Tampere University of Technology
Institute of Communications Engineering

Sampo Saaristo

Implementation of IS-IS Routing Protocol for IP versions 4 and 6

Master of Science Thesis

Subject approved by the department council on 10.04.2002
Supervisors: Prof. Jarmo Harju
MSc Heikki Vatiainen

Foreword

This Master of Science Thesis was written during spring 2002 as part of the ICEFIN project at Tampere University of Technology, Insititute of Communications Engineering. The software code the Thesis is based on was written as part of ICEFIN's predecessor, the Faster Pro project, and was started in May 2001.

I would like to thank my co-developers Ofer Wald and Hannes Gredler with whom I did the coding at SourceForge.net [1]. I would also like to thank Prof. Jarmo Harju and MSc Heikki Vatiainen for encouragement and guidance, MSc Bilhanan Silverajan for proof-reading and Anna Lahti and my family for support.

Tampere, October 1, 2002

Sampo Saaristo
Pakilantie 15 A 6
00630 Helsinki
Finland
`sambo@cs.tut.fi`

Abstract

TAMPERE UNIVERSITY OF TECHNOLOGY

Department of Information Technology

Institute of Communications Engineering

Sampo Saaristo: Implementation of IS-IS Routing Protocol for IP versions 4 and 6

Master of Science thesis: 62 pages, 20 enclosure pages

Supervisors: Prof. Jarmo Harju and Msc Heikki Vatiainen

October 2002

This Master of Science Thesis describes an implementation of Intermediate System to Intermediate System (IS-IS) routing protocol, that supports routing of Internet Protocol version 4 (IPv4) and 6 (IPv6). The thesis was done for the ICEFIN Research Laboratory at Tampere University of Technology, Institute of Communications Engineering.

IS-IS is a link state routing protocol originally designed by International Organization for Standardization (ISO). The protocol was first defined to route ConnectionLess Network Protocol (CLNP), the Open Systems Interconnection (OSI) stack equivalent to IP. However, IS-IS is designed in such a manner that it can easily be extended to support routing of any layer three protocol. The support for IP was specified by the Internet Engineering Task Force (IETF) 1990 and the extensions for IPv6 were introduced in 2000.

The implementation of IS-IS protocol was written as a module for the freely distributable GNU Zebra routing software. The software project was supported by SourceForge.net, so any interested developer was able to join and contribute to the project.

The thesis starts with a discussion on routing and routing protocols in general. Then the IS-IS routing protocol and the IETF extensions to the protocol are introduced.

Before describing the implementation of IS-IS (ISISd), the GNU Zebra routing software is introduced. The rest of the thesis deals with configuration of ISISd.

Tiivistelmä

TAMPEREEN TEKNILLINEN KORKEAKOULU

Tietotekniikan osasto

Tietoliikennetekniikan laitos

Sampo Saaristo: Implementation of IS-IS Routing Protocol for IP versions 4 and 6

Diplomityö: 62 sivua, 20 liitesivua

Tarkastajat: Prof. Jarmo Harju ja DI Heikki Vatiainen

Lokakuu 2002

Tämä diplomityö esittelee IS-IS-reititysprotokollatoteutuksen, joka tukee Internet Protokollan (IP) versioiden 4 ja 6 reititystä. Työ tehtiin ICEFIN-tutkimuslaboratoriossa Tampereen teknillisen korkeakoulun tietoliikennetekniikan laitoksella.

IS-IS on alunperin ISO-standardisointiorganisaation toimesta määritelty link-state -reititysprotokolla. IS-IS-protokolla oli alunperin määritelty CLNP:n, OSI-mallin IP:tä vastaavan protokollan, reitittämiseen. IS-IS on kuitenkin määritelty siten, että sitä voidaan helposti laajentaa tukemaan minkä tahansa verkkokerroksen protokollan reititystä. Tuki IP -versio neljälle on määritelty vuonna 1990 ja IP -versio kuudelle vuonna 2000 IETF:n toimesta.

IS-IS reititysprotokolla toteutettiin osaksi GNU Zebra -reititysohjelmistoa. Ohjelmistoprojekti toimi SourceForge.net -sivustolla, joten kuka tahansa asiasta kiinnostunut pystyi osallistumaan projektiin.

Diplomityö käsittelee aluksi reititystä ja reititysprotokollia yleensä. Tämän jälkeen seuraa IS-IS-reititysprotokollan ja sen laajennusten esittely.

Ennen IS-IS-toteutuksen (ISISd) käsittelyä esitellään GNU Zebra -reititysohjelmisto. Diplomityön loppuosa käsittelee ISISd:n konfigurointia.

Table of Contents

List of Acronyms

ARPANET	Advanced Research Projects Agency NETwork
BGP	Border Gateway Protocol
CLNP	ConnectionLess Network Protocol
CLNS	ConnectionLess-mode Network Service
CPU	Central Processing Unit
CSNP	Complete SNP
CVS	Concurrent Versions System
DIS	Designated IS
EGP	Exterior Gateway Protocol
ES	End System
FQDN	Fully Qualified Domain Name
GNU	Gnu is Not Unix
GPL	GNU Genral Public License
ID	IDentifier
IETF	Internet Engineering Task Force
IGP	Interior Gateway Protocol
IP	Internet Protocol
IPv6	Internet Protocol version 6
IPX	Internetwork Packet Exchange
IS	Intermediate System
ISO	International Organization for Standardization
IOS	Internet Operating System
LSA	Link State Advertisement

LSP	Link State PDU
MTU	Maximum Transfer Unit
NET	Network Entity Title
NLPID	Network Layer Protocol ID
NLSP	Netware Link Services Protocol
NPDU	Network Layer PDU
OS	Operating System
OSI	Open Systems Interconnection
OSPF	Open Shortest Path First
PDU	Protocol Data Unit
PSNP	Partial SNP
RFC	Request For Comments
RIP	Routing Information Protocol
SIN	Ships In the Night
SNP	Sequence Numbers PDU
SNPA	SubNetwork Point of Attachment
SPF	Shortest Path First
SPT	Shortest Path Tree
TLV	Tag Length Value
TTY	TeleTYpe
VTY	Virtual TTY

1 Introduction

This Master's thesis describes an implementation of Intermediate System to Intermediate System (IS-IS) routing protocol. This thesis was done for the ICEFIN Research Laboratory at Tampere University of Technology, Institute of Communications Engineering.

IS-IS was standardized by International Organization for Standardization (ISO) and was originally intended to be used with Connectionless Network Protocol (CLNP). Internet Engineering Task Force (IETF) also wanted a link-state routing protocol for their Internet Protocol (IPv4) and began to develop the Open Shortest Path First (OSPF) protocol. At the same time IS-IS was extended to support IPv4 (also by IETF), which was easy due to its multiprotocol nature. It took years before OSPF could be deployed, thus IS-IS was adopted by many Internet Service Providers (ISPs). All of this happened at the beginning of 1990's. Now, a decade later history seems to repeat itself, with IP version 6 (IPv6). OSPFv3, OSPF for IPv6, has been specified by IETF, but it seems to take long before it can be deployed, because it is in a sense a completely new protocol. Again IS-IS can easily be extended, this time for IPv6. The biggest routing vendors have begun to offer IS-IS with IPv6 support before OSPFv3 protocol.

The motivation for implementing the IS-IS routing protocol (ISISd) was to provide the open source community with a version of IS-IS that supports IPv6, and thus in a small part help in the deployment of IPv6. One area of interest is the feasibility of an integrated routing protocol approach, which means the usage of a single routing protocol for multiple layer three protocols.

This thesis first discusses routing, routing protocols and the history of routing protocols. Chapters 3 and 4 introduce the IS-IS routing protocol and its extensions. The implementation of IS-IS was done as a module for the GNU Zebra routing software, which is described in Chapter 5. Chapter 6 deals with the implementation of IS-IS routing protocol (ISISd) and Chapter 5 the configuration of ISISd. Finally the conclusions are drawn in Chapter 8.

2 Routing and routing protocols

Free OnLine Dictionary Of Computing (FOLDOC) [2] defines routing as *the process, performed by a router, of selecting the correct interface and next hop for a packet being forwarded.* The selection is based on the information in *forwarding database (a.k.a. route table)*, which can be maintained manually and by a *routing protocol*.

Routes inserted manually are called *static routes*, whereas routes learned through routing protocols are called *dynamic*. Static routes can be used in small networks or in places which require precise control of the routing behavior, otherwise dynamic routes are usually preferred, since static routes require manual reconfiguration every time the network topology changes.

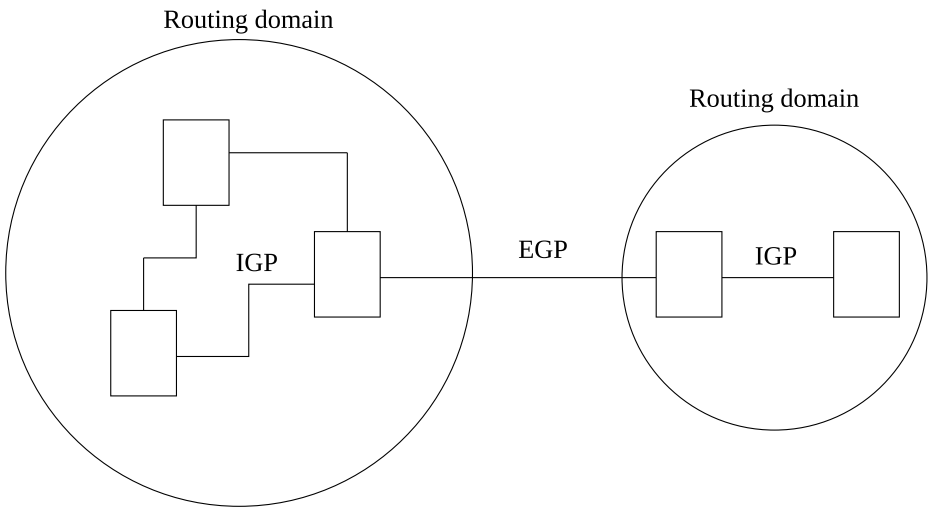

Figure 2.1: Interior/Exterior Gateway Protocols

All popular routing protocols are of type *link state* or *distance vector*. The idea of link state routing is that every router learns the topology of the whole network by exchanging link state information and can thus decide the best routes to every destination. The distance vector routing protocols implement a distributed computing of the *Bellman-Ford* algorithm, and share the results of the algorithm as vectors which hold the distance and direction of every destination. Distance vector routing is simpler to implement, but is does not scale to big networks as well as link state routing.

Routing protocols can also be grouped based on their scope. The term *Interior Gateway Protocol (IGP)* is used for protocols distributing routes inside a domain (also known as autonomous system) and the term *Exterior Gateway Protocol (EGP)* for protocols used to connect these domains (Figure 2.1).

2.1 Distance vector routing

The name *distance vector* comes from the fact that routing information is shared as vectors of the form *(ID, distance, direction)*. The *distance* can be just the plain hop count to reach the router denoted by *ID* or it can be the sum of *link costs* on the path towards it. *Direction* is the first hop of the path. This form is basically the same as in *forwarding database*, so we could state that distance vector routing is based on transmitting our route table to our neighbors.

The distance vector algorithm when applied to computer networks can be stated as follows. Each router is configured with a unique ID and a cost for each of its links. The distance vector is initialized with distance 0 for the router itself and *infinity* for every other destination. Every router transmits its distance vector every time the information changes. The most recent distance vector from every other router is saved. Each router calculates its own distance vector, by examining other router's distance vectors. The calculation is based on minimizing the cost to reach every other router. The distance vector is recalculated every time a different distance vector from a neighbor is received or state of a link to a neighbor changes.

Distance vector routing protocols suffer from slow convergence in some situations. The term *count-to-infinity* is used for the behavior, where after a topological change neighbors exchange distance vectors increasing the link cost to a neighbor that has become unreachable, until the distance reaches *infinity*, which is a parameter set by the system management.

A number of solutions to speed the slow convergence have been introduced. The term *hold-down* is used for the mechanism where the router waits for a *hold-down time*, after it has noticed that a path to a destination has gone down before switching to another path. During the *hold-down time* the router assumes that every other router has forgotten the old broken path. The scheme does not completely solve the *count-to-infinity* problem and actually slows the convergence in some cases.

Another popular method for speeding the convergence is called *split-horizon*. In this method, if router A reaches router C through router B, it reports to B its own cost to C as *infinity* and *counting-to-infinity* is avoided in many cases. [3]

2.2 Link state routing

The idea behind link state routing is simple. A router learns its neighbors (most often through a hello protocol) and constructs a Link State PDU (LSP) containing information about the neighbors. The LSP is then sent to every other router in the domain. This is depicted in Figure 2.2.

The most sensitive part of link state routing protocol is the handling of LSPs. Each router must have the latest correct LSPs from every other router in the domain, or unpredictable behaviour can occur. The LSPs are sent by a process of flooding, which means that when a router receives an LSP from link A, it propagates the LSP to every other link apart from A. Timestamps, checksums and sequence numbers are used to decide which LSPs are to be considered up-to-date and if the flooding is necessary.

When a router has all the LSPs of the domain in its link state database, it can build a shortest path tree (SPT), a tree of best paths to every other router in the domain, by running an SPT algorithm. The forwarding database (a.k.a the routing table) is then extracted from the SPT (Figure 2.3).

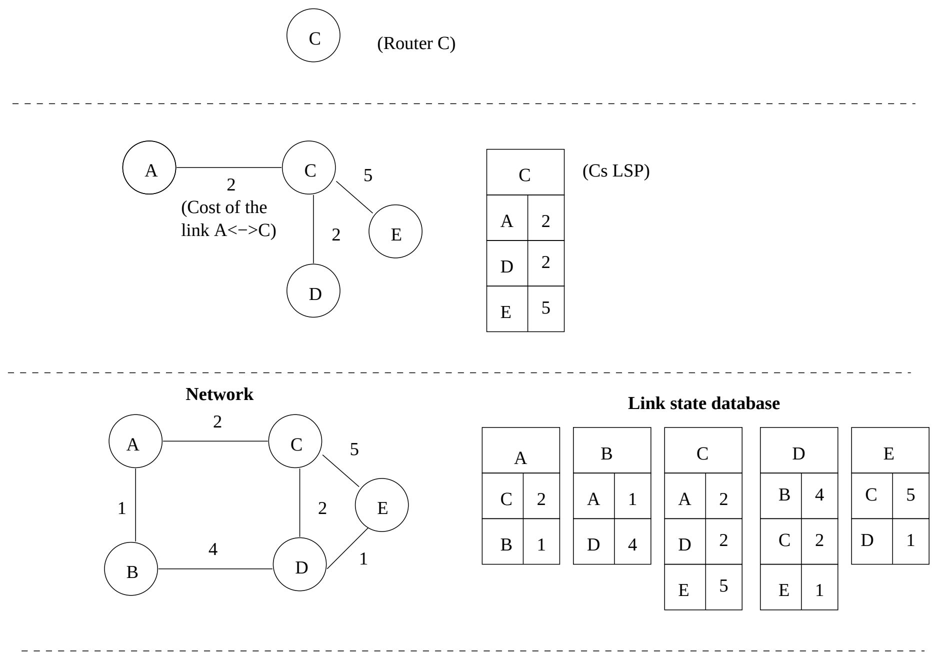

Figure 2.2: Building the LSP database

The most widely used SPT algorithm with link state routing protocols is Dijkstra. The Dijkstra algorithm deals with four different databases, which are shown in Figure 2.3. It takes the *LSP database* as input, and provides *Forwarding database* as output. *TENT* and *PATH* databases are used during the algorithm execution.

The *LSP database* contains a full set of LSPs from every router in the domain. Each LSP has a list of the neighboring routers and a cost to reach those routers from the router that originated the LSP.

The *TENT* (tentative) database is built of entries of the form *(ID, distance, direction)*, where

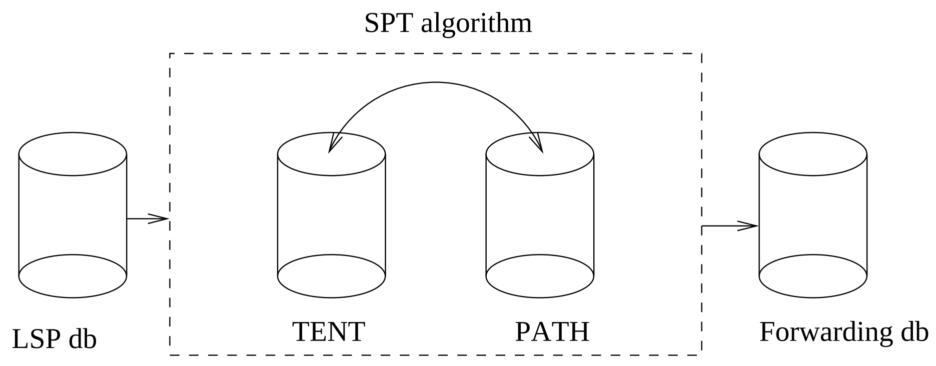

Figure 2.3: Forwarding database from the link state database

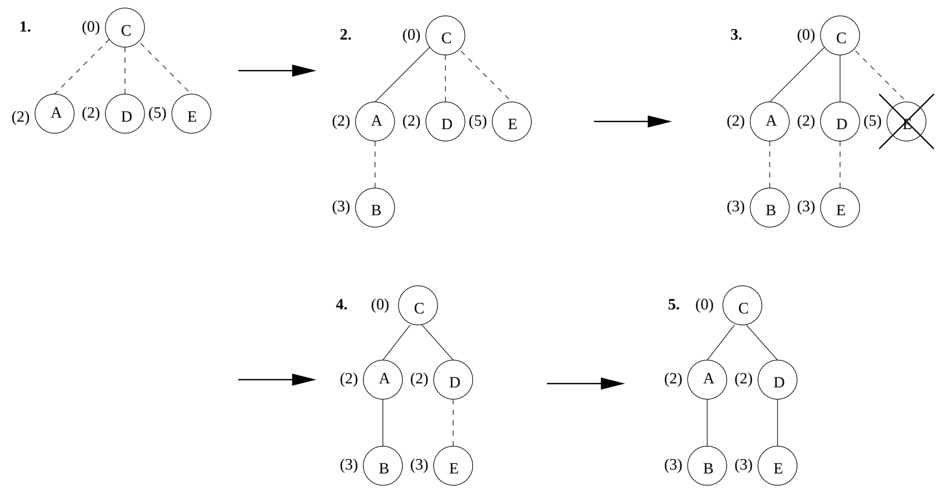

Figure 2.4: Router C executes Dijkstra

distance is the sum of costs from the current router to the destination router denoted by *ID*, and *direction* is the first hop along the path to reach that destination router.

PATH has the same form as *TENT*. If an entry is placed in *PATH* it means that no better path to reach the router with given *ID* exists. In other words *PATH* is the SPT.

In the end *Forwarding database*, which consists of entries of the form *(ID, direction)*, is extracted from *PATH*.

The algorithm can be stated as follows. (Step 1): start by adding this router with distance 0 to *PATH*. (Step 2): for router just placed in *PATH*, examine its LSP. For each neighbor listed in the LSP, compute the distance from this router. If the neighbor is not in *PATH* or *TENT* with better distance, place the neighbor in *TENT*. (Step 3), if *TENT* is empty, stop. Otherwise find the entry with minimal distance from *TENT*, place it to *PATH* and go to step 2.

Figure 2.4 shows how router C (from Figure 2.2) executes the Dijkstra algorithm. A dashed line means that the router is placed into *TENT* and a solid one that the router is placed into *PATH*. The number in parentheses beside each router is the distance from router C. The execution is as follows:

1. Place C into *PATH* (step 1). Place C's neighbors (A, D and E) into *TENT* (step 2).
2. Place A into *PATH* (step 3). Examine A's LSP, place router B into *TENT* (step 2).
3. Place D into *PATH* (step 3). Examine D's LSP, better path to E found (step 2).

4. Place B into *PATH* (step 3). No actions in step 2.
5. Place E into *PATH* (step 3). No actions in step 2. *TENT* is empty, stop (step 3).

2.3 A short history of intradomain routing protocols

Routing protocols were already used in the Internet's predecessor, the ARPANET. The first routing protocols were based on the Bellman-Ford algorithm and they included DECnet PHASE IV routing algorithm [4] and RIP [5], the latter being still in wide use.

The first link-state routing protocol, specified in [6], was also developed for use in the ARPANET, and extended in [7], which became the DECnet PHASE V routing algorithm. This protocol was adopted by ISO and named IS-IS [8]. At this time the IETF also wanted a link-state routing protocol to be used with IPv4, and started to develop their own protocol named Open Shortest Path First (OSPF) [9]. While OSPF was being specified, IS-IS was extended to support IPv4 in RFC 1195 [10]. It took years before OSPF was ready to be deployed, so IS-IS was adopted by many ISPs, while the customers used RIP in their own networks. Both IS-IS and OSPF are currently widely used and routing vendors commonly support both. [3]

2.4 Routing protocols for IPv6

No completely new routing protocols for IPv6 exist; instead the routing protocols for IPv6 are new versions or extensions to the ones available for IPv4.

The IPv6 version of RIP, called RIPng, is defined in RFC 2080 [11]. The document specifies the minimum changes to RIPv2 [12] for it handle IPv6 routing. It should however be noted that RIPng is a new protocol in a sense that it is not designed to be compatible with RIP. The limitations of the distance vector protocols have been recognized in this specification, and the protocol is thus specified to have maximum hop count of 15.

For OSPF an IPv6 version of the protocol has been defined in RFC 2740 [13], the protocol is called OSPFv3 and sometimes OSPF6, the latter being perhaps a better term since the protocol is not compatible with the OSPFv2 protocol [9]. The OSPFv3 protocol runs on per-link basis (as IS-IS) instead of per-IP-subnet basis, as OSPFv2. Also the addressing semantics have been removed from the OSPF protocol packets, so that the IPv6 addresses are only present in LSA payload (as in IS-IS PDUs). The neighboring routers are identified with a 32-bit *Router ID* instead of IP address (as with the *System ID* in IS-IS).

The support for IPv6 in IS-IS is introduced in an Internet Draft [14]. The document is only six pages long, since all that is needed are two new Tag-Length-Value (TLV) types.

The interdomain routing protocol BGP-4 [15] is also extended to support IPv6. In RFC 2858 [16] BGP-4 is extended to support multiple layer three protocols and in RFC 2545 [17] the procedures to carry IPv6 reachability information are defined. The resulting protocol that supports these extensions is usually referred to as BGP-4+.

