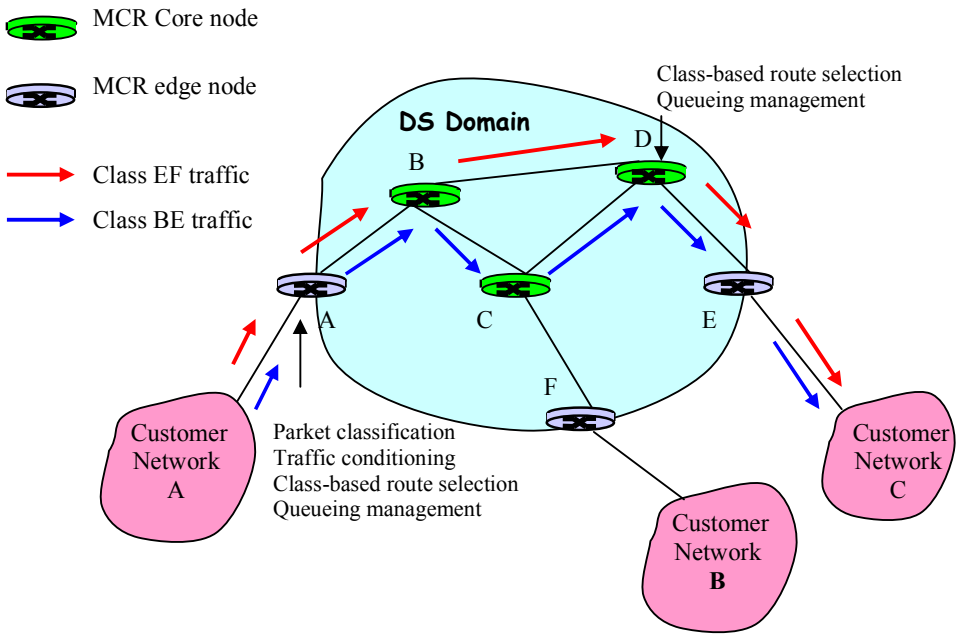

Figure 3.2: An example of the MCR DiffServ network

MCR DiffServ is the new network architecture in which the MCR scheme is used in the DiffServ architecture. An MCR DiffServ network is shown in Figure 3.2. The MCR DiffServ network is still composed of two essential elements: MCR edge nodes and MCR core nodes.

In the MCR DiffServ model, the MCR edge nodes and MCR core nodes implement all basic DiffServ functions described in section 2.1. Any interface connected to the customer networks on the MCR edge nodes is responsible for packet classification and traffic conditioning. Any interface used as the egress interface belonging to either the

MCR edge nodes or the MCR core nodes provides the PHBs for the outgoing traffic. So, the queueing and scheduling technology is used in the MCR DiffServ model in the same way as in the traditional DiffServ model.

Compared to the traditional DiffServ, a new routing scheme, the multi-class routing, is added into the MCR DiffServ. To implement the multi-class routing in the DiffServ, each MCR DiffServ node should support two new functions, which are the multi-routing management and the class-based route selection described in section 3.1.

The procedure how to handle packets on the MCR DiffServ node is described as follow:

Step 1: An MCR edge node can assign a DS codepoint to an incoming packet's IP header after traffic conditioning is performed by the MCR edge node. MCR core nodes always ignore the first step.

Step 2: One of several routing tables is selected based on the DS codepoint in the IP header.

Step 3: According to the IP destination of the packet, the next hop is determined by means of looking up the selected routing table.

Step 4: The packet is sent to the corresponding egress interface.

Step 5: The packet is sent out the egress interface after the specific PHB function is performed.

Figure 3.2 shows an example of the MCR DiffServ network. Each MCR DiffServ node provides two routing tables computed by the distinct routing algorithms. For example, traffic is sent from customer network A to customer network C. The traffic can be classified into two different classes, which are the BE class and the EF class. The traffic of the BE class traverses through the path A->B->C->D->E, whereas the traffic of the EF class goes through the path A->B->D->E. So, when the load on the path A->B->D->E is heavy, the influence on the BE class is less because the class BE traffic goes through the path A->B->C->D->E.

3.5 Communication between MCR DiffServ and DiffServ

In a DS domain, MCR DiffServ nodes and DiffServ nodes can coexist in the following way. The DS domain consists of an MCR area and a non-MCR area. The MCR area is composed of MCR DiffServ nodes, whereas the non-MCR area comprises traditional DiffServ nodes. An MCR gateway node, which can correctly execute both the MCR functions and normal DiffServ functions, is located between the MCR area and the non-

MCR area. In the MCR area, the MCR gateway is used as an MCR core node, whereas it is used as a core node in the non-MCR area. It can correctly make the route selection for the traffic that traverses through the different areas. In the DS domain, each DS codepoint only has a unique interpretation for both the MCR DiffServ nodes and the DiffServ nodes. In this thesis, we focus on the intra-DS domain rather than the inter-DS domain. Figure 3.3 illustrates an example of the communication between MCR DiffServ nodes and DiffServ nodes within a DS domain.

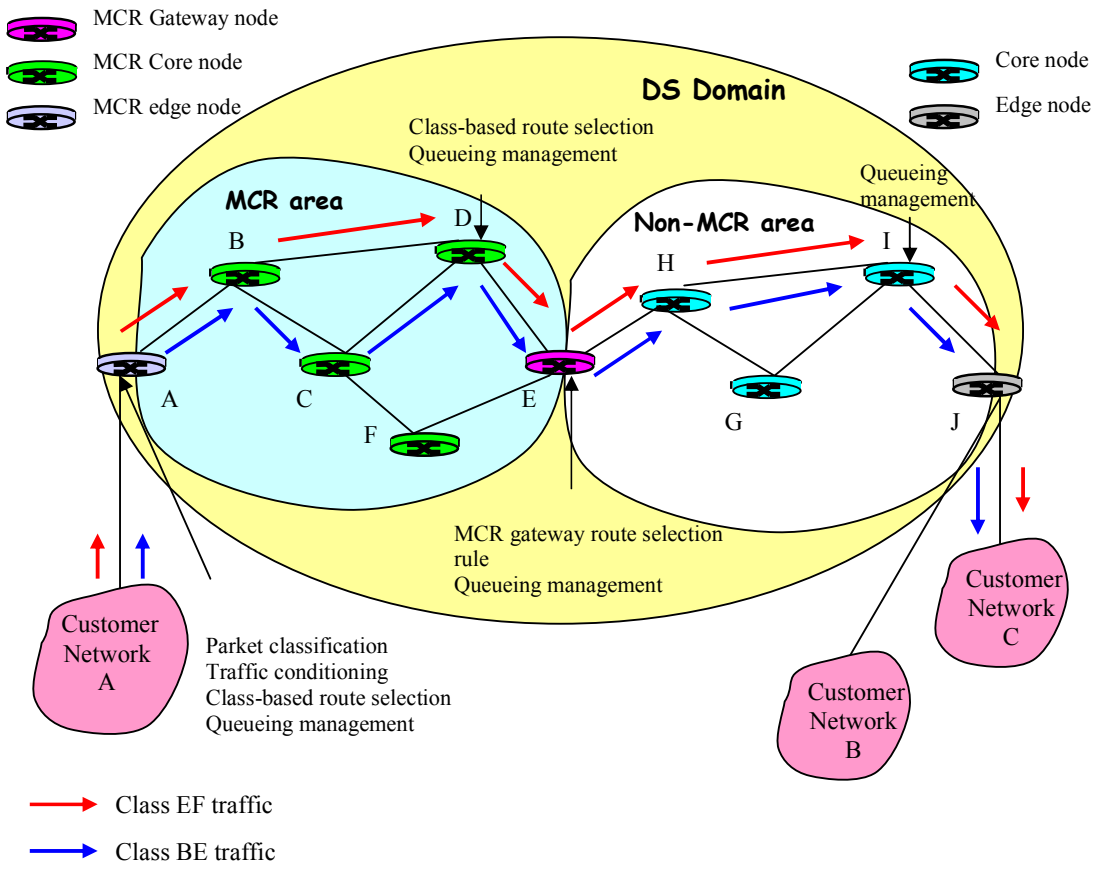

Figure 3.3: An example of communication between MCR DiffServ nodes and DiffServ nodes

All nodes, including MCR DiffServ nodes, DiffServ nodes and the MCR gateway node, can correctly establish one or more routing tables. The DiffServ nodes only have one routing table, whereas MCR DiffServ nodes build up the multiple routing tables. The routing table, called default routing table, is used for both the MCR DiffServ nodes and the DiffServ nodes. In the MCR core node or the MCR edge node, a default route to the MCR gateway node must exist in non-default routing tables. It is always used for the traffic, the destination of which is outside the MCR area. The MCR gateway is

responsible for advertising the default route information throughout the MCR area. To guarantee the correct communication between MCR area and non-MCR area, an MCR gateway default route rule is defined. In the MCR gateway node, the default route only exists in the default routing table, whereas the default route does not exist in any non-default routing tables.

Each node in the DS domain correctly makes the route selection for every packet. Each MCR DiffServ node performs the class-based route selection function to any packet in the MCR area. That means the route selection is based on the DS codepoint in the IP header of each packet. Each DiffServ node makes the route decision by using the same routing table, which is the default routing table.

To correctly make the route selection in an MCR gateway node, the MCR gateway node must obey the MCR gateway route selection rule that is described as follows. According to the DS codepoint of each IP packet, the MCR gateway node first looks up the next hop in the selected routing table. If the corresponding route in the selected routing table does not exist, the MCR gateway node will look up the next hop in the default routing table. If the corresponding route in the default routing table does not exist either, the MCR gateway node will drop the packet and then send the corresponding ICMP packet to the source of the packet. Based on the MCR gateway default route rule, a default route should exist in the default routing table. That means that if an exact route for a packet does not exist in any routing table, the MCR gateway node can still use the default route in the default routing table to deliver the packet.

Based on our principle, the BE class traffic and the EF class traffic traverse through different paths in the MCR area. The path A->B->D->E is used for the EF class, whereas the path A->B->C->D->E is selected for the BE class. But, they go through the same path E->H->I->J in the non-MCR area.

3.6 Design of the MCR DiffServ System

The MCR DiffServ system design is provided in this section. The design can be used in several operating systems, such as Linux, Solaris and FreeBSD.

An MCR DiffServ system consists of five main functional blocks: traffic conditioning block, QoS manager, class-based route selection block, MCR manager and output queueing block. The MCR DiffServ system design takes advantage of the characteristics

of the operating systems, which are composed of the user space and the kernel space. The traffic conditioning block, the class-based route selection block and the output queueing block will be implemented within the kernel. The MCR manager, which forms, controls and manages the multiple routing tables for different classes, is implemented in the user space. The QoS manager in the user space implements a QoS management mechanism. Figure 3.3 shows the MCR DiffServ system model.

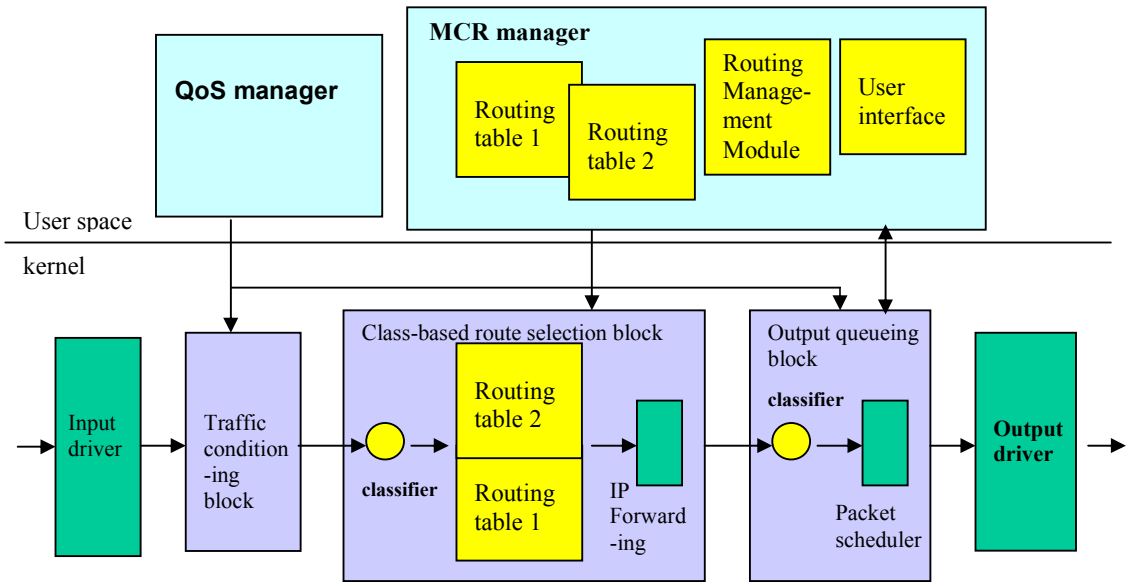

Figure 3.4: MCR DiffServ system model

The traffic conditioning block implements the functions of the DiffServ traffic conditioning. A MF classifier is used to classify the traffic in terms of the users' profile. A meter is used to meter the user' traffic. A marker is used to set the DS codepoint into the IP header of each packet. The traffic conditioning block uses a dropper to handle the out-of-profile traffic.

The class-based route selection block adds a classifier to determine which routing table should be used for each packet in terms of the DS codepoint in the IP header. If an MCR DiffServ system is used as an MCR edge node or an MCR core node, the classifier in the class-based route selection block is a BA classifier. If an MCR DiffServ system is used as an MCR gateway node, the classifier in the class-based route selection block must work in terms of the MCR gateway route selection rule. Major operating systems do not support multiple routing tables in the kernel. The kernel should be modified to store and manage the multiple routing tables. The IP forwarding block will send the packet to the correct egress after making the route selection.

The MCR manager implements the function of the multi-routing management described in section 3.2. To control the work of MCR, the MCR manager uses a user interface module to provide an interface for administrators. The routing management module is used to create the routing information, collect the information of the network topology and create multiple independent routing tables used for the class-based route selection block. If an MCR DiffServ system is used as an MCR gateway node, the routing management module can correctly create and flood the routing information about the default route. Moreover, it establishes the default route for the routing tables of the MCR gateway node based on the MCR gateway default route rule. The routing management module mainly implements the different routing protocols. It is the central module of the MCR manager, and it performs the vital MCR manager functions. The MCR manager implementation depends on the routing protocols. Now all routing protocols do not possess related mechanisms for supporting the multi-class routing. Thus, the related routing protocols, for instance OSPF, need to be extended to support the multi-class routing.

The output queueing block implements the functions of PHBs that are forwarding treatments. In the output queueing block, a BA classifier is used for selecting the correct packet scheduler. A packet scheduler implements the function of the queueing technology. The packet scheduler can be responsible for measuring the accurate state information of a link, for example, available bandwidth. The output queueing block can provide the accurate link state information for the MCR manager.

According to the users' profile information, the QoS manager configures or modifies the parameters of the components in the traffic conditioning block and the output queueing block in order to correctly control users' traffic at each interface.

3.7 Summary

MCR is introduced into the traditional DiffServ model. This is called MCR DiffServ. To support MCR, each node in a MCR DiffServ network must have two necessary functions: multi-routing management and class-based route selection. Different MCR approaches can be used for implementing the multi-routing management.

The MCR approaches can be classified into two kinds: static approach and dynamic approach. The static MCR approach basically minimizes the impact on convergence since the link weights of a router cannot be changed automatically. But, it cannot promptly

reflect any change of a network. The dynamic MCR approaches could be distributed or centralized. In general, they can measure the link weights of a router controlled by periodical timers or triggered by a significant event so that the router can automatically update its routing information. The dynamic MCR approaches could contribute to accurate balancing of the network traffic. On the other hand, the approaches result in a frequent convergence process and increasing packet loss. The time of the timers could introduce the problem of timing of the LSAs from different routers.

A MCR gateway node is added into a DiffServ network that consists of an MCR area and a non-MCR area. To correctly deliver traffic between the areas, except for the MCR gateway node, each node in the MCR area must have the default route to the MCR gateway node. Moreover, the MCR gateway node is responsible for creating and distributing the routing information about the default route. In addition, the MCR gateway default route rule and the MCR gateway route selection rule must be concurrently used in the MCR gateway node.

The MCR DiffServ adds the class-based route selection mechanism and the multi-routing management in comparison with the traditional DiffServ. One important task is to successfully create more than one routing table for each MCR DiffServ node. To finish the task, the routing protocols should be reasonably extended.

4. Design of OSPF with MCR Extensions

The MCR manager implementation is closely related to the specific routing protocol. To implement the MCR manager functions, we will focus on the design of the OSPF extensions in this chapter. As the result of the OSPF protocol extensions, the routing software will change correspondingly.

The main goals of the design of the OSPF with MCR extensions are:

- To provide two routing tables.
- To support static MCR approach and dynamic MCR approaches.
- To implement the basic MCR manager functions.
- To limit the additions to the standard OSPF version 2 protocol.
- To minimize the impact on the original OSPF, such as code and convergence, etc.

The basic functions of the MCR manager are shown as follows:

- Successfully collecting all routing information, including QoS routing information.
- Providing more than one routing algorithms, such as the SPF algorithm and the QoS routing algorithm.
- Automatically recalculating and updating routing tables if any change of the topology occurs.
- Effectively controlling and managing the routing calculation and the routing information creation. The routing information includes different types of link state advertisements.
- Offering a user interface to administrators. The administrator can check the configuration information, change the configuration information and view the OSPF information through the user interface.

4.1 OSPF Protocol Extensions

The OSPF protocol can be extended in order to support new mechanisms of the multi-routing management described in section 3.2. The extensions to the OSPF protocol should bring up only limited additions to the protocol itself. On the other hand, all of the existing OSPF mechanisms, functions, link-state advertisements, data structures and data formats will remain proper.

In this thesis, our OSPF with MCR extensions is the first version. In this version, we assume that routers supporting the MCR capability do not work with routers that do not support the MCR capability. This is a limitation of the first version and could be removed in later versions.

4.1.1 MCR Optional Capability

In order to provide the QoS routing information in the LSAs, MCR capability is introduced into the OSPF protocol. One bit in the Options field represents the MCR capability. The MCR capability enables OSPF routers to support or not support the MCR optional capability and to communicate with other routers that support the same capability. The OSPF routers supporting the MCR capability can reject to communicate with the routers that do not support the MCR capability.

M	unused	DC	EA	N/P	MC	E	unused

Figure 4.1: The modified Options format

An 'M' bit can be used as an indicator of the MCR capability for the multi-class routing mechanism. It is located at the most significant bit of the Options field. Figure 4.1 shows the modified Options field. The 'M' bit possesses two functions. One function is to support the MCR capability. Another function is to indicate that the TOS metric field could be used in the LSAs.

On the other hand, to avoid confusing the existing TOS based routing mechanism and the new multi-class routing mechanism, each TOS value encoded for the MCR capability is different from the TOS values used for the TOS capability. The TOS value encoding rules for the MCR capability will be discussed in the next section.

4.1.2 Encoding Resources as Extended TOS

To be compatible with the existing OSPF protocol, the new TOS values for the MCR capability are encoded in a way that either the existing OSPFv2 routers will ignore or misinterpret. So, the TOS values for the MCR capability are different from those for the TOS capability.

In the OSPF protocol, the number of the TOS values for the TOS capability is 16 because the RFC 1349 specification only uses four bits to define the TOS values used for the TOS

based routing. We extend the sixth bit to encode the TOS values for the MCR capability. As a result, 16 TOS values for the MCR capability are created. Table 4.1 shows the TOS values for both the TOS capability and the MCR capability. Now we only define OSPF encoding 72 for the bandwidth. The TOS values for the MCR capability have enough capability for extensions.

