ZebOS-XP®
Network Platform

Version 1.4
Extended Performance

Label Distribution Protocol
Developer Guide

December 2015

IP Infusion Inc. Proprietary

IP Infusion Inc.
3965 Freedom Circle, Suite 200
Santa Clara, CA 95054
+1 408-400-1900
http://www.ipinfusion.com/

For support, questions, or comments via E-mail, contact:
support@ipinfusion.com

Contents

Preface

This guide describes the ZebOS-XP application programming interface (API) for Label Distribution Protocol (LDP).

Audience

This guide is intended for developers who write code to customize and extend LDP.

Conventions

Table P-1 shows the conventions used in this guide.

Table P-1: Conventions

Convention	Description
Italics	Emphasized terms; titles of books
Note:	Special instructions, suggestions, or warnings
`monospaced type`	Code elements such as commands, functions, parameters, files, and directories

Contents

This guide contains these chapters:

- Chapter 1, *Introduction*
- Chapter 2, *LDP Internal Architecture*
- Chapter 3, *LDP Graceful Restart*
- Chapter 4, *LDP MD5 Authentication*
- Chapter 5, *Inter-Area Label Switched Paths*
- Chapter 6, *LDP IGP Synchronization*
- Chapter 7, *Pseudowire Redundancy*
- Chapter 8, *Data Structures*
- Chapter 9, *LDP API*
- Chapter 10, *LDP Traps*
- Chapter 11, *LDP MIBs API*

Related Documents

The following guides are related to this document:

- *Label Distribution Protocol Command Reference*
- *Network Services Module Developer Guide*
- *Network Services Module Command Reference*
- *Installation Guide*
- *Architecture Guide*

Note: All ZebOS-XP technical manuals are available to licensed customers at http://www.ipinfusion.com/support/document_list.

Support

For support-related questions, contact support@ipinfusion.com.

Comments

If you have comments, or need to report a problem with the content, contact techpubs@ipinfusion.com.

CHAPTER 1 Introduction

The Label Distribution Protocol (LDP) is a routing component of MPLS (Multi-Protocol Label Switching) technology.

Overview

The LDP daemon uses NSM services to obtain routing information. Routers send out Hello packets to establish Hello Adjacencies with other nearby routers. This allows sessions between routers to be established during which routers exchange labels in preparation for forwarding packets.

LDP generates labels for, and exchanges labels between, peer routers. It works with other routing protocols (RIP, OSPF and BGP) to create the label-switched paths (LSPs) used when forwarding packets. An LSP is the path taken by all packets that belong to the Forwarding Equivalence Class (FEC) corresponding to that LSP. This is analogous to establishing a virtual circuit in ATM (Asynchronous Transfer Mechanism). In this way, ZebOS-XP LDP assigns labels to every destination address and destination prefix provided by ZebOS-XP. The LDP interface to the MPLS forwarder adds labels to, and deletes labels from, the forwarding tables.

LDP Adjacencies

LDP defines a mechanism for discovering adjacent, LDP-capable Label Switching Routers (LSR) that participate in label switching (adjacencies). Whenever a new router comes up, it sends out a hello packet to a specified, multicast address announcing itself to the network. Every router directly connected to the network receives the packet. Receipt of a hello packet from another LSR creates a *Hello Adjacency* with that LSR. To create a Hello Adjacency with an LSR that cannot send/receive multicast packets, LDP allows a router to be manually configured to send unicast Hello packets to non-multicast LSRs. This non-multicast LSR is a *targeted peer*. Adjacencies are maintained by sending out periodic Hello packets to the multicast group, and to all targeted peers. Hello packets are sent using UDP.

LDP Sessions

LDP-capable LSRs establish a session before exchanging label information. All session messages are sent using TCP to ensure reliable delivery. After the LSRs establish a session and negotiate options, a given pair of routers may exchange label information. The labels exchanged over a session are valid only during the lifetime of the session, and routers release them when session is closed.

Forwarding Equivalence Class

A Forwarding Equivalence Class section defines a set of packets that would be forwarded on the same path by the MPLS network. To define FEC, IPv4 routes are advertised by two common methods:

Host Address. The LSR uses the address of the destination host to create this FEC: all packets going to this destination will take the same LSP. .

Prefix. The LSR uses the destination prefix to create this FEC: all packets take the LSP corresponding to the longest matching prefix. .

Label Generation

An LDP Label is a 20-bit number the LSR uses to forward a packet to its destination. When an LSR creates a new FEC, the router generates new labels and distributes them to its peers. A router keeps both incoming and outgoing labels in its database.

Label Distribution Modes

The ZebOS-XP LDP implementation supports two label distribution modes:

Downstream Unsolicited. In this mode, next hop LSRs distribute labels to peers without waiting for a label request. .

Downstream on Demand. In this mode, an LSR distributes a label to a peer only if there is a pending label request . from the peer.

Label Retention Mode

ZebOS-XP LDP supports two label retention modes:

Liberal Retention Mode. In this mode, the LSR retains all labels received from all sources. This mode helps in fast . LSP setup, in case of a change in next hop.

Conservative Retention Mode. In this mode, the LSR retains only the labels received from peers that are the next . hop for a given FEC. This mode is used by LSRs that have a constraint on the number of labels that it can retain at any given time.

LSP Control

LSPs can be set up in the following ways:

Ordered Control. In this mode, an LSR distributes a label for an FEC to its peer only if it has a corresponding label . from its next hop, or if it is the egress node.

Independent Control. In this mode, an LSR may distribute a label to its peers without waiting for a corresponding label. from its next hop.