3 IS-IS Routeing Protocol

IS-IS is defined by ISO in ISO/IEC 10589 [8]. The full name of IS-IS: *Intermediate System to Intermediate System Intra-Domain Routeing Exchange Protocol for use in Conjuction with the Protocol for Providing Connectionless-mode Network Service*, states the fact that IS-IS was originally intended to be a routing protocol for CLNP [18], which is the OSI stack equivalent to IP [19]. However, the protocol is designed in such a manner that it can easily be extended to support routing of any layer three protocol. The extensions for IP support are defined in RFC 1195 [10], which defines how IS-IS can be used as routing protocol in IP-only, OSI-only or dual environments. This chapter gives first an overview of ISO/IEC 10589 and its companion document RFC 1195. Lastly other IETF extensions are discussed.

Since the standard is defined by ISO the terms differ from the ones common in the Internet. We honour the original terms here. For readers' convenience, some of the terms and their synonyms are given in table 3.1.

ISO	**IETF**
adjacency	neighbor
circuit	interface
ES	host
IS	router
routeing	routing

Table 3.1: Terminology

In order to support large routeing domains IS-IS routeing has been organized hierarchically. A *routeing domain* can be divided in to *areas*. Routeing inside an area is referred to as *Level 1 routeing* and between the areas as *Level 2 routeing*.

3.1 Subnetwork Independent Functions

As the name suggests, *Subnetwork Independent Functions* are independent of the specific data link service underneath them. Only the broad separation of networks to two generic types of subnetworks: *General Topology Subnetwork* and *Broadcast Subnetwork*, is applied.

The standard identifies two subnetwork independent functions: *Routeing* and *Congestion Control*.

Congestion Control refers to the management of local resources by an IS. The transmission of IS-IS PDUs is controlled by timers and queues, in order to keep the amount of traffic low, and actions to be taken if the system runs out of memory are specified.

The *Routeing* function takes care of routeing an NPDU (Network Layer PDU) along a determined path. The *Routeing* function can be further divided into four processes: *Decision*, *Update*, *Forwarding* and *Receive Process* (Figure 3.1).

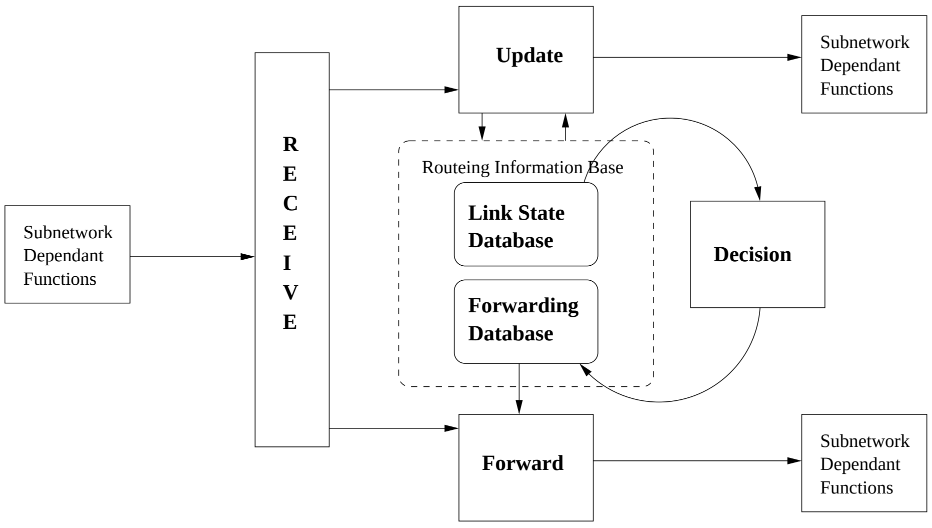
Update
Subnetwork Dependant Functions
RECEIVE
Routeing Information Base
Link State Database
Forwarding Database
Decision
Subnetwork Dependant Functions
Forward
Subnetwork Dependant Functions

Figure 3.1: Subnetwork Independent Functions

Decision Process is responsible for calculating routes to each destination in the domain. It is executed separately for Level 1 and Level 2 routeing, and separately for each supported routeing metric within a level. An IS uses the *Link State Database* to determine shortest paths to all other systems in the routeing domain.

Update Process maintains the *Link State Database*, by receiving, constructing and propagating Link State and Sequence Numbers PDUs. An LSP holds information about the adjacencies and their metric values seen by the system that originated the LSP. A system sends its own LSPs periodically and when the information in the LSP has changed. Other systems' LSPs are propagated by flooding the LSP to every other circuit with the same level than the one it was received from. Level 1 LSPs are propagated to every IS within the area and Level 2 LSPs to every Level 2 IS in the domain.

Forwarding Process supplies and manages the buffers needed for relaying NPDUs. The NPDUs are received via the *Receive Process*. The *Forwarding Process* performs lookups to the *Forwarding Database* in order to select appropriate output adjacency for forwarding the NPDU towards its destination.

Receive Process multiplexes the PDUs based on the NLPID value and performs the appropriate actions (e.g. passes the LSP PDUs to the *Update Process*).

3.1.1 Area address and System ID

The *Network Entity Title (NET)* is depicted in Figure 3.2. Every IS and ES has at least one NET. Every system holds exactly one value for *ID* portion or *System ID*, but it may be configured with more than one *Area Addresses*. The *ID* must be unique within an area. If two systems share one or more *Area Addresses*, they belong to the same area.

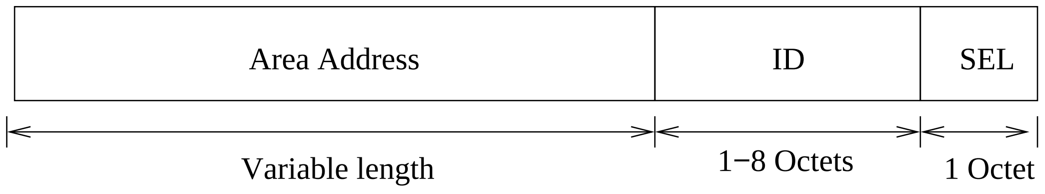

Figure 3.2: Network Entity Title

Even if operating in a pure IP environment, the IS must be configured with a NET. The *ID* is used to identify the IS and the *Area Address* the routeing area.

3.1.2 Two-level hierarchy

IS-IS provides a two-level hierarchy. The use of this hierarchy is clear when routing CLNP, since Level 1 routeing is based on the *ID* portion of the OSI address and Level 2 routeing deals with the *Area Address* portion of NET, and treats it as a prefix, so that the longest matching prefix is selected. The Level 2 has actually multiple levels since a number of hierarcies can be built with the *Area Address* part, so that a shorter prefix means a higher place in the hierarchy.

3.1.3 Handling of LSPs

Since the size of the LSP is limited by the parameter *ReceiveLSPBufferSize*, derived from the minimum MTU of the circuits participating in routeing, an IS may have to divide the single logical LSP to multiple LSPs. This is achieved by the use of last octet of *LSP ID* field called *LSP Number* (Figure 3.10). The LSP with *LSP Number* zero is handled in a special way; the values of the *LSP bits* field are only taken into account on zero LSP and the LSPs with *LSP Number* greater than zero are not considered valid if the zero LSP is not present with non-zero *Remaining Lifetime*.

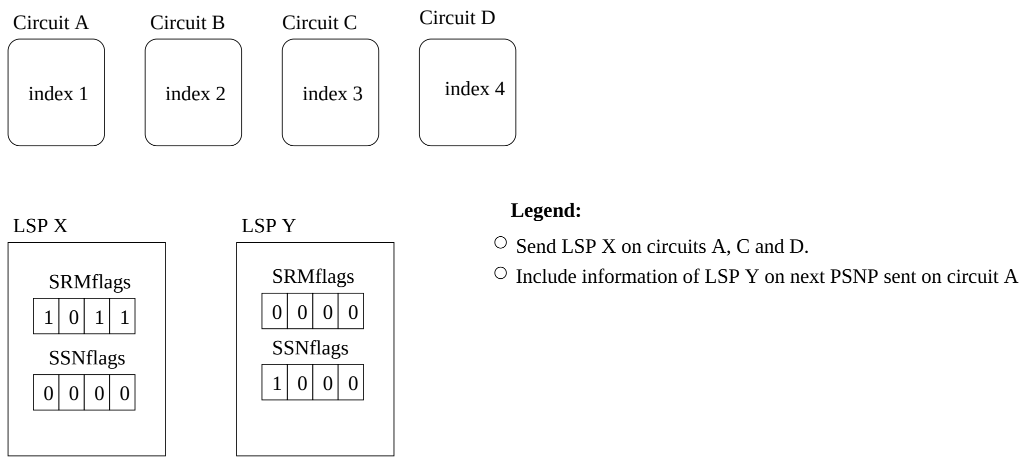

Figure 3.3: Use of SRM- and SSNflags

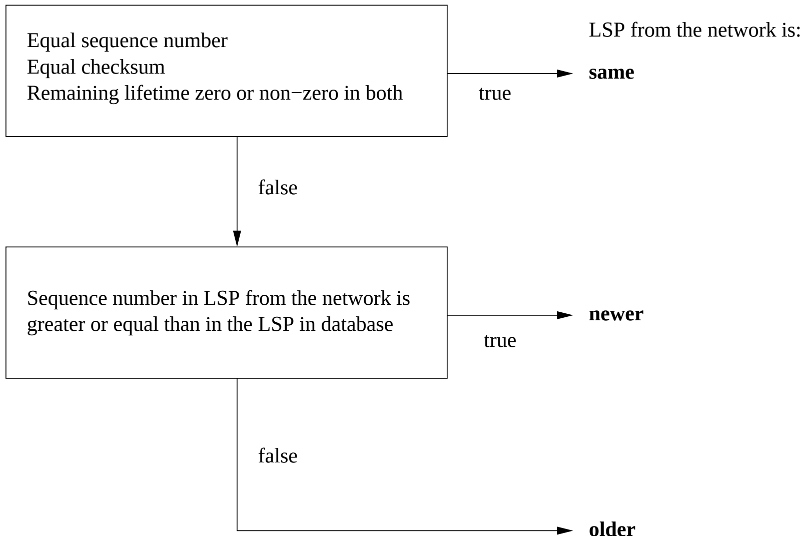
Equal sequence number
Equal checksum
Remaining lifetime zero or non−zero in both
true
LSP from the network is:
same
false
Sequence number in LSP from the network is greater or equal than in the LSP in database
true
newer
false
older

Figure 3.4: Comparison of LSPs

The sending of LSPs is controlled by *Send Routeing Message - SRM flag*. Each LSP is associated with an array of *SRM flags* with an index for each circuit. If the flag is set, it means that the LSP should be transmitted on the circuit corresponding the index of the flag. The *Send Sequence Number - SSN flag* is used in the same manner, to denote that the information of the LSP in question should be included in the next PSNP sent on the circuit. The use of these flags is illustrated in Figure 3.3.

An IS is responsible for maintaining the latest information in its LSP database. When an LSP is received it must be able to determine whether the received LSP is identical to, older or newer than the one in its database. The comparison is based on the values of *Remaining Lifetime*, *Sequence Number* and *Checksum* fields as depicted in Figure 3.4.

3.1.4 Partition repair

The standard describes a method for repairing a Level 1 area that has become partitioned due to failure of one or more links in the area. The partitions are connected with a *Level 1 repair path* through the *Level 2 subdomain* (Figure 3.5). To do this a *Partition Designated IS* must be elected on each partition, which takes care of forwarding the NPDUs destined to other partitions of the area through the repair path. Support for this method is optional. An IS that supports the partition repair option, sets a bit on its LSPs to inform other ISs that this is the case.

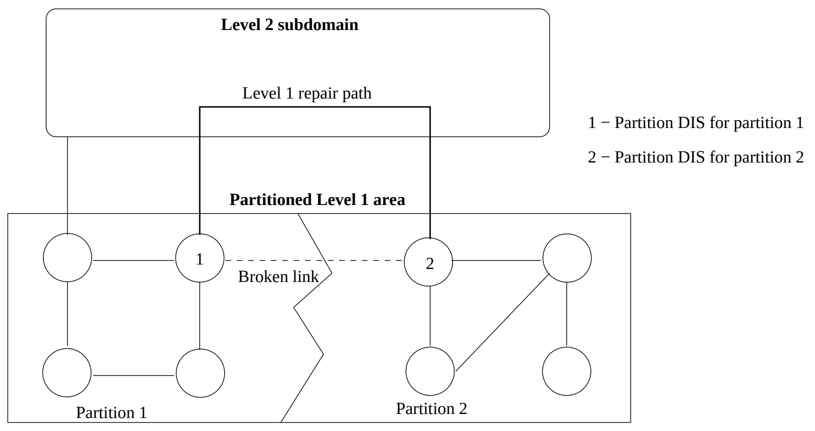

Figure 3.5: Repair of partitioned Level 1 area

3.2 Subnetwork Dependent Functions

The *Subnetwork Dependent Functions* hide the characteristics of different kinds of subnetworks from the *Subnetwork Independent Functions*.

Subnetwork Dependent Functions include reception and transmission of IS-IS PDUs over a specific subnetwork and exchanging the *IS to IS Hello* PDUs in order to establish intermediate and end system adjacencies.

3.2.1 LAN designated intermediate system

In order to reduce the amount of routeing traffic, broadcast subnetworks are handled in a special way. Each LAN is represented as a *pseudonode*. The *designated intermediate system (DIS)* that is acting as a *pseudonode* is elected by the ISs on the LAN based on the value of *priority field* in *IS to IS LAN Hello PDUs*. Level 1 and Level 2 intermediate systems elect the *DIS* separately. If two or more systems have the same value for the *priority* field, a tie-breaker is used by comparing the *SNPA* address, so that the numerically higher *SNPA* has a bigger priority of becoming the *DIS*.

Each IS on the LAN reports only a single link to a *pseudonode*, rather than reporting a link to every other IS on the broadcast subnetwork (see Figure 3.6). The *DIS* then constructs an LSP for the whole LAN.

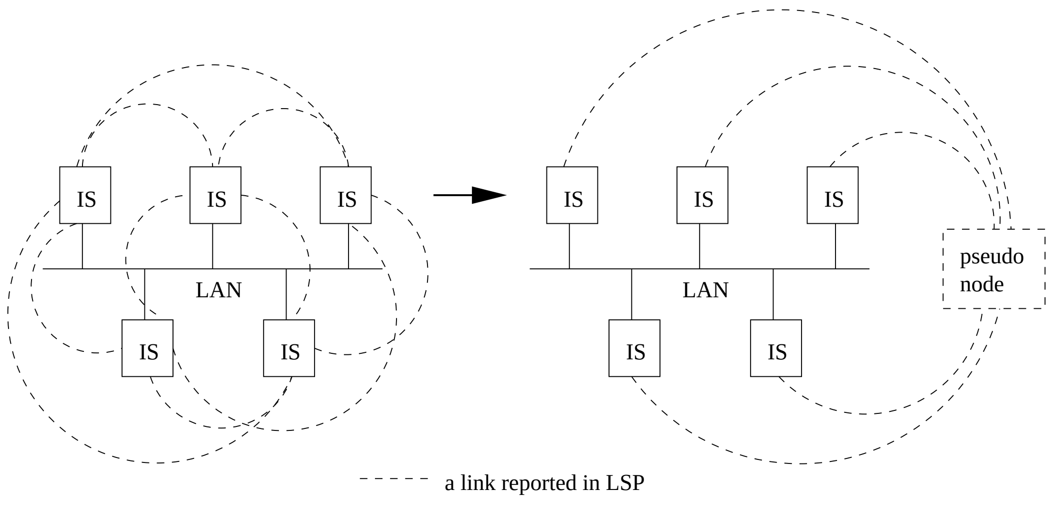

Figure 3.6: LAN pseudonode

3.2.2 Maintaining adjacencies

Each IS learns its neighbors by exchanging *IS to IS Hello (IIH) PDUs* .

On non-broadcast subnetworks that are not marked (with *external domain* attribute) as belonging to an external domain, *point-to-point IIH* PDUs are transmitted, whenever a *point-to-point IIH* PDU is received or the *iSISHelloTimer* expires.

On broadcast subnetworks *Level 1 and Level 2 LAN IIH PDUs* are exchanged to form Level 1 and Level 2 adjacencies. The information of adjacencies on state *UP* is then included in LSPs. The adjacency is accepted if the neighboring IS has the same maximum value for

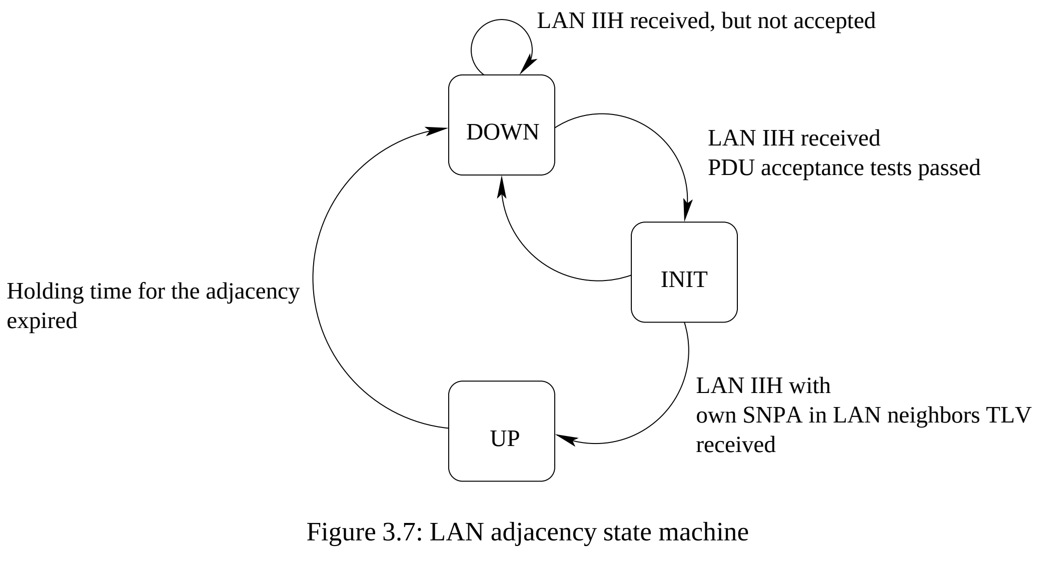
LAN IIH received, but not accepted
DOWN
LAN IIH received
PDU acceptance tests passed
INIT
Holding time for the adjacency expired
LAN IIH with
own SNPA in LAN neighbors TLV
received
UP

Figure 3.7: LAN adjacency state machine

number of Area Addresses and the same System ID length. In addition Level 1 adjacencies are accepted only if a match in one or more Area Addresses is found. If the *LAN IIH PDU* passes the acceptance tests just described, the adjacency is set to *INIT* state. To ensure two-way connectivity an adjacency is set to *UP* state, only if the neighboring IS reports our *SNPA* as its neighbor in the *LAN Neighbors TLV*. A simplified state machine for LAN adjacencies is shown in Figure 3.7.

3.3 Format of IS-IS PDUs

The nine different PDU types and their type codes are listed in Table 3.2. The PDUs are of three basic types, namely *Hello PDUs*, *Link State PDUs (LSPs)* and *Sequence Number PDUs (SNPs)*. *Hello PDUs* are used between adjacent ISs to inform and learn of each other's existence and capabilities. *LSPs* are used to tell other ISs about the adjacencies and the state of the adjacencies the IS in question has. *SNPs* contain information about *LSPs*. They are sent as acknowledgements and to inform other ISs of the contents of our *link state database*.

PDU Type	**Type code**
Hello	
Level 1 LAN IS to IS Hello PDU	15
Level 2 LAN IS to IS Hello PDU	16
Point-to-Point IS to IS Hello PDU	19
LSP	
Level 1 Link State PDU	18
Level 2 Link State PDU	20
SNP	
Level 1 Complete Sequence Numbers PDU	24
Level 2 Complete Sequence Numbers PDU	25
Level 1 Partial Sequence Numbers PDU	26
Level 2 Partial Sequnce Numbers PDU	27

Table 3.2: IS-IS PDU types

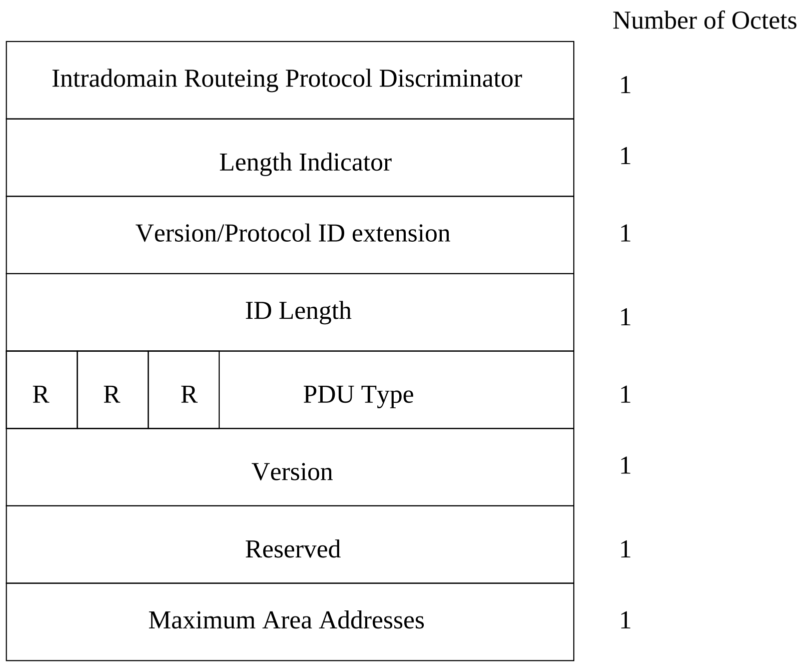

Figure 3.8: Common part of IS-IS fi xed header

All nine PDU types have the same format for the first eight octets (Figure 3.8). The *Intradomain Routeing Protocol Discriminator* field holds the value 131, as defined for IS-IS in ISO 9577 [20] and *Length Indicator* field is set to the length of fixed header in octets. *Version/Protocol ID Extension* and *Version* fields are always set to 1. The *PDU Type* holds one of the values listed in table 3.2.