OSPF encoding	TOS for TOS capability		OSPF encoding	TOS for MCR capability	
0	0000	normal service	64	10 0000	
2	0001	minimize monetary cost	66	10 0001	
4	0010	maximize reliability	68	10 0010	
6	0011		70	10 0011	
8	0100	maximize throughput	72	10 0100	bandwidth
10	0101		74	10 0101	
12	0110		76	10 0110	
14	0111		78	10 0111	
16	1000	minimize delay	80	10 1000	
18	1001		82	10 1001	
20	1010		84	10 1010	
22	1011		86	10 1011	
24	1100		88	10 1100	
26	1101		90	10 1101	
28	1110		92	10 1110	
30	1111		94	10 1111	

Table 4.1: TOS values for TOS capability and MCR capability

4.1.3 Encoding Bandwidth Resource

The bandwidth needs to be encoded before it can be represented in the TOS metric field of each Router-LSA. The actual metric field and the TOS metric field of each Router-LSA are only 16 bits long. The maximum encoding value of the metric or TOS metric is 65535. But, the Gbits/s bandwidth of links has been used in the current Internet. The bandwidth must be converted to the available encoding value. Note that we assume that the maximum bandwidth in the networks is 10 Gbits/s.

The following equation (4-1) defines how to convert the bandwidth into the corresponding encoding value for the TOS metric in each Router-LSA.

$$TOS\ metric = \frac{bandwidth}{152.59(kbits/s)} \qquad (4\text{-}1)$$

The bandwidth that is more than 152.59kbits/s can use the equation (4-1) to calculate its encoding value, whereas the bandwidth, which is less than 159.59kbits/s, is directly equal to 1. For example, the bandwidth of 10Gbits/s is converted into 65535.

However, the bandwidth does not need to be converted before it is inserted into the TOS metric field in each AS-external-LSA. The actual TOS metric field in the AS-external-LSA is 24 bits long so that it can accommodate the bandwidth of 10 Gbits/s.

4.1.4 OSPF Packets and LSA Formats

Even though the OSPF protocol is extended, none of the formats of OSPF packets needs to be changed at all. Only the 'M' bit is introduced into the Options fields in all Hello packets, Database Description packets and all LSAs.

To support MCR capability, different types of the LSAs adopt the formats that can include the TOS information, and new types of the LSAs will be introduced. Except for the Options field, the format of Network-LSA supporting the MCR capability is in accordance with that of the Network-LSA used for the standard OSPF protocol.

4.1.4.1 Router-LSA Format

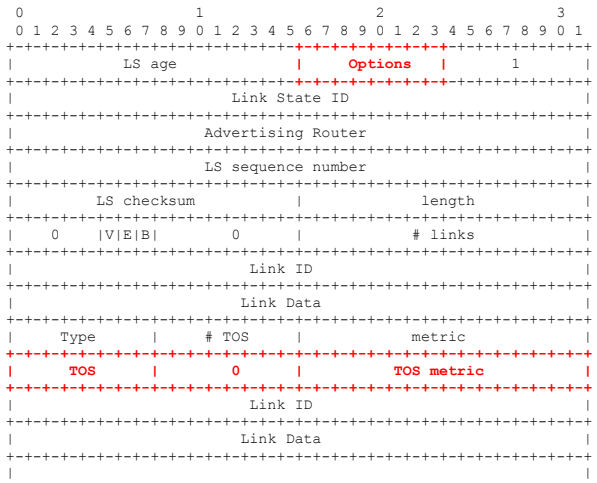

Figure 4.2: The MCR Router-LSA format

To support multi-class routing, two metrics, known as hop count and the bandwidth, need to be present in all Router-LSAs. Figure 4.2 illustrates the Router-LSA with two metrics. The #TOS field is set to 1. The metric field, which is TOS 0, shows the hop count. The TOS field is set to 72. That means that the bandwidth will be placed in the TOS metric field. The information about the link includes two metrics. Other fields in the Router-LSA are consistent with the standard OSPFv2 protocol.

4.1.4.2 Summary-LSA Format

Two metrics can be present in the Summary-LSAs. Figure 4.3 shows the format of the Summary-LSAs for multi-class routing. According to the routing table, all Summary-LSAs are created by the ABRs. If the cost of each route is a single value, only the metric field is used in the Summary-LSAs. Otherwise, the TOS metric field is needed. For example, the BSP algorithm only creates one cost for each route. So, in the Summary-LSA, only the metric field is present, and the TOS metric is not in use. The WSP algorithm produces two costs as a pair for each route so that the TOS metric in the Summary-LSAs is needed.

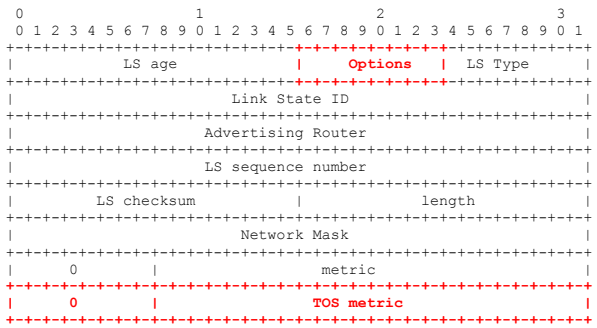

Figure 4.3: The MCR Summary-LSA format

To support more than one routing table, new types of Summary-LSAs need to be introduced because the type 3 Summary-LSA and type 4 Summary-LSA as a pair are used for a single routing table. Each new routing table needs a pair of the Summary-LSAs for itself. For example, for the second routing table, the type 12 Summary-LSA is used for the route, the destination of which is an IP network. The type 13 Summary-LSA is defined for the route, the destination of which is an AS boundary router.

Based on different types of Summary-LSAs for different routing tables, the route summarization mechanism can be separately performed for the different routing tables. Thus, aggregating IP addresses for one routing table does not affect other routing tables.

4.1.4.3 AS-external-LSA Format

Two metrics of the hop count and the bandwidth should be present in each AS-external-LSA in order to support the multi-class routing scheme. Figure 4.4 illustrates the AS-external-LSA with two metrics. The metric field, which is for TOS 0, represents the hop count. The TOS field is set to 72, meaning that the bandwidth will be placed in the TOS

metric field. The external routing information includes two metrics. The TOS value cannot be more than 127 because the most significant bit is used to represent the type of the external metric. Other fields in the AS-external-LSA are consistent with the standard OSPFv2 protocol.

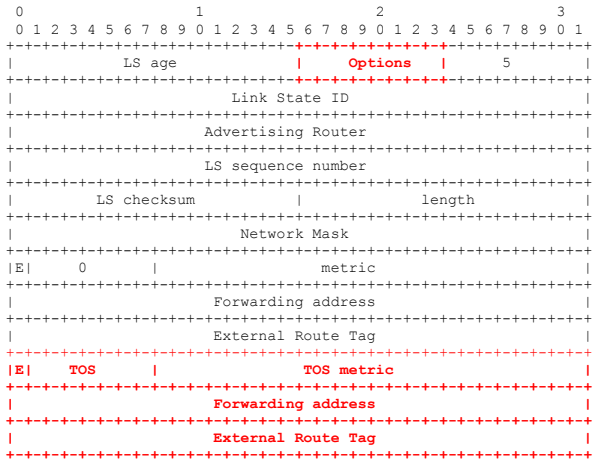

Figure 4.4: The MCR AS-external-LSA format

4.1.5 Calculation of Routing Tables

Compared to the standard OSPF protocol, each router supporting the MCR capability creates more than one routing table by means of using different routing algorithms. Based on the extensions to the OSPF protocol, all LSAs can provide one non-TOS metric and one TOS metric for different routing algorithms.

The routing table calculation performed by any routing algorithm still includes three essential steps: the intra-area routing calculation, the inter-area routing calculation and the external routing calculation. The real routing algorithm is performed to compute the intra-area routes, and the distance-vector approach is used for calculating the inter-area routes and the external routes. During the course of creating a routing table, the routing algorithm examines all types of the LSAs stored in the link state database.

One routing algorithm is independent of other routing algorithms. One routing table calculation does not have any influence on other routing table calculations. As a consequence, the process of creating the Summary-LSAs related to the different routing tables can be separately performed.

4.2 Software Architecture

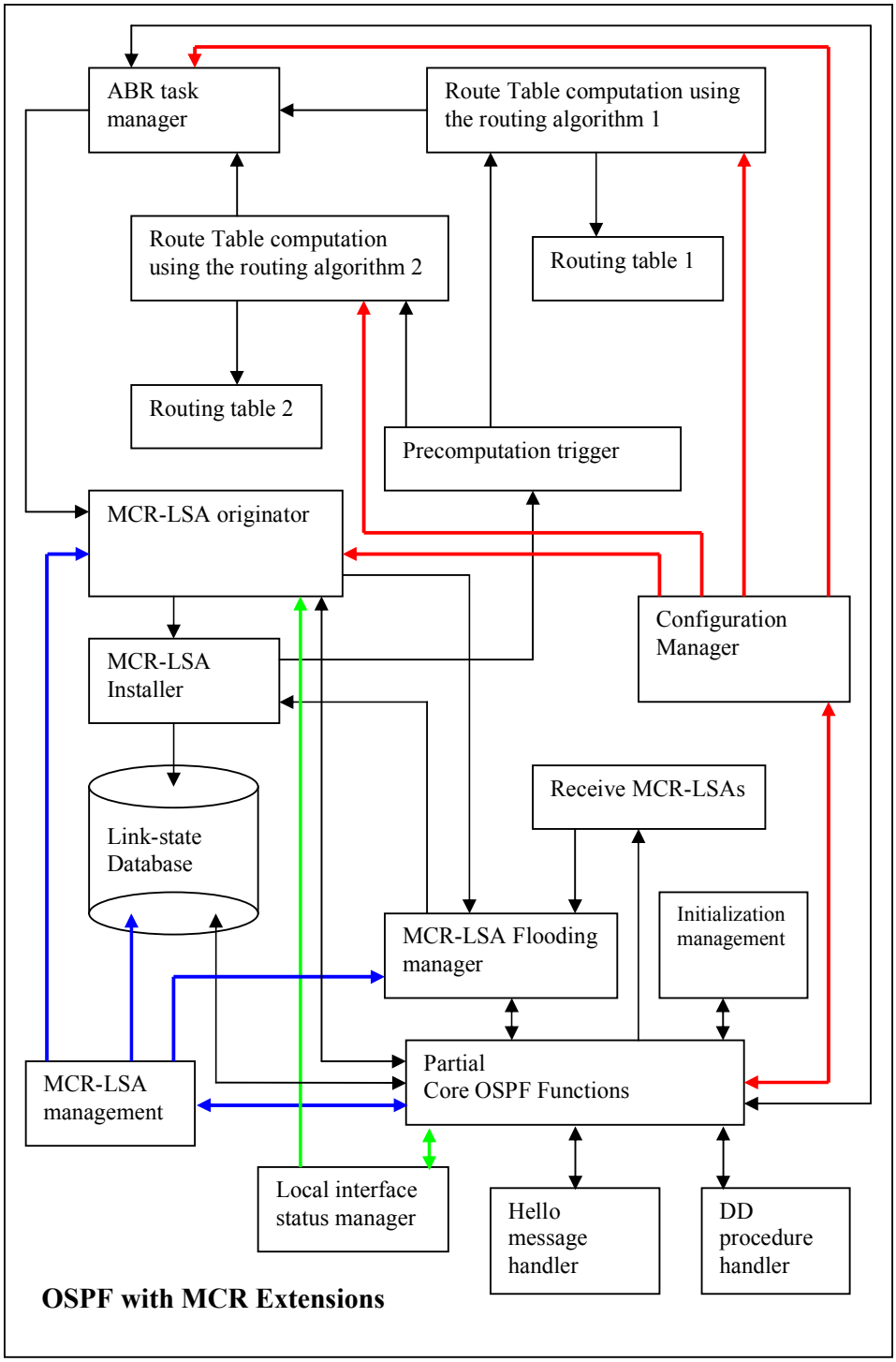

Figure 4.5: The software architecture of the OSPF with MCR extensions

The software architecture is modular so that the MCR extensions to the OSPF protocol are localized to the specific modules. To produce minimal change, the software is

designed to reuse the existing OSPF code. Figure 4.5 illustrates the software architecture of the OSPF with MCR extensions. Different colours, which are used for arrows in the Figure, only make the relationships among different modules in the software architecture visible. The software is divided into the following modules.

Pre-computation trigger module determines which route table computation module should be performed and when it should be done. This module can trigger different route table computations.

Route table computation module creates one independent routing table through performing a specific routing algorithm. This module implements both the whole routing table calculation and the incremental update calculation based on the specific routing algorithm, which is either the SPF algorithm or a QoS routing algorithm. To implement the MCR scheme, more than one route table computation module exists in the software. Only two route table computation modules are shown in Figure 4.5. The software architecture is allowed to add new route table computation modules. One route table computation module is independent of the other route table computation modules. If a router is an ABR, each route table computation module of the router will invoke the ABR task manager module after the corresponding routing table is completely calculated.

ABR task manager module checks which route in the routing table is allowed to create a corresponding Summary-LSA and determines the areas into which the Summary-LSA should be advertised. During the process of checking one routing table, the route summarization is done before the ABR manager module requests the MCR-LSA originator module to originate the Summary-LSAs. Furthermore, this module is responsible for indicating the correct types of the Summary-LSAs for the diverse routing tables because each routing table must have its own types of the Summary-LSAs. For example, type 3 and type 4 Summary-LSAs are used for the first routing table, whereas the new type 12 and type 13 Summary-LSAs are used for the second routing table. In an OSPF router, the ABR task manager module is performed by some core OSPF functions when the router's ABR configuration is changed.

MCR-LSA originator module produces different types of MCR LSAs with the correct Option field. This module cannot completely reuse the existing OSPF code because the MCR LSA formats are different from the standard OSPFv2 LSA formats. On the other hand, it must create the new types of the Summary-LSAs. Based on the local interface status, this module produces the router's Router-LSA and Network-LSA. Under the

condition of the configuration manager module and the core OSPF functions, it can create the AS-external-LSAs if a router is configured as an ASBR. Note that one of the core OSPF functions is the external routing information redistribution. The MCR-LSA originator module requests the MCR-LSA flooding manager module to flood an LSA after the LSA is successfully created. The MCR-LSA originator module replaces the normal LSA originator module in the core OSPF functions.

MCR-LSA installer module installs different types of the MCR LSAs received from other routers or originated by the router itself into the link-state database. The data structure of the link-state database has to be changed because new types of Summary-LSAs are introduced for multi-class routing. If an LSA is a new one or different from the database copy, the module requests the pre-computation trigger module to trigger the corresponding route table computation. The MCR-LSA installer module substitutes the normal LSA installer module in the core OSPF functions.

MCR-LSA flooding manager module floods different LSAs to the correct areas after a router receives LSAs from the other routers or the router itself. It requests the MCR-LSA installer module to update the router's link-state database. The MCR-LSA flooding manager module replaces the normal LSA flooding manager module in the core OSPF functions since the new types of LSAs are introduced into the software.

Configuration manager module provides all management functions for administrators. It can manage and control several modules in the OSPF with MCR extension. Administrators can adjust the frequency of performing a certain route table computation by means of setting different parameters to timers of the route table computation module. The route summarization needs to be configured manually in order to know the ranges of aggregated IP addresses. The configuration manager module extends some functions for the normal configuration module in the core OSPF functions.

Hello message handler module reads the Hello packets received from the neighboring routers and writes the new Hello packets sending to the other routers. Except for the value of the Options field, the format of the OSPF Hello packets is not altered. The M-bit in the Options field must be set during the course of creating every new Hello packet. When a router receives any Hello packet, the router must check the Options field in order to guarantee that the communication is only established with the neighboring routers supporting the MCR capability. This module replaces the original one in the core OSPF functions.

DD message handler module reads the Database Description packets received from the neighboring routers and writes the new Database Description packets sending to the other routers. Except for the value of the Options field, the format of the OSPF Database Description packets is not changed. The M-bit in the Options field must be set during the course of creating every new Database Description packet. When a router receives any Database Description packet, the router must check the Options field in order to guarantee that the communication is only established with the neighboring routers supporting the MCR capability. This module substitutes the original one in the core OSPF functions.

MCR-LSA management module controls and manages all LSAs stored in the link-state database. This module provides several functions, such as LSA refreshment management, self-originated LSA flush management, LSA MaxAge management, database summary function and LSA lookup function, etc. The LSA refreshment management periodically refreshes LSAs stored in the refresh queue of a router by means of requesting the MCR-LSA originator module. Using the MCR-LSA flooding manager module, the self-originated LSA flush management can flush all self-originated LSAs throughout a routing domain. The LSA MaxAge management periodically checks the LS age of every LSA in the link-state database. Any LSA with the MaxAge must be deleted from the link-state database. Database summary function creates a list of LSAs that make up the entire link-state database. The list is sent to a neighbor in the Database Description packets when the neighbor enters Database Exchange state. The LSA lookup function is used to determine whether an LSA exists in the link-state database. Those functions should be involved with all types of the LSAs. Compared to the original LSA management module in the existing OSPF code, functions of the MCR-LSA management module handle new types of Summary-LSAs, such as the type 12 and type 13 Summary-LSAs.

Local interface status manager module controls each local interface of a router and monitors the output cost and the output bandwidth of each interface. This module must simultaneously provide the output bandwidth of the interface used for TOS 72 metric and the output cost used for TOS 0 metric. Some new functions need to be introduced into the OSPF core functions. Different MCR approaches deploy diverse methods to implement the local interface status manager module.

- Static MCR approach: When any weight of an interface is changed manually, the local interface manager module requests the MCR-LSA originator module to create a new Router-LSA.

- Dynamic distributed MCR with Time based triggering: The local interface manager module uses a timer with time *T* to measure weights of interfaces at *T* interval. If any weight changes, the local interface manager module will request the MCR-LSA originator module to create a new Router-LSA.
- Dynamic distributed MCR with significant event triggering: The local interface manager module uses a threshold trigger for each significant event. It deploys two timers, which are *Tmin* timer and *Tmax* timer. When a significant event takes place and *Tmin* is expired, the local interface manager module will request the MCR-LSA originator module to create an updated Router-LSA. If no significant events happen before the *Tmax* timer times out, the measurement of weights is done by the local interface manager module. If any change of weights is found, the local interface manager module will request the MCR-LSA originator module to create a new Router-LSA.

Initialization management module initializes different data structures for the OSPF software. Some original data structures should be changed because of the extensions to the OSPF protocol. Thus, the modified data structures must be correctly initialized before they are used by the different OSPF functions, including the OSPF core functions and new functions.

The link-state database still includes two parts: area database and global database. The area database stores an area's Router-LSAs, Network-LSAs and Summary-LSAs. The Summary-LSAs include original types and the new types of Summary-LSAs. All AS-external-LSAs are stored in the global database.

Major core OSPF functions still remain unchanged in the software architecture. However, in order to implement the MCR extensions to the OSPF protocol, some original modules are extended and some new modules are introduced into the software. Some modules, which are related to the new types of Summary-LSAs, can effectively use the existing OSPF functions as long as the code for handling the new types of the Summary-LSAs is added. Fortunately, the code for handling the new types of Summary-LSAs is similar to that for the type 3 and type 4 Summary-LSAs.