Loop Detection

Loop detection can be enabled to detect routing loops in LSPs. There are two methods supported for the loop detection mechanism:

Hop Count. During setup of an LSP, the LSP passes hop count with the LSP setup messages. This hop count is . incremented by each node router participating in LSP establishment. If the hop count exceeds the maximum configured value, the LSP setup process is stopped, and a notification message is passed back to the message originator.

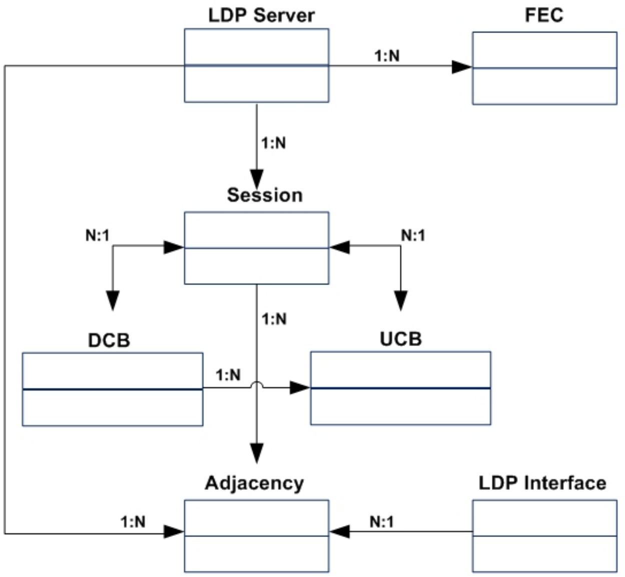
Path Vector. A path vector contains a list of LSR identifiers. This is passed as a part of LSP setup messages. Each . LSR participating in the LSP establishment adds its own LSR identifier to the path vector. If an LSR finds its own identifier in the path vector, it drops the message, and sends a message back to the originator.

Using these messages ensures that a loop is detected while establishing a label switched path, before any data is passed over that LSP.

CHAPTER 2

LDP Internal Architecture

The following diagram depicts the main entities used for implementing LDP, and the relationship between them. The relationship between a pair of entities (for example, session, adjacency) is illustrated using arrows.



A→B (1:N) = One instance of entity A can have more than one instance of entity B.

A→B (N:1) = One instance of entity B can belong to more than one instance of entity A.

A→B (N:M) = One instance of entity A can have mulitple instances of entity B and one instance of entity B can have mulitple instances of entity A.

Figure 2-1: LDP Entities

- **LDP Server.** This is a global entity which provides access to all other main entities (such as session, FEC). There is only one instance of this entity in the whole system.
- **FEC.** This global entity stores all routing updates received from NSM. These routing entries are used to define FECs for which MPLS labels are exchanged between LDP peers.
- **Session.** A session entity is created to communicate with an LDP peer. Among other information, it contains a list of labels exchanged with an LDP peer.
- **Adjacency.** An adjacency entity is used to discover and maintain sessions with peer LDP capable routers. It identifies the interface being used to reach an LDP peer, and also, the LSR ID of the peer.
- **DCB.** The Downstream Control Block (DCB) maintains control information regarding labels received from peer LDP routers. There is a DCB associated with every label received from a peer.
- **UCB.** The Upstream Control Block (UCB) maintains control information regarding labels sent to peer LDP routers. There is a UCB associated with every label sent to peer LDP routers.
- **LDP Interface.** The set of interfaces and associated information received from NSM.

LDP Initialization Finite State Machine

LDP session establishment procedure is implemented as a Finite State Machine (FSM). The following State Transition diagram illustrates the possible states and transitions in response to events.