All ISs in a routing domain must have the same values for *ID Length* and *Maximum Area Addresses*. The *ID Length* tells the length of System ID (Section 3.1.1) and possible values are 0-8 and 255, where 0 means that the default value of 6 and 255 that null length ID fields are used. The *Maximum Area Addresses* field holds the maximum number of area addresses permitted. Possible values are 0-254, where 0 means that the default value of 3 is used.

3.3.1 Hello PDUs

LAN IS to IS (LAN IIH) PDUs are sent on broadcast circuits in order to learn about neighboring ISs. The *Level 1 LAN IIH PDUs* are sent to Multi-destination Address *AllL1Iss* (`0x01-80-C2-00-00-14` in Ethernet) and *Level 2 LAN IIH PDUs* to *AllL2Iss* (`0x01-80-C2-00-00-15`).

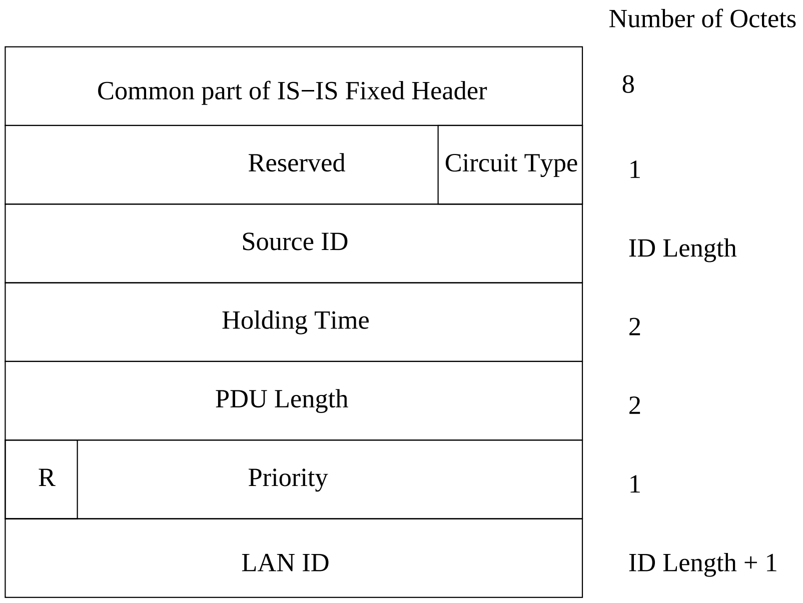

Figure 3.9: Level 1 and 2 LAN IS-IS Hello PDU fixed header

Figure 3.9 shows the format for *LAN IIH PDUs*. The format of fixed header is the same for both *Level 1* and *Level 2 LAN IIH PDUs*. *Circuit Type* tells the level of the IS. Value 1 means Level 1 IS, value 2 Level 2 IS, and value 3 that both levels are supported. In *Level 1 LAN IIH PDU* the value is 1 or 2 and in *Level 2 LAN IIH PDU* 2 or 3.

Holding Time is the time in seconds the adjacency for this IS should be considered valid and *PDU Length* holds the entire length of the PDU in octets. The *Priority* is set to value 0-127. Greater value means higher priority for the transmitting IS for becoming the *LAN Designated IS (DIS)* on the circuit. *Source ID* is set to the *system ID* of the transmitting IS and *LAN ID* is the *system ID* of the *LAN DIS*, appended with an octet assigned by the *DIS*.

Point-to-point IIH PDUs are transmitted on nonbroadcast circuits. The only differences to *LAN IIH PDUs* are that the *Priority* field is not present and that the *LAN ID* field is replaced by 1 octet *Local Circuit ID*, assigned to this circuit when it is created.

3.3.2 Link State PDUs

Level 1 Link State PDUs (LSPs) are flooded by *Level 1* and *Level 2* ISs on the area to inform other *Level 1* ISs of the state of the *Level 1* adjacencies the originating IS has. *Level 2 LSPs* are sent by *Level 2* ISs throughout the *Level 2* domain, to inform other *Level 2* ISs of the

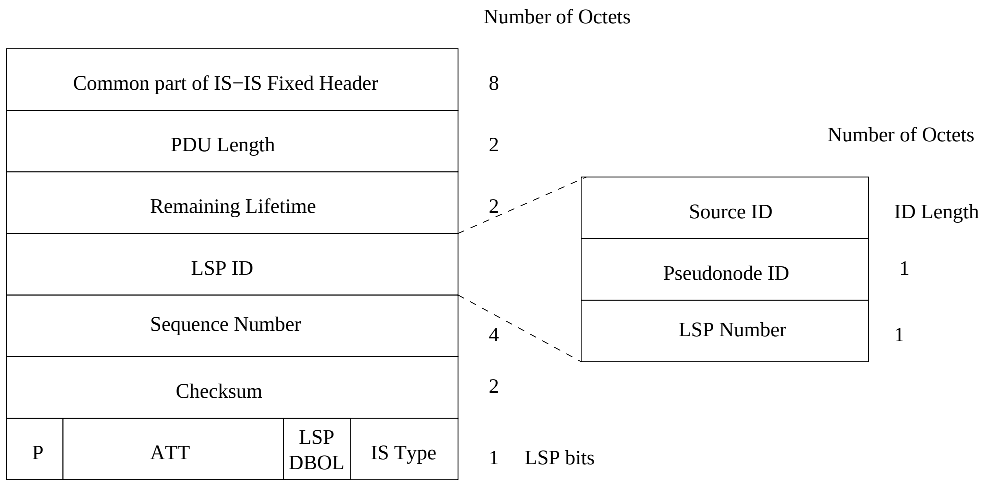

Figure 3.10: Level 1 and Level 2 Link State PDU Fixed header

Level 2 adjacencies and address prefixes reachable through the advertising IS.

Level 1 and Level 2 LSPs have similar formats for the fixed header (Figure 3.10). *PDU Length* is the length of the entire PDU in octets and*Remaining Lifetime* is set to the number of seconds before the LSP is to be considered expired.

LSP ID consists of *System ID* and *Pseudonode ID* of the originating IS, appended with *LSP Number*. The *Pseudonode ID* is non-zero if this is an LSP originated by a *LAN Designated IS*. The *LSP Number* is used in LSP fragmentation.

Sequence Number holds the 32-bit sequence number of the LSP and *Checksum* is a 16-bit value calculated from *Source ID* to the end of the PDU. The calculation is based on the Fletcher checksum defined in ISO 8473 [18].

The *P* bit is set if this IS supports *Partition Repair* and *ATT* field indicates the type of service with which this IS is connected to other areas. The bits are from 4-7: Default, Delay, Expense and Error Metric. The *LSPDBOL* bit is set, if this IS is in the *LSP database overload* state. *IS Type* field indicates IS level. It is handled differently from the *Circuit Type* field in *IIH PDUs*, so that value 1 means *Level 1 IS* and value 3 *Level 2 IS*. Value 2 is not used.

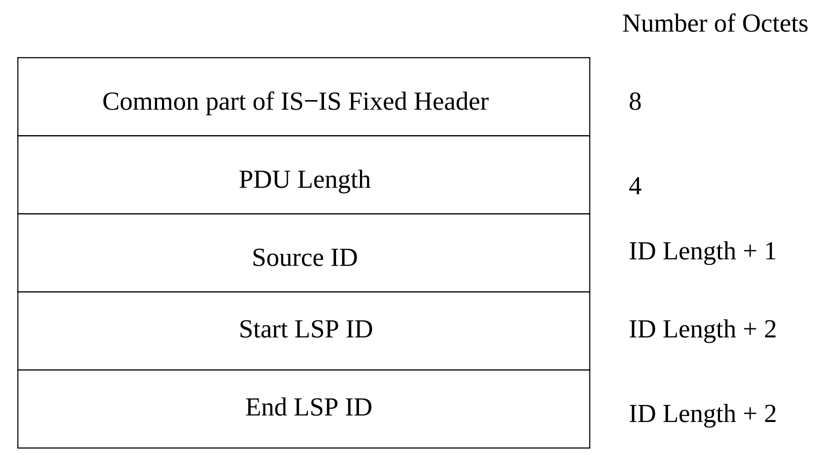

Figure 3.11: Level 1 and Level 2 complete sequence numbers PDU ﬁ xed header

3.3.3 Sequence Number PDUs

The *Level 1 and Level 2 complete sequence numbers (CSNP)* PDUs are sent to inform other intermediate systems of the contents of the transmitting IS's link state database.

Level 1 and Level 2 CSNPs have similar format for fixed header (Figure 3.11). *PDU Length* and *Source ID* have the same meaning as for example *IIH PDUs*.

The *Start LSP ID* and *End LSP ID* fields specify a range of LSPs reported in this *CNSP*. The LSPs are reported in the variable length part of the PDU as *LSP Entries* TLVs.

The *Level 1 and Level 2 partial sequence number (PSNP)* PDUs are sent as acknowledgments and to request LSPs from another IS. The format is shown in Figure 3.12. As with the *CSNPs* the variable length part contains the compact information of the LSPs in *LSP Entries* TLVs.

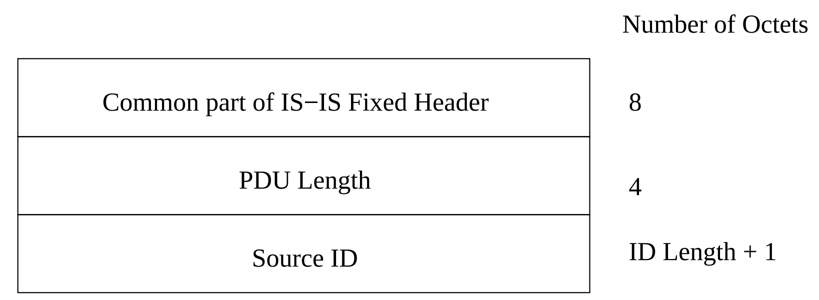

Figure 3.12: Level 1 and Level 2 partial sequence numbers PDU ﬁ xed header

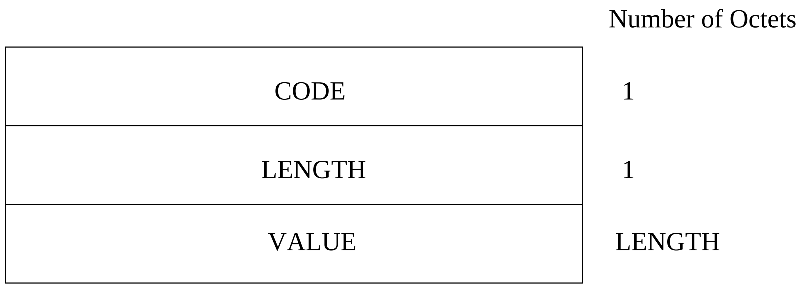

Figure 3.13: TLV format

3.3.4 TLVs

The variable length part of IS-IS PDUs consists of TLVs, which all have the format depicted in Figure 3.13.

The TLV names and their codepoints are shown in Table 3.3. The table also shows in which PDU types the TLV can be present. A short introduction to each of these TLV types follows.

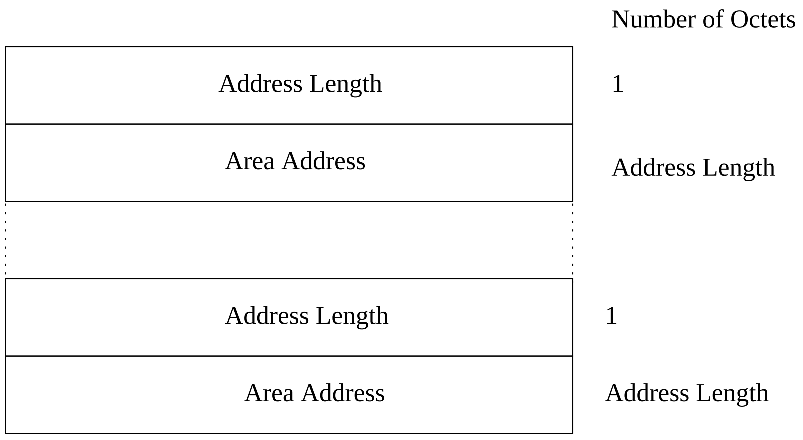

Figure 3.14: Area Addresses TLV

The *Area Addresses TLV* has the value part shown in Figure 3.14. The TLV contains the list of *Area Addresses* configured to the transmitting IS.

The *IS Reachability TLV* is used in LSPs to distribute reachability information (Figure 3.15). The *Virtual Flag* holds a boolean value. If set it means that this neighbor is actually a virtual Level 2 path to repair partitioned Level 1 area. The next four octets define the metrics to reach the IS denoted by *Neighbor ID*. The bit *S* before each metric is set if that metric type is unsupported and the bit *I/E* is set if the metric is external. The default metric must be supported, thus the bit *S* for default metric is always transmitted as zero.

TLV Name	TLV Code	Hello	LSP	SNP
Defined in ISO/IEC 10589				
Area Addresses	1	X	X	
IS Reachability	2		X	
ES Neighbors	3		X	
Partition DIS	4		X	
Prefix Neighbors	5		X	
LAN Neighbors	6	X		
LAN Neighbors	7	X		
Padding	8	X		
LSP Entries	9			X
Authentication	10	X	X	X
LSPBufferSize	14		X	
Defined in RFC 1195				
IP Internal Reachability	128		X	
Protocols Supported	129	X	X	
IP External Reachability	130		X	
IDRPI	131		X	X
IP Interface Address	132	X	X	

Table 3.3: TLVs defi ned in [8] and [10]

There are two types of *LAN Neighbors* TLVs. These TLVs are used in *LAN IIH* PDUs to inform about LAN neighbours. The one with code 6 contains simply a list of neighboring LAN addresses that are six octet long. If the media has a LAN address of a different length than six, the TLV with code 7 is used, which contains a length field before the list of LAN addresses.

The *ES Neighbors TLV*, is used in *Level 1 LSPs* to supply reachability information of End Systems. The only difference to *IS Reachability TLV* is that if the metric values are the same, more than one ES can be listed after the metric fields. The *ES Neighbors TLV* is not used, when operating in an IP-only mode.

Partition Designated IS TLV is present in *Level 2 LSP*, if the area is partitioned. The support

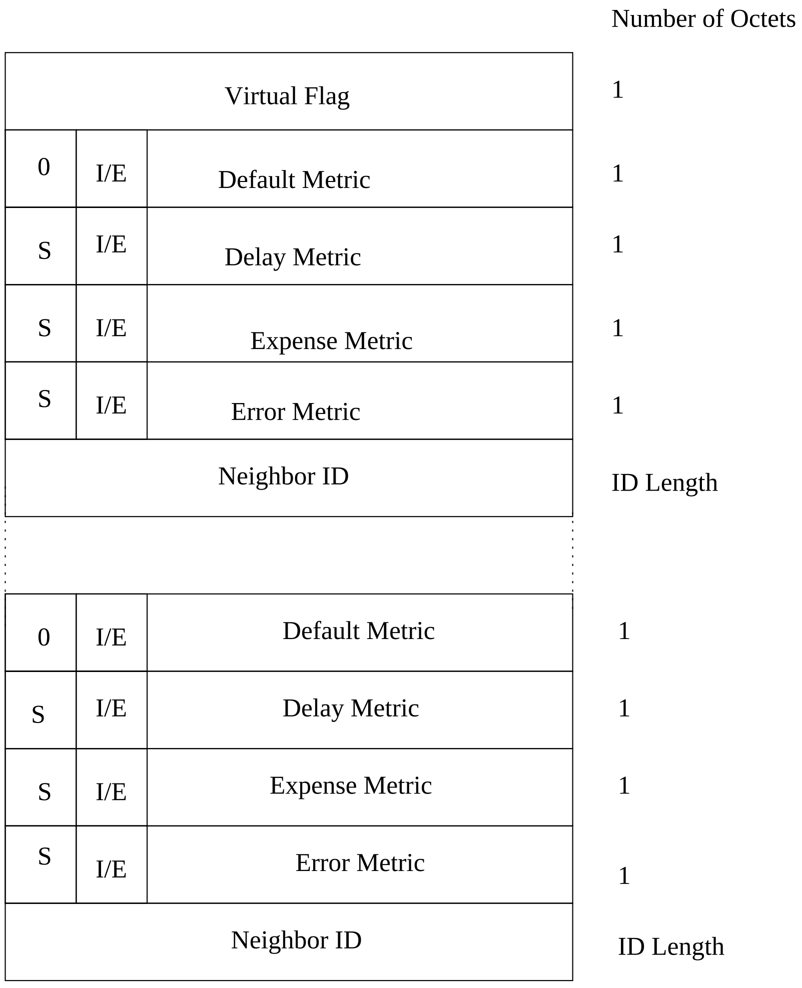

Figure 3.15: IS Reachability TLV

for this TLV as for the partition repair function is not mandatory. The TLV holds the System ID of the *Partition Designated IS*.

The *Prefix Neighbors TLV* is used in *Level 2 LSPs* to propagate (OSI) address reachability information. The TLV has the format depicted in Figure 3.16. The *Address Prefix Length* is the length of the *Address Prefix* in semi-octets. The *Address Prefix* is padded with null half-octets, if the *Address Prefix Length* is odd. This TLV is omitted in an IP only operation.

The *Padding TLV* is used in *IIH Hello PDUs* to pad the PDU to the maximum size of the underlying media. The PDU holds padding from 0-255 octets of arbitary value. This TLV is used to ascertain that neighbors have been configured with same MTU.

LSP Entries TLV is used in *Sequence Numbers PDUs*, to provide information of the LSPs

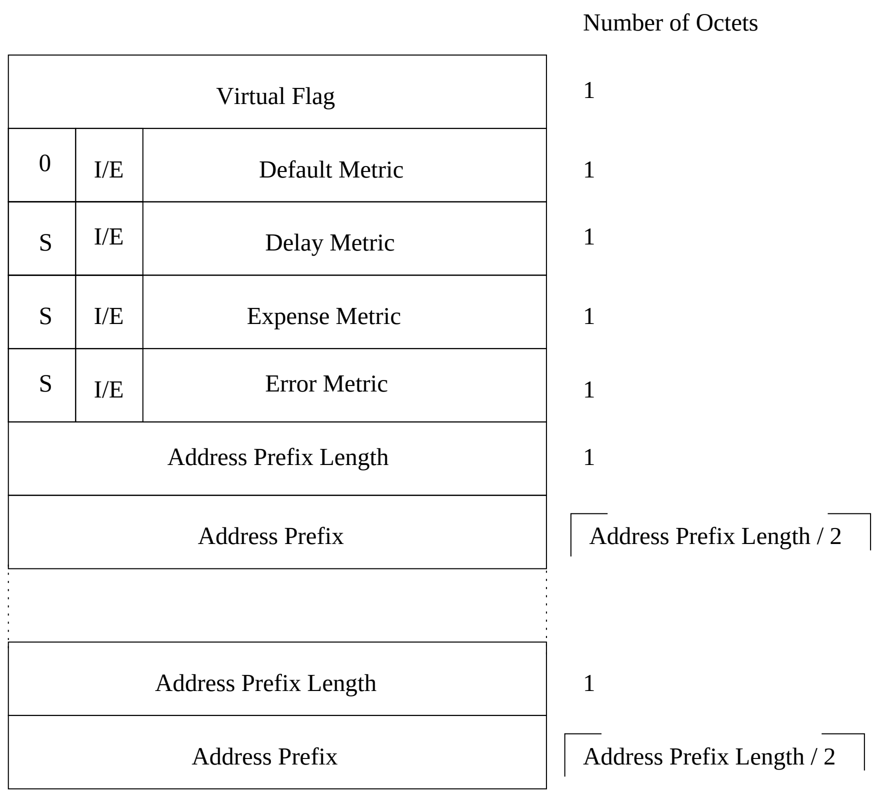

Figure 3.16: Prefi x Neighbors TLV

in a compact form. The *LSP Entries TLV* has the format shown in Figure 3.17.

The *Authentication TLV* has a simple form where the *Authentication Type* field specifies the used authentication method followed by the *Authentication Value*. Both [10] and [8] specify only a cleartext password authentication (*Auhentication Type 1*). Encrypted authentication is proposed in an Internet Draft [21].

The *LSPBufferSize TLV* is an optional TLV, which can be included in any LSP to inform other intermediate systems of the buffersize used by the originating IS. The values range from 512 to 1492.

The rest of this subsection deals with TLVs defined in RFC 1195.

The *Protocols Supported TLV* has a simple form, where the value part consists of one octet *Network Layer Protocol Identifiers (NLPID)*, having values registered in ISO 9577 [20]. This TLV is used to tell other intermediate systems which layer 3 protocol the transmitting IS is prepared to route.

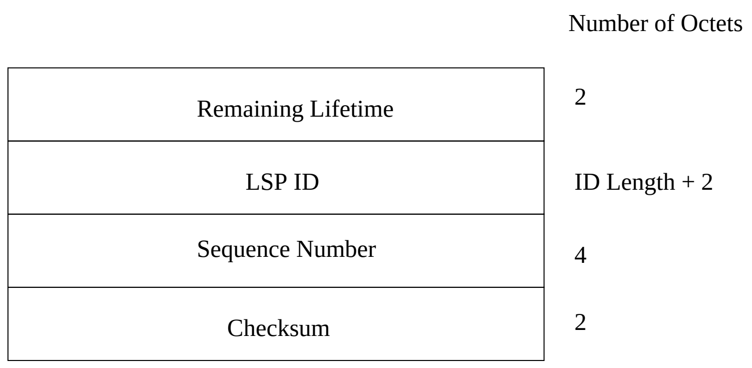

Figure 3.17: LSP Entries TLV

The *IP Interface Address TLV* consists of the four octet IP address(es) of the intermediate system that originated the PDU.

The *IP Internal Reachability TLV* and *IP External Reachability TLV* have the same format which is the same as the format of *IS Reachability* except that the *Neigbor ID* field is replaced by *IP Address* and *Subnet Mask* fields. The *IP Internal Reachability* is used in LSPs to inform other intermediate systems of the IP addresses within the routing domain, that can be reached through this IS. The *IP External Reachability TLV* can be present only in Level 2 LSPs and it is used to advertise IP addresses outside the routing domain that can be reached through the originating IS. The *I/E* bit is only present before the *Default Metric* field, and is set always to zero on *IP Internal Reachability TLV*.

3.4 IETF extensions

In addition to IP support defined in RFC 1195 the IS-IS working group on IETF has defined a number of extensions to IS-IS, which are now discussed. The discussion is limited to proposals that have reached the RFC state, with exception to IPv6 support, which is still defined in an Internet-Draft.