4.3 Discussion

In the first version of OSPF with MCR extensions, we make an assumption for encoding method for the bandwidth information so that if the assumption is changed, the equation (4-1) must be modified too. It is not entirely satisfactory for different MCR approaches, especially dynamic MCR approaches. Thus, two possible encoding methods for the bandwidth information could be introduced in a later version of the OSPF with MCR extensions.

The first encoding method is to use a non-linear code, which represents the bandwidth of a link. For example, the accuracy of one or two percent is used to represent the bandwidth information. In a Router-LSA, the TOS metric field that is only 16 bits long can carry the bandwidth information, the range of which is from 1 kbit/s to Terabits. However, the drawback of this encoding method is that the calculation of the weight function of some routing algorithms, for instance, BSP algorithm, could become heavy on hardware that does not support floating point arithmetic.

The second encoding method is that the network administrator specifies the granularity of the bandwidth coding in the MCR DiffServ network. Different granularities may be used for diverse networks. For example, the granularity used in a 3G access network distinguishes from that in a data network. The following equation (4-2) can be used with a configuration parameter, which is *reference_bandwidth*. Using the corresponding commands, a network administrator can change the parameter *reference_bandwidth* within certain bounds. The limitation of the equation (4-1) can be removed due to the parameter *reference_bandwidth*. However, the parameter *reference_bandwidth* cannot be altered dynamically.

$$TOS\ metric = \frac{bandwidth}{reference\_bandwidth(kbits/s)} \qquad (4\text{-}2)$$

5. Static MCR Implementation Based on Zebra Software

This chapter describes the implementation of the MCR extensions to the OSPF protocol based on the Zebra OSPF software. The goal of our implementation is:

- To implement the basic MCR manager functions.
- To implement the static MCR approach
- To implement the SP algorithm and the BSP algorithm.
- To show two routing tables and other OSPF information by means of the VTY commands.

5.1 Zebra Software Overview

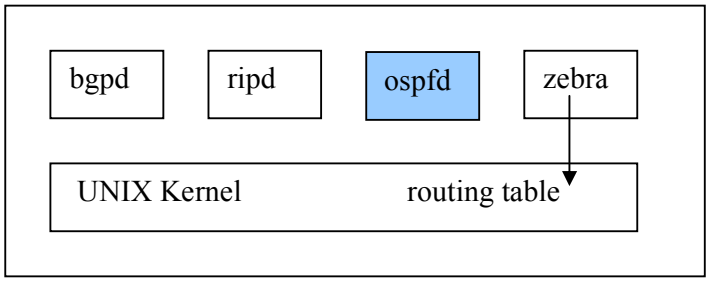

Figure 5.1: Zebra system architecture

Zebra software is a multi-process architecture that has been designed for extensibility. Figure 5.2 shows the Zebra software architecture. Zebra adopts a collection of several separate daemons that work together to form the routing table for UNIX or Linux Kernel. Each specific routing protocol has the corresponding daemon. For example, *ospfd* is the daemon for OSPF version 2. The *zebra* daemon is the kernel routing table manager. It is used to change the entries of the kernel routing table. Generally speaking, we implement the MCR manager through modifying the OSPF code of the Zebra software. For more detailed information about the OSPF code of the Zebra software, please see appendix A.

5.2 Software Architecture Based on the Zebra OSPF

Our OSPF code, which is modified to support the MCR capability, is consistent with the design of the OSPF with MCR extensions. Figure 5.2 shows the zebra OSPF with the MCR extensions. Different colours, which are used for arrows in the Figure, only make the relationships among different modules in the software architecture visible. All modules defined in the design are implemented by means of the corresponding functions in the OSPF code. The SP algorithm has been provided by the original OSPF code. Thus, the BSP algorithm and related functions need to be implemented.

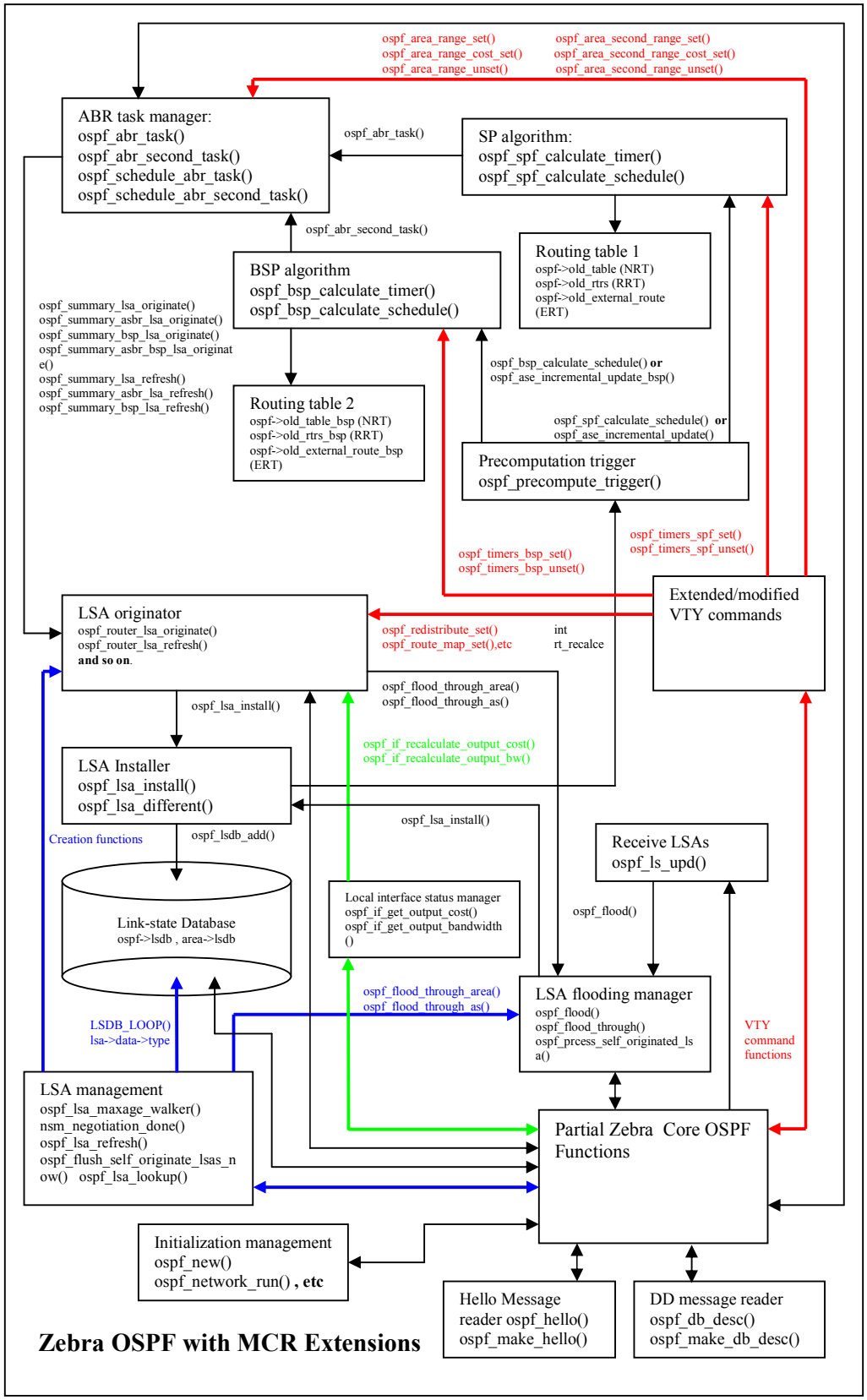

Figure 5.2: The MCR software architecture of Zebra OSPF software

The function ospf_spf_calculate_timer() and ospf_spf_calculate_schedule() are used to implement the functions of the route table computation module. The function ospf_spf_calculate_timer() implements the whole routing table calculation based on the SP algorithm. Having received a new LSA or an LSA with a changed TOS 0 metric, an OSPF router performs the SP algorithm one time. This situation can result in a frequent calculation of the SP algorithm. To avoid the above problem, the function ospf_spf_calculate_schedule() is used to set a timer for the SP algorithm calculation. Administrators can change the values of the timer's parameters through VTY commands. If a router is configured as an ABR, the function ospf_spf_calculate_timer() will invoke the function ospf_abr_task() to check the first routing table and create the Summary-LSAs. Those functions belong to the original OSPF code.

The new function ospf_bsp_calculate_timer() and ospf_bsp_calculate_schedule() are used to implement the functions of the route table computation module. They are responsible for creating the second routing table for the MCR. The function ospf_bsp_calculate_timer () completely implements the whole routing table calculation based on the BSP algorithm. To avoid frequent BSP algorithm calculation, the function ospf_bsp_calculate_schedule() is used to set a timer for the BSP algorithm calculation. Administrators can change the values of the timer's parameters through VTY commands. If a router is configured as an ABR, the function ospf_bsp_calculate_timer() will invoke the function ospf_abr_second_task() to check the second routing table and create the Summary-LSAs.

The ABR task manager module adopts independent functions to handle the different routing tables. The function ospf_abr_task(), which is the original OSPF code, handles the first routing table, whereas the new function ospf_abr_second_task() is used to implement the ABR functions for the second routing table. The route summarization mechanism is implemented in both functions. Administrators can respectively configure the ranges of aggregated IP addresses for each routing table. The function ospf_abr_task() can invoke the functions that create the type 3 and type 4 Summary-LSAs when it determines that the Summary-LSAs should be created. The function ospf_abr_second_task() can invoke the functions that create the type 12 and type 13 Summary-LSAs when it determines that the Summary-LSAs should be created. When a router detects any change of the ABR status, the function ospf_schedule_abr_task() and the function ospf_schedule_abr_second_task() are invoked by certain core OSPF functions, for instance, the function ospf_network_free().

The new function ospf_precompute_trigger() is used as a pre-computation trigger. According to a parameter rt_recalc of this function, it can trigger an entire routing table calculation or an incremental update calculation. When the function ospf_precompute_trigger() invokes the function ospf_bsp_calculate_schedule() or the function ospf_spf_calculate_schedule(), the entire routing table calculation is performed. While the function ospf_ase_incremental_update() or the function ospf_ase_incremental_update_bsp() is invoked, the incremental update calculation is done.

The modified function ospf_lsa_install() implements the functions of the MCR-LSA Installer module. It invokes the original function ospf_lsdb_add() to insert or update LSAs in the link-state database. The modified function ospf_lsa_different() is invoked by the function ospf_lsa_install() in order to determine whether the received LSA is different from the database copy. If the function ospf_lsa_install() finds the received LSA is a new one or different from the database copy, it invokes the function ospf_precompute_trigger() to trigger a certain routing table recalculation in terms of the value of the parameter rt_recalc. When a router receives an LSA from another router or originates an LSA by itself, the router performs the function ospf_lsa_install(). The original function ospf_lsa_install() can be basically used as long as the code for handling the type 12 and type 13 Summary-LSAs is added.

Each type of the LSAs possesses a set of creation functions in the OSPF code. For example, the function ospf_router_lsa_originate() and ospf_router_lsa_refresh() are used for creating Router-LSAs. According to the formats of the MCR LSAs, some functions must be correspondingly modified. The new functions used for the type 12 and type 13 Summary-LSAs must be introduced. More detailed information about the creation functions is in Appendix B. The modified function ospf_lsa_install() is an essential function used by those creation functions because each LSA successfully originated by the router itself must be installed into the router's link-state database. Then, those LSAs will be flooded throughout areas or an entire routing domain by invoking the original function ospf_flood_through_area() or ospf_flood_throug_as().

The original function ospf_flood() can implement the actual flooding procedure for the MCR extensions. If a received LSA does not exist in the database or it is more recent than the database copy, a router performs the function ospf_flood(). Except for flooding the LSA out the correct interfaces of the router, the function ospf_flood() invokes the function ospf_lsa_install() to update the router's link-state database. In the ospf_flood(),

the modified function ospf_flood_through(), which is used to determine the flooding method in terms of the type of the LSAs, adds the code for handling the type 12 and type 13 Summary_LSAs. The modified function ospf_process_self_orginated_lsa() is used to determine how to deal with a self-originated LSA received from the other routers. When receiving a self-originated type 12 or type 13 Summary-LSA from another router, the function ospf_process_self_originated_lsa() used by the function ospf_flood() needs to perform the function ospf_abr_second_task() to the second routing table.

Compared to the original function ospf_make_hello(), the modified function ospf_make_hello() sets the M-bit in the Options field for every Hello packet. When a router receives Hello packets, it performs the modified function ospf_hello() to check the M-bit in the Options field. The router rejects to communicate with other routers that cannot support the MCR capability.

The modified function ospf_make_db_desc() sets the M-bit in the Options field for every Database Description packet. When a router receives Database Description packets, it performs the modified function ospf_db_desc() to check the M-bit in the Options field. The router only communicates with other routers supporting the MCR capability.

The MCR-LSA management module is implemented by several functions. The modified function ospf_lsa_maxage_walker() provides the MCR-LSA MaxAge management. The Database summary is implemented by the modified function nsm_negotiation_done(). The modified function ospf_lsa_refresh() refreshes LSAs based on their types. The self-originated LSA flush management is implemented by the modified function ospf_flush_self_originate_lsas_now(). The modified function ospf_lsa_lookup can determine whether an LSA exists in the database. Compared to the original functions, they add the code for handling the type 12 and type 13 Summary-LSAs.

To implement the local interface status manager module, the original function ospf_if_get_output_cost() obtains the output cost of a link, whereas the new function ospf_if_get_output_bandwidth() acquires the bandwidth of the link. Administrators can configure the output cost and the bandwidth through simple VTY commands. The output cost and the bandwidth are used for the metric and TOS metric of a link. When either output cost or the bandwidth of any link of a router is changed, the original function ospf_if_recalculate_output_cost() or the new function ospf_if_recalculate_output_bw() triggers to create a new Router-LSA for the router.

Some important data structures are modified to support the MCR capability. Functions, which initialize those data structures, should be adequately changed in parallel. For example, the structure ospf adds new pointers, which point to new route_table structures used for the second routing table. The function ospf_new() that initializes an OSPF instance use the function route_table_init() to initialize new route_table structures. All modified data structures and related functions are described in Appendix C.

The Configuration manager module extends some new VTY commands in our OSPF software. Some original VTY commands are modified because certain OSPF functions need new input information. More detailed information about VTY commands is depicted in Appendix D.

In the modified Zebra OSPF code, the new routing table is created for the BSP algorithm. The BSP routing table, which is the same as the SP routing table, consists of three elements: Network Routing Table (NRT), Router Routing Table (RRT) and External Routing Table (ERT). The NRT records the routes, the destination type of which is the "Network". The RRT stores the routes, the destination type of which is the "ABR" or "ASBR". All external routes created by using the external routing information are stored in the ERT.

Major core Zebra OSPF functions remain unchanged in our implementation. To support the MCR capability, some new functions are created. A lot of modifications for the core OSPF functions result from the introduction of the new types of Summary-LSAs. Fortunately, the modifications are not complex because the code for handling the type 12 and type 13 Summary-LSAs is similar to that for type 3 and type 4 Summary-LSAs.

5.3 BSP Algorithm Implementation

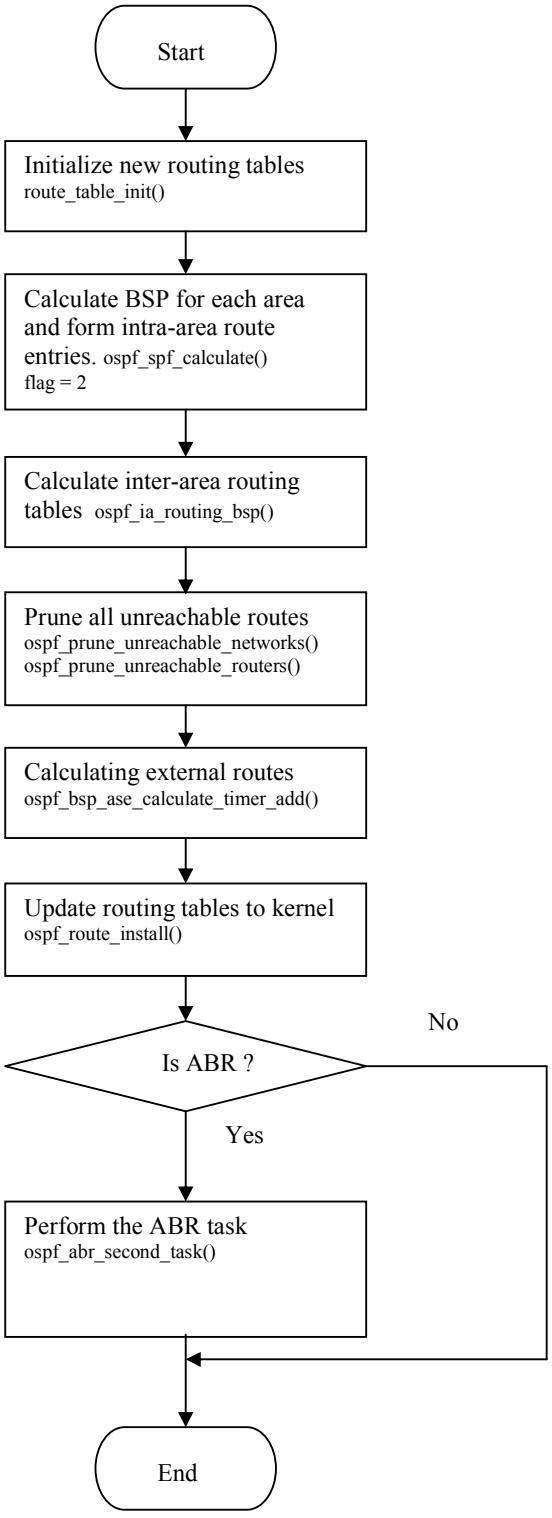

Figure 5.3: Flow chart of the whole BSP routing table calculation

In our OSPF code, the function ospf_bsp_calculate_timer() implements the whole BSP routing table calculation (See Figure 5.3). First, a NRT and a RRT need to be initialised by performing the function route_table_init(). Then, the function ospf_spf_calculate() computes a SPF tree for each area and inserts the intra-area routes into the NRT and RRT. Next, inter-area routes are created into the NRT and RRT by the function ospf_ia_routing_bsp(). All unreachable routes in the NRT are deleted by the function ospf_prune_unreachable_networks(), whereas the unreachable routes in the RRT are removed by the function ospf_prune_unreachable_routers(). After that, ospf_bsp_ase_calculate_timer_add() calculates the external routes and inserts them into the ERT. According to the results of the BSP algorithm calculation, the routing table in the kernel is updated by the function ospf_route_install(). Finally, if a router is configured as an ABR, the function ospf_abr_second_task() is performed in order to handle the ABR tasks.

The following subchapters describe how to implement the intra-area routing calculation, the inter-area routing calculation and the external routing calculation.

5.3.1 BSP Intra-area Routing Calculation

The function ospf_spf_calculate() is responsible for computing a SPF tree for each area. It creates route entries into the routing table in terms of the SPF tree. Note that the function ospf_spf_calculate() is used for both the SP algorithm and BSP algorithm. When a parameter “flag” of the function ospf_spf_calculate() is equal to 2, it is used for the BSP algorithm. Three types of the vertices, including router vertices, transit network vertices and stub network vertices, are used in the routing calculation. The BSP algorithm calculation includes two stages. The first stage only applies the BSP algorithm to the router vertices and the transit network vertices. The second stage handles all stub network vertices.