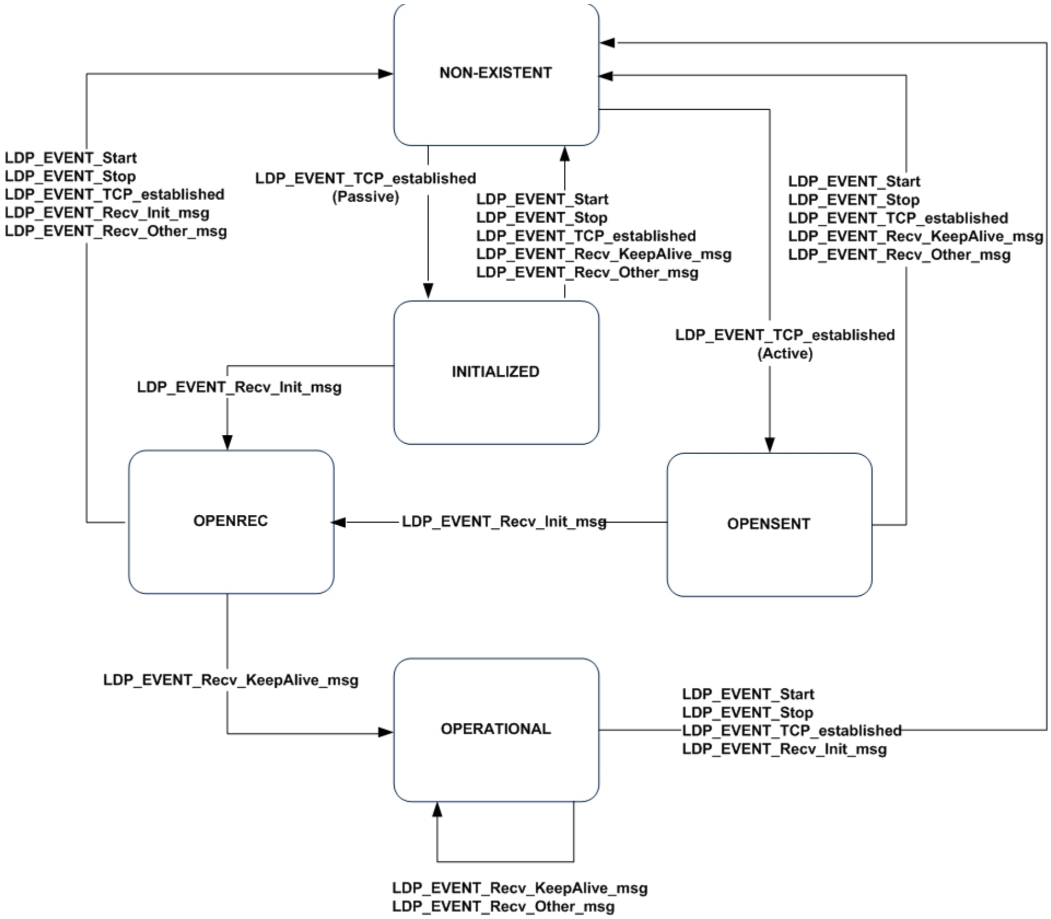


Figure 2-2: LDP Initialization FSM

Session Initialization State Transition

Following are the five states defined by LDP initialization FSM

LDP_STATE_NON_EXISTENT

A session enters this state as soon as it is created. It might reenter this state if an error occurs and the session ends.

Valid Event # 1 (`LDP_EVENT_Start`)

> Start the FSM. If the LSR is passive, it waits for a connection from it's active peer. If the LSR is active, it tries to connect to its passive peer.
>
> Next State: `LDP_STATE_NON_EXISTENT`

Valid Event # 2 (`LDP_EVENT_TCP_Established`)

> For active LSRs, the connect socket is available with read/write, this means the connection to the passive peer succeeded. If the LSR is passive, the select call returned, this means the active peer connected to its passive peer.

Next State: `LDP_STATE_INITIALIZED`.

LDP_STATE_INITIALIZED

For active sessions, this state is reached when the TCP connection is initiated. For passive sessions, this state is reached when the sessions starts listening on the well-known LDP port.

Valid Event # 1 (`LDP_EVENT_Stop`)

Stop the FSM, and clean up the session parameters, which includes cancelling all events and all spawned threads.

Next State: `LDP_STATE_NON_EXISTENT`

Valid Event # 2 (`LDP_EVENT_Recv_Init_msg`)

Wait for an initialization message from the peer. (This applies only to passive peers.)

Next State: `LDP_STATE_OPEN_REC`.

LDP_STATE_OPENSENT

This state is only reached by active sessions. This state indicates that the active peer has transmitted an `INITIALIZATION` message to its peer, and is waiting for an `INITIALIZATION` message in reply.

Valid Event # 1 (`LDP_EVENT_Stop`)

Close all sockets, and stop all threads for the session.

Next State: `LDP_STATE_NON_EXISTENT`.

Valid Event # 2 (`LDP_EVENT_Recv_Init_msg`):

Wait for an initialization message from the peer. (This applies only to active peers.)

Next State: `LDP_STATE_OPEN_REC`.

LDP_STATE_OPENREC

This state is reached by both active and passive sessions. The event that drives both into this state is the receipt of an `INITIALIZATION` from the peer. For passive sessions, an `INITIALIZATION` message is sent in reply, along with a keepalive message.

Valid Event # 1 (`LDP_EVENT_Stop`)

Close all sockets, and stop all threads for the session.

Next State: `LDP_STATE_NON_EXISTENT`.

Valid Event # 2 (`LDP_EVENT_Recv_KeepAlive_msg`)

The session is now finalized for both ends.

Next State: `LDP_STATE_OPERATIONAL`.

LDP_STATE_OPERATIONAL

This state is reached upon the receipt of a keepalive message for both active and passive sessions. Once this state is reached, all LDP messages might be legally exchanged.

Valid Event # 1 (`LDP_EVENT_Stop`)

Close all sockets, and stop all threads for the session.

Next State: `LDP_STATE_NON_EXISTENT`.

Valid Event # 2 (`LDP_EVENT_Recv_KeepAlive_msg`)

Session is still good.

Next State: `LDP_STATE_OPERATIONAL`. (No change in state.)

Valid Event # 3 (`LDP_EVENT_Recv_Other_msg`)

Receive other LDP specific messages.

Next State: `LDP_STATE_OPERATIONAL`. (No change in state.)

Note: Given two LSRs, LSRa and LSRb, if the integer representation LSRa's IP address is greater than the integer representation of LSRb's IP address, LSRa will act as the Active LSR in the TCP connection setup (client for the TCP connection), and LSRb will act as the Passive LSR in the TCP connection setup (server for the TCP connection).

Session Events

LDP_EVENT_Start. Try and set up the TCP connection. For active, this means try and connect. For passive, this means listen to the socket for incoming connections. .

LDP_EVENT_Stop. Invoked when the session needs to be restarted. Could be due to an error, or due to a management decision. .

LDP_EVENT_TCP_established. TCP connection is successful. .

LDP_EVENT_Recv_Init_msg. Receive initialization message. Refer to state explanation in *Session Initialization State Transition*. .

LDP_EVENT_Recv_KeepAlive_msg. Receive keepalive message. Refer to state explanation in *Session Initialization State Transition*. .

LDP_EVENT_Recv_Other_msg. All other legal LDP messages, including notifications. .

Timers associated with LDP

ZebOS-XP LDP uses the following timers/timeouts.