3.4.1 Dynamic hostnames

The dynamic hostname exchange mechanism is defined in RFC 2763 [22]. The RFC defines a TLV with type 137 that is used to carry the hostname in LSPs. With this mechanism the

System ID can be mapped to 1-255 character string, which is the symbolic name of the intermediate system that originated the LSP. The symbolic name is configurable by system administration, but it is recommended that the name is set to the *Fully Qualified Domain Name (FQDN)* of the IS.

3.4.2 IS-IS mesh groups

Support for mesh groups is defined in RFC 2973 [23]. The mesh groups are used to reduce the flooding of LSPs in a full mesh topology of point-to-point links, such as ATM and Frame Relay networks. In these kind of networks of N nodes, the standard LSP mechanism results in N-2 extra transmissions of each LSP.

The RFC defines two new attributes for every circuit: the *meshGroupEnabled*, which is in state *meshInactive*, *meshBlocked*, or *meshSet* and an integer variable *meshGroup*. The value of the *meshGroup* attribute is meaningful only if the attribute *meshGroupEnabled* is in state *meshSet*. Circuits that are in state *meshSet* and that have the same value of *meshGroup* are said to belong to the same mesh group. LSPs are not flooded over circuits in *meshBlocked* state, and an LSP received on a circuit C is not flooded out circuits that belong to C's mesh group.

With numbered mesh groups a large full mesh topology can be divided into a smaller group

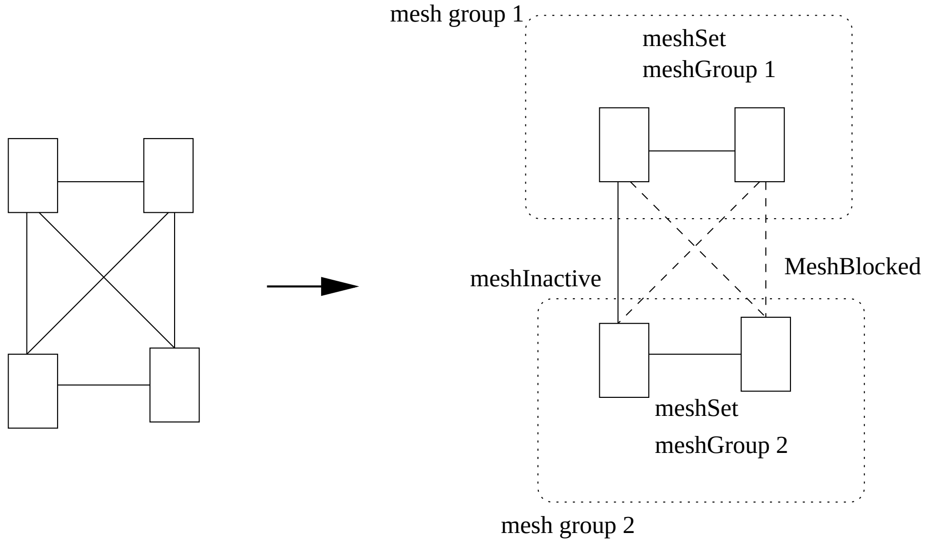

Figure 3.18: Full mesh topology divided into two mesh groups

of full mesh sub-topologies (mesh groups). These mesh groups are connected by "transit" circuits which are *meshInactive*. The remaining circuits between the mesh groups are configured as *meshBlocked* to reduce flooding redundancy. This is depicted in Figure 3.18.

It should be noted that the configuration of mesh groups is done manually and that both circuits at the ends the of point-to-point link should have the same values for the attributes. Suggestions have been made that these attributes could be negotiated or checked upon adjacency initialization.

3.4.3 Domain wide prefi x distribution

“Domain prefix distribution with two-level IS-IS” is an extension to RFC 1195, defined in RFC 2966 [24]. The RFC specifies how IP-prefixes can be distributed between Level 1 and Level 2 and vice versa.

In IS-IS, as defined in RFC 1195, Level 1 intermediate systems forward IP packets with destinations outside the Level 1 area to the nearest IS that is attached to Level 2. An IS that belongs to both levels should be manually configured with a summarization of the level 1 area, to be injected into Level 2. This leads to routes that are not necessarily optimal.

If the IP prefixes were distributed throughout the domain, as proposed in the RFC 2966, the optimal routes would always be found. However the domain wide prefix distribution affects the scalability since it adds memory consumption and SPT algorithm calculation complexity.

The RFC proposes that the high order bit of the default metric in *IP Internal and External Reachability* TVLs, which is defined to be 0 (reserved), is redefined to be an *up/down* bit. An IS that is inserting Level 2 prefixes into Level 1 LSPs, must set this bit (*down*). The bit must be clear (*up*) in all other IP prefixes in Level 1 and Level 2 LSPs. The prefixes with the *up/down* bit set that are learned via Level 1 routeing, must never be advertised back into Level 2, to avoid routing loops.

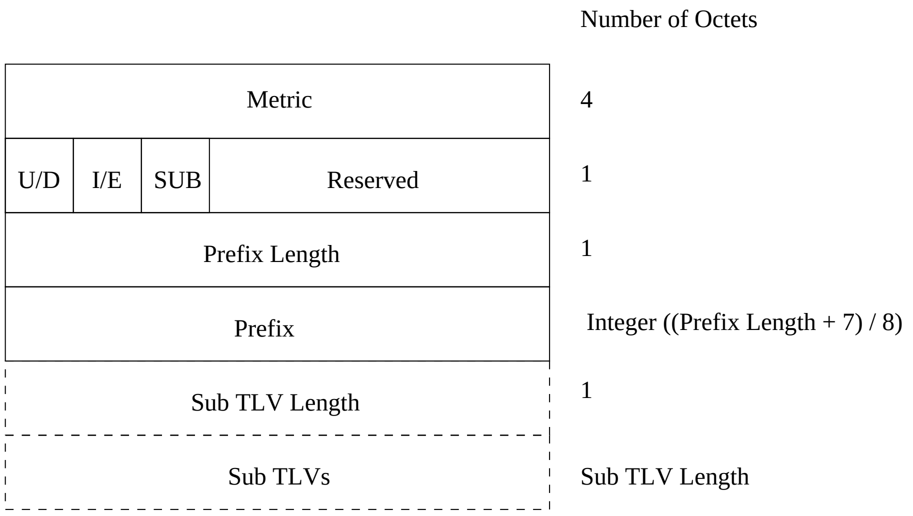

Figure 3.19: IPv6 Reachability TLV

3.4.4 IPv6 support

The extensions for IPv6 [25] support on IS-IS is proposed in an Internet-Draft [14]. The draft extends the IP support (defined in RFC 1195), with two new TLV types.

The *IPv6 Interface Address TLV* maps directly to the *IP Interface TLV* and consists of (0-15) 16 octet IPv6 addresses. In *IIH PDUs* this TLV must contain only the link-local IPv6 addresses assigned to the circuit which is sending the Hello. On LSP the TLV must contain only the non-link-local IPv6 addresses assigned to the intermediate system that originated the LSP.

The *IPv6 Reachability TLV* is shown in Figure 3.19. The TLV utilizes the extended metric field proposed in [26], where only the default metric is present but it is 32 bits long. The *Up/Down* bit has the semantics defined in [24] and *Internal/External* bit is used to tell whether the prefix was learned through IS-IS or some external routing protocol. The *Prefix Length* tells the length of the IPv6 Address *Prefix*, which is packed so that only the required number of prefix octets is present. If the *Sub options* bit is set, it means that optional *Sub TLVs* are present.

This method allows routing of both IPv4 and IPv6 using a single intra-domain routing protocol.

4 Integrated routing with IS-IS

The traditional and prevailing approach to routing of different network layer protocols is that each of these protocols has its own specific routing protocol. For example, we might use OSPF for IPv4, NLSP for IPX and IS-IS for CLNS. This approach is called “ships in the nigth (SIN)”, because each routing protocol operates on its own without any knowledge of other layer three and routing protocols.

The opposite to the SIN approach is integrated routing, where a single routing protocol handels the routing of multiple network protocols. RFC 1195 specified a dual-stack approach for OSI and TCP/IP with IS-IS known as Integrated IS-IS. The main advantage of integrated routing is that the integrated routing protocol does not have to compete for the recources (CPU, memory, bandwidth) with other routing protocols. Other advantages of integrated routing, according to the RFC, are that the integrated routing simplifies network management and can be configured to meet the strict timing requirements of link state routing more easily.

RFC 1195 allows mixing of pure-IP, pure-OSI and dual areas within a dual routing domain but restricts mixing of routers inside the areas so that for example in a pure-IP area within a dual routing domain only pure-IP and dual routers, i.e. routers with dual OSI and TCP/IP stack, can co-exist. Table 4.1 tries to clarify these resctrictions.

The topological restrictions placed by the RFC 1195 (Table 4.1) own to the fact that the SPT algorithm must find the correct paths for all supported layer three protocols. It might first seem that the restrictions cannot be avoided, but this is not the case.

router type / area type	pure IP	pure OSI	dual
pure IP	X		
pure OSI		X	
dual	X	X	X

Table 4.1: Mixing of router and area types within a dual routing domain

When we look at the information in IS-IS link state and adjacency databases, we notice that the information of which protocols are supported can be found from entries in both the databases. Also, RFC 1195 states that ISs are to become neighbors if they have a single matching protocol in the *Protocols Supported* TLV. Given this information ISs will operate correctly if they calculate the SPT separately for different protocols.

In order to find the correct paths to destinations in an area where some ISs support protocol X, some Y, and some both, the SPT would be run once per supported protocol per routing level. For example an IS that supports both IPv4 and IPv6 runs the SPT algorithm twice per level. First the IS takes only the LSPs and adjacencies that support IPv4 in consideration, when running the SPT, before doing the same for IPv6. This would not be integrated routing in a strict sense, but most, if not all, the advantages of integrated routing would still prevail.

Hints to a more advanced solution are given in section 3.8 of RFC 1195; a mechanism called automatic encapsulation is left for further study. Currently automatic encapsulation is defined in an Internet-Draft [27].

The draft introduces a new TLV type "encapsulation capability", which lists the protocols an IS supporting automatic encapsulation is able to encapsulate and unencapsulate. An IS supporting this feature adds the TLV in its LSPs.

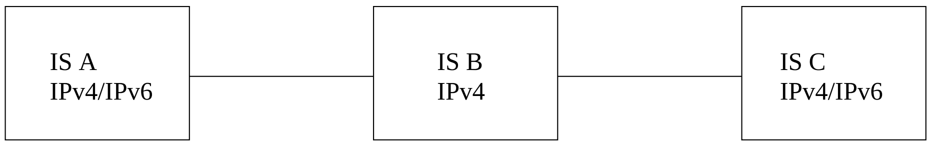

Figure 4.1: Example network utilizing automatic encapsulation

The use of automatic encapsulation is best illustrated with an example. In the network of Figure 4.1 forwarding of IPv4 is no problem, but IS A and IS C cannot send IPv6 to each other without encapsulating it inside IPv4. If IS C has informed in its LSPs that it can unencapsulate PDUs which contain IPv6 inside IPv4, IS A can now encapsulate IPv6 traffic inside IPv4 packets, and send them to IS C. IS C will then unencapsulate the IPv6 traffic and forward it onwards.

The idea is straightforward, but the changes implied by the method are not. In addition to the new TLV type automatic encapsulation requires changes to the basic operation of IS-IS. DIS election must be run separately among ISs supporting different subsets of the protocols supported on a LAN. The forwarding and receive processes must be extended to support encapsulation and unencapsulation of network layer PDUs and even the update and decision processes must be modified. Especially the fact that hardware based forwarding engines would need to support the (un)encapsulation of network layer PDUs might decrease the appeal of this method.

5 GNU Zebra routing software

The GNU Zebra routing software [28] is a free software distributed under the GNU General Public License (GPL) [30]. Zebra supports a number of TCP/IP based routing protocols (listed in Table 5.1), each of which are run separately as an own process. Zebra can be run on free BSD variants, GNU/Linux and Solaris platforms.

5.1 Zebra architecture

The Zebra architecture is depicted in Figure 5.1. Each protocol daemon uses the basic socket API to receive and transmit protocol specific PDUs (e.g. OSPF PDUs). Each of these processes is connected to the Zebra daemon with the Zebra protocol. The Zebra process passes the routing information to the kernel.

Protocol	**Daemon name**	**Defined in**
BGP-4 and BGP-4+	bgpd	[15] [16]
RIPv1 and RIPv2	ripd	[5] [12]
RIPng	ripngd	[11]
OSPFv2	ospfd	[9]
OSPFv3	ospf6d	[13]
Zebra	zebra	

Table 5.1: Protocols supported by Zebra

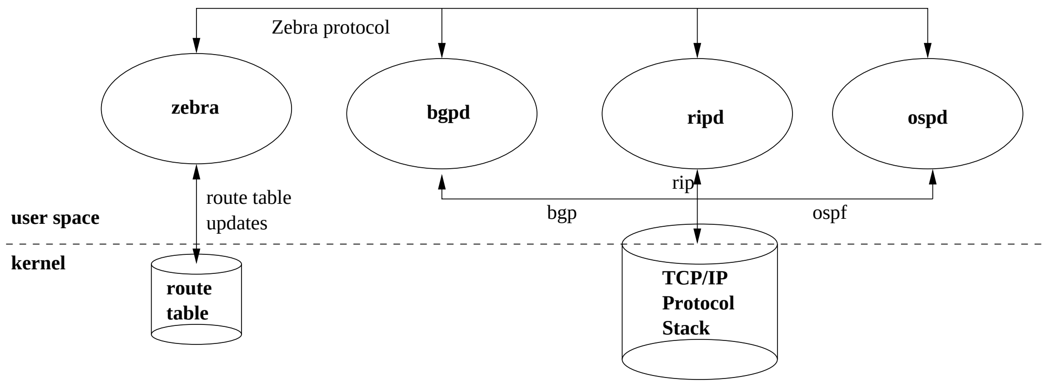

Figure 5.1: Zebra architecture

5.2 Zebra protocol

The Zebra protocol is a TCP based protocol used among the various routing processes and the Zebra process. In addition to route distribution, the protocol is used to pass interface state and address information among the processes. The Zebra protocol PDU is shown in Figure 5.2.

The *Length* is the entire length of the PDU and *Command* holds one of the values listed in Table 5.2. Only these two fields are mandatory. The *Type* field is used to identify the routing protocol (such as OSPF, RIP) using the Zebra daemon and the *Flags* and *Message*

	Number of Octets
Length	2
Command	1
Type	1
Flags	1
Message	1
Variable	Length − 6

Figure 5.2: Zebra PDU

Zebra Protocol command	**Code**
ZEBRA_INTERFACE_ADD	1
ZEBRA_INTERFACE_DELETE	2
ZEBRA_INTERFACE_ADDRESS_ADD	3
ZEBRA_INTERFACE_ADDRESS_DELETE	4
ZEBRA_INTERFACE_UP	5
ZEBRA_INTERFACE_DOWN	6
ZEBRA_IPV4_ROUTE_ADD	7
ZEBRA_IPV4_ROUTE_DELETE	8
ZEBRA_IPV6_ROUTE_ADD	9
ZEBRA_IPV6_ROUTE_DELETE	10
ZEBRA_REDISTRIBUTE_ADD	11
ZEBRA_REDISTRIBUTE_DELETE	12
ZEBRA_REDISTRIBUTE_DEFAULT_ADD	13
ZEBRA_REDISTRIBUTE_DEFAULT_DELETE	14
ZEBRA_IPV4_NEXTHOP_LOOKUP	15
ZEBRA_IPV6_NEXTHOP_LOOKUP	16
ZEBRA_IPV4_IMPORT_LOOKUP	17
ZEBRA_IPV6_IMPORT_LOOKUP	18

Table 5.2: Zebra protocol command types

fields are used to describe the contents of the *Variable* part.

5.3 Zebra library

Zebra library provides a number of functions to the routing protocol implementations. The source code for these functions resides, within the Zebra source package [31], in the `./zebra/lib/` -directory. An introduction to these functions follows.

5.3.1 Generic datastructures

In addition to the basic datastructures: linked-list (`linklist.h`), hash-table (`hash.h`) and vector (`vector.h`), the library functions include a stream buffer datastructure (`stream.h`), which can be used in the processing of PDUs. A stream buffer is depicted in Figure 5.3. The stream struct contains a pointer to dynamically allocated data of variable length (`size`). This data could for example be a protocol PDU read from a socket. The

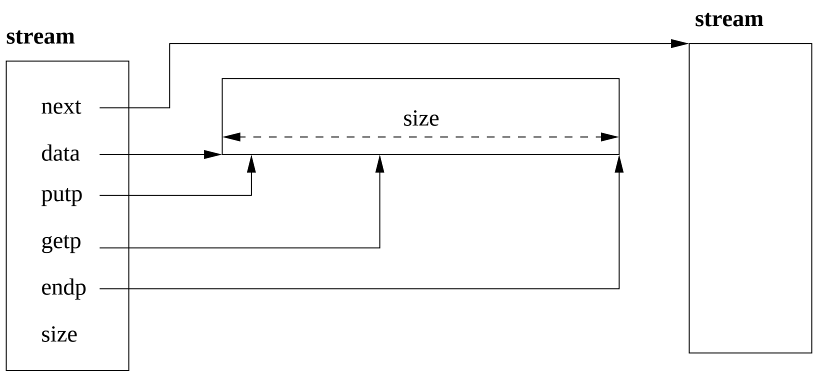

Figure 5.3: Stream buffer

`getp` points to the place where data was last read, and advances after a read. The library contains functions for reading 1, 2, 4 or user specified number of octets. Similarly the `putp` pointer is used for writing the data.

5.3.2 Memory handling

The library includes functions for handling of dynamically allocated data. The functions are actually wrappers for the standard memory routines `malloc()`, enhanced with memory tagging. The tagging of memory means, that when allocating or freeing memory, a memory type is specified. This type can be used for logging the memory usage, and helps to trace the cause of memory leaks.

5.3.3 Threads

The thread functions are used to control execution of events. The name “thread” is a bit misleading since the library is actually single-threaded. The execution of the threads is controlled with `select()` system call, which takes sets of file descriptors and timers as arguments and can be used to execute different functions based on the status of these arguments. The library can be used to register timers and functions to run when the timers expire, or for example, to register a file desciptor, so that a specified functions is called, whenever there is data to be read from the file descriptor.

```
/* Structure for the zebra client. */
struct zclient
{
  /* Socket to zebra daemon. */
  int sock;
  /* Flag of communication to zebra is enabled or not.
     Default is on.
     This flag is disabled by ‘no router zebra’ statement. */
  int enable;
  /* Connection failure count. */
  int fail;
  /* Input buffer for zebra message. */
  struct stream *ibuf;
  /* Output buffer for zebra message. */
  struct stream *obuf;
  /* Read and connect thread. */
  struct thread *t_read;
  struct thread *t_connect;
  /* Redistribute information. */
  u_char redist_default;
  u_char redist[ZEBRA_ROUTE_MAX];
  /* Redistribute default. */
  u_char default_information;
  /* Pointer to the callback functions. */
  int (*interface_add) (int, struct zclient *, zebra_size_t);
  int (*interface_delete) (int, struct zclient *, zebra_size_t);
  int (*interface_up) (int, struct zclient *, zebra_size_t);
  int (*interface_down) (int, struct zclient *, zebra_size_t);
  int (*interface_address_add) (int, struct zclient *, zebra_size_t);
  int (*interface_address_delete) (int, struct zclient *,
                                 zebra_size_t);
  int (*ipv4_route_add) (int, struct zclient *, zebra_size_t);
  int (*ipv4_route_delete) (int, struct zclient *, zebra_size_t);
  int (*ipv6_route_add) (int, struct zclient *, zebra_size_t);
  int (*ipv6_route_delete) (int, struct zclient *, zebra_size_t);
};
```

Figure 5.4: `struct zclient`

5.3.4 Zebra client API

A routing protocol daemon registers to the Zebra process through the Zebra client API (`zclient.h`). Each protocol holds a copy of the `zclient` struct shown in Figure 5.4. The `zclient` contains the socket (`sock`) for communications between the zebra process and the routing daemon, as well as threads and stream buffers used in the communication. Each routing daemon is responsible for setting the function pointers (the ten last fields of `struct zclient`) with addresses of these functions that it has implemented.

5.3.5 Route table

The maintainance of the route table can be done with the aid of route table library functions (`table.h`). The table is stored as a binary tree and consists of entries shown in Figure 5.5.

```
/* Each routing entry. */
struct route_node
{
  /* Actual prefix of this radix. */
  struct prefix p;
  /* Tree link. */
  struct route_table *table;
  struct route_node *parent;
  struct route_node *link[2];
#define l_left   link[0]
#define l_right  link[1]
  /* Lock of this radix */
  unsigned int lock;
  /* Each node of route. */
  void *info;
  /* Aggregation. */
  void *aggregate;
};
```

Figure 5.5: `struct route_node`

The `prefix` is the network address and mask and protocol family of the route. The `info` points to protocol specific data set by the routing protocol that was responsible for creating the entry.

5.3.6 Filters and routemap

Zebra provides the basic routines for route filtering and redistribution as library functions. The basic access-lists for routes can be built with the filters specified in `filter.h`. These filters have the form [ip-address, mask and action (*permit/deny*)].

Route maps (`routemap.h`) are similar to access-lists, but they can also be used to actual route mangling and policy routing and can be used to give routes a local preference value. The greater the value, the more likely a route matching the rule will be preferred. The routes can be matched based on access-lists, next-hop values, metric values and so on. The route map functionalities also include the setting of BGP specific parameters. Access-lists and route maps can be used in conjunction and to control route redistribution (`distribute.h`).