Figure 5.4 shows the flow chart of the BSP algorithm calculation procedure for intra-area routes:

Step 1 The function ospf_spf_calculate(), the parameter “flag” of which is 2, starts.

Step 2: the function ospf_spf_calculate() initializes algorithm’s data structures, including two SPF trees and a candidate list. The two SPF trees, the data type of which is the structure route_table, are used to record router vertices and network vertices respectively.

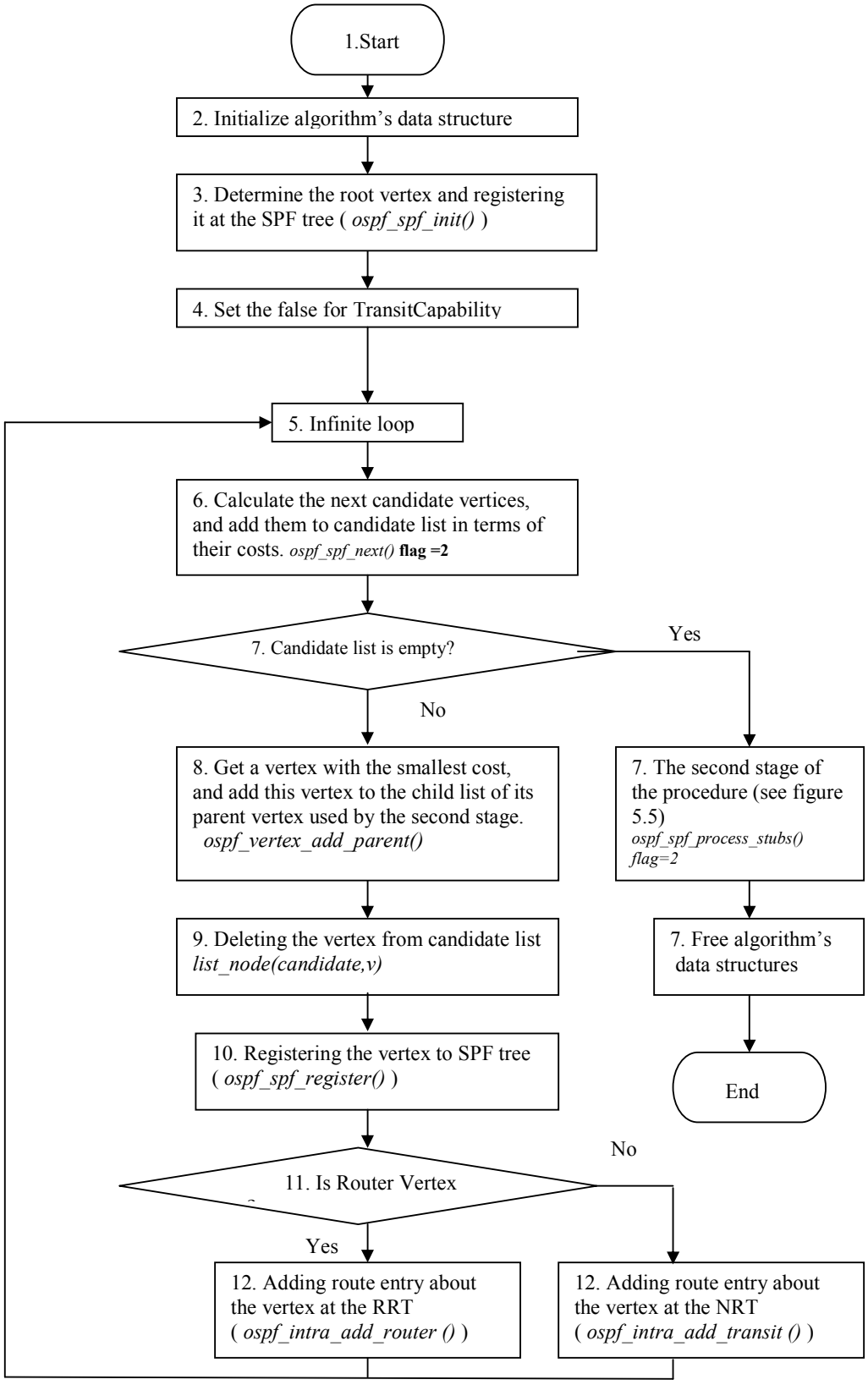

Figure 5.4: Flow chart of BSP algorithm for calculating intra-area routes

Step 3: The root vertex is determined by performing the function ospf_spf_init() and registered into the SPF tree though the function ospf_spf_register().

Step 4: The function ospf_spf_calculate() sets the area's TransitCapability to False. The area's TransitCapbility could be modified during the execution of the function ospf_spf_next(). If the area's Transit Capability is set to True during the first stage of the calculation procedure, it makes the function process_transit_summary_lsa() perform when inter-area routes are calculated by the function ospf_ia_routing_bsp().

Step 5: An infinite loop starts.

Step 6: The function ospf_spf_next() chooses one next candidate vertex with the smallest inversed bandwidth from the candidate list. Then, it uses the selected vertex to determine new candidate vertices. Next, it calculates inversed bandwidth from the root vertex to those candidate vertices and places the candidate vertices into the candidate list based on the value of their inversed bandwidths. The BSP cost calculation is defined in equation (5-1).

$$BSP\ cost = chosen\ vertex's\ cost + \frac{65535}{link'sTOS72metric} \qquad (5\text{-}1)$$

Step 7: The candidate list is checked. If the candidate list is empty, the loop is terminated. Then, the second stage of the calculation procedure begins (see Figure 5.5). After that, all data structures of the BSP algorithm will be free. Lastly, the function ospf_spf_calculate() is terminated.

Step 8: The first vertex is chosen from the candidate list. The function ospf_vertex_add_parent() adds this vertex to a child list of its parent vertex. The child list will be used to calculate routes for the stub network vertices at the second stage.

Step 9: The function listnode_delete() removes the chosen vertex from the candidate list.

Step 10: The chosen vertex is registered into the SPF tree through performing the function ospf_spf_register().

Step 11: the function ospf_spf_calculate() check the type of the chosen vertex.

Step 12: If the vertex is a router vertex, the corresponding route is added or updated to the RRT by performing the function ospf_intra_add_router(). Otherwise, the function ospf_intra_add_transit() inserts the corresponding route into the NRT. The procedure goes back to the Step 5.

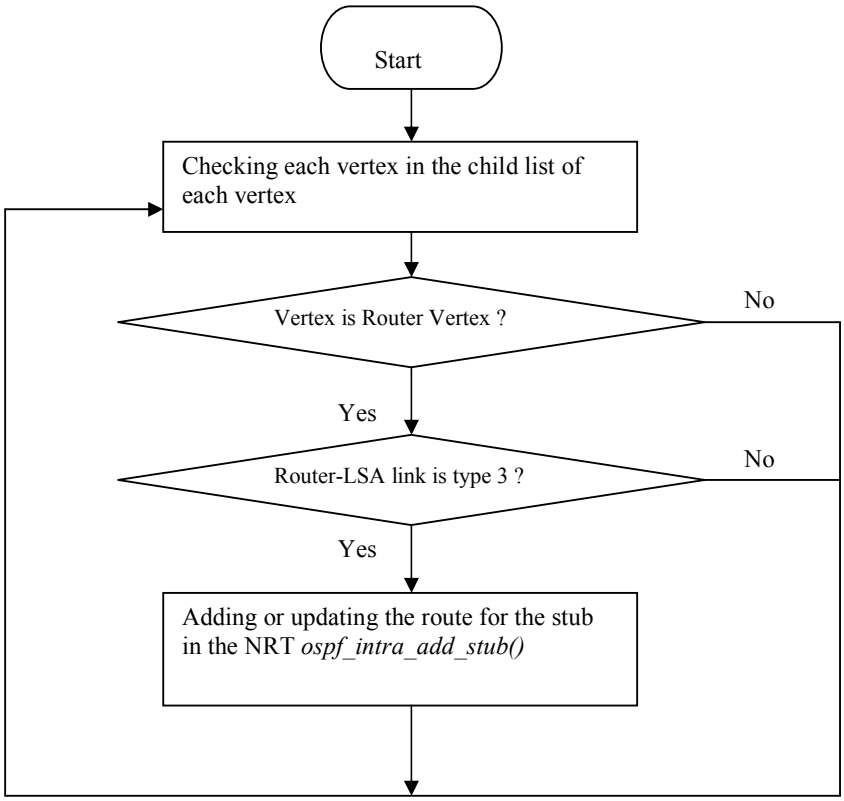

Figure 5.5: Flow chart of the second stage of the BSP calculation procedure

The function ospf_spf_process_stubs() is used to implement the second stage of the calculation. Note that this function is used for both the SP algorithm and BSP algorithm. When the parameter "flag" of this function is equal to 2, it can be used for the BSP algorithm. Figure 5.5 shows the flow chart of the second stage of the calculation. The second stage only calculates routes for the stub network vertices. The function ospf_spf_process_stubs() checks each vertex listed in the child list of every vertex. If the chosen vertex is the router vertex, it checks the vertex's Router-LSA. If the type of the link in the Router-LSA is type 3, the route related to this stub network vertex is created or updated in the NRT. The function ospf_intra_add_stub() is responsible for calculating the cost from the root vertex to the stub network vertex and inserting the route into the NRT. The following equation (5-2) defines how to calculate the BSP cost for the stub network vertex.

$$\textit{Cost of stub network vertex} = \textit{parent vertex's cost} + \frac{65535}{Link'sTOS72metric} \qquad (5\text{-}2)$$

5.3.2 BSP Inter-area Routing Calculation

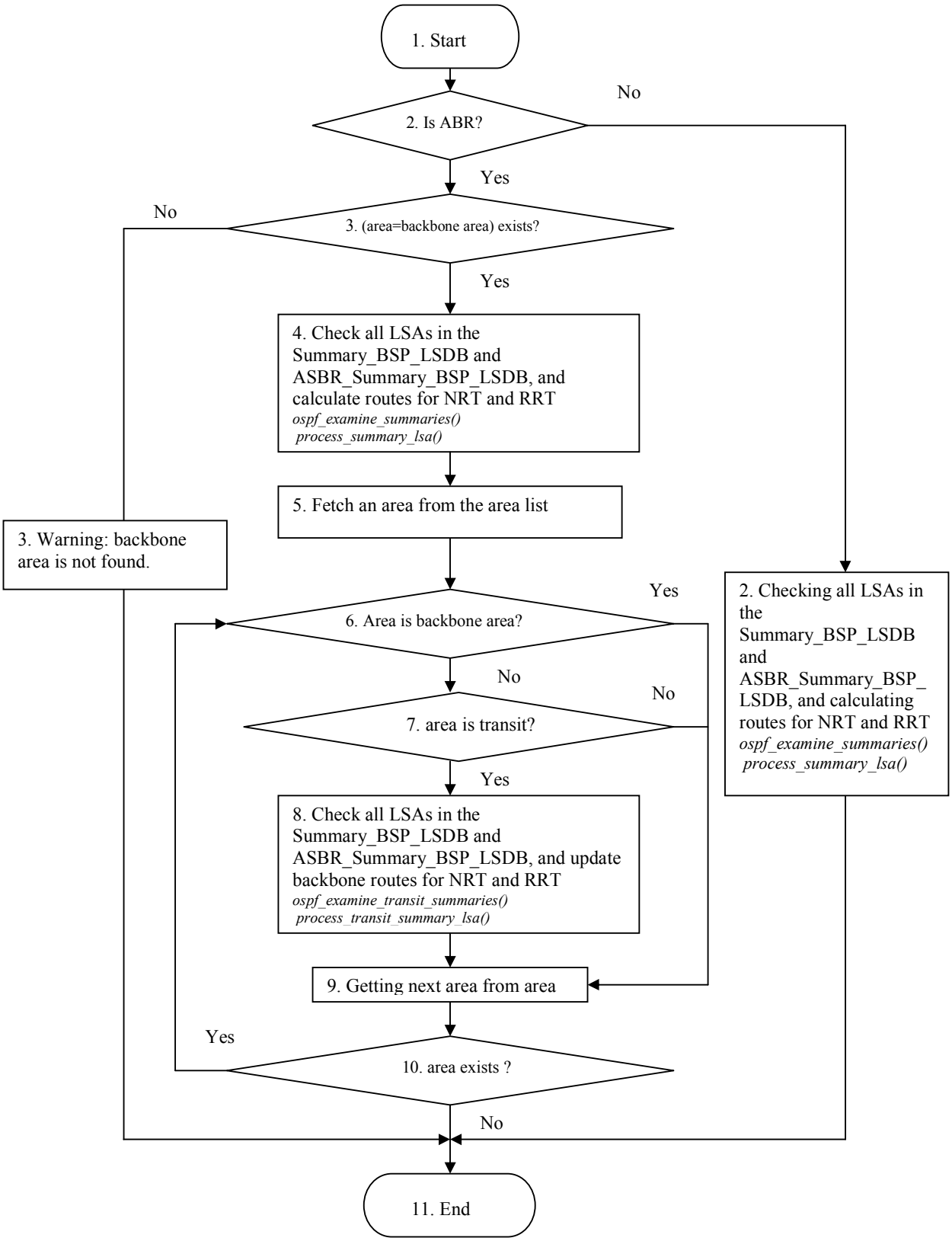

Figure 5.6: Flow chart of calculating BSP inter-area routes

The BSP inter-area routing calculation adopts the distance-vector approach to compute every inter-area route. The cost of each inter-area route is equal to the cost of an ABR route plus the metric of the Summary-LSA created by the ABR. Both the cost of the ABR route and the metric of the Summary-LSAs are directly associated with the inversed bandwidth. Metric of each Summary-LSA is set to the corresponding route's cost stored in the advertising ABR's routing table. The calculating router obtains the cost of a certain ABR router from its' routing table.

The new function ospf_ia_routing_bsp() implements the inter-area routing calculation for the BSP algorithm. Figure 5.6 shows the flow chart of our implementation for the inter-area routing calculation. The following steps are used to explain how the inter-area calculation works.

Step 1: The function ospf_ia_routing_bsp starts.

Step 2: The function ospf_ia_routing_bsp() determines whether a router is an ABR router. If the router is not an ABR router, the function ospf_examine_summaries() examines all type 12 and type 13 Summary-LSAs stored in each area' database. The function process_summary_lsa() computes and inserts inter-area routes into the NRT or RRT. Last, the process terminates.

Step 3: If the router is an ABR, the function ospf_ia_routing_bsp() checks whether the backbone area exists. If the backbone area does not exist, the function terminates with a warning message.

Step 4: Otherwise, the function ospf_examine_summaries() examines all type 12 and type 13 Summary-LSAs related to the intra-area routes of the backbone area. The function process_summary_lsa() computes the inter-area routes for the NRT and RRT.

Step 5: An area is obtained from the area list of the OSPF instance.

Step 6: The function ospf_ia_routing_bsp() needs to determine whether the chosen area is the backbone area. If the area is the backbone area, the process goes to Step 9.

Step 7: the function ospf_ia_routing_bsp() needs to determine whether the chosen area is a transit area. If the area is not a transit area, the process goes to Step 9.

Step 8: Otherwise, the function ospf_examine_transit_summaries() examines all type 12 and type 13 Summary-LSAs. The function process_transit_summary_lsa() updates certain backbone routes in the NRT and RRT. The goal of this step is to eliminate the shortest route problems caused by a virtual link.

Step 9: The next area is fetched from the area list.

Step 10: If the area exists, the process goes to Step 6.

Step 11: Otherwise, the process terminates.

5.3.3 BSP External Routing Calculation

Each AS-external-LSA contains one forwarding address, which is specified by an ASBR. If the forwarding address is 0.0.0.0, packets delivered to another AS will be directly sent to the ASBR. Otherwise, the packets will be sent to the router, the IP address of which is the specific forwarding address in the AS-external-LSA. Thus, every external route is involved with the corresponding ASBR route or the route to specific forwarding address. We assume that X is a cost, which is the shortest cost from the router itself to the ASBR or the forwarding address router. And X is gained from the routing table. Y is a type 1 or 2 metric specified in the AS-external-LSA.

In the AS-external-LSA, bit E defines the type of an external metric. If bit E is zero, the TOS 72 metric field is used as Type 1 external metric. The route's path type is OSPF_PATH_TYPE1_EXTERNAL. The cost of the corresponding external route, which is related to destination N, is defined in the equation (5-3).

$$external\ route's\ cost = X + \frac{16777215}{Y} \tag{5-3}$$

On the other hand, if bit E is set to one, the TOS 72 metric field is used as Type 2 external metric. The cost of the external route is divided into two parts. The link state component of the external route's cost is X, and the type 2 cost of the external route is (16777215/Y). The path type of the route is OSPF_PATH_TYPE2_EXTERNAL. The cost of an external route to destination N is calculated by the function ospf_ase_calculate_new_route().

When an AS-external-LSA is received, it is not necessary to recalculate the entire routing table, meaning that the inter-area route calculation and intra-area route calculation do not need to be performed. Thus, there are two different processes of the external route calculation.

- Entire external routing table calculation: The entire process of calculating all external routes is executed. (See Figure 5.7). This calculation is always used by the whole BSP routing table calculation.
- Incremental update calculation: Only a route related to the AS-external-LSA is recalculated. (See Figure 5.8)

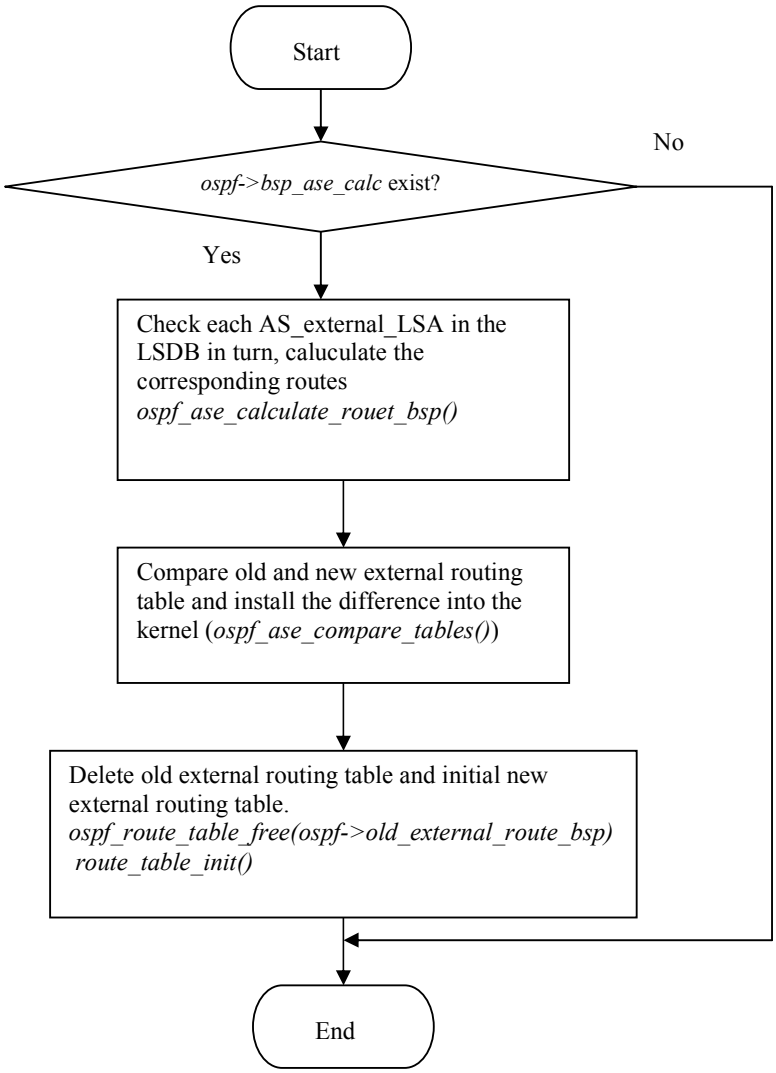

Figure 5.7: Flow chart of the function ospf_bsp_ase_calculate_timer()

The entire external routing table calculation is implemented by the new function ospf_bsp_ase_calculate_timer(). Figure 5.7 shows the flow chart of the function ospf_bsp_ase_calculate_timer(). This function is invoked during the course of performing the function ospf_bsp_ase_calculate_timer_add() (see Figure 5.3). The function ospf_ase_calculate_route_bsp() checks all AS-external-LSAs stored in the global link-state database, calculates the cost of the corresponding external routes and installs them into the ERT of the BSP algorithm. Then, according to each entry of the external routing table, the function ospf_ase_compare_tables() can install or update the related route entries into the kernel. At last, the old ERT is freed by the function ospf_route_table_free(), whereas the new ERT is initialized by the function route_table_init().