- Session Re-Connect Timer: Two peers always try and maintain a session between each other. As soon as an LDP_EVENT_Stop event is received, the session reconnect timer is started. This timer has a minimum value of 15 seconds, and is rapidly increased to the maximum of 120 seconds, and held constant.
- Session Keep Alive Interval Timer: This interval can be configured per LSR or per interface. This interval governs the frequency with which keep-alive packets are sent to a peer. It is advised that this value be no greater than one-third the value of the keep-alive timeout.
- Session Keep Alive Timeout Timer: This timeout interval can be configured per LSR or per interface. This timeout defines the interval an LSR waits for the receipt of a keep-alive packet before it sends an LDP_EVENT_Stop event to the FSM. This timeout is reset each time the session holding this timeout receives a keep-alive packet.
- Adjacency Hello Interval Timer: This interval can be configured per LSR or per interface. This interval governs the frequency hello packets are sent to a peer. Set this value no greater than one-third the value of the hello timeout.
- Adjacency Hello Timeout Timer: This timeout can be configured per LSR or per interface. This timeout defines the time an LSR waits for the receipt of a hello packet before it deletes the hello adjacency for this peer. This timeout is reset when the session holding this timeout receives a hello packet.
- Request Retry Timer: This timer is restarted every time a loop is detected. The LSR waits for one request retry timer cycle before it tries to request labels for the given FEC again.
- TCP Session Re-Connect Timer: This timer is restarted every time the TCP connection phase between two peers is broken down, before it ever reaches the “established” phase. The time value is currently hard-coded to be 3 seconds.

CHAPTER 3 LDP Graceful Restart

The ZebOS-XP Graceful Restart feature for LDP reduces the impact on MPLS Forwarding due to the restart of the LDP module.

Introduction

Under normal conditions, LDP LSRs clear FEC-Label bindings learned from the restarting LSR and the restarting LSR may signal new labels after completion of graceful restart. As a result, MPLS Forwarding is impacted during the restart. With graceful-restart capability enabled, adjacent routers exchange each others' restart capability in their Initialization messages, and a restart capability per session is established at session startup. Subsequently, whenever a router goes down, it preserves its MPLS forwarding table entries. Peer routers detect session shutdown and preserve FEC-Label bindings subject to the session being restart-capable.

When LDP restarts gracefully, minimal or no changes are made to the forwarding table entries and MPLS forwarding continues uninterrupted, thereby achieving Non-Stop Forwarding (NSF). This mechanism ensures that MPLS forwarding remains intact during transient changes in the control plane.

Features Supported

The ZebOS-XP LDP graceful restart feature is based on the following standard:

- RFC 3478 - Graceful Restart Mechanism for Label Distribution Protocol.

Routers that separate control and management tasks from data forwarding tasks are well-suited to the graceful restart feature. Network personnel initiate graceful restarts. Graceful restart is possible when the network topology is stable and the restarting router and its peers retain their forwarding table entries.

Non-Stop Forwarding

The following capabilities in ZebOS-XP LDP support Non Stop Forwarding (NSF) on routers. These features make the Graceful Restart feature robust.

- When LDP restarts, NSM marks the LDP label block and LDP installed forwarding entries as stale.
- When LDP restarts within the reconnect time-out period, NSM synchronizes the stale entries to LDP. LDP maintains this stale copy/shadow database. Whenever a label is to be allocated, LDP looks up the stale copy/ shadow database for the preserved label. If an entry is found for a corresponding FEC, the label is marked as in use, and the same label is advertised. If no preserved or stale label is available for an FEC, LDP allocates a new label.
- When NSM receives FTN and ILM add messages from LDP, it unsets the stale mark for the forwarding entry and does FTM and ILM updates, resulting in minimal or no change to the MPLS forwarding entry.
- When the recovery timer expires, LDP cleans up the shadow database of preserved or stale forwarding entries, and unsets the label block stale flag.
- If LDP does not start within the reconnect time-out period, the forwarding timer expiration in NSM removes the stale FTM and ILM entries.

- If LDP starts, but then exits from Graceful Restart, it unsets the stale label block flag, cleans up the shadow database, and sends a message to NSM to cleanup the stale entries.

Restart Capability Exchange

Graceful restart capability is disabled by default. This can be modified using the `graceful-restart (enable|disable)` command. It sets the Instance level capability with respect to graceful restart for a router.

If graceful-restart capability is enabled for a router, it communicates the information to its peer while sending the Initialization message, using the FT session TLV.

The following describes the layout of FT session TLV:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1|0| FT Session TLV (0x0503)   |      Length (= 12)            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     FT Flags                  |      Reserved                 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                FT Reconnect Timeout (in milliseconds)         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                Recovery Time (in milliseconds)                |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The FT Flags field is made up of the following information:

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|R|         Reserved    |S|A|C|L|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Routers supporting Graceful Restart should set the L bit in FT Flags. The FT Reconnect time-out is the time for which the restarting router's peer retains forwarding entries across the restart. It is negotiated in the initialization messages as the minimum of received value and per-router-neighbor liveness time configured. The default value is 120 seconds.

Recovery time is applicable after restart. It is the maximum time until which stale entries are maintained after the session is re-initiated between the restarting router and its peer. It is negotiated in the initialization messages exchanged after restart as the minimum of received value and per router max-recovery time configured. Its default value is 120 seconds. Prior to restart, it is encoded as 0.

Although the Graceful Restart capability is enabled at the instance level, session level capability depends on the peer's support for graceful restart. Whenever the Instance level graceful-restart capability is set or reset, all sessions in that router are restarted to reflect the modified capability.