In the example network of Figure 5.6 the operator wants to redistribute everything but network 192.168.1.0 into the IS-IS Level 2 domain as an external route with 0 metric. Figure 5.7 shows the related part of router Dilbert's configuration. First a filter that matches the route is specified (`access-list 42`). The route map `Dogbert` utilizes this filter and specifies that routes matching the filter are denied. The `redistribute` command in

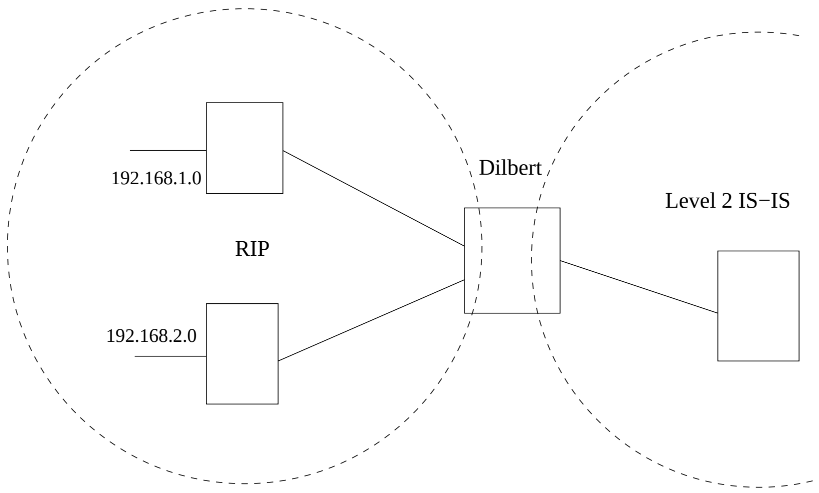

Figure 5.6: Example network

```
router rip
  network 192.168.1.0
  network 192.168.2.0
  default-information originate
!
router isis
  net 47.0023.0000.0003.0300.0100.0000.0000.0001.00
  is-type level-2-only
  redistribute rip metric 0 route-map Dogbert metric-type external level-2
!
access-list 42 permit 192.168.1.0
!
route-map Dogbert deny 10
  match ip address 42
!
```

Figure 5.7: Router Dilbert's confi guration

turn utilizes the route map `Dogbert` and the desired behaviour is achieved.

5.3.7 User interface

A user can connect to a protocol daemon by opening a telnet session to a specific port on the host. The library functions include "Virtual terminal" routines (`vty.h`), so that a user can issue commands to routing processes on the fly. Also the use of configuration files or combination of both methods is possible.

The Zebra software provides a similar command line interface to the one in IOS of Cisco routers [32]. The basic functions and macros to implement these commands are provided by the library (`command.h`).

6 ISISd - Implementation of IS-IS routing protocol

The implementation of IS-IS was done as an open source project at SourceForge.net [1]. SourceForge.net provides all the necessary services needed when running a software project for free. The services include version control (CVS), a web based bug-tracking system and mailing-lists. The ISISd project began on 17th of May 2001 and has released six versions of the software [29].

ISISd is implemented as a module for Zebra. The code is written in C and released under the GPL [30].

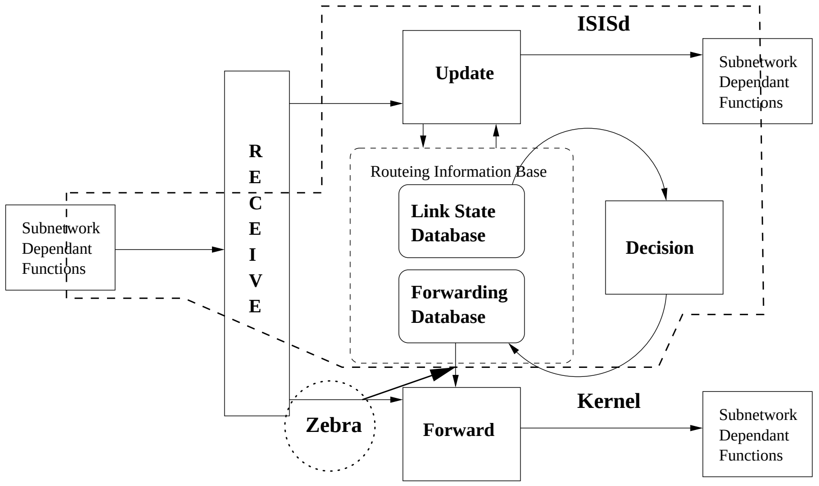
ISISd
Update
Subnetwork Dependant Functions
R E C E I V E
Routeing Information Base
Link State Database
Forwarding Database
Decision
Subnetwork Dependant Functions
Zebra
Forward
Kernel
Subnetwork Dependant Functions

Figure 6.1: ISISd and the Subnetwork Independent Functions

Figure 6.1 shows how ISISd relates to the functional organization of IS-IS depicted in Fig-

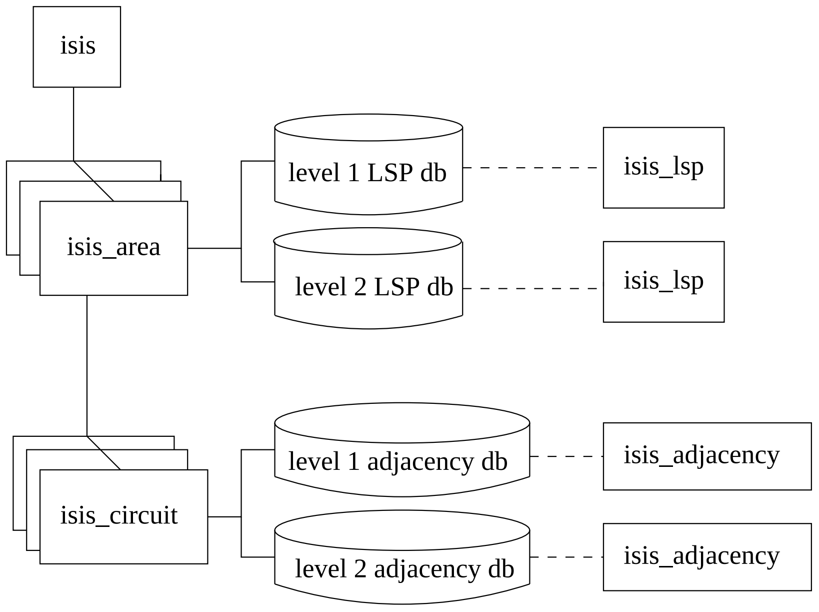

Figure 6.2: Major datastructures of ISISd

ure 3.1. ISISd handles the *Update* and *Decision* processes and passes the routing information to OS kernel through Zebra. The kernel then takes care of the *Forwarding* process.

A description of the software follows. We start by introducing the major datastructures and move then to the various functionalities.

6.1 Major datastructures

The major datastructures and their relation is depicted in Figure 6.2. The `struct isis` is the starting point for all the datastructures. There is only one of these structs per ISISd process. The `struct isis` holds a list of IS-IS areas and each area a list of circuits configured to the area. The area holds the LSP databases for both levels. Circuits in turn keep the adjacency databases. Next each of these structs is investigated in more detail.

6.1.1 `struct isis`

The `struct isis` is shown in Figure 6.3. The struct holds the *SystemID* of the IS, which has a fixed length of six octets in this implementation (`ISIS_SYS_ID_LEN`). The list of Area Addresses that are configured manually as well as the value of the *maximumAreaAd-*

```
struct isis
{
  u_long process_id;
  int sysid_set;
  u_char sysid[ISIS_SYS_ID_LEN];    /* SystemID for this IS */
  struct list *area_list;           /* list of IS-IS areas */
  struct list *init_circ_list;
  struct list *nexthops;            /* IPv4 next hops from this IS */
#ifdef HAVE_IPV6
  struct list  *nexthops6;          /* IPv6 next hops from this IS */
#endif /* HAVE_IPV6 */
  u_char max_area_addrs;            /* maximumAreaAdresses */
  struct area_addr *man_area_addrs; /* manualAreaAddresses */
  u_int32_t debugs;                 /* bitmap for debug */
  time_t uptime;                    /* when did we start */
};
```

Figure 6.3: `isisd.h - struct isis`

dresses parameter are also present. The `area_list` holds all the IS-IS areas. The level of debugging is controlled with commands that affect the variable `debugs`. The next-hop lists are stored in the `isis` struct, instead of `isis_area` or `isis_circuit` (discussed in the two following sub-sections), as a memory saving trick, since IPv4 and IPv6 next-hops can be identical for different areas. The `init_circ_list` holds the list of circuits the Zebra process has informed us, but are not yet configured to take part in IS-IS routing.

```
struct isis_area
{
  struct isis *isis;                          /* back pointer */
  dict_t *lspdb[ISIS_LEVELS];                 /* link-state dbs */
  struct isis_spftree *spftree[ISIS_LEVELS];  /* The v4 SPTs */
  struct route_table *route_table;            /* IPv4 routes */
#ifdef HAVE_IPV6
  struct isis_spftree *spftree6[ISIS_LEVELS]; /* The v4 SPTs */
  struct route_table *route_table6;           /* IPv6 routes */
#endif
  int min_bcast_mtu;
  struct list *circuit_list;                  /* IS-IS circuits */
  struct flags flags;
  struct thread *t_tick;                      /* LSP walker */
  /*
   * Configurables
   */
  /* identifies the routing instance   */
  char *area_tag;
  /* Counters */
  u_int32_t circuit_state_changes;
 };
};
```

Figure 6.4: `isisd.h - struct isis_area`

6.1.2 `struct isis_area`

The essential parts of `struct isis_area` are shown in Figure 6.4. The LSP databases for both levels are implemented with red-black tree (`dict_t`) that is part of the freely distributable kazlib software package [33]. The `struct isis_area` holds an SPF tree per supported protocol (IPv4 and IPv6) per level as well as IPv4 and IPv6 route tables derived from the trees. The `circuit_list` field points to a list of circuits that belong to this area. The `isis_area` struct holds also a number of configurables that are initially set to default values and then changed according to the users' commands. The area instance is identified, with a user specified `area_tag`.

6.1.3 `struct isis_circuit`

Each circuit participating in IS-IS routing is presented with `struct isis_circuit`, parts of which are shown in Figure 6.5. The struct holds a pointer to the interface struct received from Zebra that maps directly to the physical interface of the IS. The socket for sending and receiving IS-IS PDUs as well as threads to control the sending are also present. The `lsp_queue` is needed because transmitting of LSPs must be controlled so that LSPs are not sent more often than the parameter *lSPThrottleInterval*. The `circ_type` defines

```
struct isis_circuit {
  int state;
  u_char circuit_id;             /* l1/l2 p2p/bcast CircuitID */
  struct isis_area *area;        /* back pointer to the area */
  struct interface *interface;   /* interface info from z */
  int fd;                        /* IS-IS l1/2 socket */
  struct list *lsp_queue;        /* LSPs to be txed (both levels) */
  int idx;                       /* idx in S[RM|SN] flags */
#define CIRCUIT_T_BROADCAST  0
#define CIRCUIT_T_P2P        1
  int        circ_type;          /* type of the physical interface */
  union {
    struct isis_bcast_info bc;
    struct isis_p2p_info p2p;
  } u;
  /*
   * Configurables
   */
  long lsp_interval;
   /*
    * Counters as in 10589--11.2.5.9
    */
   u_int32_t adj_state_changes;    /* changesInAdjacencyState */
 };
```

Figure 6.5: `isis_circuit.h - struct isis_circuit`

the type of the physical media, and thus the contents of the union `u`. On point-to-point circuits only the neighbor adjacency and thread for sending *point-to-point IIH PDUs* is needed. The struct `isis_bcast_info` is more complex. It contains the threads for running the *LAN DIS* election, for refreshing the pseudonode LSPs and for sending the *LAN IIH PDUs*. The adjacency databases for both levels are also located inside the struct. The union is followed by a number of circuit related configurables and counters.

6.1.4 `struct isis_lsp`

Every LSP installed is described with `struct isis_lsp`, shown in Figure 6.6. The idea is that a received LSP is saved “as is” (`pdu`) from the network and then pointers to the various fields of LSP are provided. The variable part of the LSP can be referenced through the `tlv_data` field, which is a struct consisting of lists of TLVs. Each TLV type has its own list and the list items point to the TLVs in the PDU.

The `struct isis_lsp` contains also the *SRM* and *SSN flags* and a pointer to the adjacency the LSP was received from. The time values (`installed, last_generated` and `last_sent`) are used in debugging information and to control the regeneration and sending of the LSP. When LSP expires an IS must keep a copy of its header for *zeroAgeLifeTime*, before the LSP can be removed entirely from the database. The field `age_out` is used for this purpose. When the LSP is generated by this system the flag `own_lsp` is set and the `adj` is then set to `NULL`.

```
struct isis_lsp
{
  struct isis_fixed_hdr *isis_header;     /* normally == pdu */
  struct isis_link_state_hdr *lsp_header; /* pdu+isis_hdr_len */
  struct stream *pdu;                     /* full pdu lsp */
  u_int32_t SRMflags[ISIS_MAX_CIRCUITS];
  u_int32_t SSNflags[ISIS_MAX_CIRCUITS];
  u_int32_t rexmit_queue[ISIS_MAX_CIRCUITS];
  int level;                              /* L1 or L2? */
  int purged;                             /* purged this one */
  time_t installed;
  time_t last_generated;
  time_t last_sent;
  int own_lsp;
  /* used for 60 second counting when rem_lifetime is zero */
  int age_out;
  struct isis_adjacency *adj;
  struct tlvs tlv_data;               /* Simplifies TLV access */
};
```

Figure 6.6: `isis_lsp.h - struct isis_lsp`

6.1.5 `struct isis_adjacency`

Adjacent systems are described by `struct isis_adjacency` (Figure 6.7). The struct contains neighbors' SNPA address, *System ID* and *Area Addresses* copied from neigbors' *IIH* PDU. If IPv4 or IPv6 addresses are present they are saved in the adjacency. For adjacencies learnt from a broadcast circuit the *LAN ID* and *Priority* fields are also copied.

The `hold_time` is updated and the `t_expire` thread cancelled each time a *IIH* is heard. The current implementation removes the adjacency immediately when the holding time runs out. A suggestion has been made that the adjacency would be saved in a down state for some time and then removed with a special garbage collection function. This way it would be easier for the network manager to notice the fact that a neighbor has gone down.

The adjacencies are stored in a Zebra library hash-table, and the *System ID* is used as the hash-key.

The state of the adjacency is in the variable `adj_state`. The LAN adjacencies go from *down*-state, through *init*-state to *up*-state as described in section 3.2.2.

```
struct isis_adjacency{
  u_char snpa[ETH_ALEN];           /* NeighbourSNPAAddress */
  u_char sysid[ISIS_SYS_ID_LEN];   /* neighbourSystemIdentifier */
  u_char lanid[ISIS_SYS_ID_LEN+1]; /* LAN id on bcast circuits */
  enum isis_adj_state adj_state;   /* adjacencyState */
  enum isis_adj_usage adj_usage;   /* adjacencyUsage */
  struct list *area_addrs;         /* areaAdressesOfNeighbour */
  struct nlpids nlpids;            /* protocols spoken ... */
  u_char ipv4_addrs_count;
  struct in_addr ipv4_addrs[366];
#ifdef HAVE_IPV6
  u_char ipv6_addrs_count;
  struct in6_addr ipv6_addrs[91];
#endif /* HAVE_IPV6 */
  u_char prio[ISIS_LEVELS];        /* priorityOfNeighbour for DIS*/
  int circuit_t;                   /* from hello PDU hdr */
  int level;                       /* level (1 or 2) */
  enum  isis_system_type sys_type; /* neighbourSystemType */
  u_int16_t hold_time;             /* entryRemainingTime */
  u_int32_t last_upd;
  struct thread *t_expire;         /* expire after hold_time  */
  struct isis_circuit *circuit;    /* back pointer */
};
```

Figure 6.7: `isis_adjacency.h - struct isis_adjacency`

6.2 Processing of PDUs

The code for sending and receiving IS-IS PDUs is located in the `isis_network.[ch]` files. The IS-IS PDUs are sent on link-level (OSI layer 2) so special actions need to be taken in order to send and receive the packets.

In the GNU/Linux platform the socket is created with `socket(2)` system call and then bound to a physical interface with `bind(2)` system call. The family of the socket is set to `PF_PACKET` and the socket type to `SOCK_DGRAM`.

In order to receive packets destined to multicast addresses *AllL1ISs* and *AllL2ISs* the socket is bound to corresponding physical layer multicast groups with `setsockopt(2)` as defined in `packet(7)` manual pages.

When the socket family is `AF_PACKET` a device independent physical layer address `sockaddr_ll` is used (Figure 6.8). This address is filled when sending or receiving PDUs through the socket. The `sll_pkttype` is used to notice when a PDU is created by some other interface on the localhost. When this is the case the `sll_pkttype` has the value `PACKET_OUTGOING` and the PDU can be omitted.

```
struct sockaddr_ll {
  unsigned short  sll_family;    /* Always AF_PACKET */
  unsigned short  sll_protocol;  /* Physical layer protocol */
  int             sll_ifindex;   /* Interface number */
  unsigned short  sll_hatype;    /* Header type */
  unsigned char   sll_pkttype;   /* Packet type */
  unsigned char   sll_halen;     /* Length of address */
  unsigned char   sll_addr[8];   /* Physical layer address */
};
```

Figure 6.8: `packet(7) - struct sockaddr_ll`

Since operating systems other than GNU/Linux do not have the `packet(7)` socket, they require different methods for sending and receiving the PDUs. The FreeBSD port of the software utilizes Berkeley Packet Filter (BPF) and sockets bound to special BPF device files. This method has some drawbacks. The `select()` system call does not handle these kind of sockets well and BPF requires the interface to be in a promiscious mode. Promiscious mode means that every packet in the network is read by the device driver of the interface and causes a security risk. A better solution for sending and receiving the PDUs is being planned.

IS-IS PDUs sent on broadcast circuits are prepended with three octet LLC-header field. The values for the field are defined in ISO 9577 [20] and always have the value `0xFE-FE-03` as defined for routed ISO PDUs. PDUs received from broadcast circuits that do not have this kind of LLC header are discarded by the implementation.

After the PDU is read from the socket it can be passed to processing functions which are in the files `isis_pdu.[ch]` and `isis_tlv.[ch]`.

First the PDU is handled with a common wrapper function for all the PDU types called `isis_handle_pdu()` (Appendix A). This function performs basic validity checks on the IS-IS fixed header, such as the values of *ID Length* and *Maximum Area Addresses* fields. If the PDU passes these checks it is passed on to a processing function based on its type. For example the *Level 1* and *Level 2 LAN IIH PDUs* are passed to the `process_lan_hello()` function.

The variable part of IS-IS PDUs is handled in files `isis_tlv.[ch]`. All TLVs are parsed in a single function `parse_tlvs()` (Appendix A), which takes the variable part of IS-IS PDU, addresses of two 32-bit variables (`expected` and `found`) and a pointer to `struct tlvs` as arguments. The `expected` and `found` are used as bitfields consisting of flags for each TLV type. The caller sets the bits for the TLV types it is interested in and the function sets the bits in `found` for the TLV types it came across while parsing the PDU. The `struct tlvs` consists of lists for each TLV type and is filled when parsing the PDU. This way the data in the variable part can be easily accessed. Only the TLVs of types reported in `expected` are saved for the caller.

The functions for writing the TLVs are also in files `isis_tlv.[ch]`. Each TLV type has its own function for writing (Appendix A). The functions take a list of TLVs to be written and a PDU to which to write them as arguments.

6.3 LAN Designated IS election

The LAN DIS election is implemented in files `isis_dr.[ch]`. When the circuit is created or configured with a new level, the implementation must first wait a random time between 0 and configurable *iSISHelloTimer* seconds, before commencing the LAN DIS election on the circuit. The DIS election is not run if there are no adjacencies on the circuit. The DIS election is controlled with threads and flags `run_dr`, that are located in the `struct isis_circuit`.

The election is performed by the function `run_dr_elect()`. The function first builds a list of IS adjacencies that are in *up*-state from the adjacency database, and then goes through the list to find the IS adjacency with highest priority. The SNPA addresses are compared if more than one IS has the highest priority. When we have found the IS adjacency with highest priority the priority of the adjacency is compared to priority configured for the circuit running the election. If our circuit has a lower priority and we are the DIS, a function `isis_dr_resign()` is called, that purges our pseudonode LSP and resigns from the DIS role on the circuit. If we have a higher or equal priority with the adjacency and we were not a LAN DIS previously, a function `isis_dr_commence()` is called, which calls the functions for generating and transmitting our pseudonode LSP.

6.4 Circuit state machine

ISO 10589 defines only two states for circuits, namely *up* and *down*. However, the implementation utilizes more states due to the modular nature of Zebra. ISISd has four states for circuits: state *N/A* means that nothing is known of the circuit, circuits learned from Zebra, but not yet configured for IS-IS are in state *init*, circuits configured for IS-IS but not attached to a network interface (learnt from Zebra) are in state *configured*, and circuits that are both configured and attached to an interface are in state *up*.

```
/*
 * Circuit states
 */
#define C_STATE_NA   0
#define C_STATE_INIT 1 /* Connected to interface */
#define C_STATE_CONF 2 /* Configured for ISIS    */
#define C_STATE_UP   3 /* CONN | CONF            */

/*
 * Circuit events
 */
#define ISIS_ENABLE    1
#define IF_UP_FROM_Z   2
#define ISIS_DISABLE   3
#define IF_DOWN_FROM_Z 4
```

Figure 6.9: `isis_csm.h - circuit state machine states and events`

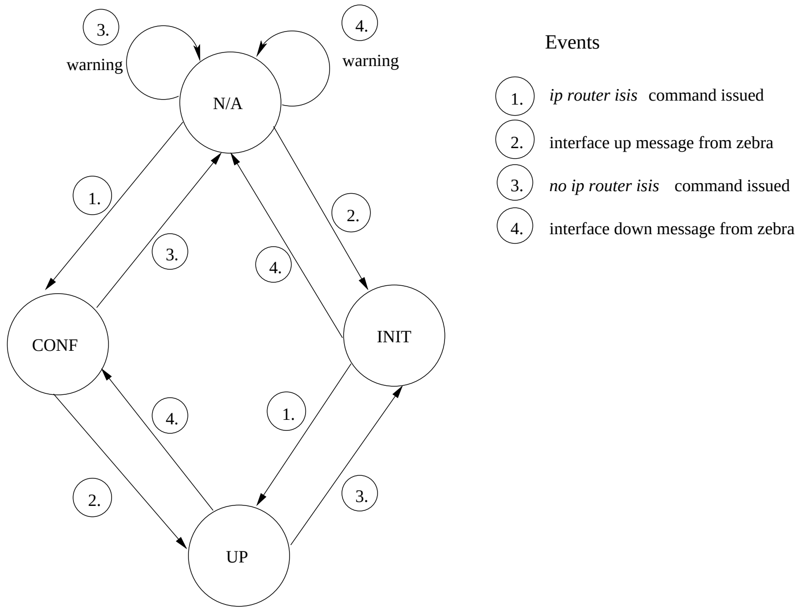
3.
warning
4.
warning
N/A
Events
1. ip router isis command issued
2. interface up message from zebra
3. no ip router isis command issued
4. interface down message from zebra
1.
2.
3.
4.
CONF
INIT
4.
1.
2.
3.
UP

Figure 6.10: Circuit state machine

The state machine for the circuits is implemented in files `isis_csm.[ch]`. The state machine is run by a single function `isis_csm_state_change()`, which takes one of the events, shown in Figure 6.9, and the circuit as arguments.