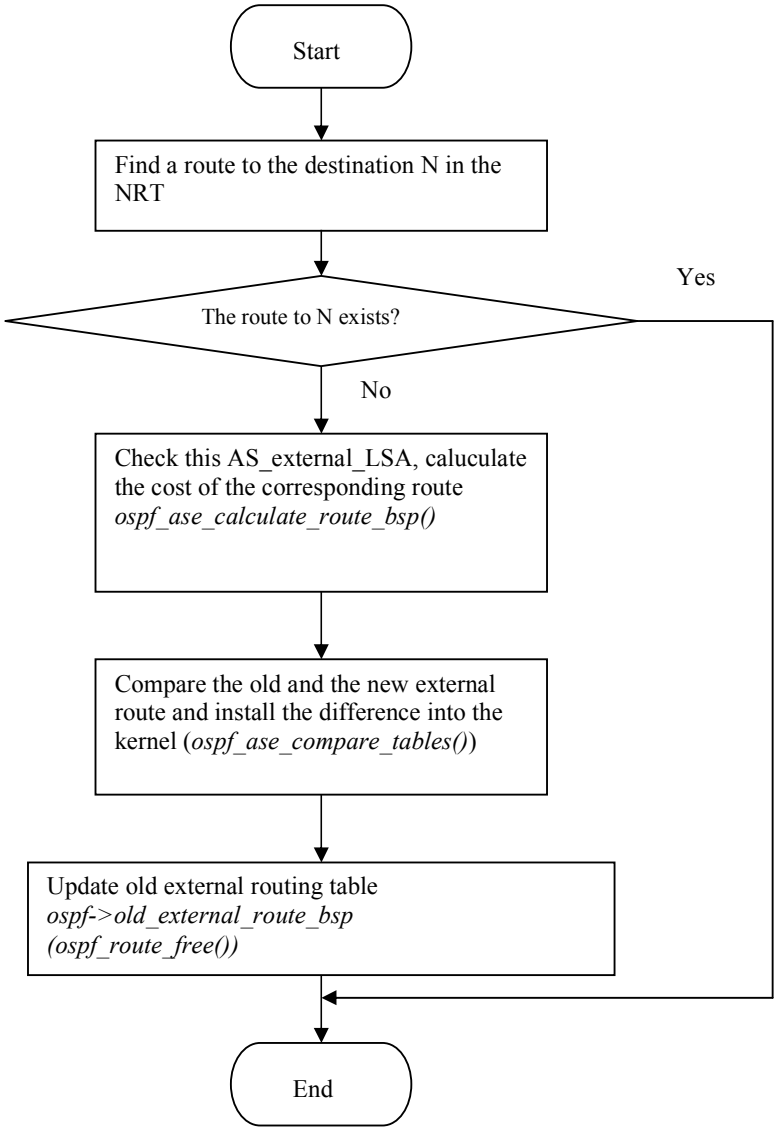

Figure 5.8: Flow chart of the function ospf_ase_incremental_update_bsp()

The incremental update calculation is triggered during the course of installing an AS-external-LSA. The function ospf_external_lsa_install() is responsible for installing AS-external-LSAs into the global link-state database. The incremental update calculation only checks and calculates one AS-external-LSA instead of all AS-external-LSAs stored in the database.

The incremental update calculation is implemented by the new function ospf_ase_incremental_update_bsp() (see figure 5.8). Firstly, the function finds a route to the same destination N in the network routing table. If the route to N exists, the process ends up because an inter-area or intra-area route has priority over an external route. Secondly, based on the received AS-external-LSA, a new external route is created by the

function ospf_ase_calculate_route_bsp(). Thirdly, the new external route is compared with the old one stored in the old external routing table (ospf->old_external_route_bsp). The corresponding route entry in the kernel is updated by the function ospf_ase_compare_tables(). Finally, the old external routing table is updated if the change has happened.

5.4 ABR Task Manager Implementation

Tasks of the ABR consist of performing the route summarization, creating Summary-LSAs and flooding those Summary-LSAs to the correct areas. The new function ospf_abr_second_task() implements the ABR tasks for the second routing table.

Figure 5.9 shows the flow chart of the function ospf_abr_second_task(). The following steps describe how to create Summary-LSAs in terms of the route entries in the routing table NRT and RRT.

Step 1: The function ospf_abr_second_task starts.

Step 2: The function ospf_abr_second_task checks whether both the NRT and the RRT of the BSP algorithm exist. If one of them does not exist, the process terminates.

Step 3: The function ospf_abr_unapproved_summaries_second() sets all self-originated type 12 and type 13 Summary-LSAs to the unapproved state. The goal of this function is to determine whether a Summary-LSA should be operated. If a Summary-LSA is handled during the course of from Step 6 to Step 8, the Summary-LSA will be set to the approved state.

Step 4: The function ospf_abr_prepare_aggregate_second() prepares aggregation for the second area ranges if the second area ranges are configured by the function ospf_area_second_range_set(). The goal of this step is to make preparation for applying the route summarization to the second routing table.

Step 5: The function confirms whether this is an ABR router. If the router is not an ABR router, the process goes to Step 9.

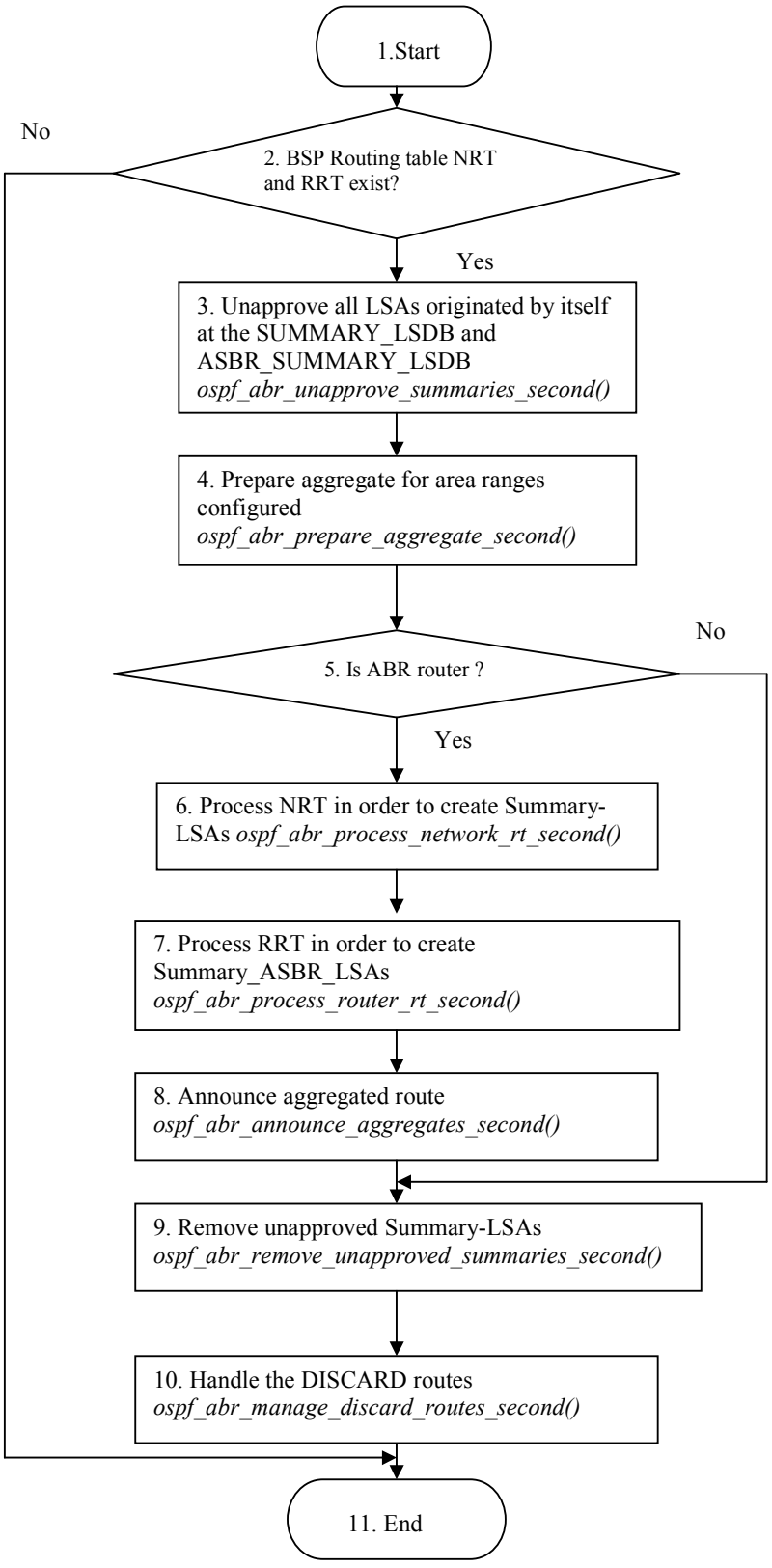

Figure 5.9: Flow chart of the function ospf_abr_second_task()

Step 6: According to each entry in the NRT, the corresponding type 12 Summary-LSAs are correctly created and flooded through areas by the function ospf_abr_process_network_rt_second(). If the destination in the NRT belongs to one of the second area ranges, the route summarization is performed instead of the Summary-LSA creation. The function ospf_abr_update_aggregate_second() implements the route summarization in our OSPF code.

Step 7: According to each entry in the RRT, the corresponding type 13 Summary-LSAs are correctly created and flooded through areas by the function ospf_abr_process_router_rt_second().

Step 8: If the aggregated routes related to the second area ranges exist, the related type 12 Summary-LSAs are created and flooded through the area by the new function ospf_abr_announce_aggregates_second().

Step 9: The function ospf_abr_remove_unapproved_summaries() checks the state of the self-originated type 12 and type 13 Summary_LSAs. If any Summary-LSA stored in the database is in the unapproved state, the Summary-LSA is removed from the database and flushed throughout the areas.

Step 10: The function ospf_abr_manage_discard_routes_second() handles all routes, the type of which is the DISCARD. In the routing table, the DISCARD type routes are used for the aggregate routes. If the route summarization is performed by the router, the corresponding DISCARD routes should be installed into the second routing table.

Step 11: the process ends up.

5.5 MCR-LSA Originator Implementation

The main method to implement the MCR-LSA originator is to modify the original OSPF functions that are used to create different types of the LSAs. The new functions about creating LSAs of type 12 and type 13 can reuse the functions for type 3 and type 4 Summary-LSA as long as the LS type field is set to 12 or 13. Those original functions only provide one TOS 0 metric. The TOS 72 metric needs to be added into the modified functions. Some new members are added into the data structures, which are involved with creating LSAs. For example, Figure 5.10 shows the original structure router_lsa and the modified structure used for the Router-LSAs. To create any Router-LSA, the function link_info_set() must be necessarily used. Figure 5.11 shows the original function link_info_set() and the modified function, which is responsible for setting the link information into the Router-LSA. The modified function link_info_set() can simultaneously set the output cost and bandwidth into the Router-LSA.

```
/* Original structure */
/* OSPF Router-LSAs structure. */
struct router_lsa
{
 struct lsa_header header;
 u_char flags;
 u_char zero;
 u_int16_t links;
 struct
 {
  struct in_addr link_id;
  struct in_addr link_data;
  u_char type;
  u_char tos;          /* set to 1 */
  u_int16_t metric;
 } link[1];
};
```

```
/* Modified Structure */
/* OSPF Router-LSAs structure. */
struct router_lsa
{
 struct lsa_header header;
 u_char flags;
 u_char zero;
 u_int16_t links;
 struct
 {
  struct in_addr link_id;
  struct in_addr link_data;
  u_char type;
  u_char tos;          /* set to 1 */
  u_int16_t metric;
  /* modified by wy -- add the second metric */
  u_char tos1;         /* set to 72 */
  u_char zero1;        /* set to zero */
  u_int16_t tos_metric;  /* the TOS 72 metric */
 } link[1];
};
```

Figure 5.10: The original and the modified structure about Router-LSA

```
/* Original Function*/
/* OSPF Router-LSAs structure. */
/* Set a link information. */
void
link_info_set (struct stream *s, struct in_addr id,
               struct in_addr data, u_char type, u_char tos,
               u_int16_t cost, u_int16_t cost_tos)
{
/* not support TOS */
 stream_put_ipv4 (s, id.s_addr);       /* Link ID. */
 stream_put_ipv4 (s, data.s_addr);     /* Link Data. */
 stream_putc (s, type);                /* Link Type. */
 stream_putc (s, (u_char) 1);          /* TOS = 0. */
 stream_putw (s, cost);                /* Link Cost. */
}
```

```
/* Modified Function */
/* Set a link information. */
void
link_info_set (struct stream *s, struct in_addr id,
               struct in_addr data, u_char type, u_char tos,
               u_int16_t cost, u_int16_t cost_tos)
{

 stream_put_ipv4 (s, id.s_addr);       /* Link ID. */
 stream_put_ipv4 (s, data.s_addr);     /* Link Data. */
 stream_putc (s, type);                /* Link Type. */
 stream_putc (s, (u_char) 1);          /* TOS = 1. */
 stream_putw (s, cost);                /* Link Cost. */
 stream_putc (s, (u_char) 72);     /* value of TOS 72
is bandwidth */
 stream_putc (s, (u_char) 0);          /* zero field */
 stream_putw (s, cost_tos);    /* Link TOS  72Cost. */
}
```

Figure 5.11: The original and the modified function link_info_set()

5.6 MCR-LSA Installer Implementation

The modified function ospf_lsa_install() implements the MCR-LSA installer. Compared to the original function, this modified function only adds some code used for handling type 12 and type 13 Summary-LSAs. The new function ospf_summary_bsp_lsa_install() handles the type 12 Summary-LSA installation, whereas the new function ospf_summary_asbr_bsp_lsa_install() handles the type 13 Summary-LSA installation. Figure 5.12 shows the original function and the modified function ospf_lsa_install().

```
/* Original Function*/
struct ospf_lsa *
ospf_lsa_install (struct ospf *ospf, struct ospf_interface
*oi, struct ospf_lsa *lsa)
{….
/* Do comparision and record if recalc needed. */
  rt_recalc = 0;
  if (  old == NULL || ospf_lsa_different(old, lsa))
    rt_recalc = 1;
…
/* Do LSA specific installation process. */
  switch (lsa->data->type)
    {
    case OSPF_ROUTER_LSA:
      new = ospf_router_lsa_install (ospf, lsa, rt_recalc);
      break;
    case OSPF_NETWORK_LSA:
      assert (oi);
      new = ospf_network_lsa_install (ospf, oi, lsa,
rt_recalc);
      break;
    case OSPF_SUMMARY_LSA:
      new = ospf_summary_lsa_install (ospf, lsa,
rt_recalc);
      break;
    case OSPF_ASBR_SUMMARY_LSA:
      new = ospf_summary_asbr_lsa_install (ospf, lsa,
rt_recalc);
      break;
    case OSPF_AS_EXTERNAL_LSA:
      new = ospf_external_lsa_install (ospf, lsa, rt_recalc);
      break;
    default: /* NSSA, or type-6,8,9....nothing special */
      break;
    }

….

}
```

```
/* Modified Function */
struct ospf_lsa *
ospf_lsa_install (struct ospf *ospf, struct ospf_interface
*oi, struct ospf_lsa *lsa)
{….
/* Do comparision and record if recalc needed. */
  rt_recalc = 0;
  if (  old == NULL)
    rt_recalc = 1;
  else
    rt_recalc = ospf_lsa_different(old, lsa);
…
/* Do LSA specific installation process. */
  switch (lsa->data->type)
    {
    case OSPF_ROUTER_LSA:
      new = ospf_router_lsa_install (ospf, lsa, rt_recalc);
      break;
    case OSPF_NETWORK_LSA:
      assert (oi);
      new = ospf_network_lsa_install (ospf, oi, lsa,
rt_recalc);
      break;
    case OSPF_SUMMARY_LSA:
      new = ospf_summary_lsa_install (ospf, lsa,
rt_recalc);
      break;
    case OSPF_ASBR_SUMMARY_LSA:
      new = ospf_summary_asbr_lsa_install (ospf, lsa,
rt_recalc);
      break;
    case OSPF_SUMMARY_BSP_LSA:
      new = ospf_summary_bsp_lsa_install (ospf, lsa,
rt_recalc);
      break;
      /* modified by wy */
    case OSPF_ASBR_SUMMARY_BSP_LSA:
      new = ospf_summary_asbr_bsp_lsa_install (ospf,
lsa, rt_recalc);
      break;
    case OSPF_AS_EXTERNAL_LSA:
      new = ospf_external_lsa_install (ospf, lsa, rt_recalc);
      break;
    default: /* NSSA, or type-6,8,9....nothing special */
      break;
    }
….
}
```

Figure 5.12: The original and the modified function ospf_lsa_install()

In addition, all specific installation functions, such as ospf_router_lsa_insall() and ospf_network_lsa_install(), add the new function ospf_precompute_trigger that trigger a certain route table calculation based on the value of the parameter rt_recalc. If the function ospf_lsa_install() determines that a received LSA is a new one, the rt_recalc is directly equal to 1, meaning that the entire routing table calculations need to be performed. Otherwise, the rt_recalc is equal to the return value of the function ospf_lsa_different(), which is used to determine whether the LSA is different from the database copy.

5.7 Pre-computation Trigger Implementation

The new function ospf_precompute_trigger() is used to implement the pre-computation trigger module. The function ospf_precompute_trigger() is shown in Figure 5.13. If the parameter rt_recalc is equal to 0, any algorithm calculation does not happen. The function ospf_bsp_calculate_schedule() is invoked if the parameter rt_recalc is 1 or 3. The function ospf_spf_calculate_schedule() is performed if the parameter rt_recalc is 1 or 2., The function ospf_ase_incremental_update() is invoked if the parameter rt_recalc is 4 or 5. If the parameter rt_recalc is 4 or 6, the function ospf_ase_incremetal_update_bsp is performed.

```
/* New Function*/
/* Precompute_trigger. */
void
ospf_precompute_trigger (struct ospf *ospf,
                struct ospf_lsa *new, int rt_recalc)
{
 switch(rt_recalc)
   {
   case 0:
    break;
   case 1:
    ospf_spf_calculate_schedule (ospf);
    ospf_bsp_calculate_schedule (ospf);
    break;
   case 2:
    ospf_spf_calculate_schedule (ospf);
    break;
   case 3:
    ospf_bsp_calculate_schedule (ospf);
    break;
   case 4:
    ospf_ase_incremental_update (ospf, new);
    ospf_ase_incremental_update_bsp (ospf, new);
    break;
   case 5:
    ospf_ase_incremental_update (ospf,new);
    break;
   case 6:
    ospf_ase_incremental_update_bsp (ospf,new);
    break;
   default:
    break;
   }
}
```

Figure 5.13: The new function ospf_precompute_trigger()

6. Testing

Results of four tests, which are shown through VTY commands, are described in this chapter. Based on the results, we discuss the feasibility and issues of our OSPF with MCR extensions.