Graceful Restart Mechanism

The following section describes graceful restart operation with respect to the restarting router and peer router. The session under consideration should be restart capable as described in the following behavior.

Restarting Router

LDP Termination

Graceful-restart can be triggered on a router by any of the following actions:

- Terminating the LDP process
- Executing `restart ldp graceful` command

Upon either of the above actions, LDP cleans up its sessions and related parameters before flushing out LDP Instance. If the router is restart capable, the following are performed before termination:

- The router signals to NSM that LDP is restarting and passes reconnect and recovery timers that need to be preserved across the restart by NSM. This is done using the ldp_nsm_preserve_set() procedure.
- It also blocks sending of FTN-ILM delete messages to NSM when corresponding control blocks are deleted at the LDP level. This ensures that Forwarding table is preserved across the restart.

LDP-NSM Interaction

When the LDP client disconnects from the NSM server, NSM checks whether restart options have been set for this client. If restart options have been set, then client-specific entries in the MPLS RIB and label pools are marked stale, and the Reconnect timer is started with the timer value received from LDP. At the expiration of the reconnect timer, stale entries are removed from the MPLS RIB and corresponding label pools are released to the label pool manager.

When LDP is restarted, NSM stops the reconnect timer, unsets the stale mark for label pools, passes the restart options and label pools back to the LDP client. This restart option contains the configured values for reconnect time and recovery time and is opaque from NSM's point of view.

Restart LDP Process

When LDP restarts within the reconnect time-out period, it receives the stored restart options from NSM via the ldp_nsm_recv_service() function. This is a handle by which LDP ascertains that the restart is a Graceful Restart and starts the LDP instance-level recovery timer. LDP receives all stale entries from NSM via the ldp_nsm_recv_stale_entry_add() function. The LDP message handler ldp_nem_recv_stale_entry_add() adds the information to the stale/shadow database by invoking the ldp_fec_stale_add() function.

LDP then proceeds with its normal session initiation operations. The received values for reconnect and recovery time are encoded in the Initialization message sent out to its peers. In the meantime, the LDP session is established. When allocating a label for an FEC, LDP does a lookup, using ldp_fec_state_lookup(), on the shadow database to find any preserved or stale labels. If an entry is found for a corresponding FEC, the same label is used and marked in use in label pool. If no preserved label is found for an FEC, LDP allocates a new label and FEC-label bindings are exchanged with peer routers and the forwarding table is updated accordingly.

When NSM receives FTN and ILM add messages from LDP, it unsets the stale mark for the forwarding entry and does FTN and ILM updates resulting in no change or minimal change in the MPLS forwarding entry.

At the expiration of the recovery time, LDP cleans up the stale/shadow database by sending a “remove stale” message to NSM using ldp_nsm_stale_remove (). This cleans up stale entries from the MPLS RIB. At this point, the restarting router should have stabilized after the graceful restart.

Peer Router

Processing Session Down

Any instance of a restart-capable session going down is detected by a router that LDP has gracefully restarted at its peer. It performs the following operations upon receipt of a SHUTDOWN notification for the session:

1. It starts the reconnect timer for that session. The reconnect timer value is based on that negotiated for the session.
2. It marks upstream and downstream control blocks of the session as “stale” and retains them. As a result, Label-FEC bindings are retained at the control plane level on the peer router.
3. When the reconnect timer expires, DCBs and UCBs for the session that have been marked “stale” are removed and are cleaned up. As a result, corresponding forwarding entries also get flushed.

Session Restart

When the restarting router adjacency is recreated, the peer router stops the reconnect timer. Upon the receipt of an initialization message with an FT Session TLV, the peer router starts the session-level recovery timer.

Meanwhile, LDP exchanges FEC-Label binding with its peer. During label exchange for FECs, new DCBs or UCBs are created only as a last resort. Instead, they are looked up based on FEC from the session’s stale DCB and UCB lists. Changes, if any, are made to their respective labels and the same data is updated in FTN and ILM tables. Upon re-activation of a DCB or a UCB, its stale marking is removed.

At the expiration of the session's recovery timer, DCBs or UCBs belonging to the session that are still marked “stale” are cleaned up. As a result, corresponding forwarding entries also get flushed. At this point, the restarting router is said to have stabilized after the graceful restart.

Commands

For descriptions of these commands, see the *Label Distribution Protocol Command Reference*:

- `restart ldp graceful` - forces restarting LDP process to be a Graceful Restart.
- `(no) ldp graceful-restart timers neighbor-liveness` - configures neighbor-liveness time for a restart-capable router.
- `(no) ldp graceful-restart timers max-recovery` - configures max-recovery time for a restart-capable router.
- `graceful-restart (enable|disable)` - sets graceful-restart capability for a router (disabled by default).

CHAPTER 4 LDP MD5 Authentication

LDP MD5 authentication allows a user to enable LDP MD5 password authentication on a per-peer basis. In this way, password requirements can be set for LDP neighbors to 1) help prevent unauthorized peers from establishing LDP sessions, and 2) block spoofed TCP messages.

LDP MD5 authentication provides a mechanism to protect against the introduction of spoofed TCP segments into LDP session connection streams.

This configurable mechanism is based on use of the TCP MD5 Signature option, as specified in RFC2385, for use by BGP. RFC1321 describes the full specification for the MD5 hash function.

LDP MD5 Authentication Features

To accommodate MD5 authentication, the LDP module supports the following:

- CLI support to set a per LDP peer password used in generating the MD5 digest for each TCP segment used by the LDP session messages.
- MD5 routines to trigger when an MD5 password is configured for an LDP peer.
- If there is an existing session with the peer when the MD5 password configuration is done, the existing session is torn down, and a new session is formed.