The state machine is shown in Figure 6.10. As one can see from the figure the up state can be reached through either *init* or *configured* state.

6.5 Dynamic hostnames

ISISd supports the dynamic hostname exchange mechanism [22]. The hostname cache is implemented in files `isis_dynhn.[ch]`. The hostnames learnt from the dynamic hostname TLVs are stored in `struct isis_dynhn` (Figure 6.11). The struct holds the hostname, *System ID* and level of the IS. The variable `refresh` is updated every time the TLV is seen, so that outdated hostname information can be removed. The printing functions

```
struct isis_dynhn {
  u_char id[ISIS_SYS_ID_LEN];
  struct hostname name;
  time_t refresh;
  int level;
};
```

Figure 6.11: `isis_dynhn.h - struct isis_dynhn`

show the hostname instead of the *System ID* of the IS, if the hostname matching *System ID* is found from the cache.

6.6 Implementation of SPT algorithm

An IS must run the SPF periodically and each time a change in LSP or adjacency database is noticed. The functions that implement the SPT algorithm are in files `isis_spf.[ch]`. Every area builds its own SPT for both routing levels. The SPT resides in `struct isis_spftree`, shown in Figure 6.12. The struct contains the thread for periodic SPF and the timestamp the SPF was last run. The *PATH* and *TENT* databases can be accessed through the struct. SPF should not be run more often than the configurable *minimumSPFInterval*, parameters `pending` and `lastrun` are used in scheduling the consecutive runs of SPF, so that this requirement is met.

The databases *PATH* and *TENT* consist of vertices described by `struct isis_vertex` (shown in Figure 6.13). The vertex holds a pointer to LSP it was derived from as well as adjacencies that are the starting points of equal cost paths towards this vertex. The number of adjacencies is limited by the parameter *maximumPathSplits*.

The `type` of the vertex enforces the value of the `union N`. For IS and ES vertices `N` holds the value of `System ID`, appended with a non-zero octet if the vertex describes a pseudonode. If the vertex describes address reachability it holds the network address and mask in the prefix struct.

The distance from the IS performing the SPF is stored in the variable `d_N` and the depth in

```
struct isis_spftree
{
  struct thread *t_spf_periodic;  /* periodic spf threads  */
  time_t                lastrun;  /* for scheduling */
  int                   pending;  /* already scheduled */
  struct isis_vertex     *paths;  /* the SPT */
  struct list            *tents;  /* TENT */
  struct list             *list;  /* list for search */

  u_int32_t             timerun;  /* statistics */

};
```

Figure 6.12: `isis_spf.h - struct isis_spftree`

```
/*
 * Triple <N, d(N), {Adj(N)}>
 */
struct isis_vertex
{
  enum vertextype type;

  union {
    u_char id [ISIS_SYS_ID_LEN + 1];
    struct prefix prefix;
  } N;

  struct isis_lsp *lsp;
  u_int32_t d_N;   /* d(N) Distance from this IS      */
  u_int16_t depth; /* The depth in the imaginary tree */

  struct list *Adj_N; /* {Adj(N)}  */
};
```

Figure 6.13: `isis_spf.h - struct isis_vertex`

the SPT in variable `depth`.

The SPT algorithm implementation is based on Annex C of ISO 10589 [8], which describes the basic Dijkstra algorithm enhanced with support for multiple equal cost paths. The support for multiple equal cost paths implies load splitting, which means that if more than one route of equal costs exists for the destination, the traffic towards the destination is divided equally to the paths. Current Linux kernels support load splitting.

If the IS supports both IPv4 and IPv6, it runs the SPT separately for both protocols.

6.7 Installing routes

While adding an address prefix to the *PATH* database the implementation calls the function `isis_route_create()` (Appendix A) (in file `isis_route.c`). The function takes the prefix, cost and depth (a hopcount) of the prefix and a list of adjacencies to reach

```
struct isis_route_info {
#define ISIS_ROUTE_FLAG_ZEBRA_SYNC 0x01
#define ISIS_ROUTE_FLAG_ACTIVE     0x02
  u_char flag;
  u_int32_t cost;
  u_int32_t depth;
  struct list *nexthops;
#ifdef HAVE_IPV6
  struct list *nexthops6;
#endif /* HAVE_IPV6 */
};
```

Figure 6.14: `isis_route.h - struct isis_route_info`

that prefix as arguments. Each area holds its own route table, implemented by the Zebra library (Section 5.3.5). The implementation specific route information is stored in `struct isis_route_info` (Figure 6.14). The next hop information is derived from the list of adjacencies by the function.

The `isis_route_create()` (Appendix A) first makes a new `isis_route_info` struct and then checks if a route for the prefix exists. If there already is a route for the prefix, the route information is compared. If the route has the same attributes nothing needs to be done. If the new route has a better cost, the old route is replaced with the new one. If the routes have the same costs but different next-hop information, the next-hop lists are merged.

After every SPF execution a thread is activated that executes a function called `isis_route_validate()`(Appendix A). This function goes through all the routes of the area and calls the `isis_route_update()` function for each route. The `isis_route_update()`, that lies in `isis_zebra.[ch]` files, in turn calls appropriate function for routes not marked "being in sync" with Zebra (`ISIS_ROUTE_FLAG_ZEBRA_SYNC`). Finally the routes are inserted to or deleted from the kernel by the Zebra daemon.

6.8 Zebra connection

As explained in Section 5.3.4, each routing process fills the `struct zclient` with its own implementation of the functions. ISISd does this in the function `isis_zebra_init()` shown in Figure 6.15. This function is called when starting the ISISd process.

After we have registered to the Zebra daemon with the `isis_zebra_init()` function, the Zebra daemon can inform us about the configuration and changes in the configuration it sees. For example if a new network interface is to be created, the `isis_zebra_if_add()` function would eventually be called.

```
void
isis_zebra_init ()
{

  zclient = zclient_new ();
  zclient_init (zclient, ZEBRA_ROUTE_ISIS);
  zclient->interface_add = isis_zebra_if_add;
  zclient->interface_delete = isis_zebra_if_del;
  zclient->interface_up = isis_zebra_if_state_up;
  zclient->interface_down = isis_zebra_if_state_down;
  zclient->interface_address_add = isis_zebra_if_address_add;
  zclient->interface_address_delete = isis_zebra_if_address_del;
  zclient->ipv4_route_add = isis_zebra_read_ipv4;
  zclient->ipv4_route_delete = isis_zebra_read_ipv4;
#ifdef HAVE_IPV6
  zclient->ipv6_route_add = isis_zebra_read_ipv6;
  zclient->ipv6_route_delete = isis_zebra_read_ipv6;
#endif /* HAVE_IPV6 */

  return;
}
```

Figure 6.15: `isis_zebra.c - isis_zebra_init() function`

6.9 Future work

ISISd supports Linux and as of latest version (0.0.6) also FreeBSD. Support for other platforms that Zebra supports will be added in the later releases, next in line is OpenBSD.

Today ISISd can be used as a standalone routing protocol; it does not yet interact with other routing protocols than IS-IS. The Zebra library functions provide mechanisms for redistribution of routes among different routing protocols. Route redistribution and auto summarization of routes will be added in later releases of ISISd.

7 Configuring ISISd

This section introduces the configuration options available in ISISd. The options described are the ones implemented in version 0.0.6 of the software. More options may be included in later versions.

7.1 Invoking ISISd

ISISd is started by typing `isisd` and possibly options on the command line. Next the command line options are explained. The options follow the GNU style, where every option has a long and short form.

`[-d | -daemon]` "daemonizes" the ISISd, which means that the process forks and exits from the tty, closes the standard input, output and error file descriptors, and changes working directory to the root of the filesystem.

`[-f | -config-file]` is used to set the configuration file of isisd. The option requires the configuration file name as argument. If this option is not specified, the configuration file defaults to */usr/local/etc/isisd.conf*.

`[-P | -vty_port]` is used to specify the port ISISd vty will listen on. The option requires the port number as argument. If not specified the port defaults to *2607*.

`[-v | -version]` prints the version of ISISd and exits.

`[-h | -help]` prints a short help describing the command line options.

7.2 Connecting to vty

A connection to ISISd vty can be achieved by opening a telnet session to a port specified for ISISd. A password (specified in the configuration file) is required. Figure 7.1 shows how the connection is done. After issuing a valid password the user enters *view* mode which has the commands shown in the Figure.

```
sambo@jaarli~: telnet localhost 2607
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.

Hello, this is zebra (version 0.93-pre1).
Copyright 1996-2001 Kunihiro Ishiguro.


User Access Verification

Password:*****
isisd>?
  enable    Turn on privileged mode command
  exit      Exit current mode and down to previous mode
  help      Description of the interactive help system
  list      Print command list
  quit      Exit current mode and down to previous mode
  show      Show running system information
  terminal  Set terminal line parameters
  who       Display who is on vty
```

Figure 7.1: Connecting to ISISd vty

7.3 Show commands

Show commands can issued in *view* mode. Figure 7.2 displays the top level of show commands. On `clns` branch there is only one command `show clns neighbors`, which is an alias for the `show isis neighbors` command.

```
isisd> show ?
  clns     clns network information
  history  Display the session command history
  isis     IS-IS information
  memory   Memory statistics
  version  Displays zebra version
```

Figure 7.2: `show` commands of ISISd

```
isisd> show memory
  all    All memory statistics
  bgp    BGP memory
  isis   ISIS memory
  lib    Library memory
  ospf   OSPF memory
  ospf6  OSPF6 memory
  rip    RIP memory
  <cr>
isisd> show memory isis
ISIS             : 1
ISIS TMP         : 0
ISIS circuit     : 2
ISIS LSP         : 8
ISIS adjacency   : 3
ISIS area        : 1
ISIS area address: 1
ISIS TLV         : 5
ISIS dyn hostname: 3
ISIS SPFtree     : 2
ISIS vertex      : 25
ISIS route info  : 16
ISIS nexthop     : 0
ISIS nexthop6    : 18
```

Figure 7.3: Output of `show memory` command

The memory usage can be viewed with the `show memory` command. (Figure 7.3). Figure 7.3 depicts the output of `show memory isis` command. The viewing of memory is possible because of the memory tagging provided by the Zebra library (Section 5.3.2).

The `show` commands under `isis` branch are listed in Figure 7.4.

```
isisd> show isis ?
  database   IS-IS link state database
  hostname   IS-IS Dynamic hostname mapping
  neighbors  IS-IS neighbor adjacencies
  topology   IS-IS paths to Intermediate Systems
```

Figure 7.4: The `show isis` commands

The contents of link state database can be viewed, with `show isis database` command (Figure 7.5). The command lists the information in LSP headers in the LSP database. If the command is appended with `detail` command, the contents of the variable part of LSPs is also shown.

The contents of the dynamic hostname cache can be viewed with `show isis hostname` command (Figure 7.6). The commands shows the *System ID* to hostname mappings and the level of IS. The mapping of the IS itself is prepended with the ∗-sign.

The adjacency database can be viewed with `show isis neighbors` command (Figure 7.7). The command displays the hostname, the interface it was learned from, the state,

```
isisd> show isis database
Area FOO:
  IS-IS Level-2 link-state database:
  LSP ID                 Sequence   Checksum Lifetime Attributes
  broker-gw.00-00        0x00006fcb   0x65b1      977 L2
  ipv6-gw.00-00          0x000034b5   0xbcf0      640 L2
  ipv6-gw.02-00          0x00001c45   0xe021      638 L2
  isis-4700.00-0         0x000037a9   0x68b3      763 L2
  isis-4700.01-00        0x00003258   0x1123     1007 L2
  isisd.00-00          * 0x00000005   0xa5b9      416 L2

    6 LSPs
isisd> show isis database detail
Area FOO:
IS-IS Level-2 Link State Database:

  broker-gw.00-00
    Sequence: 0x00006fcb Checksum: 0x65b1 Lifetime: 718
    Attributes: L1 L2, Installed: 8m8s ago
    Speaks: IPv6
    IS      broker-gw.06, Metric: 10
    IS      broker-gw.05, Metric: 10
    IS      broker-gw.01, Metric: 10
    IS      isis-4700.01, Metric: 10
    Hostname: broker-gw
```

Figure 7.5: Output of `show isis database` command

the SNPA address and *Holding Time* of the adjacencies. If the flag `detail` is given, also the priority, network addresses and supported protocols are shown. The `DIS flaps` refers to the number of *LAN DIS* changes noticed and the `LAN-id` the *System ID* of the pseudonode on the LAN.

The contents of the *PATH* database(s) can be viewed with the `show isis topology` command. The command lists the shortest paths to other ISs on the area. Level 1 and Level 2 are listed separately as well as routes to ISs that support IPv4 and IPv6. First column is the System ID or dynamic hostname of the destination IS. Metric is the total cost to reach the IS and the following three columns are the system ID, interface and SNPA address of the first hop on the path towards the IS denoted by the first column.

```
isisd> show isis hostname
Level  System ID      Dynamic Hostname
2      0000.0000.0001 broker-gw
2      0000.0000.0004 ipv6-gw
2      0000.0000.0042 isis-4700
     * dead.beef.0043 isisd
```

Figure 7.6: The `show isis hostname` command

```
isisd> show isis neighbors
Area FOO:
  System Id    Interface   L  State        Holdtime SNPA
  broker-gw    eth0        2  Up           25       0001.631e.8038
  ipv6-gw      eth0        2  Up           23       0060.3e2f.de00
  isis-4700    eth0        2  Up           8        0060.70cb.f108
isisd> show isis neighbors detail
Area FOO:
  broker-gw
    Interface: eth0, Level: 2, State: Up, Expires in 22s
    Adjacency flaps: 0, Last: 1h11m26s ago
    Circuit type: L2, Speaks: IPv6
    SNPA: 0001.631e.8038, LAN id: isis-4700.01
    Priority: 64, is not DIS, DIS flaps: 1, Last: 1h10m26s ago
    IPv6 Addresses:
      fe80::201:63ff:fe1e:8038
```

Figure 7.7: Output of `show isis neighbors` command

```
kameisis> show isis topology
Area FOO:
IS-IS paths to level-2 routers that speak IPv6
System Id    Metric  Next-Hop   Interface  SNPA
kameisis     --
broker-gw    10      broker-gw  xl0        0001.631e.8038
v6rtr1       20      broker-gw  xl0        0001.631e.8038
v6rtr2       20      broker-gw  xl0        0001.631e.8038
teemu-gw     20      broker-gw  xl0        0001.631e.8038
ipv6-gw      20      broker-gw  xl0        0001.631e.8038
isis-4700    20      broker-gw  xl0        0001.631e.8038
```

Figure 7.8: The output of `show isis topology` command

7.4 Router confi guration commands

The configuration commands are given in *configuration* mode, which is reached by first entering the *privileged* mode with `enable` command and then the *configuration* mode, with command `configure terminal`. An IS-IS routing instance is created in the *router-configuration* mode. The *router-configuration* mode for IS-IS is entered by giving `router isis` command followed by a local symbol for the area. Figure 7.9 shows how the *router-configuration* mode is entered and the commands available in the mode.

With the `hostname` command, the dynamic hostname added in TLVs generated by this system can be specified. By default the hostname is the network hostname learned from Zebra.

The `is-type` command specifies the level for the routing instance. It can be Level 1, Level 2 or both. By default the first instance is Level 1 and Level 2 and the rest are Level 1.

The `lsp-gen-interval` command is used to specify the minimum time between suc-

```
isisd> ena
Password:****
isisd# configure terminal
isisd(config)# router isis AREA1
isisd(config_router)#?
  end                End current mode and change to enable mode.
  exit               Exit current mode and down to previous mode
  help               Description of the interactive help system
  hostname           Dynamic hostname for IS-IS
  is-type            IS Level for this routing process (OSI only)
  list               Print command list
  lsp-gen-interval   Minimum interval between regenerating same LSP
  lsp-lifetime       Maximum LSP lifetime
  metric-style       Use old-style (ISO 10589) or
                     new-style packet formats
  net                A Network Entity Title for this process
  no                 Negate a command or set its defaults
  quit               Exit current mode and down to previous mode
  spf-interval       Minimum interval between SPF calculations
  write              Write running configuration to memory, network,
                     or terminal
```

Figure 7.9: Entering router confi guration mode

cessive regenerations of our own LSPs. The default value in this implementation is 10 seconds. The possible values are range from 1 to 120 seconds.

The maximum value of *remaining lifetime* field set to our own LSPs can be specified with the `lsp-lifetime` command. The default value is 1200 seconds and the possible values 380-65535 seconds.

By default ISISd uses the traditional style metrics defined in [8] in its TLVs. New style metrics, where other than *default* metric are omitted, but the *default* metric is 3 octets long in is defined in [26]. This new style can be enabled with `metric-style wide` command and disabled with `metric-style narrow` command.

At least one *Network Entity Title* for the routing instance must be specified with the `net` command. Up to parameter *Maximum Area Adresses* number of *NETs* can be specified. All *NETs* must have the same *System ID* part.

The `spf-interval` command can be used to specify the minimum delay between consecutive runs of the SPF algorithm. The default value for ISISd is 5 seconds and possible values range from 1 to 120 seconds.

7.5 Interface confi guration commands

The *interfaces configuration* mode is entered with the `interface` command in the *configuration* mode. The name of the interface to be configured must be specified (Figure 7.10).

To enable IS-IS routing on the interface, command `ip router isis` or `ipv6 router isis`, or both, is given. The command must include the local tag of the routing instance the interface is configured to (for example `ip router isis AREA1`). Other IS-IS specific configuration commands are shown in Figure 7.11.

The level of adjacencies formed on the interface can be specified with `circuit-type` command. The interface can be configured to form Level 1 and Level 2 adjacencies, if the routing instance supports both levels. By default the `isis circuit-type` is the same as the `is-type` of the routing instance.

When acting as a pseudonode for LAN, *CSNPs* are transmitted periodically on the circuit. The interval between periodic *CSNP* transmissions can be changed with the `isis csnp-interval` command. The default value in this implementation is 10 seconds.

If the *IIH PDUs* are to be padded to the full MTU of the circuit, the command `isis hello padding` is specified. By default the hello padding is on.

The average time between periodic *IIH PDU* transmissions can be set with `isis hello-interval` command. The value is specified in milliseconds and the default is 10000 milliseconds. The `isis hello-multiplier` command can be used in conjunction with the `isis-hello interval` command in order to control the actual value of

```
isisd(config)# interface eth0
isisd(config-if)#?
  description  Interface specific description
  end          End current mode and change to enable mode.
  exit         Exit current mode and down to previous mode
  help         Description of the interactive help system
  ip           Interface Internet Protocol config commands
  ipv6         IPv6 interface subcommands
  isis         IS-IS commands
  list         Print command list
  no           Negate a command or set its defaults
  quit         Exit current mode and down to previous mode
  write        Write running configuration to memory, network, or terminal
```

Figure 7.10: Entering interface confi guration mode

```
isisd(config-if)# isis ?
  circuit-type      Configure circuit type for interface
  csnp-interval     Set CSNP interval in seconds
  hello             Add padding to IS-IS hello packets
  hello-interval    Set Hello interval in milliseconds
  hello-multiplier  Set multiplier for Hello holding time
  metric            Set default metric for circuit
  priority          Set priority for Designated Router election
```

Figure 7.11: IS-IS specifi c commands in interface confi guration mode

Holding Time in the *IIH PDUs* transmitted by the IS on the interface. The *Holding Time* is set to the value of hello-interval in seconds multiplied by the value of hello-multiplier, which is 3 by default.

The *default metric* for the circuit can be set with the command `metric` and the priority for becoming the *LAN Designated IS* with the command `priority`.

8 Conclusions

IS-IS is unique among today's routing protocols, because of its multiprotocol design. Integrated routing of CLNS and IPv4 might not interest many, but IS-IS used as single routing protocol for both IPv4 and IPv6 is a possibility not to be overlooked.

The differences between OSPFv3 and IS-IS are not major, and the former is actually closer to IS-IS than OSPFv2. As the addition of IPv6 support to IS-IS was easy to implement, IS-IS was the first link state routing protocol that supports IPv6 inside the routers of biggest vendors. Which of these protocols will be the major link state routing protocol for IPv6 in the future, time will tell, but integrated IS-IS might at least be of aid in the transition from IPv4 to IPv6.

The goal of this thesis was to implement a version of IS-IS (ISISd) that supports the routing of IPv4 and IPv6. The implementation was done as an open source project and had a changing number of developers from all over the world. ISISd was implemented as a module for the GNU Zebra routing software. It is not merged to official Zebra package yet, but exists as a separate patch.

As of summer 2002 ISISd has been taking care of IPv6 routing on a FreeBSD workstation that acts as a router between a production and a test network at the Institute of Communications Engineering of Tampere University of Technology.

References

[1] Open Source Development Network. *SourceForge.net*. `http://sourceforge.net/`.

[2] D. Howe. *Free On-Line Dictionary Of Computing*. `http://wombat.doc.ic.ac.uk/foldoc/`, 1993.

[3] R. Perlman. *Interconnections: Bridges, Routers, Switches and Internetworking Protocols*. Addison-Wesley, 2000. Second edition.

[4] Digital Equipment Corporation. *DIGITAL Network Architecture - Routing Layer Functional Specification*, 1980.

[5] RFC 1058; C. Hedrick. *Routing Information Protocol*, June 1988.

[6] J. McQuillan et.al. *The New Routing Algorithm for the Arpanet*. IEEE Transactions on Communications, May 1980.

[7] R. Perlman. *Fault-Tolerant Broadcast of Routing Information*, December 1983.

[8] ISO/IEC 10589. *Intermediate System to Intermediate System Intra-Domain Routeing Exchange Protocol for use in Conjunction with the Protocol for Providing the Connectionless-mode Network Service (ISO 8473)*, July 2000. Second edition.

[9] RFC 2328; J. Moy. *OSPF Version 2*, April 1998.