6.1 Basic Function Test

Goals of the basic function test are

- To succeed in creating two independent routing tables.
- To correctly produce all types of MCR-LSAs
- To demonstrate the correctness of the BSP algorithm calculation.

Configuration

Our test network topology is shown in Figure 6.1. The information about configuration, such as output cost and bandwidth, etc, is indicated in the network topology. The operating system of each computer is Linux. Computer Debian, which is configured as an ASBR, redistributes its default route to the test environment. Computer PC59 is configured as a standard ABR. Computer Debian1 is a normal OSPF router.

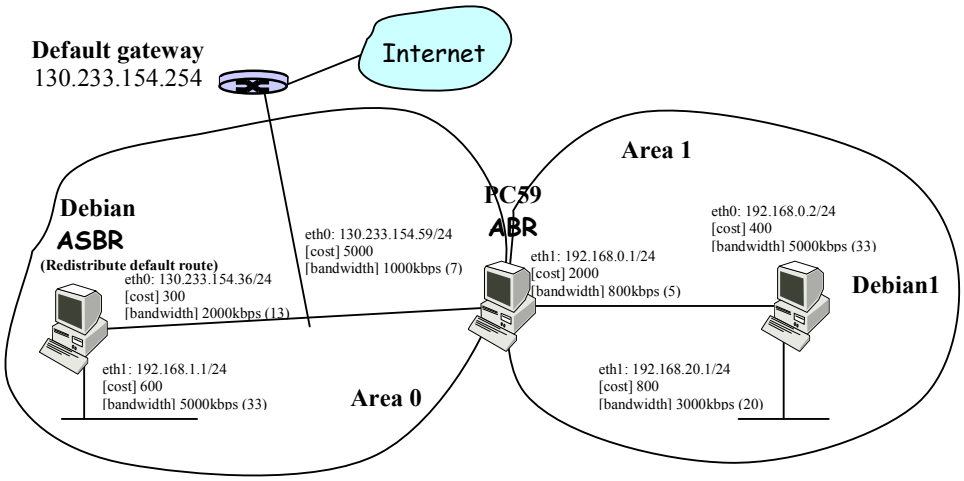

Figure 6.1: A basic function testing environment

Results

1) Routing tables

- PC59' routing table:

SP routing table

```
PC59_ospfd# show ip ospf route
============ OSPF network routing table ============
N    130.233.154.0/24      [5000] area: 0.0.0.0
```

```
                            directly attached to eth0
N    192.168.0.0/24         [2000] area: 0.0.0.1
                            directly attached to eth1
N    192.168.1.0/24         [5600] area: 0.0.0.0
                            via 130.233.154.36, eth0
N    192.168.20.0/24        [2800] area: 0.0.0.1
                            via 192.168.0.2, eth1

============ OSPF router routing table =============
R    192.168.1.1            [5000] area: 0.0.0.0, ASBR
                            via 130.233.154.36, eth0

============ OSPF external routing table ===========
N E2 0.0.0.0/0              [5000/300] tag: 0
                            via 130.233.154.254, eth0

============ OSPF BSP network routing table ============
N    130.233.154.0/24       [9362] area: 0.0.0.0
                            directly attached to eth0
N    192.168.0.0/24         [13107] area: 0.0.0.1
                            directly attached to eth1
N    192.168.1.0/24         [11348] area: 0.0.0.0
                            via 130.233.154.36, eth0
N    192.168.20.0/24        [16384] area: 0.0.0.1
                            via 192.168.0.2, eth1

============ OSPF BSP router routing table =============
R    192.168.1.1            [9362] area: 0.0.0.0, ASBR
                            via 130.233.154.36, eth0

============ OSPF BSP external routing table ===========
N E2 0.0.0.0/0              [9362/168] tag: 0
                            via 130.233.154.254, eth0
```

BSP routing table

Inter-area routes do not exist in the PC59's SP routing table and BSP routing table because the PC59 is a unique ABR in the test environment.

- Debian's routing table

```
debian_ospfd# show ip ospf route
============ OSPF network routing table ============
N    130.233.154.0/24       [300] area: 0.0.0.0
                            directly attached to eth0
N IA 192.168.0.0/24         [2300] area: 0.0.0.0
                            via 130.233.154.59, eth0
N    192.168.1.0/24         [600] area: 0.0.0.0
                            directly attached to eth1
N IA 192.168.20.0/24        [3100] area: 0.0.0.0
                            via 130.233.154.59, eth0

============ OSPF router routing table =============
R    192.168.0.1            [300] area: 0.0.0.0, ABR
                            via 130.233.154.59, eth0

============ OSPF external routing table ===========

============ OSPF BSP network routing table ============
N    130.233.154.0/24       [5041] area: 0.0.0.0
                            directly attached to eth0
N IA 192.168.0.0/24         [18148] area: 0.0.0.0
                            via 130.233.154.59, eth0
N    192.168.1.0/24         [1986] area: 0.0.0.0
                            directly attached to eth1
N IA 192.168.20.0/24        [21425] area: 0.0.0.0
                            via 130.233.154.59, eth0

============ OSPF BSP router routing table =============
R    192.168.0.1            [5041] area: 0.0.0.0, ABR
                            via 130.233.154.59, eth0

============ OSPF BSP external routing table ===========
debian_ospfd#
```

SP routing table

BSP routing table

In the Debian's BSP routing table and SP routing table, there are no external routes because the external routing information is redistributed by the Debian itself.

- Debian1's routing table

SP routing table

```
debian1_ospfd# show ip ospf route
============ OSPF network routing table ============
N IA 130.233.154.0/24      [5400] area: 0.0.0.1
                           via 192.168.0.1, eth0
N    192.168.0.0/24        [400] area: 0.0.0.1
                           directly attached to eth0
N IA 192.168.1.0/24        [6000] area: 0.0.0.1
                           via 192.168.0.1, eth0
N    192.168.20.0/24       [800] area: 0.0.0.1
                           directly attached to eth1

============ OSPF router routing table =============
R    192.168.0.1           [400] area: 0.0.0.1, ABR
                           via 192.168.0.1, eth0
R    192.168.1.1        IA [5400] area: 0.0.0.1, ASBR
                           via 192.168.0.1, eth0

============ OSPF external routing table ===========
N E2 0.0.0.0/0             [5400/300] tag: 0
                           via 192.168.0.1, eth0

============ OSPF BSP network routing table ============
N IA 130.233.154.0/24      [11348] area: 0.0.0.1
                           via 192.168.0.1, eth0
N    192.168.0.0/24        [1986] area: 0.0.0.1
                           directly attached to eth0
N IA 192.168.1.0/24        [13334] area: 0.0.0.1
                           via 192.168.0.1, eth0
N    192.168.20.0/24       [3277] area: 0.0.0.1
                           directly attached to eth1

============ OSPF BSP router routing table =============
R    192.168.0.1           [1986] area: 0.0.0.1, ABR
                           via 192.168.0.1, eth0
R    192.168.1.1        IA [11348] area: 0.0.0.1, ASBR
                           via 192.168.0.1, eth0

============ OSPF BSP external routing table ===========
N E2 0.0.0.0/0             [11348/168] tag: 0
                           via 192.168.0.1, eth0
```

BSP routing table

2) PC59's general OSPF information

```
PC59_ospfd# show ip ospf
 OSPF Routing Process, Router ID: 192.168.0.1
 Supports only TOS 0 and TOS 72 routes
 This implementation conforms to RFC2328
 RFC1583Compatibility flag is disabled
 SPF schedule delay 5 secs, Hold time between two SPFs 10 secs
 BSP schedule delay 5 secs, Hold time between two BSPs 10 secs
 Refresh timer 10 secs
 This router is an ABR, ABR type is: Standard (RFC2328)
 Number of external LSA 1
 Number of areas attached to this router: 2

 Area ID: 0.0.0.0 (Backbone)
   Number of interfaces in this area: Total: 1, Active: 2
   Number of fully adjacent neighbors in this area: 1
   Area has no authentication
   SPF algorithm executed 7 times
   BSP algorithm executed 7 times
   Number of LSA 7

 Area ID: 0.0.0.1
   Shortcutting mode: Default, S-bit consensus: no
   Number of interfaces in this area: Total: 1, Active: 2
   Number of fully adjacent neighbors in this area: 1
   Area has no authentication
   Number of full virtual adjacencies going through this area: 0
   SPF algorithm executed 7 times
```

```
BSP algorithm executed 7 times
Number of LSA 9
```

The PC59 whose router ID is 192.168.0.1 is configured as the standard ABR. It attaches the area 0 and area 1. The area 0, which is the backbone area, has 7 LSAs, whereas the area 1 has 9 LSAs. The PC59 stores one AS-external-LSA. The SP algorithm and the BSP algorithm have independent parameters of their calculation timers. In this test, both the SP algorithm and BSP algorithm are executed 7 times.

3) PC59's Link-state database

```
PC59_ospfd# show ip ospf database

       OSPF Router with ID (192.168.0.1)

                Router Link States (Area 0.0.0.0)

Link ID         ADV Router      Age  Seq#       CkSum  Link count
192.168.0.1     192.168.0.1     1109 0x80000003 0x9de9 1
192.168.1.1     192.168.1.1      997 0x80000004 0xc4fa 2

                Net Link States (Area 0.0.0.0)

Link ID         ADV Router      Age  Seq#       CkSum
130.233.154.59  192.168.0.1     1109 0x80000001 0x3720

                Summary Link States (TYPE 3) (Area 0.0.0.0)

Link ID         ADV Router      Age  Seq#       CkSum  Route
192.168.0.0     192.168.0.1      980 0x80000002 0x85aa 192.168.0.0/24
192.168.20.0    192.168.0.1     1451 0x80000002 0x05f3 192.168.20.0/24

                Second Summary Link States (TYPE 12) (Area 0.0.0.0)

Link ID         ADV Router      Age  Seq#       CkSum  Route
192.168.0.0     192.168.0.1      780 0x80000002 0x6c2c 192.168.0.0/24
192.168.20.0    192.168.0.1     1551 0x80000002 0x05a5 192.168.20.0/24

                Router Link States (Area 0.0.0.1)

Link ID         ADV Router      Age  Seq#       CkSum  Link count
192.168.0.1     192.168.0.1      784 0x80000007 0x3abe 1
192.168.20.1    192.168.20.1     751 0x80000006 0x9c4d 2

                Net Link States (Area 0.0.0.1)

Link ID         ADV Router      Age  Seq#       CkSum
192.168.0.1     192.168.0.1      788 0x80000002 0x0219

                Summary Link States (TYPE 3) (Area 0.0.0.1)

Link ID         ADV Router      Age  Seq#       CkSum  Route
130.233.154.0   192.168.0.1     1160 0x80000002 0x943a 130.233.154.0/24
192.168.1.0     192.168.0.1     1031 0x80000001 0x9b76 192.168.1.0/24

                ASBR-Summary Link States (TYPE 4) (Area 0.0.0.1)

Link ID         ADV Router      Age  Seq#       CkSum
192.168.1.1     192.168.0.1     1104 0x80000001 0xfd6c

                Second Summary Link States (TYPE 12) (Area 0.0.0.1)

Link ID         ADV Router      Age  Seq#       CkSum  Route
130.233.154.0   192.168.0.1      790 0x80000002 0x1496 130.233.154.0/24
192.168.1.0     192.168.0.1     1031 0x80000001 0x6f0f 192.168.1.0/24

                Second ASBR-Summary Link States (TYPE 13) (Area 0.0.0.1)

Link ID         ADV Router      Age  Seq#       CkSum
192.168.1.1     192.168.0.1     1104 0x80000001 0x7dc8
```

```
                AS External Link States (Global database)

Link ID         ADV Router      Age  Seq#       CkSum  Route
0.0.0.0         192.168.1.1     1111 0x80000001 0xf3bc E2 0.0.0.0/0 [0x0]
```

In the PC59, the link-state database can be divided into 3 parts: area 0 database, area 1 database and global database. The area 0 database stores 7 LSAs, and the area 1 database stores 9 LSAs. Only one AS-external-LSA is inserted into the global database. Based on its SP routing table and BSP routing table, we can obviously note that the PC59 correctly creates different types of Summary-LSAs.

4) Detailed link state advertisements

- Router LSA

```
 LS age: 1016
Options: 130
Flags: 0x2 : ASBR
LS Type: router-LSA
Link State ID: 192.168.1.1
Advertising Router: 192.168.1.1
LS Seq Number: 80000004
Checksum: 0xc4fa
Length: 56
 Number of Links: 2

  Link connected to: a Transit Network
   (Link ID) Designated Router address: 130.233.154.59
   (Link Data) Router Interface address: 130.233.154.36
    Number of TOS metrics: 1
     TOS 0 Metric: 300
     TOS [72] Metric: 13

  Link connected to: Stub Network
   (Link ID) Net: 192.168.1.0
   (Link Data) Network Mask: 255.255.255.0
    Number of TOS metrics: 1
     TOS 0 Metric: 600
     TOS [72] Metric: 33
```

This Router-LSA is originated by the Debian, which is the ASBR. The Debian possesses two links in the area 0. One link is connected to a transit network 130.233.154.0/24 whose DR IP address is 130.233.154.59. Another is directly attached to the stub network 192.168.1.0/24. Each link carries one TOS 0 metric and one TOS 72 metric.

- Type 12 Summary-LSA

```
LS age: 888
 Options: 130
 LS Type: summary-BSP-LSA
 Link State ID: 192.168.0.0 (summary Network Number)
 Advertising Router: 192.168.0.1
 LS Seq Number: 80000002
 Checksum: 0x6c2c
 Length: 28
 Network Mask: /24
       TOS: 0  Metric: 13107
```

The above type 12 Summary-LSA originated by the PC59 only carries one metric because the BSP algorithm computes one cost for each route.

- Type 13 Summary-LSA

```
 LS age: 670
Options: 130
LS Type: summary-ASBR-BSP-LSA
Link State ID: 192.168.1.1 (AS Boundary Router address)
Advertising Router: 192.168.0.1
LS Seq Number: 80000002
Checksum: 0x7bc9
Length: 28
Network Mask: /32
      TOS: 0  Metric: 9362
```

The above type 13 Summary-LSA originated by the PC59 only carries one metric for the ASBR. The ASBR's IP address is 192.168.1.1 that is the router ID of the Debian.

- AS-external-LSA

```
 LS age: 1299
Options: 130
LS Type: AS-external-LSA
Link State ID: 0.0.0.0 (External Network Number)
Advertising Router: 192.168.1.1
LS Seq Number: 80000001
Checksum: 0xf3bc
Length: 48
Network Mask: /0
      Metric Type: 2 (Larger than any link state path)
      TOS: 1
      Metric 0 : 300
      Metric [72] : 100000
      Forward Address: 130.233.154.254
      External Route Tag: 0
```

The AS-external-LSA originated by the Debian describes the information about a default route 0.0.0.0/0. This LSA provides the PC59 and the Debian1 with a default gateway, the IP address of which is 130.233.154.254. Both the metric and the TOS 72 metric are present in this AS-external-LSA.

5) Information of the PC59's neighboring routers

```
PC59_ospfd# show ip ospf neighbor detail
Neighbor 192.168.1.1, interface address 130.233.154.36
   In the area 0.0.0.0 via interface eth0
   Neighbor priority is 1, State is Full, 5 state changes
   DR is 130.233.154.59, BDR is 130.233.154.36
   Options 130 |M|-|-|-|-|-|E|*|
   Dead timer due in 00:00:38
   Database Summary List 0
   Link State Request List 0
   Link State Retransmission List 0
   Thread Inactivity Timer on
   Thread Database Description Retransmision off
   Thread Link State Request Retransmission on
   Thread Link State Update Retransmission on

Neighbor 192.168.20.1, interface address 192.168.0.2
   In the area 0.0.0.1 via interface eth1
   Neighbor priority is 1, State is Full, 10 state changes
   DR is 192.168.0.1, BDR is 192.168.0.2
```

```
    Options 130 |M|-|-|-|-|-|E|*|
    Dead timer due in 00:00:30
    Database Summary List 0
    Link State Request List 0
    Link State Retransmission List 0
    Thread Inactivity Timer on
    Thread Database Description Retransmision off
    Thread Link State Request Retransmission off
    Thread Link State Update Retransmission on

PC59_ospfd#
```

From the above information, the PC59 establishes two adjacencies with two neighboring routers. The neighboring routers are the same as the PC59 that supports the MCR capability because their Options field is 130.

Conclusions

Our *ospfd* daemon, which supports the MCR capability, can work normally. It can use the BSP algorithm to successfully create the second routing table. All types of LSAs, including Summary-LSAs and AS-external-LSAs, can be correctly created by the *ospfd* daemon. The BSP algorithm can rightly compute routes on the basis of LSAs stored in the link-state database. All information can be checked by means of the corresponding VTY commands.

6.2 Pre-computation Trigger Test

Our aims of the pre-computation trigger test are:

- To demonstrate that each routing algorithm can be independently executed.
- To correctly trigger either the entire routing table calculation or the incremental update calculation.

Configuration

The pre-computation trigger test still adopts the above network topology, which is shown in Figure 6.1. Configuration files are not changed at all.

Test Steps

1) Each router starts to run the *ospfd* daemon. Routing tables are completely created on each router.

2) The PC59's general OSPF information and routing tables are recorded into a log file.

3) The Debian changes the value of its eth1' output cost though the following VTY commands.

```
debian_ospfd(config)# interface eth1
debian_ospfd(config-if)# ospf cost 100
debian_ospfd(config-if)# exit
debian_ospfd(config)# exit
```

4) On the PC59, the general OSPF information and routing table are checked again and recorded into the log file.

5) The Debian sets a changed value of the metric-bandwidth. VTY commands are shown as follows:

```
debian_ospfd(config)# router ospf
debian_ospfd(config-router)# default-information originate metric 300 metric-
bandwidth 10000 metric-type 2
debian_ospfd(config-router)# exit
debian_ospfd(config)# exit
```

6) On the PC59, the related information is recorded into the log file.