CHAPTER 5 Inter-Area Label Switched Paths

ZebOS-XP LDP supports Inter-Area Label Switched Paths (LSP) that span multiple IGP (Internal Gateway Protocol) areas in an Autonomous System (AS). The procedure allows the use of a label if the Forwarding Equivalence Class (FEC) Element matches an entry in the Routing Information Base (RIB). Instead of mandating an exact match for the two entries, this new feature defines matching as an IP longest-match search based on a Longest Prefix Match (LPM).

Inter-Area LSP Features

RFC 5238 defines extensions to LDP to support inter-area LSP via label-matching procedure in which an LSR receives a label-matching message from a neighbor, and uses the label for MPLS forwarding when its routing table contains a longest match for the FEC. This differs from the original label-matching procedure for LDP that requires an *exact* match. ZebOS-XP LDP supports both procedures, as well as IPv4 and IPv6 label messaging and label mapping. The newer label-mapping procedure allows multiple FECs to be associated with a single RIB element, so a single RIB prefix change can propagate label-mapping messages and label-withdraw messages.

- When an FEC to label mapping is received from a neighbor, it is used for MPLS forwarding if its routing table contains a matching entry. An IP longest match lookup is performed. The lookup result may be an exact match, a longest match, or no match.
- When a new prefix arrives in the RIB, the LSR determines whether it is a better match for some of the existing FECs. If the new prefix is a better match and from a different next hop, old labels are withdrawn and new label mappings messages are sent for all peers to which these FECs were originally advertised.
- When a prefix is removed from the RIB, the LSR identifies all FEC elements using RIB prefix as a best match and updates them with the new best match RIB prefix. At the same time, the LSR withdraws old labels and sends new label mapping messages to all peers. When no new best match is found, a label withdraw message is sent to all upstream peers.

Label Mapping Procedures

The label mapping procedure does an exact match for a received FEC in its local RIB using the route_node_lookup() routine, which looks for an exact match to the prefix in the LDP RIB. In this implementation, when the inter-area LSP flag is set to true, the route_node_lookup routine searches for a longest match for the prefix that is stored as part of the ldp_fec_cb data structure. The search result indicates which RIB prefix is used as a best match for each FEC element. When no match is found, the received label is released with a label release message.

The command `inter-area LSP` configured an inter-area LSP based on the LPM label-matching procedure. The command stores a flag in the LDP_DISTANCE structure to show whether the flag is set to true or false. Default value is false. When a label mapping message received from a peer is processed, the new flag is checked, and if it is set to true, a longest prefix match-based search is initiated in the local LDP RIB. When the flag is set to false, the original (RFC 3036) label-mapping procedure initiated. In either case, it is used when there is an exact match in the local RIB.

Two tables, fec_matched_lpm_rib_table4 and fec_matched_lpm_rib_table6, are maintained for IPv4 and IPv6 FECs, respectively. These tables contain the matched entries (route_node) from the LDP RIB as keys. The route_node data consists of the list of ldp_fec_cb structures. All FECs matching the RIB entry are added to the list and sorted.

A table is populated when a label mapping message is received from a peer. The FEC passed in the label matching message is matched using an LPM search. When a match is found, a RIB entry is added to one of the fec_matched_lpm_rib_tables, and the related ldp_fec_cb structure is appended to the list. In the ldp_fec_cb structure, a flag is set that identifies the entry as an LPM match in LDP. A back-pointer to the matched RIB node is also set in ldp_fec_cb.

Entries in fec_matched_lpm_rib_table4 or fec_matched_lpm_rib_table6 are removed when label withdraw messages are received from downstream peers, or when matching RIB nodes are withdrawn by the IGP. Entries in the tables are updated when there are changes to matching RIB prefixes, for example, a new prefix or a new next hop.

Note: For all self-originated labels, the corresponding route_node entries in the ldp_fec_cb structure are NULL

Adding New Prefix to LDP RIB

When a new prefix is added to an LDP RIB, a new routine, ldp_update_fec_cb_lpm_entries_new_prefix (), is triggered. It matches the new prefix using the route_node_lookup () routine with other entries in the fec_matched_lpm_rib_table4 or fec_matched_lpm_rib_table6 to search for a better match. When the new prefix is determined to be a better match than other entries in the table, and it is from a new next hop, the related fec_db_list is retrieved, old labels are withdrawn, and new label mapping messages are sent to all upstream peers.

Deleting Prefix from LDP RIB

When a prefix is deleted from an LDP RIB, the deleted prefix is looked up in the fec_matched_lpm_rib_table4 or fec_matched_lpm_rib_table6 to retrieve the fec_cb list. and each FEC from fec_cb elements are looked up in the LDP RIB for a new longest match. If a new match is found and it is from a new nexthop, the old label is withdrawn and a new label is advertised to all upstream peers. If no match is found, a label withdraw message is sent to all upstream peers.

Updating Prefix With New Next Hop

When a next hop is changed for an entry in the LDP RIB, a lookup is preformed on the fec_matched_lpm_rib_table4 or fec_matched_lpm_rib_table6 to retrieve the fec_cb list. The old labels advertised for each of these FEC elements are withdrawn and new label mapping messages are sent to all upstream peers.

Removing Inter-Area LSP at Runtime

All FECs using best match in the LDP RIB are stored in the fec_matched_lpm_rib_table4 data structure or in the fec_matched_lpm_rib_table6 data structure. When the configuration is removed, the table(s) is (are) browsed, and all FECs are withdrawn from the upstream peers to which they were advertised.