[10] RFC 1195; R. Callon. *Use of OSI IS-IS for Routing in TCP/IP and Dual Environments*, December 1990.

[11] RFC 2080; G. Malkin. *RIPng for IPv6*, January 1997.

[12] RFC 1723; G. Malkin. *RIP Version 2 - Carrying Additional Information*, November 1994.

[13] RFC 2740; R. Coltun and D. Ferguson. *OSPF for IPv6*, December 1999.

[14] C. E. Hopps. *Routing IPv6 with IS-IS*. `draft-ietf-isis-ipv6-02.txt`, April 2001. Internet-draft, work in progress.

[15] RFC 1771; Y. Rekhter and T. Li. *A Border Gateway Protocol 4 (BGP-4)*, March 1995.

[16] RFC 2858; R. Chandra, T. Bates, Y. Rekhter and D. Katz. *Multiprotocol Extensions for BGP-4*, June 2000.

[17] RFC 2545; P. Marques and F. Dupont. *Use of BGP-4 Multiprotocol Extensions for IPv6 Inter-Domain Routing*, 1999.

[18] ISO 8473. *Protocol for Providing the Connectionless-Mode Network Service*, March 1987.

[19] RFC 791; J. Postel. *Internet Protocol*, September 1981.

[20] ISO/IEC TR 9577. *Protocol Identification in the network layer*, 1990.

[21] T. Li and R.J. Atkinson. *IS-IS Cryptographic Authentication*. `draft-ietf-isis-hmac-03.txt`, July 2001. Internet-draft, work in progress.

[22] RFC 2763; N. Shen and H. Smith. *Dynamic Hostname Exchange Mechanism for IS-IS*, February 2000.

[23] RFC 2973; D. Katz, R. Baley and J. Parker. *IS-IS Mesh Groups*, October 2000.

[24] RFC 2966; T. Przygienda, T. Li and H. Smit. *Domain-wide Prefix Distribution with Two-Level IS-IS*, October 2000.

[25] RFC 2460; S. Deering and R. Hinden. *Internet Protocol, Version 6 (IPv6) Specification*, December 1998.

[26] H. Smith T. Li. *IS-IS extensions for Traffic Engineering*. `draft-ietf-isis-traffic-04.txt`, August 2001. Internet-draft, work in progress.

[27] P. Christian. *IS-IS Automatic Encapsulation*. `draft-ietf-isis-auto-encap-01.txt`, July 2002. Internet-draft, work in progress.

[28] K. Ishiguro. *GNU Zebra routing software*. `http://www.zebra.org/`, January 2001.

[29] ISISd project. *ISISd project homepage* `http://isisd.sourceforge.net`, May 2001

[30] Free Software Foundation. *GNU General Public License*. `http://www.gnu.org/copyleft/gpl.html`, June 1991.

[31] K. Ishiguro. *FTP site of Zebra*. `ftp://ftp.zebra.org/pub/zebra`.

[32] Cisco Systems. *Cisco IOS*. http://www.cisco.com/warp/public/732/jump.shtml.

[33] K. Kylheku. *Kazlib*. `http://users.footprints.net/~kaz/kazlib.html`.

Appendix A - Code snippets

In this appendix the source code of the following functions is shown:

- `isis_handle_pdu()`
- `parse_tlvs()`
- `add_tlv()`
- `tlv_add_area_addrs()`
- `isis_dr_elect()`
- `add_to_paths()`
- `isis_run_spf()`
- `isis_route_create()`
- `isis_route_validate()`

The `isis_handle_pdu()` is a common wrapper for all IS-IS PDUs. It performs basic validity checks to the received PDU and passes it to function specific to the type of the PDU.

```
/*
 * PDU Dispatcher
 */
int
isis_handle_pdu (struct isis_circuit *circuit, u_char *ssnpa)
{

  struct isis_fixed_hdr *hdr;
  struct esis_fixed_hdr *esis_hdr;

  int retval=ISIS_OK;

  /*
   * Let’s first read data from stream to the header
   */
  hdr = (struct isis_fixed_hdr*)STREAM_DATA(circuit->rcv_stream);

  if ((hdr->idrp != ISO10589_ISIS) && (hdr->idrp != ISO9542_ESIS)){
    zlog_warn ("Not an IS-IS or ES-IS packet IDRP=%02x", hdr->idrp);
    return ISIS_ERROR;
  }

  /* now we need to know if this is an ISO 9542 packet and
   * take real good care of it, waaa!
   */
  if (hdr->idrp == ISO9542_ESIS){
    esis_hdr = (struct esis_fixed_hdr*)STREAM_DATA(circuit->rcv_stream);
    stream_set_getp (circuit->rcv_stream, ESIS_FIXED_HDR_LEN);
    /* FIXME: Need to do some acceptence tests */
    /* example length... */
    switch (esis_hdr->pdu_type) {
      case ESH_PDU:
        /* FIXME */
        break;
      case ISH_PDU:
        retval = process_is_hello (circuit);
        break;
      default:
        return ISIS_ERROR;
      }
    return retval;
  } else {
    stream_set_getp (circuit->rcv_stream, ISIS_FIXED_HDR_LEN);
  }

  /*
   * and then process it
   */
  if (hdr->length < ISIS_MINIMUM_FIXED_HDR_LEN) {
    zlog_err ("Fixed header length = %d", hdr->length);
    return ISIS_ERROR;
  }



  if (hdr->version1 != 1) {
    zlog_warn ("Unsupported ISIS version %u", hdr->version1);
    return ISIS_WARNING;
  }
  /* either 6 or 0 */
  if ((hdr->id_len | ISIS_SYS_ID_LEN) != ISIS_SYS_ID_LEN)  {
    zlog_err ("IDFieldLengthMismatch: ID Length field in a received PDU  %u, "
                      "while the parameter for this IS is %u", hdr->id_len,
                       ISIS_SYS_ID_LEN);
    return ISIS_ERROR;
  }

  if (hdr->version2 != 1) {
    zlog_warn ("Unsupported ISIS version %u", hdr->version2);
    return ISIS_WARNING;
  }
  /* either 3 or 0 */
  if ((hdr->max_area_addrs | isis->max_area_addrs) != isis->max_area_addrs) {
    zlog_err ("maximumAreaAddressesMismatch: maximumAreaAdresses in a "
                      "received PDU %u while the parameter for this IS is %u",
                       hdr->max_area_addrs, isis->max_area_addrs);
    return ISIS_ERROR;
  }

  switch (hdr->pdu_type) {
  case L1_LAN_HELLO:
    retval = process_lan_hello (ISIS_LEVEL1, circuit, ssnpa);
    break;
  case L2_LAN_HELLO:
    retval = process_lan_hello (ISIS_LEVEL2, circuit, ssnpa);
    break;
  case P2P_HELLO:
    retval = process_p2p_hello (circuit);
    break;
  case L1_LINK_STATE:
    retval = process_lsp (ISIS_LEVEL1, circuit, ssnpa);
    break;
  case L2_LINK_STATE:
    retval = process_lsp (ISIS_LEVEL2, circuit, ssnpa);
    break;
  case L1_COMPLETE_SEQ_NUM:
    retval = process_csnp (ISIS_LEVEL1, circuit, ssnpa);
    break;
  case L2_COMPLETE_SEQ_NUM:
    retval = process_csnp (ISIS_LEVEL2, circuit, ssnpa);
    break;
  case L1_PARTIAL_SEQ_NUM:
    retval = process_psnp (ISIS_LEVEL1, circuit, ssnpa);
    break;
  case L2_PARTIAL_SEQ_NUM:
    retval = process_psnp (ISIS_LEVEL2, circuit, ssnpa);
    break;
  default:
    return ISIS_ERROR;
  }

  return retval;
}
```

The variable part of received IS-IS PDUs is handled by a single function `parse_tlvs()`. The encountered TLVs are saved into the `struct tlvs` if the caller has so requested by setting flags in the `expected` corresponding those TLVs.

```
/*
 * Parses the tlvs found in the variant length part of the PDU.
 * Caller tells with flags in "expected" which TLV’s it is interested in.
 */
int
parse_tlvs (char *areatag, u_char *stream, int size, u_int32_t *expected,
            u_int32_t *found, struct tlvs *tlvs)
{
  u_char                          type, length;
  struct lan_neigh                    *lan_nei;
  struct area_addr                  *area_addr;
  struct is_neigh                      *is_nei;
  struct te_is_neigh                *te_is_nei;
  struct es_neigh                      *es_nei;
  struct lsp_entry                  *lsp_entry;
  struct in_addr                    *ipv4_addr;
  struct ipv4_reachability         *ipv4_reach;
  struct te_ipv4_reachability   *te_ipv4_reach;
#ifdef HAVE_IPV6
  struct in6_addr                   *ipv6_addr;
  struct ipv6_reachability         *ipv6_reach;
  int                           prefix_octets;
#endif /* HAVE_IPV6 */
  u_char                              virtual;
  int              value_len, retval = ISIS_OK;
  u_char                   *pnt = stream;

  *found = 0;
  memset (tlvs, 0, sizeof (struct tlvs));

  while (pnt < stream + size - 2) {
    type = *pnt;
    length = *(pnt+1);
    pnt += 2;
    value_len = 0;
    if ( pnt + length > stream + size ) {
      zlog_warn ("ISIS-TLV (%s): TLV (type %d, length %d) exceeds packet "
                 "boundaries", areatag, type, length);
      retval = ISIS_WARNING;
      break;
    }
    switch (type) {
    case AREA_ADDRESSES:
      /* +-------+-------+-------+-------+-------+-------+-------+-------+
       * |                        Address Length                         |
       * +-------+-------+-------+-------+-------+-------+-------+-------+
       * |                         Area Address                          |
       * +-------+-------+-------+-------+-------+-------+-------+-------+
       * :                                                               :
       */
      *found |= TLVFLAG_AREA_ADDRS;
#ifdef EXTREME_TLV_DEBUG
      zlog_info ("TLV Area Adresses len %d", length);
#endif /* EXTREME_TLV_DEBUG */
      if (*expected & TLVFLAG_AREA_ADDRS) {
        while (length > value_len) {
          area_addr = (struct area_addr*)pnt;
          value_len += area_addr->addr_len + 1;
          pnt +=  area_addr->addr_len + 1;
          if (!tlvs->area_addrs) tlvs->area_addrs = list_new ();
          listnode_add (tlvs->area_addrs, area_addr);
        }
      } else {
        pnt += length;
      }
      break;

    case IS_NEIGHBOURS:
      *found |= TLVFLAG_IS_NEIGHS;
#ifdef EXTREME_TLV_DEBUG
      zlog_info ("ISIS-TLV (%s): IS Neighbours length %d",
                 areatag,
                 length);
#endif /* EXTREME_TLV_DEBUG */
      if (TLVFLAG_IS_NEIGHS & *expected) {
      /* +-------+-------+-------+-------+-------+-------+-------+-------+
       * |                        Virtual Flag                           |
       * +-------+-------+-------+-------+-------+-------+-------+-------+
       */
        virtual = *pnt; /* FIXME: what is the use for this? */
        pnt++;
        value_len ++;
      /* +-------+-------+-------+-------+-------+-------+-------+-------+
       * |   0   |  I/E  |               Default Metric                  |
       * +-------+-------+-------+-------+-------+-------+-------+-------+
       * |   S   |  I/E  |               Delay Metric                    |
       * +-------+-------+-------+-------+-------+-------+-------+-------+
       * |   S   |  I/E  |               Expense Metric                  |
       * +-------+-------+-------+-------+-------+-------+-------+-------+
       * |   S   |  I/E  |               Error Metric                    |
       * +-------+-------+-------+-------+-------+-------+-------+-------+
       * |                        Neighbour ID                           |
       * +---------------------------------------------------------------+
       * :                                                               :
       */
        while (length > value_len) {
          is_nei = (struct is_neigh*)pnt;
          value_len += 4 + ISIS_SYS_ID_LEN + 1;
          pnt += 4 + ISIS_SYS_ID_LEN + 1;
          if (!tlvs->is_neighs) tlvs->is_neighs = list_new ();
          listnode_add (tlvs->is_neighs, is_nei);
        }
      } else {
        pnt += length;
      }
      break;

    case TE_IS_NEIGHBOURS:
      /* +-------+-------+-------+-------+-------+-------+-------+-------+
       * |                        Neighbour ID                           | 7
       * +---------------------------------------------------------------+
       * |                        TE Metric                              | 3
       * +---------------------------------------------------------------+
       * |                        SubTLVs Length                         | 1
       * +---------------------------------------------------------------+
       * :                                                               :
       */
      *found |= TLVFLAG_TE_IS_NEIGHS;
#ifdef EXTREME_TLV_DEBUG
      zlog_info ("ISIS-TLV (%s): Extended IS Neighbours length %d",
                 areatag,
                 length);
#endif /* EXTREME_TLV_DEBUG */
      if (TLVFLAG_TE_IS_NEIGHS & *expected) {
        while (length > value_len) {
          te_is_nei = (struct te_is_neigh*)pnt;
          value_len += 11;
          pnt += 11;
          /* FIXME - subtlvs are handled here, for now we skip */
          value_len += te_is_nei->sub_tlvs_length;
          pnt += te_is_nei->sub_tlvs_length;


          if (!tlvs->te_is_neighs) tlvs->te_is_neighs = list_new ();
          listnode_add (tlvs->te_is_neighs, te_is_nei);
        }
      } else {
        pnt += length;
      }
      break;

    case ES_NEIGHBOURS:
      /* +-------+-------+-------+-------+-------+-------+-------+-------+
       * |   0   |  I/E  |               Default Metric                  |
       * +-------+-------+-------+-------+-------+-------+-------+-------+
       * |   S   |  I/E  |               Delay Metric                    |
       * +-------+-------+-------+-------+-------+-------+-------+-------+
       * |   S   |  I/E  |               Expense Metric                  |
       * +-------+-------+-------+-------+-------+-------+-------+-------+
       * |   S   |  I/E  |               Error Metric                    |
       * +-------+-------+-------+-------+-------+-------+-------+-------+
       * |                        Neighbour ID                           |
       * +---------------------------------------------------------------+
       * |                        Neighbour ID                           |
       * +---------------------------------------------------------------+
       * :                                                               :
       */
#ifdef EXTREME_TLV_DEBUG
      zlog_info ("ISIS-TLV (%s): ES Neighbours length %d",
                 areatag,
                 length);
#endif /* EXTREME_TLV_DEBUG */
      *found |= TLVFLAG_ES_NEIGHS;
      if (*expected & TLVFLAG_ES_NEIGHS) {
        es_nei = (struct es_neigh*)pnt;
        value_len += 4;
        pnt += 4;
        while (length > value_len) {
        /* FIXME FIXME FIXME - add to the list */
        /*          sys_id->id = pnt;*/
          value_len += ISIS_SYS_ID_LEN;
          pnt += ISIS_SYS_ID_LEN;
        /*  if (!es_nei->neigh_ids) es_nei->neigh_ids = sysid;*/
        }
        if (!tlvs->es_neighs) tlvs->es_neighs = list_new ();
        listnode_add (tlvs->es_neighs, es_nei);
      } else {
        pnt += length;
      }
      break;

    case LAN_NEIGHBOURS:
      /* +-------+-------+-------+-------+-------+-------+-------+-------+
       * |                        LAN Address                            |
       * +-------+-------+-------+-------+-------+-------+-------+-------+
       * :                                                               :
       */
      *found |= TLVFLAG_LAN_NEIGHS;
      #ifdef EXTREME_TLV_DEBUG
      zlog_info ("ISIS-TLV (%s): LAN Neigbours length %d",
                 areatag,
                 length);
      #endif /* EXTREME_TLV_DEBUG */
      if (TLVFLAG_LAN_NEIGHS & *expected) {
        while (length > value_len) {
          lan_nei = (struct lan_neigh*)pnt;
          if (!tlvs->lan_neighs) tlvs->lan_neighs = list_new ();
          listnode_add (tlvs->lan_neighs, lan_nei);
          value_len += ETH_ALEN;
          pnt += ETH_ALEN;
        }
      } else {
        pnt += length;
      }
      break;

    case PADDING:
#ifdef EXTREME_TLV_DEBUG
      zlog_info ("TLV padding %d", length);
#endif /* EXTREME_TLV_DEBUG */
      pnt += length;
      break;

    case LSP_ENTRIES:
      /* +-------+-------+-------+-------+-------+-------+-------+-------+
       * |                    Remaining Lifetime                         | 2
       * +-------+-------+-------+-------+-------+-------+-------+-------+
       * |                           LSP ID                              | id+2
       * +-------+-------+-------+-------+-------+-------+-------+-------+
       * |                   LSP Sequence Number                         |<A0>4
       * +-------+-------+-------+-------+-------+-------+-------+-------+
       * |                          Checksum                             | 2
       * +-------+-------+-------+-------+-------+-------+-------+-------+
       */
#ifdef EXTREME_TLV_DEBUG
      zlog_info ("LSP Entries length %d",
                 areatag,
                 length);
#endif /* EXTREME_TLV_DEBUG */
      *found |= TLVFLAG_LSP_ENTRIES;
      if (TLVFLAG_LSP_ENTRIES & *expected) {
        while (length > value_len) {
          lsp_entry = (struct lsp_entry*)pnt;
          value_len += 10 + ISIS_SYS_ID_LEN;
          pnt +=  10 + ISIS_SYS_ID_LEN;
          if (!tlvs->lsp_entries) tlvs->lsp_entries = list_new ();
          listnode_add (tlvs->lsp_entries, lsp_entry);
        }
      } else {
        pnt += length;
      }
      break;

    case CHECKSUM:
      /* +-------+-------+-------+-------+-------+-------+-------+-------+
       * |                   16 bit fletcher CHECKSUM                    |
       * +-------+-------+-------+-------+-------+-------+-------+-------+
       * :                                                               :
       */
      *found |= TLVFLAG_CHECKSUM;
#ifdef EXTREME_TLV_DEBUG
      zlog_info ("ISIS-TLV (%s): Checksum length %d",
                 areatag,
                 length);
```

```
#endif /* EXTREME_TLV_DEBUG */
      if (*expected & TLVFLAG_CHECKSUM) {
        tlvs->checksum = (struct checksum*)pnt;
      }
      pnt += length;
      break;

    case PROTOCOLS_SUPPORTED:
      /* +-------+------+-------+------+-------+------+------+-------+
       * |                       NLPID                               |
       * +-------+------+-------+------+-------+------+------+-------+
       * :                                                           :
       */
      *found |= TLVFLAG_NLPID;
#ifdef EXTREME_TLV_DEBUG
      zlog_info ("ISIS-TLV (%s): Protocols Supported length %d",
                 areatag,
                 length);
#endif /* EXTREME_TLV_DEBUG */
      if (*expected & TLVFLAG_NLPID) {
        tlvs->nlpids = (struct nlpids*)(pnt-1);
      }
      pnt += length;
      break;

    case IPV4_ADDR:
      /* +-------+------+-------+------+-------+------+------+-------+
       * +                IP version 4 address                       + 4
       * +-------+------+-------+------+-------+------+------+-------+
       * :                                                           :
       */
      *found |= TLVFLAG_IPV4_ADDR;
#ifdef EXTREME_TLV_DEBUG
      zlog_info ("ISIS-TLV (%s): IPv4 Address length %d",
                 areatag,
                 length);
#endif /* EXTREME_TLV_DEBUG */
      if (*expected & TLVFLAG_IPV4_ADDR) {
        while (length > value_len) {
          ipv4_addr = (struct in_addr*)pnt;
          zlog_info ("ISIS-TLV (%s) : IP ADDR %s, pnt %p", areatag,
                     inet_ntoa (*ipv4_addr), pnt);
          if (!tlvs->ipv4_addrs) tlvs->ipv4_addrs = list_new();
          listnode_add (tlvs->ipv4_addrs, ipv4_addr);
          value_len += 4;
          pnt += 4;
        }
      } else {
        pnt += length;
      }
      break;

    case AUTH_INFO:
      zlog_info ("ISIS-TLV (%s): IS-IS Authentication Information",
                 areatag);
      zlog_info ("  -- Unsupported as of yet");
      pnt += length;
      break;

    case DYNAMIC_HOSTNAME:
      *found |= TLVFLAG_DYN_HOSTNAME;
#ifdef EXTREME_TLV_DEBUG
      zlog_info ("ISIS-TLV (%s): Dynamic Hostname length %d",
                 areatag,
                 length);
```

```
#endif /* EXTREME_TLV_DEBUG */
      if (*expected & TLVFLAG_DYN_HOSTNAME) {
        /* the length is also included in the pointed struct */
        tlvs->hostname = (struct hostname*)(pnt - 1);
      }
      pnt += length;
      break;

    case TE_ROUTER_ID:
      /* +-------------------------------------------------------------+
       * +                          Router ID                          + 4
       * +-------------------------------------------------------------+
       */
      *found |= TLVFLAG_TE_ROUTER_ID;
#ifdef EXTREME_TLV_DEBUG
      zlog_info ("ISIS-TLV (%s): TE Router ID %d",
                 areatag,
                 length);
#endif /* EXTREME_TLV_DEBUG */
      if (*expected & TLVFLAG_TE_ROUTER_ID) {
        tlvs->router_id = (struct te_router_id*)(pnt);
      }
      pnt += length;
      break;

    case IPV4_INT_REACHABILITY:
      /* +-------+-------+-------+-------+-------+-------+-------+-------+
       * |   0   |  I/E  |               Default Metric                  | 1
       * +-------+-------+-------+-------+-------+-------+-------+-------+
       * |   S   |  I/E  |               Delay Metric                    | 1
       * +-------+-------+-------+-------+-------+-------+-------+-------+
       * |   S   |  I/E  |               Expense Metric                  | 1
       * +-------+-------+-------+-------+-------+-------+-------+-------+
       * |   S   |  I/E  |               Error Metric                    | 1
       * +-------+-------+-------+-------+-------+-------+-------+-------+
       * |                        ip address                             | 4
       * +---------------------------------------------------------------+
       * |                        address mask                           | 4
       * +---------------------------------------------------------------+
       * :                                                               :
       */
      *found |= TLVFLAG_IPV4_INT_REACHABILITY;
#ifdef EXTREME_TLV_DEBUG
      zlog_info ("ISIS-TLV (%s): IPv4 internal Reachability length %d",
                 areatag,
                 length);
#endif /* EXTREME_TLV_DEBUG */
      if (*expected & TLVFLAG_IPV4_INT_REACHABILITY) {
        while (length > value_len) {
          ipv4_reach = (struct ipv4_reachability*)pnt;
          if (!tlvs->ipv4_int_reachs) tlvs->ipv4_int_reachs = list_new();
          listnode_add (tlvs->ipv4_int_reachs, ipv4_reach);
          value_len += 12;
          pnt += 12;
        }
      }
      else {
        pnt += length;
      }
      break;

    case IPV4_EXT_REACHABILITY:
      /* +-------+-------+-------+-------+-------+-------+-------+-------+
       * |   0   |  I/E  |               Default Metric                  | 1
       * +-------+-------+-------+-------+-------+-------+-------+-------+
       * |   S   |  I/E  |               Delay Metric                    | 1
       * +-------+-------+-------+-------+-------+-------+-------+-------+
       * |   S   |  I/E  |               Expense Metric                  | 1
       * +-------+-------+-------+-------+-------+-------+-------+-------+
       * |   S   |  I/E  |               Error Metric                    | 1
       * +-------+-------+-------+-------+-------+-------+-------+-------+
       * |                        ip address                             | 4
       * +---------------------------------------------------------------+
       * |                        address mask                           | 4
       * +---------------------------------------------------------------+
       * :                                                               :
       */
      *found |= TLVFLAG_TE_IPV4_REACHABILITY;
#ifdef EXTREME_TLV_DEBUG
      zlog_info ("ISIS-TLV (%s): IPv4 external Reachability length %d",
                 areatag,
                 length);
#endif /* EXTREME_TLV_DEBUG */
      if (*expected & TLVFLAG_TE_IPV4_REACHABILITY) {
        while (length > value_len) {
          ipv4_reach = (struct ipv4_reachability*)pnt;
          if (!tlvs->ipv4_ext_reachs) tlvs->ipv4_ext_reachs = list_new();
          listnode_add (tlvs->ipv4_ext_reachs, ipv4_reach);
          value_len += 12;
          pnt += 12;
        }
      }
      else {
        pnt += length;
      }
      break;

    case TE_IPV4_REACHABILITY:
      /* +-------+-------+-------+-------+-------+-------+-------+-------+
       * |                        TE Metric                              | 4
       * +-------+-------+-------+-------+-------+-------+-------+-------+
       * |  U/D  | sTLV? |               Prefix Mask Len                 | 1
       * +-------+-------+-------+-------+-------+-------+-------+-------+
       * |                           Prefix                              | 0-4
       * +---------------------------------------------------------------+
       * |                         sub tlvs                              |
       * +---------------------------------------------------------------+
       * :                                                               :
       */
      *found |= TLVFLAG_TE_IPV4_REACHABILITY;
#ifdef EXTREME_TLV_DEBUG
      zlog_info ("ISIS-TLV (%s): IPv4 extended Reachability length %d",
                 areatag,
                 length);
#endif /* EXTREME_TLV_DEBUG */
      if (*expected & TLVFLAG_TE_IPV4_REACHABILITY) {
        while (length > value_len) {
          te_ipv4_reach = (struct te_ipv4_reachability*)pnt;
          if (!tlvs->te_ipv4_reachs) tlvs->te_ipv4_reachs = list_new();
          listnode_add (tlvs->te_ipv4_reachs, te_ipv4_reach);
          /* this trickery is permitable since no subtlvs are defined */
          value_len += 5 + ((te_ipv4_reach->control & 0x3F) ?
                            ((((te_ipv4_reach->control & 0x3F)-1)>>3)+1) : 0);
          pnt +=  5 + ((te_ipv4_reach->control & 0x3F) ?
                       ((((te_ipv4_reach->control & 0x3F)-1)>>3)+1) : 0);
        }
      }
      else {
        pnt += length;
      }
      break;

#ifdef  HAVE_IPV6
    case IPV6_ADDR:
      /* +-------+------+------+------+------+------+------+-------+
       * +                IP version 6 address                     + 16
       * +-------+------+------+------+------+------+------+-------+
       * :                                                         :
       */
      *found |= TLVFLAG_IPV6_ADDR;
#ifdef EXTREME_TLV_DEBUG
      zlog_info ("ISIS-TLV (%s): IPv6 Address length %d",
                 areatag,
                 length);
#endif /* EXTREME_TLV_DEBUG */
      if (*expected & TLVFLAG_IPV6_ADDR) {
        while (length > value_len) {
          ipv6_addr = (struct in6_addr*)pnt;
          if (!tlvs->ipv6_addrs) tlvs->ipv6_addrs = list_new();
          listnode_add (tlvs->ipv6_addrs, ipv6_addr);
          value_len += 16;
          pnt += 16;
        }
      } else {
        pnt += length;
      }
      break;

    case IPV6_REACHABILITY:
      /* +-------+------+------+------+------+------+------+-------+
       * |                Default Metric                           | 4
       * +-------+------+------+------+------+------+------+-------+
       * |                       Control Informantion              |
       * +---------------------------------------------------------+
       * |                       IPv6 Prefix Length                |--+
       * +---------------------------------------------------------+  |
       * |                       IPv6 Prefix                       |<-+
       * +---------------------------------------------------------+
       */
      *found |= TLVFLAG_IPV6_REACHABILITY;
      if (*expected & TLVFLAG_IPV6_REACHABILITY) {
        while (length > value_len) {
          ipv6_reach = (struct ipv6_reachability*)pnt;
          prefix_octets = ((ipv6_reach->prefix_len + 7) / 8);
          value_len += prefix_octets + 6;
          pnt +=  prefix_octets + 6;
          /* FIXME: sub-tlvs */
          if (!tlvs->ipv6_reachs) tlvs->ipv6_reachs = list_new();
          listnode_add (tlvs->ipv6_reachs, ipv6_reach);
        }
      } else {
        pnt += length;
      }
      break;
#endif /* HAVE_IPV6 */

    case WAY3_HELLO:
      /* +---------------------------------------------------------+
       * |                  Adjacency state                        | 1
       * +---------------------------------------------------------+
       * |                  Extended Local Circuit ID              | 4
       * +---------------------------------------------------------+
       * |                  Neighbor System ID (If known)          | 0-8
       *                                     (probably 6)
       * +---------------------------------------------------------+
       * |                  Neighbor Local Circuit ID (If known)        | 4
       * +---------------------------------------------------------------+
       */
      *found |= TLVFLAG_3WAY_HELLO;
      if (*expected & TLVFLAG_3WAY_HELLO) {
        while (length > value_len) {
        /* FIXME: make this work */
/*           Adjacency State (one octet):
              0 = Up
              1 = Initializing
              2 = Down
            Extended Local Circuit ID (four octets)
            Neighbor System ID if known (zero to eight octets)
            Neighbor Extended Local Circuit ID (four octets, if Neighbor
              System ID is present) */
          pnt += length;
        }
      } else {
        pnt += length;
      }

      break;

    default:
      zlog_warn ("ISIS-TLV (%s): unsupported TLV type %d, length %d",
                 areatag,
                 type,
                 length);

      retval = ISIS_WARNING;
      pnt += length;
      break;
    }
  }

  return retval;
}
```