Results

1) Step 2's OSPF Information

```
PC59_ospfd# show ip ospf route
============ OSPF network routing table ============
N    130.233.154.0/24      [5000] area: 0.0.0.0
                           directly attached to eth0
N    192.168.0.0/24        [2000] area: 0.0.0.1
                           directly attached to eth1
N    192.168.1.0/24        [5600] area: 0.0.0.0
                           via 130.233.154.36, eth0
N    192.168.20.0/24       [2800] area: 0.0.0.1
                           via 192.168.0.2, eth1

============ OSPF router routing table =============
R    192.168.1.1           [5000] area: 0.0.0.0, ASBR
                           via 130.233.154.36, eth0

============ OSPF external routing table ===========
N E2 0.0.0.0/0             [5000/300] tag: 0
                           via 130.233.154.254, eth0

============ OSPF BSP network routing table ============
N    130.233.154.0/24      [9362] area: 0.0.0.0
                           directly attached to eth0
N    192.168.0.0/24        [13107] area: 0.0.0.1
                           directly attached to eth1
N    192.168.1.0/24        [11348] area: 0.0.0.0
                           via 130.233.154.36, eth0
N    192.168.20.0/24       [16384] area: 0.0.0.1
                           via 192.168.0.2, eth1

============ OSPF BSP router routing table =============
R    192.168.1.1           [9362] area: 0.0.0.0, ASBR
                           via 130.233.154.36, eth0

============ OSPF BSP external routing table ===========
N E2 0.0.0.0/0             [9362/168] tag: 0
                           via 130.233.154.254, eth0


PC59_ospfd#
PC59_ospfd# show ip ospf
 ……
  Area ID: 0.0.0.0 (Backbone)
   Number of interfaces in this area: Total: 1, Active: 2
   Number of fully adjacent neighbors in this area: 1
   Area has no authentication
   SPF algorithm executed 4 times
   BSP algorithm executed 4 times
   Number of LSA 7

 Area ID: 0.0.0.1
   Shortcutting mode: Default, S-bit consensus: no
   Number of interfaces in this area: Total: 1, Active: 2
   Number of fully adjacent neighbors in this area: 1
   Area has no authentication
```

```
   Number of full virtual adjacencies going through this area: 0
   SPF algorithm executed 4 times
   BSP algorithm executed 4 times
   Number of LSA 9
```

This is the number of executing the whole routing table calculation based on a certain routing algorithm

2) Step 4's OSPF Information

```
PC59_ospfd# show ip ospf route
============ OSPF network routing table ============
N    130.233.154.0/24      [5000] area: 0.0.0.0
                           directly attached to eth0
N    192.168.0.0/24        [2000] area: 0.0.0.1
                           directly attached to eth1
N    192.168.1.0/24        [5100] area: 0.0.0.0
                           via 130.233.154.36, eth0
N    192.168.20.0/24       [2800] area: 0.0.0.1
                           via 192.168.0.2, eth1

============ OSPF router routing table =============
R    192.168.1.1           [5000] area: 0.0.0.0, ASBR
                           via 130.233.154.36, eth0

============ OSPF external routing table ===========
N E2 0.0.0.0/0             [5000/300] tag: 0
                           via 130.233.154.254, eth0

============ OSPF BSP network routing table ============
N    130.233.154.0/24      [9362] area: 0.0.0.0
                           directly attached to eth0
N    192.168.0.0/24        [13107] area: 0.0.0.1
                           directly attached to eth1
N    192.168.1.0/24        [11348] area: 0.0.0.0
                           via 130.233.154.36, eth0
N    192.168.20.0/24       [16384] area: 0.0.0.1
                           via 192.168.0.2, eth1

============ OSPF BSP router routing table =============
R    192.168.1.1           [9362] area: 0.0.0.0, ASBR
                           via 130.233.154.36, eth0

============ OSPF BSP external routing table ===========
N E2 0.0.0.0/0             [9362/168] tag: 0
                           via 130.233.154.254, eth0

PC59_ospfd# show ip ospf
 ……
 Area ID: 0.0.0.0 (Backbone)
   Number of interfaces in this area: Total: 1, Active: 2
   Number of fully adjacent neighbors in this area: 1
   Area has no authentication
   SPF algorithm executed 5 times
   BSP algorithm executed 4 times
   Number of LSA 7

 Area ID: 0.0.0.1
   Shortcutting mode: Default, S-bit consensus: no
   Number of interfaces in this area: Total: 1, Active: 2
   Number of fully adjacent neighbors in this area: 1
   Area has no authentication
   Number of full virtual adjacencies going through this area: 0
   SPF algorithm executed 5 times
   BSP algorithm executed 4 times
   Number of LSA 9
```

The SP algorithm is executed one time, but the BSP is not performed.

Compared to the Step 2's general OSPF information, only the SP algorithm is executed one time after the PC59 receives the Debian's Router-LSA with the changed TOS 0 metric. In the SP routing table, the cost of the route to the destination 192.168.1.0/24 is altered after the PC59's *ospfd* daemon recalculates the SP algorithm.

3) Step 6's OSPF Information

```
PC59_ospfd# show ip ospf
 ……
 Area ID: 0.0.0.0 (Backbone)
   Number of interfaces in this area: Total: 1, Active: 2
   Number of fully adjacent neighbors in this area: 1
   Area has no authentication
   SPF algorithm executed 5 times
   BSP algorithm executed 4 times
   Number of LSA 7

 Area ID: 0.0.0.1
   Shortcutting mode: Default, S-bit consensus: no
   Number of interfaces in this area: Total: 1, Active: 2
   Number of fully adjacent neighbors in this area: 1
   Area has no authentication
   Number of full virtual adjacencies going through this area: 0
   SPF algorithm executed 5 times
   BSP algorithm executed 4 times
   Number of LSA 9

PC59_ospfd# show ip ospf route
============ OSPF network routing table ============
N    130.233.154.0/24      [5000] area: 0.0.0.0
                           directly attached to eth0
N    192.168.0.0/24        [2000] area: 0.0.0.1
                           directly attached to eth1
N    192.168.1.0/24        [5100] area: 0.0.0.0
                           via 130.233.154.36, eth0
N    192.168.20.0/24       [2800] area: 0.0.0.1
                           via 192.168.0.2, eth1

============ OSPF router routing table =============
R    192.168.1.1           [5000] area: 0.0.0.0, ASBR
                           via 130.233.154.36, eth0

============ OSPF external routing table ===========
N E2 0.0.0.0/0             [5000/300] tag: 0
                           via 130.233.154.254, eth0

============ OSPF BSP network routing table ============
N    130.233.154.0/24      [9362] area: 0.0.0.0
                           directly attached to eth0
N    192.168.0.0/24        [13107] area: 0.0.0.1
                           directly attached to eth1
N    192.168.1.0/24        [11348] area: 0.0.0.0
                           via 130.233.154.36, eth0
N    192.168.20.0/24       [16384] area: 0.0.0.1
                           via 192.168.0.2, eth1

============ OSPF BSP router routing table =============
R    192.168.1.1           [9362] area: 0.0.0.0, ASBR
                           via 130.233.154.36, eth0

============ OSPF BSP external routing table ===========
N E2 0.0.0.0/0             [9362/1678] tag: 0
                           via 130.233.154.254, eth0
```

Both the SP algorithm and the BSP algorithm are not executed.

No entire routing table calculation takes place in terms of the general OSPF information. Only incremental update calculation for the BSP routing table is executed. The cost of the default route is correspondingly changed in the BSP routing table.

Conclusions

In our *ospfd* daemon, the SP algorithm calculation is independent of the BSP algorithm calculation. The pre-computation trigger module can correctly work. It effectively avoids any unnecessary routing calculation. It facilitates to speed up the convergence of OSPF.

6.3 Route Summarization Test

The Goal of the route summarization test is to demonstrate the correctness of creating the Summary-LSAs related to the aggregated IP addresses.

Configuration

The route summarization test is shown in Figure 6.2. The configuration information is indicated in the Figure. On PC59, 130.233.0.0/16 used as a range of aggregated IP addresses is configured for area 0, and it is used for the SP routing table. The range 192.168.0.0/16 is set for area 1, and it is used for the BSP routing table.

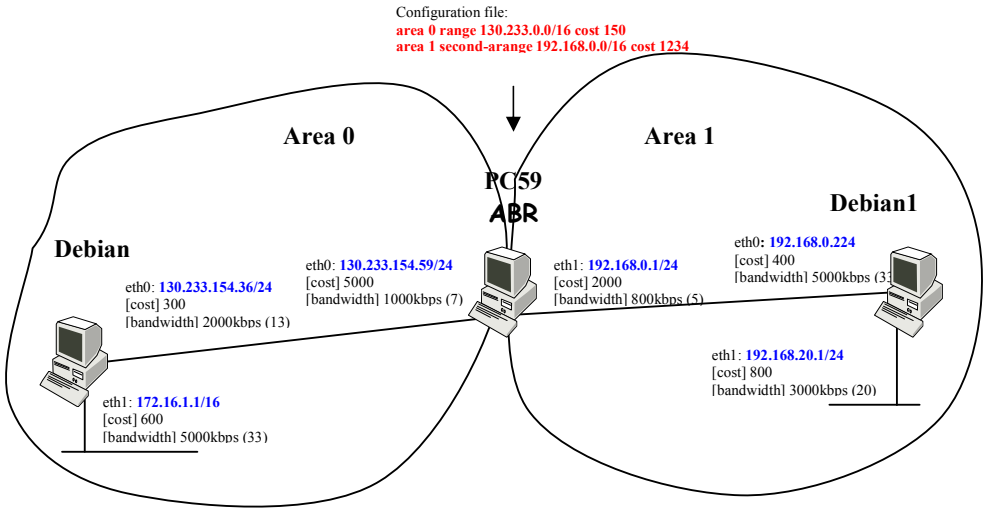

Figure 6.2: A route summarization test environment

Results

1) The PC59's routing tables

```
PC59_ospfd# show ip ospf route
============ OSPF network routing table ============
D IA 130.233.0.0/16        Discard entry
N    130.233.154.0/24      [5000] area: 0.0.0.0
                           directly attached to eth0
N    172.16.0.0/16         [5600] area: 0.0.0.0
                           via 130.233.154.36, eth0
N    192.168.0.0/24        [2000] area: 0.0.0.1
                           directly attached to eth1
N    192.168.20.0/24       [2800] area: 0.0.0.1
                           via 192.168.0.2, eth1

============ OSPF router routing table =============

============ OSPF external routing table ===========

============ OSPF BSP network routing table ============
N    130.233.154.0/24      [9362] area: 0.0.0.0
                           directly attached to eth0
N    172.16.0.0/16         [11348] area: 0.0.0.0
                           via 130.233.154.36, eth0
D IA 192.168.0.0/16        Discard entry
N    192.168.0.0/24        [13107] area: 0.0.0.1
```

```
                            directly attached to eth1
N    192.168.20.0/24        [16384] area: 0.0.0.1
                            via 192.168.0.2, eth1

============ OSPF BSP router routing table =============

============ OSPF BSP external routing table ===========

PC59_ospfd#
```

A label ‘D IA’ represents a route related to a range of aggregated IP addresses. Any route with the label ‘D IA’ is not used for the route selection.

2) Type 3 Summary-LSAs originated by the PC59

```
PC59_ospfd# show ip ospf database summary

       OSPF Router with ID (192.168.0.1)


                Summary Link States (TYPE 3) (Area 0.0.0.0)

  LS age: 620
  Options: 130
  LS Type: summary-LSA
  Link State ID: 192.168.0.0 (summary Network Number)
  Advertising Router: 192.168.0.1
  LS Seq Number: 80000001
  Checksum: 0x87a9
  Length: 28
  Network Mask: /24
        TOS: 0  Metric: 2000
  LS age: 620
  Options: 130
  LS Type: summary-LSA
  Link State ID: 192.168.20.0 (summary Network Number)
  Advertising Router: 192.168.0.1
  LS Seq Number: 80000001
  Checksum: 0x07f2
  Length: 28
  Network Mask: /24
        TOS: 0  Metric: 2800

                Summary Link States (TYPE 3) (Area 0.0.0.1)

  LS age: 1056
  Options: 130
  LS Type: summary-LSA
  Link State ID: 130.233.0.0 (summary Network Number)
  Advertising Router: 192.168.0.1
  LS Seq Number: 80000001
  Checksum: 0x1c53
  Length: 28
  Network Mask: /16
        TOS: 0  Metric: 150
  LS age: 565
  Options: 130
  LS Type: summary-LSA
  Link State ID: 172.16.0.0 (summary Network Number)
  Advertising Router: 192.168.0.1
  LS Seq Number: 80000001
  Checksum: 0xd2ec
  Length: 28
  Network Mask: /16
        TOS: 0  Metric: 5600
PC59_ospfd#
```

The PC59 creates a type 3 Summary-LSA containing the range 130.233.0.0/16 instead of the normal one with 130.233.154.0/24. It advertises this Summary-LSA into the area 1.

3) Type 12 Summary-LSAs originated by the PC59

```
PC59_ospfd# show ip ospf database bsummary

       OSPF Router with ID (192.168.0.1)


                Second Summary Link States (TYPE 12) (Area 0.0.0.0)

  LS age: 635
  Options: 130
  LS Type: summary-BSP-LSA
  Link State ID: 192.168.0.0 (summary Network Number)
  Advertising Router: 192.168.0.1
  LS Seq Number: 80000001
  Checksum: 0x0227
  Length: 28
  Network Mask: /16
        TOS: 0  Metric: 1234

                Second Summary Link States (TYPE 12) (Area 0.0.0.1)

  LS age: 1070
  Options: 130
  LS Type: summary-BSP-LSA
  Link State ID: 130.233.154.0 (summary Network Number)
  Advertising Router: 192.168.0.1
  LS Seq Number: 80000001
  Checksum: 0x1695
  Length: 28
  Network Mask: /24
        TOS: 0  Metric: 9362
  LS age: 580
  Options: 130
  LS Type: summary-BSP-LSA
  Link State ID: 172.16.0.0 (summary Network Number)
  Advertising Router: 192.168.0.1
  LS Seq Number: 80000001
  Checksum: 0xa685
  Length: 28
  Network Mask: /16
        TOS: 0  Metric: 11348
PC59_ospfd#
```

The PC59 aggregates two intra-area routes, which belong to area 1, into a single type 12 Summary-LSA, and it advertises this Summary-LSA into the area 0.

Conclusions

Our *ospfd* daemon can successfully create type 3 and type 12 Summary-LSAs about aggregated IP addresses. The route summarization for different routing tables is separately executed. Based on the correct configuration about the route summarization, the *ospfd* daemon can decrease the number of the Summary-LSAs.

6.4 Multi-path Test

The Goal of the multi-path test is to demonstrate that the *ospfd* daemon supporting the MCR capability can provide multiple paths to the same destination.

Configuration

The multi-path test is shown in Figure 6.3. The configuration information is indicated in the Figure.

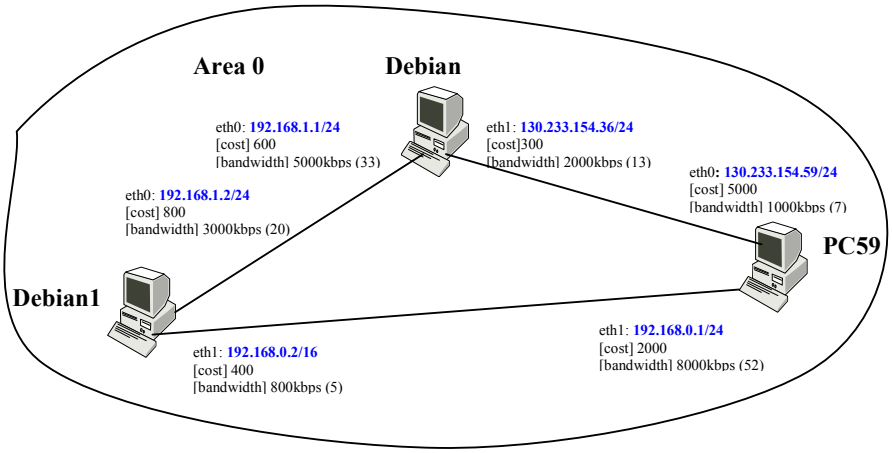

Figure 6.3: A multi-path test environment

Results

```
debian1_ospfd# show ip ospf route
============ OSPF network routing table ============
N    130.233.154.0/24      [1100] area: 0.0.0.0
                           via 192.168.1.1, eth1
N    192.168.0.0/24        [400] area: 0.0.0.0
                           directly attached to eth0
N    192.168.1.0/24        [800] area: 0.0.0.0
                           directly attached to eth1

============ OSPF router routing table =============

============ OSPF external routing table ===========

============ OSPF BSP network routing table ============
N    130.233.154.0/24      [8318] area: 0.0.0.0
                           via 192.168.1.1, eth1
N    192.168.0.0/24        [9578] area: 0.0.0.0
                           via 192.168.1.1, eth1
N    192.168.1.0/24        [3277] area: 0.0.0.0
                           directly attached to eth1

============ OSPF BSP router routing table =============

============ OSPF BSP external routing table ===========
```

Debian1's SP routing table

Debian1's BSP routing table

The above routing tables belong to the Debian1. In the SP routing table, the egress of the route to 192.168.0.0/24 is eth0. The egress of the route to 192.168.0.0/24 is eth1 in the BSP routing table. Thus, the traffic to the same destination 192.168.0.0./24 traverses through different paths in the two routing tables.

Conclusions

On some network topologies, different routing algorithms can create diverse routes to the same destination.

6.5 Discussion

Our OSPF with MCR extensions is localized to some new modules and some modified modules, which reuse the existing OSPF modules with only necessary changes. It influences on the OSPF convergence because of the introduction of new elements, such as the second routing table, a new routing algorithm and new LSAs, etc.

The time of computing the routing tables definitely increases in the OSPF with MCR extensions because two routing tables need to be considered. An OSPF router, which is running our *ospfd* daemon, simultaneously creates two routing tables. The computing time becomes approximately double if the *ospfd* daemon always recalculates two routing tables after receiving a changed LSA. Fortunately, our OSPF code uses the pre-computation trigger to avoid any unnecessary routing calculation, and it adopts the incremental update calculations to prevent any unnecessary whole routing table calculation.

The number of Summary-LSAs could double. Some new types of Summary-LSAs are introduced into the OSPF protocol because more than one routing table exists. However, the route summarization mechanism can efficiently aggregate several Summary-LSAs into a single Summary-LSA. Our *ospfd* daemon separately executes the route summarization for each routing table. Thus, the route summarization mechanism can decrease the number of Summary-LSAs.

In our OSPF code, the output cost and bandwidth of each interface of a router are configured by VTY commands. The costs that are inserted into the router' Router-LSA can not be changed automatically. Similarly, the external routing information, which is related to AS-external-LSAs, is configured manually. This does not frequently produce the procedure of the OSPF convergence.

In brief, our OSPF with MCR extensions is basically feasible because several methods are used to minimize the impact on the OSPF convergence. However, as a matter of fact, the bandwidth of a router's interface dynamically changes with the usage of the interface. When the BSP algorithm uses the available bandwidth of each link to compute the second routing table, the second routing table can offer more up-to-date routes for the traffic. But, using the available bandwidth could result in frequently updating the Router-LSAs. That means that frequent OSPF convergence would occur.

7. Conclusions and Future work

7.1 Conclusions

In this thesis, we introduced the MCR scheme into the DiffServ architecture. In our work, the MCR DiffServ architecture focused on the intra-DS domain. We discussed different MCR approaches that include a static MCR approach and dynamic MCR approaches. We discussed how to communicate between an MCR area and a non-MCR area. The design of the MCR DiffServ system was provided in the thesis. The MCR approaches can be used for the MCR DiffServ system. We designed the OSPF with MCR extensions, and implemented the static MCR based on the Zebra OSPF software.

Our OSPF with MCR extensions can achieve the functions of the MCR manager for the MCR DiffServ system. It can provide more than one routing table. Those routing tables are independent. Distinct routing algorithms can be used to compute the different routing tables. The routes to the same destination could be different in the diverse routing tables.

The static MCR approach was implemented for the MCR DiffServ system. When a router's link weights, for instance, bandwidth, are changed by the administrator, the router will create its updated LSAs and flood them throughout an area or a whole routing domain.

The second routing table was added into our OSPF with MCR extensions. The corresponding new types of Summary-LSAs for the new routing table must be introduced. Several modules, which are related to different types of LSAs, must be modified on the basis of the legacy OSPF code.