CHAPTER 6 LDP IGP Synchronization

In come networks, there is dependency on edge-to-edge Label Switched Paths (LSPs) setup by the Label Distribution Protocol (LDP), such as networks that are used for Multiprotocol Label Switching (MPLS) Virtual Private Network (VPN) applications. For these applications, it is not possible to rely on Internet Protocol (IP) forwarding if the MPLS LSP is not operating correctly. Black-holing of labeled traffic can occur in situations where the Interior Gateway Protocol (IGP) is operational on a link on which LDP is not. While the link can still be used for IP forwarding, it is not useful for MPLS forwarding, for example, in MPLS VPN applications or Border Gateway Protocol (BGP) route-free cores.

The MPLS LDP-IGP Synchronization feature ensures that the Label Distribution Protocol (LDP) is fully established before the IGP path is used for packet forwarding. It is valuable in situations where a router is the ingress and the determination of whether to take the MPLS LSP or IGP path is made there.

Overview

LDP-IGP synchronization is an interface level feature. It can be selectively enabled in the required interfaces. For each interface there are two commands available for synchronization, one each for IS-IS and OSPF. Once configured, the IGP saves the required information, and also notifies LDP. In between, the IGP increases the link cost to maximum and sends advertisements to its peer. This discourages the peers from taking routes that pass through it. When all LDP sessions hosted on the interface become operational and complete label exchanges for all the FECs, LDP sends a notification to IGP. This is known as LDP convergence. The IGP then advertises normal traffic cost, so all traffic coming to the interface takes the MPLS LSP path established by LDP and not be IP-routed.

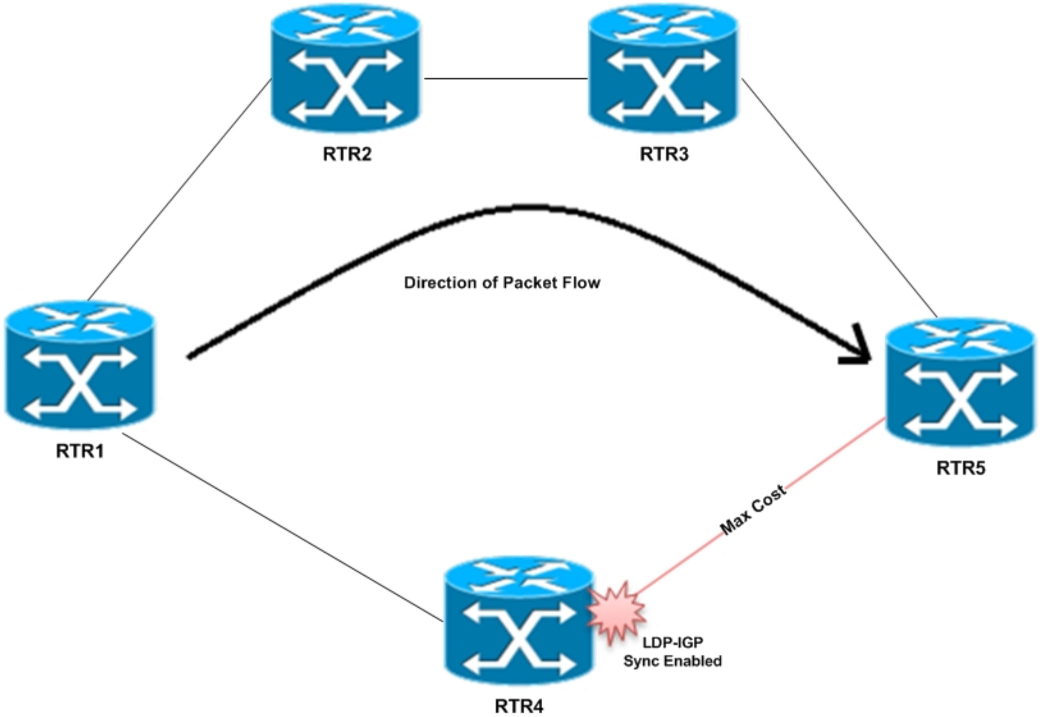


Figure 6-1: LDP-IGP Synchronization Topology

The diagram illustrates the topology implemented for LDP-IGP synchronization. Initially, packets from RTR1 to RTR5 take the minimum cost path RTR1-RTR4-RTR5. When LDP-IGP synchronization is enabled on an interface at RTR4, it advertises the maximum cost for that interface link. Because of this, all packets take the route RTR1-RTR2-RTR3-RTR4. RTR4 advertises normal cost after LDP-IGP synchronization is achieved at the interface. After synchronization is enabled, packets take the route RTR1-RTR4-RTR5.

Architecture

The functions that support LDP-IGP synchronization span the Network Services Module (NSM) and the Internal Gateway Protocols, specifically IS-IS and OSPF. These modules interact with each other via IPC messages to set or unset the synchronized state.

Operation

To achieve LDP-IGP synchronization, it must to be explicitly enabled for the interface on which synchronization is desired. Once enabled, LDP is notified to keep track of all sessions attached to the interface. Until the session is fully established, the IGP keeps advertising the maximum path cost for the interface. This discourages traffic from passing through that interface. It is expected that there is an alternate path available for traffic to flow in this state. However, if none is available, traffic may still flow through this interface, regardless of cost.

Once LDP finishes exchanging all labels, it notifies the IGP and then the IGP starts advertising normal cost on the interface. At this point, any incoming traffic takes the LSP established by LDP instead of the IGP path.