The writing of TLVs is performed by different functions for each TLV type. Each of the functions utilizes the function `add_tlv()`. Below writing of *Area Addresses* TLV is shown.

```
int
add_tlv (u_char tag, u_char len, u_char *value, struct stream *stream)
{

  if (STREAM_SIZE (stream) - stream_get_putp (stream)  < len + 2) {
    zlog_warn ("No room for TLV of type %d", tag);
    return ISIS_WARNING;
  }

  stream_putc (stream, tag);  /* TAG */
  stream_putc (stream, len);  /* LENGTH */
  stream_put (stream, value, (int)len); /* VALUE */

#ifdef EXTREME_DEBUG
  zlog_info ("Added TLV %d len %d", tag, len);
#endif /* EXTREME DEBUG */
  return ISIS_OK;
}
```

```
int
tlv_add_area_addrs (struct list *area_addrs, struct stream *stream)
{
  struct listnode *node;
  struct area_addr *area_addr;

  u_char value [255];
  u_char *pos = value;

  for (node = listhead (area_addrs); node; nextnode (node)) {
    area_addr = getdata (node);
    if (pos - value + area_addr->addr_len > 255)
      goto err;
    *pos = area_addr->addr_len;
    pos ++;
    memcpy (pos, area_addr->area_addr, (int)area_addr->addr_len);
    pos += area_addr->addr_len;
  }

  return add_tlv (AREA_ADDRESSES, pos - value, value, stream);

 err:
  zlog_warn ("tlv_add_area_addrs(): TLV longer than 255");
  return ISIS_WARNING;
}
```

The election of LAN DIS is performed by the function `isis_dr_elect()`. The function first finds a candidate for the LAN DIS from the adjacencies on the circuit and then compares its priority to the priority of the IS itself.

```
int
isis_dr_elect (struct isis_circuit *circuit, int level)
{
  struct hash *adjdb;
  struct listnode *node;
  struct isis_adjacency *adj, *adj_dr = NULL;
  struct list *list = list_new ();
  u_char own_prio, biggest_prio = 0;
  int cmp_res, retval = ISIS_OK;

  own_prio = circuit->u.bc.priority[level-1];
  adjdb = circuit->u.bc.adjdb[level - 1];
  if(!adjdb) {
    zlog_warn ("isis_dr_elect() adjdb == NULL");
    retval = ISIS_WARNING;
    list_delete (list);
    goto out;
  }
  hash_iterate (adjdb,
                (void (*) (struct hash_backet *, void *))
                isis_adj_build_up_list, list);
  /*
   * Loop the adjacencies and find the one with the biggest priority
   */
  for (node = listhead (list); node; nextnode (node)) {
    adj = getdata (node);
    /* clear flag for show output */
    adj->dis_record[level-1].dis = ISIS_IS_NOT_DIS;
    adj->dis_record[level-1].last_dis_change = time(NULL);
    if (adj->prio[level-1] > biggest_prio) {
      biggest_prio = adj->prio[level-1];
      adj_dr = adj;
    } else if (adj->prio[level-1] == biggest_prio) {
      /*
       * Comparison of MACs breaks a tie
       */
      if (adj_dr) {
        cmp_res = memcmp (adj_dr->snpa, adj->snpa, ETH_ALEN);
        if (cmp_res < 0) {
          adj_dr = adj;
        }
        if (cmp_res == 0)
          zlog_warn ("isis_dr_elect(): multiple adjacencies with same SNPA");
      } else {
        adj_dr = adj;
      }
    }
  }
  if (!adj_dr) {
    /*
     * Could not find the DR - means we are alone and thus the DR
     */
    if ( !circuit->u.bc.is_dr[level - 1]) {
      list_delete (list);
      list = NULL;
      return isis_dr_commence (circuit, level);
    }
    goto out;
  }
  /*
   * Now we have the DR adjacency, compare it to self
   */
  if (adj_dr->prio[level-1] < own_prio || (adj_dr->prio[level-1] == own_prio &&
                                  memcmp (adj_dr->snpa, circuit->u.bc.snpa,
                                          ETH_ALEN) < 0)) {
    if (!circuit->u.bc.is_dr[level - 1]) {
      /*
       * We are the DR -> commence
       */
      list_delete (list);
      return isis_dr_commence (circuit, level);
    }
  } else {
    /* ok we have found the DIS - lets mark the adjacency */
    /* set flag for show output */
    adj_dr->dis_record[level - 1].dis = ISIS_IS_DIS;
    adj_dr->dis_record[level - 1].last_dis_change = time(NULL);
    /* now loop through a second time to check if there has been a DIS change
     * if yes rotate the history log
     */
    for (node = listhead (list); node; nextnode (node)) {
      adj = getdata (node);
      isis_check_dr_change(adj, level);
    }
    /*
     * We are not DR - if we were -> resign
     */
    if (circuit->u.bc.is_dr[level - 1]) {
      list_delete (list);
      return isis_dr_resign (circuit, level);
    }
  }
 out:
  if (list)
    list_delete (list);
  return retval;
}
```

The `isis_run_spf()` function takes care of running the Dijkstra algorithm. First it pre-loads the TENT database and then processes it in a loop adding vertices to the PATH database when appropriate. The `add_to_paths()` function calls `isis_route_create()` for vertices that contain IPv4 or IPv6 address reachability information.

```
void
add_to_paths (struct isis_spftree *spftree, struct isis_vertex *vertex,
              struct isis_area *area)
{

#ifdef EXTREME_DEBUG
  u_char buff[BUFSIZ];
#endif /* EXTREME_DEBUG */
  listnode_add (spftree->paths, vertex);

#ifdef EXTREME_DEBUG
  zlog_info ("ISIS-Spf: added  %s %s depth %d dist %d to PATHS",
             vtype2string(vertex->type), vid2string(vertex, buff),
             vertex->depth, vertex->d_N);
#endif /* EXTREME_DEBUG */
  if (vertex->type > VTYPE_ES) {
    if (listcount(vertex->Adj_N) > 0)
      isis_route_create ((struct prefix *)&vertex->N.prefix,
                         vertex->d_N, vertex->depth, vertex->Adj_N, area);
    else if (isis->debugs & DEBUG_SPF_EVENTS)
      zlog_info ("ISIS-Spf: no adjacencies do not install route");
  }

  return;
}

int
isis_run_spf (struct isis_area *area, int level, int family)
{
  int retval = ISIS_OK;
  struct listnode *node;
  struct isis_vertex *vertex;
  struct isis_spftree *spftree = NULL;
  u_char lsp_id[ISIS_SYS_ID_LEN + 2];
  struct isis_lsp *lsp;

  if (family == AF_INET)
    spftree = area->spftree[level - 1];
#ifdef HAVE_IPV6
  else if (family == AF_INET6)
    spftree = area->spftree6[level - 1];
#endif

  assert (spftree);

  /*
   * C.2.5 Step 0
   */
  init_spt (spftree);
  /*              a) */
  isis_spf_add_self (spftree, area, level);
  /*              b) */
  retval = isis_spf_preload_tent (spftree, area, level, family);
  /*
   * C.2.7 Step 2
   */
  if (listcount (spftree->tents) == 0) {
    zlog_warn ("ISIS-Spf: TENT is empty");
    spftree->lastrun = time (NULL);
    return retval;
  }

  while (listcount (spftree->tents) > 0) {
    node = listhead (spftree->tents);
    vertex = getdata (node);
    /* Remove from tent list */
    list_delete_node (spftree->tents, node);
    if (isis_find_vertex (spftree->paths, vertex->N.id, vertex->type))
      continue;
    add_to_paths (spftree, vertex, area);
    if (vertex->type == VTYPE_PSEUDO_IS ||
        vertex->type == VTYPE_NONPSEUDO_IS) {
      memcpy (lsp_id, vertex->N.id, ISIS_SYS_ID_LEN + 1);
      LSP_FRAGMENT(lsp_id) = 0;
      lsp = lsp_search (lsp_id, area->lspdb[level - 1]);
      if (lsp) {
        if (LSP_PSEUDO_ID (lsp_id)) {
          isis_spf_process_pseudo_lsp (spftree, lsp, vertex->d_N,
                                       vertex->depth, family);

        } else {
          isis_spf_process_lsp (spftree, lsp, vertex->d_N, vertex->depth,
                                family);
        }
      } else {
        zlog_warn ("ISIS-Spf: No LSP found for %s", rawlspid_print (lsp_id));
      }
    }
  }

  thread_add_event (master, isis_route_validate, area, 0);
  spftree->lastrun = time (NULL);
  spftree->pending = 0;

  if (level == 1)
    spftree->t_spf_periodic =  thread_add_timer (master,
                                                 isis_run_spf_l1, area,
                                                 isis_jitter
                                                 (PERIODIC_SPF_INTERVAL, 10));
  else
    spftree->t_spf_periodic =  thread_add_timer (master,
                                                 isis_run_spf_l2, area,
                                                 isis_jitter
                                                 (PERIODIC_SPF_INTERVAL, 10));

  return retval;
}
```

The routes are inserted to the area's route table by the function `isis_route_create()`. The function creates a routing entry and checks if a similar entry exists. If there already exists a route to the given prefix, the routes are compared. The comparison is based on the cost of the route. If the routes have the same cost the next hop lists are merged.

```
struct isis_route_info *
isis_route_create (struct prefix *prefix, u_int32_t cost, u_int32_t depth,
                   struct list *adjacencies, struct isis_area *area)
{
  struct route_node *route_node;
  struct isis_route_info *rinfo_new, *rinfo_old, *route_info = NULL;
  u_char buff[BUFSIZ];
  u_char family;

  family = prefix->family;
  /* for debugs */
  prefix2str (prefix, buff, BUFSIZ);

  rinfo_new = isis_route_info_new (cost, depth, family, adjacencies);
  if (!rinfo_new) {
    zlog_err ("ISIS-Rte (%s): isis_route_create: out of memory!",
              area->area_tag);
    return NULL;
  }

  if (family == AF_INET)
    route_node = route_node_get (area->route_table, prefix);
#ifdef HAVE_IPV6
  else if (family == AF_INET6)
    route_node = route_node_get (area->route_table6, prefix);
#endif /* HAVE_IPV6 */
  else
    return NULL;
  rinfo_old = route_node->info;
  if (!rinfo_old) {
    if (isis->debugs & DEBUG_RTE_EVENTS)
      zlog_info ("ISIS-Rte (%s) route created: %s", area->area_tag, buff);
    SET_FLAG(rinfo_new->flag, ISIS_ROUTE_FLAG_ACTIVE);
    route_node->info = rinfo_new;
    return rinfo_new;
  }

  if (isis->debugs & DEBUG_RTE_EVENTS)
    zlog_info ("ISIS-Rte (%s) route already exists: %s", area->area_tag, buff);

  if (isis_route_info_same (rinfo_new, rinfo_old, family)) {
    if (isis->debugs & DEBUG_RTE_EVENTS)
      zlog_info ("ISIS-Rte (%s) route unchanged: %s", area->area_tag, buff);
    isis_route_info_delete (rinfo_new);
    route_info = rinfo_old;
  } else if (isis_route_info_same_attrib (rinfo_new, rinfo_old)) {
    /* merge the nexthop lists */
    if (isis->debugs & DEBUG_RTE_EVENTS)
        zlog_info ("ISIS-Rte (%s) route changed (same attribs): %s",
                   area->area_tag, buff);
#ifdef EXTREME_DEBUG
    zlog_info ("Old nexthops");
    nexthops6_print (rinfo_old->nexthops6);
    zlog_info ("New nexthops");
    nexthops6_print (rinfo_new->nexthops6);
#endif /* EXTREME_DEBUG */
    isis_route_info_merge (rinfo_new, rinfo_old, family);
    isis_route_info_delete (rinfo_new);
    route_info = rinfo_old;
  } else {
    if (isis_route_info_prefer_new (rinfo_new, rinfo_old)) {
      if (isis->debugs & DEBUG_RTE_EVENTS)
        zlog_info ("ISIS-Rte (%s) route changed: %s", area->area_tag, buff);
      isis_route_info_delete (rinfo_old);
      route_info = rinfo_new;
    } else {
      if (isis->debugs & DEBUG_RTE_EVENTS)
        zlog_info ("ISIS-Rte (%s) route rejected: %s", area->area_tag, buff);
      isis_route_info_delete (rinfo_new);
      route_info = rinfo_old;
    }
  }

  SET_FLAG (route_info->flag, ISIS_ROUTE_FLAG_ACTIVE);
  route_node->info = route_info;

  return route_info;
}
```

The thread for `isis_route_validate()` is scheduled to be executed soon after the Dijkstra algorithm. The functions inspects the routes of the area and calls the function `zebra_route_update()` which in turn informs the Zebra daemon of the changes in the route table.

```
int
isis_route_validate (struct thread *thread)
{
  struct isis_area *area;
  struct route_table *table;
  struct route_node *rode;
  struct isis_route_info *rinfo;
  u_char buff[BUFSIZ];
#ifdef HAVE_IPV6
  int v6done = 0;
#endif
  area = THREAD_ARG (thread);
  table = area->route_table;
#ifdef HAVE_IPV6
 again:
#endif
  for (rode = route_top (table); rode; rode = route_next (rode)) {
    if (rode->info == NULL)
      continue;
    rinfo = rode->info;

    if (isis->debugs & DEBUG_RTE_EVENTS) {
      prefix2str (&rode->p, buff, BUFSIZ);
      zlog_info ("ISIS-Rte (%s): route validate: %s %s %s",
                 area->area_tag,
                 (CHECK_FLAG (rinfo->flag, ISIS_ROUTE_FLAG_ZEBRA_SYNC) ?
                  "sync'ed": "nosync"),
                 (CHECK_FLAG (rinfo->flag, ISIS_ROUTE_FLAG_ACTIVE) ?
                  "active": "inactive"), buff);
    }

    isis_zebra_route_update (&rode->p, rinfo);
    if (!CHECK_FLAG (rinfo->flag, ISIS_ROUTE_FLAG_ACTIVE))
      isis_route_delete (&rode->p, area->route_table);
  }
#ifdef HAVE_IPV6
  if (v6done)
    return ISIS_OK;
  table = area->route_table6;
  v6done = 1;
  goto again;
#endif

  return ISIS_OK;
}
```

Appendix B - Files of ISISd

Filename	Lines	Content
dict.[ch]	1640	the Kazlib red-black tree
isis_adjacency.[ch]	632	the adjacency database
isis_circuit.[ch]	2301	circuit related routines
isis_common.h	56	common definitions for all the modules
isis_constants.h	151	some constants used in the implementation
isis_csm.[ch]	233	circuit state machine
isis_dr.[ch]	419	LAN DIS routines
isis_dynhn.[ch]	165	dynamic hostname cache
isis_events.[ch]	381	event wrapper for events such as system type change
isis_flags.[ch]	129	implementation of SRM and SSN flags
isis_lsp.[ch]	2428	LSP database and LSP generation routines
isis_main.c	328	the main() routine
isis_misc.[ch]	521	miscellaneous helper routines
isis_network.[ch]	659	OSI Layer 2 network I/O
isis_pdu.[ch]	2534	handling of IS-IS PDUs
isis_route.[ch]	667	handling of routes
isis_spf.[ch]	1343	the SPT algorithm
isis_tlv.[ch]	1258	processing of TLVs
isis_zebra.[ch]	607	interface to Zebra
isisd.[ch]	2003	top level datastuctures and related routines
iso_checksum.[ch]	221	implementation of the Fletcher cheksum
total	18676	