Some useful mechanisms were adopted in our OSPF with MCR extensions in order to minimize the impact on the OSPF convergence. To decrease the number of Summary-LSAs, the route summarization mechanism should be implemented for each routing table. To prevent any unnecessary whole routing table calculation, the incremental update calculation is needed for each routing algorithm. To avoid any unnecessary routing calculation, the pre-computation trigger should correctly determine which routing algorithm is executed.

7.2 Future work

For the MCR DiffServ system, the class-based route selection block should be reasonably designed and implemented in the computer operating systems, for example, FreeBSD.

Furthermore, the *zebra* daemon of the Zebra software should provide the management functions for multiple routing tables in the kernel.

For our OSPF with MCR extensions, different dynamic MCR approaches need to be further discussed and designed. New methods to encode bandwidth are needed because our encoding method is not suitable for the dynamic MCR approaches. In the software architecture of the OSPF with MCR extensions, the local interface status manager module needs to be implemented for the dynamic MCR approaches, for example, dynamic distributed MCR with significant event triggering. The measurement of link weights is absolutely important for any dynamic MCR approach since the measurement of link weights could have an impact on convergence. For this reason, methods to measure link weights, such as delay and residual bandwidth, need to be determined reasonably.

The performance and scalability of the different MCR approaches, especially dynamic approaches, need to be tested in different network environments. The impact of MCR on convergence and packet loss is one of the most important criteria in establishing whether the idea of MCR is beneficial to the users and network operators.

Based on our static MCR implementation, new QoS routing algorithms, for example, WSP algorithm, can be implemented in our OSPF code. New routing tables can be introduced into the software architecture of the OSPF with MCR extensions. Each routing table in the software architecture deploys different routing algorithms to compute route entries.

We need to carry out some further research work for the MCR DiffServ architecture. The research work should include the inter-DS domain since this thesis only discusses the intra-DS domain.

Within a DS domain, the communication between MCR areas and non-MCR areas needs to be further discussed in detail. Our OSPF with MCR extensions needs to be further extended to support new functions and mechanisms of the MCR gateway.

Reference

[1] S. Blake, et al. An Architecture for Differentiated Services. RFC2475. December 1998.

[2] J. Wang, Y. Wang and K. Nahrstedt. Quantitative Study of Differentiated Service Model Using UltraSAN. Tech. Report UIUCDCS-R-2001-2237, Department of Computer Science, University of Illinois at Urbana-Champaign, July 2001.

[3] P. Zhang, Xiaole Bai and Raimo Kantola. A Routing Scheme for Optimizing Multiple Classes in a DiffServ Network. IEEE, 2004.

[4] Zheng. Wang. Internet QoS : Architectures and Mechanisms for Quality of Service.

[5] J. Moy. OSPF Version 2. RFC2328. April 1998.

[6] J. Moy. OSPF Version 2 . RFC1583. March 1994.

[7] A. Zinin, A. Lindem and D. Yueng. Alternative Implementations of OSPF Area Border Routers. RFC3509. April 2003.

[8] Christian Huitema. Routing in the Internet. Second edition. Prentice-hall, Inc. 2000.

[9] R. Coltun. The OSPF Opaque LSA Option.RFC2370 . July 1998.

[10] G. Apostolopoulos, D. Williams, S. Kamat, R. Guerin, A. Orda and T. Przyqienda. QoS Routing Mechanisms and OSPF Extensions. RFC2676. August 1999.

[11] Qingming Ma and Peter Steenkiste. On Path Selection for Traffic with Bandwidth Guarantees. IEEE. 1997.

[12] J. Wang, and K. Nahrstedt. Hop-by-Hop Routing Algorithms for Premium-class Traffic in DiffServ Networks. ACM SIGCOMM Computer Communications Review. November 2002.

[13] Wooguil Pak and Saewoong Bahk. Partial Optimization Method of Topology Aggregation for Hierarchical QoS. IEEE. 2002.

[14] G. Apostolopoulos, R. Guerin and S. Kamat. Implementation and Performance Measurements of QoS Routing Extensions to OSPF. IEEE. 1999

[15] Hedia Kochkar, Takeshi Ikenaga, Yoshiaki Hori and Yuji Oie. Multi-Class QoS Routing with Multiple Routing Tables. IEEE. 2003.

[16] Kenjiro Cho. The Design and Implementation of the ALTQ Traffic Management System. PH.D thesis. Keio University. January 2001.

[17] Marcin Matuszewski, Raimo Kantola, Risto Sarala, OSPF Convergence and Its Impact on VoIP, accepted at ATNAC 2004.

[18] Cisco Company, CISCO Nonstop Forwarding and Timer Manipulation for Fast Convergence.
http://www.cisco.com/en/US/tech/tk869/tk769/technologies_white_paper09186a00801dce40.shtml

Appendix A. Files of Zebra OSPF Software

Source codes of Zebra OSPF software consist of several source files. The simple summary of each file displays as follows. Those files are located at the directory zebra-0.94/ospfd.

- File ospf_main.c is used to initialise OSPF module, go into a loop and process threads.
- File ospfd.h defines important data structures, such as structure ospf_master, ospf and ospf_area.
- File ospfd.c provides some functions and fundamental feature of OSPF
- File ospf_packet.c is responsible for sending and receiving all kinds of OSPF packets, such as Hello packets, Database Description packet and Link State Update packet. Structure ospf_packet, ospf_header, ospf_hello, ospf_db_desc and ospf_fifo are defined in the file ospf_packet.h.
- File ospf_zebra.c is used to provide application interface between ZebOS module and OSPF module. It is involved with redistributing the external routing information.
- File ospf_interface.c contains functions about interfaces and virtual link. Structure ospf_interface is defined in the file ospf_interface.h.
- File ospf_ism.c is for OSPF version 2 interface state machine that is consistent with RFC2319.
- File ospf_neighbor.c is used to define neighbors and adjacencies. Structure ospf_neighbor is defined in the file ospf_neighbor.h.
- File ospf_nsm.c is for OSPF version 2 neighbor state machine.
- File ospf_network.c is concerned about network related functions. For example, joining or leaving ALLDRouter multicast group.
- File ospf_flood.c provides functions about flooding procedure.
- File ospf_lsdb.c provides functions about managing link state database. Structure ospf_lsdb is defined in the file ospf_lsdb.h.
- File ospf_route.c provides functions about OSPF routing table. Structure ospf_route and ospf_path are defined in the file ospf_route.h.
- File ospf_lsa.c handles each type of LSAs and installs LSAs to LSDB. Each kind of LSAs' data structure is defined in the file ospf_lsa.h, such as structure lsa_header, ospf_lsa and router_lsa.
- File ospf_abr.c specifies how to create Summary-LSAs for ABR. The LSA about aggregation and default route are handled here. Structure ospf_area_range is defined in the file ospf_abr.h.

- File ospf_asbr.c provides ASBR functions. Structure external_info is defined in the file ospf_asbr.h.
- File ospf_spf.c calculates a SPF for an area. If router has more than one area, the calculation must be performed for each area. Structure vertex and vertex_nexthop are defined in the file ospf_spf.h.
- File ospf_ia.c calculates inter-area routes.
- File ospf_ase.c calculates As External routes.

The below files support various data structures, which are used by above files. Those files are located at the directory zebra-0.94/lib.

- File table.c describes a binary tree structure used for storing different data that is identified by IP data structure, for instance, routing table and link state database (LSDB). An efficient binary tree structure is easy to be added, modified and deleted. Structure route_table and route_node are defined in the file table.h.
- File prefix.c describes IP Data structure. Using data type union, it can use one structure describe three or four different types of IP addresses, such as IPv4 address and IPv6 address. Structure prefix and prefix_ipv4 are defined in the file prefix.h.
- File linklist.c describes a set of list functions and macros. It can be used for LSAs. Structure list and list_node are defined in the file linklist.h.
- File stream.c describes useful functions and macros for handling OSPF packet streams. Structure stream is defined in the file stream.h.
- File thread.c provides macros and functions about thread management. Structure thread, thread_master and thread_list are defined in the file thread.h.

Appendix B. Creation Functions for MCR-LSA Originator

Each type of LSAs has its own creation functions, which are listed in Table B.1. The creation functions for the type 12 and type 13 Summary-LSAs are new functions in our implementation. The creation functions for the Network-LSAs and type 3 and type 4 Summary-LSAs are original OSPF functions. The creation functions for Router-LSAs and AS-external-LSAs are modified in terms of the formats of the MCR LSAs. All following creation functions belong to the file ospf_lsa.c.

Type	Creation Functions	state
Router-LSA	ospf_router_lsa_originate() ospf_router_lsa_refresh()	modified
Network-LSA	ospf_network_lsa_originate() ospf_network_lsa_refresh()	original
Type 3 Summary-LSA	ospf_summary_lsa_originate() ospf_summary_lsa_refresh()	original
Type 4 Summary-LSA	ospf_summary_asbr_lsa_originate() ospf_summary_asbr_lsa_refresh()	original
Type 12 Summary-LSA	ospf_summary_bsp_lsa_originate() ospf_summary_bsp_lsa_refresh()	new
Type 13 Summary-LSA	ospf_summary_asbr_bsp_lsa_originate() ospf_summary_asbr_bsp_lsa_refresh()	new
AS-external-LSA	ospf_external_lsa_originate() ospf_external_lsa_refresh()	modified

Table B.1: Creation functions for MCR-LSA originator module

Appendix C. All Modified Data Structures and Related Functions

Table C.1 shows all modified data structures and related functions, which are used for our OSPF code.

Modified Data Structures	Location	Modified Functions
ospf	ospfd.c	ospf_new() ospf_finish() ospf_network_run() ospf_timer_bsp_set() ospf_timer_bsp_unset()
ospf_area	ospfd.c	ospf_area_new() ospf_area_free() ospf_are
ospf_interface	ospf_interface.c	ospf_if_new() ospf_if_cleanup() ospf_if_get_output_bandwidth()
ospf_vl_data	ospf_interface.c	ospf_vl_set_params() ospf_vl_up_check()
route_map_set_values	ospf_asbr.c	ospf_reset_route_map_set_value() ospf_route_map_set_compare()
	ospf_zebra.c	ospf_redistribute_set() ospf_redistribute_unset() ospf_redistribute_default_set() ospf_redistribute_default_unset()

Table C.1: All modified data structures and related functions

Appendix D. VTY Commands

New VTY commands are shown in Table D.1. All modified VTY commands are shown in Table D.2. Table D.3 shows all original VTY commands, displays of which are changed.

VTY commands	Description	Related Functions	Function state
1) **timer bsp** <x> <y> 2) **no timer bsp**	Set parameters to the timer of the BSP algorithm	ospf_timer_bsp_set() ospf_timer_bsp_unset() DEFUN(ospf_timers_bsp_cmd) DEFUN(no_ospf_timers_bsp_cmd)	new
		ospf_vty_init()	modified
1) **route-map** x **(permit| deny)** x **set metric_bandwidth** x 2) **route-map** x **(permit| deny)** x **no set metric_bandwidth** x	Set a bandwidth for a route map profile	route_set_metric_bandwidth() route_set_metric_bandwidth_compile() route_set_metric_bandwidth_free() struct route_map_rule_cmd route_set_metric_bandwidth_cmd DEFUN(set_metric_bandwidth_cmd) DEFUN(no_set_metric_bandwidth_cmd)	new
		ospf_route_map_init()	modified
1) **area** x.x.x.x **second-range** x.x.x.x/y 2) **area** x.x.x.x **second-range** x.x.x.x/y **advertise** 3) **area** x.x.x.x **second-range** x.x.x.x/y **cost** <x> 4) **area** x.x.x.x **second-range** x.x.x.x/y **advertise cost** <x>	Set an area range for the second routing table. The range are used for the route summarization	ospf_area_range_second_set() ospf_area_range_second_cost_set() DEFUN(ospf_area_second_range_cmd) ALIAS(ospf_area_second_range_cost_cmd) ALIAS(ospf_area_second_range_advertise_cmd) ALIAS(ospf_area_second_range_advertise_cost_cmd)	new
		ospf_vty_init()	modified

1) no **area** x.x.x.x **second-range** x.x.x.x/y 2) no **area** x.x.x.x **second-range** x.x.x.x/y **advertise cost** <x> 3) no **area** x.x.x.x **second-range** x.x.x.x/y **cost** <x>	Unset an area range for the second routing table	ospf_area_range_second_unset() DEFUN(no_ospf_area_second_range_cmd) ALIAS(no_ospf_area_second_range_cost_cmd) ALIAS(no_ospf_area_second_range_advertise_cmd) ALIAS(no_ospf_area_second_range_advertise_cost_cmd)	new
		ospf_vty_init()	modified
1) **area** x.x.x.x **second-range** x.x.x.x/y **substitute** x.x.x.x/z	Set a substituted area range for the second routing table	ospf_area_range_second_substitue_set() DEFUN(ospf_area_second_range_substitute_cmd)	new
		ospf_vty_init()	modified
1) no **area** x.x.x.x **second-range** x.x.x.x/y **substitute** x.x.x.x/z	Unset a substituted area range for the second routing table	ospf_area_range_second_substitue_unset() DEFUN(no_ospf_area_second_range_substitute_cmd)	new
		ospf_vty_init()	modified
1**) ip ospf bandwidth** <x> 2) **no ip ospf bandwidth**	Set output bandwidth for an interface	ospf_if_recalculate_output_bw() DEFUN(ip_ospf_bandwidth_cmd) DEFUN(no_ip_ospf_bandwidth_cmd)	new
		ospf_vty_init()	modified

Table D.1: New VTY commands

VTY commands	Description	Related Functions	Function state
1) **redistribute** x **metric** <x> **metric-bandwidth** <x> **metric-type** (1|2) **route-map** WORD 2) **redistribute** x **metric** <x> **metric-bandwidth** <x> **metric-type** (1|2) 3) **redistribute** x **metric** <x> **metric-bandwidth** <x> 4) **redistribute** x **metric-type** (1|2)**metric** <x> **metric-bandwidth** <x>	Redistribute external routing information into OSPF. Those (commands should be	ospf_redistribute_set() DEFUN(ospf_redistribute_source_metric_type_routemap_cmd) ALIAS(ospf_redistribute_source_metric_type_cmd) ALIAS(ospf_redistribute_source_metric_cmd)	modified

route-map WORD 5) **redistribute** x **metric-type** (1|2)**metric** <x> **metric-bandwidth** <x> 6) **redistribute** x **metric-type** (1|2) 7) **redistribute** x **metric-type** (1|2) **route-map** WORD 8) **redistribute** x 9) **redistribute** x **route-map** WORD 10) **redistribute** x **metric** <x> **metric-bandwidth** <x> **route-map** WORD	included metric and bandwidth.)	DEFUN(ospf_redistribute_source_type_metric_routemap_cmd) ALIAS(ospf_redistribute_source_type_metric_cmd) ALIAS(ospf_redistribute_source_type_cmd) ALIAS(ospf_redistribute_source_cmd) DEFUN(ospf_redistribute_source_metric_routemap_cmd) DEFUN(ospf_redistribute_source_routemap_cmd) DEFUN(ospf_redistribute_source_type_routemap_cmd)	
1) default-information originate metric <x> **metric-bandwidth** <x> **metric-type** (1|2) **route-map** WORD 2) **default-information originate metric** <x> **metric-bandwidth** <x> **metric-type** (1|2) 3) **default-information originate metric** <x> **metric-bandwidth** <x> 4) **default-information originate** 5) **default-information originate metric** <x> **metric-bandwidth** <x> **route-map** WORD 6) **default-information originate metric-type** (1|2) **metric** <x> **metric-bandwidth** <x> **route-map** WORD 7) **default-information originate metric-type** (1|2) **metric** <x> **metric-bandwidth** <x> 8) **default-information originate metric-type** (1|2) 9) **default-information originate metric-type** (1|2) **route-map** WORD 10) **default-information originate route-map** WORD 11) **default-information originate always metric** <x> **metric-bandwidth** <x> **metric-type** (1|2) **route-map** WORD 12) **default-information originate always metric** <x> **metric-bandwidth** <x> **metric-type** (1|2) 13) **default-information originate always metric** <x> **metric-bandwidth**	Redistribute default external routing information into OSPF	ospf_redistribute_default_set() DEFUN(ospf_default_information_originate_metric_type_routemap_cmd) ALIAS(ospf_default_information_originate _metric_type_cmd) ALIAS(ospf_default_information_originate_metric_cmd) ALIAS(ospf_default_information_originate_cmd) DEFUN(ospf_default_information_originate_metric_routemap_cmd) DEFUN(ospf_default_information_originate_type_metric_routemap_cmd) ALIAS(ospf_default_information_originate_type_metric_cmd) ALIAS(ospf_default_information_originate_type_cmd) DEFUN(ospf_default_information_originate_type_routemap_cmd) DEFUN(ospf_default_information_originate_routemap_cmd) DEFUN(ospf_default_information_originate_always_metric_type_	modified

<x> 14) **default-information originate always** 15) **default-information originate always metric** <x> **metric-bandwidth** <x> **route-map** WORD 16) **default-information originate always route-map** WORD 17) **default-information originate always metric-type** (1|2) **metric** <x> **metric-bandwidth** <x> **route-map** WORD 18) **default-information originate always metric-type** (1|2) **metric** <x> **metric-bandwidth** <x> 19) **default-information originate always metric-type** (1|2) 20) **default-information originate always metric-type** (1|2) **route-map** WORD		routemap_cmd) ALIAS(ospf_default_information_originate_always_metric_type_cmd) ALIAS(ospf_default_information_originate_always_metric_cmd) ALIAS(ospf_default_information_originate_always_cmd) DEFUN(ospf_default_information_originate_always_metric_routemap_cmd) DEFUN(ospf_default_information_originate_always_routemap_cmd) DEFUN(ospf_default_information_originate_always_type_metric_routemap_cmd) ALIAS(ospf_default_information_originate_always_type_metric_cmd) ALIAS(ospf_default_information_originate_always_type_cmd) DEFUN(ospf_default_information_originate_always_type_routemap_cmd)	
1) **default-metric** <x> **default-bandwidth-metric** <x> 2) **no default-metric**	Set a default metric and bandwidth for external information	DEFUN(ospf_default_metric_cmd) DEFUN(no_ospf_default_metric_cmd)	modified
		ospf->default_metric_bw	new

Table D.2: Modified VTY commands

Original VTY Command	Description	Modified Functions/Definitions
show ip ospf route	Show the SP routing table and the BSP routing	DEFUN(show_ip_ospf_route_cmd)

	table	
show ip ospf database	Show all link-state database (LSDB)	DEFUN(show_ip_ospf_database_cmd)
show ip ospf database x	Show a certain LSDB based on the type of LSAs	show_lsa_summary() show_ip_ospf_database_router_links() show_summary_bsp_lsa_detail() show_summary_asbr_bsp_lsa_detail() show_as_external_lsa_detail() int (*show_function[]) () char *show_deatabase_desc[] char *show_database_header[] ALIAS(show_ip_ospf_database_type_cmd) #define OSPF_LSA_TYPES_CMD_STR
show ip ospf	Show the information about OSPF	DEFUN(show_ip_ospf_cmd)
show ip ospf neighbor detail	Show the detailed information about all neighbors	ospf_options_dump()
show ip ospf interface	Show the detailed information about all interfaces	ospf_if_get_input_bandwidth() show_ip_ospf_interface_sub() DEFUN(show_ip_ospf_interface_cmd)

Table D.3: VTY commands with changed displays