Sequence

LPD Synchronization Up at Initialization

The IGP protocol sends a session query to LDP via NSM and also starts advertising the maximum cost over the link. LDP is not operational at this point, so it sends a session state “DOWN” message to IGP. If LDP is “OPERATIONAL” and label exchange is completed within the hold-down time, it sends session state “UP” message to IGP. Now the IGP starts advertising a normal cost for the link. If the hold-down timer expires before LDP convergence takes place, IGP will continue to advertise a normal cost for the link.

LDP Down at Initialization of Synchronization

When the LDP process is down during LDP-IGP Synchronization initiation, NSM informs the IGP by sending an “LDP down” message. While this takes place, the IGP continues to advertise maximum path cost. If LDP comes up, NSM checks on which interfaces synchronization is enabled and sends a session query message for each of them to LDP. However, the hold-down timer may have expired, so it might be advertising normal cost. This means that even after LDP convergence is achieved and it sends a session state UP message, it will not have any impact on the IGP.

LDP Down or Session Down After Synchronization is Achieved

If the IGP receives an LDP session DOWN message after synchronization is achieved, it starts the hold-down timer and advertises the maximum cost on the link. After label exchange is complete, the IGP session receives a session UP message from LDP via the NSM. When the UP message is received, the IGP again starts advertising normal cost for the link.

IGP Event Handling

Table 6-1 summarizes the event handling for LDP-IGP synchronization.

Table 6-1: IGP event handling

		LDP Session Down or LDP DOWN	LDP Session UP
IGP Sync down	Hold-down timer not configured	Set IGP Sync state to Max Cost	Advertise Normal Cost and change state to IGP Sync up
	Hold-down-timer is On	No Action	Advertise Normal Cost, stop hold-down timer, and change state to IGP Sync up
	Hold-down timer is Off; either it timed out or was not started	If IGP Sync state is unknown, advertise Max Cost, start timer and set sync state to unknown	Change state to IGP Sync up
IGP Sync up	Hold-down timer is On	Error condition	Error condition
	Hold-down timer is off or not configured	Advertise Max Cost, start hold-down-timer, and change state to IGP Sync down	No action

LDP Convergence

LDP is considered fully operational on a link when an LDP hello adjacency exists on it, an associated LDP session (matching the LDP Identifier of the hello adjacency) is established to the peer at the other end of the link, and all label bindings have been exchanged over the session. At the present time, the latter condition cannot generally be verified by a router and some estimation may have to be used. The implementation strategy employed is use of a configurable hold-down timer to allow LDP session establishment before declaring LDP fully operational.

LDP has a configurable delay timer, so that after starting it goes into the operational state. If the delay timer expires and LDP remains operational, LDP notifies the IGP that it has converged. In case of multiple sessions on an interface, the delay timer should be started when the last of the sessions become operational. If the LDP session goes down and no longer remains operational, LDP becomes non-converged and notifies the IGP.

If LDP is already converged and a new route and subsequent FEC is added, LDP is non-converged. However, if it immediately notifies the IGP the link cost increases and existing traffic gets diverted. To avoid this, LDP waits for a configured time and again checks if convergence is still lost. If so, it notifies the IGP. For every incremental addition of FECs after convergence is achieved, the timer is restarted, if it was already running.

CHAPTER 7 Pseudowire Redundancy

The LDP module supports Pseudowire (PW) Redundancy for single-segment Pseudowires (SS-PW). A static CLI command is used to simulate the Active and Standby modes (rather than a multi-chassis hardware communication protocol) to achieve PW redundancy protection.

In a PW application, protection for the pseudowires can be provided by the PSN layer. Sometimes, a TE LSP signaled by RSVP-TE can be used as a PSN tunnel for a specific PW. In such a case, TE can provide FRR in order to protect the end-to-end LSP in the PSN layer.

However, FRR-based protection schemes cannot protect against failure of PE nodes and access circuits. PW redundancy is designed to protect against these failures. Multi-homed customer edge (CE) devices can be connected to two provider edge (PE) nodes via access circuits to provide protection.

In a Hierarchical Virtual Private VLAN Service (HVPLS), the MTU-s can create spoke circuits to two PEs, using one to protect the other.

Note: Only 1:1 protection is supported. At most, two pseudowires are allowed to be bound to one virtual port (port or port + VLAN).

Multi-Momed CE PW Redundancy

The logic diagram on the next page depicts two Multi-homed CE devices connected to two PE devices with Access Circuits (AC). In this scenario, CE1 is dual-homed to PE1 and PE2, and CE2 is dual-homed to PE3 and PE. As a consequence, multiple pseudowires need to be signaled: PW1 between PE1 and PE3; PW2 between PE1 and PE4; PW4 between PE2 and PE3; PW4 between PE2 and PE4.

Simultaneously, only one PE node can be chosen as Active for a CE; the other PE node acts as Standby. A protocol is required to communicate between a CE and the PEs to designate which PE is the Active node and which is the Standby node. The solution ZebOS-XP LDP provides is a command that simulates switchover between the PE devices. Executing the command results in PE node runtime-mode changes between Active and Standby status.

Scenario Description

Assume that PE1 and PE3 have been chosen to be the Active nodes. In this scenario, the following takes place:

1. PE1 announces PW1 and PW2 as Active.
2. PE3 announces PW1 and PW3 as Active.
3. PE2 announces PW3 and PW4 as Standby.
4. PE4 announces PW2 and PW4 as Standby.
5. PW1 is used for forwarding, because only it has had both endpoints announced as Active.

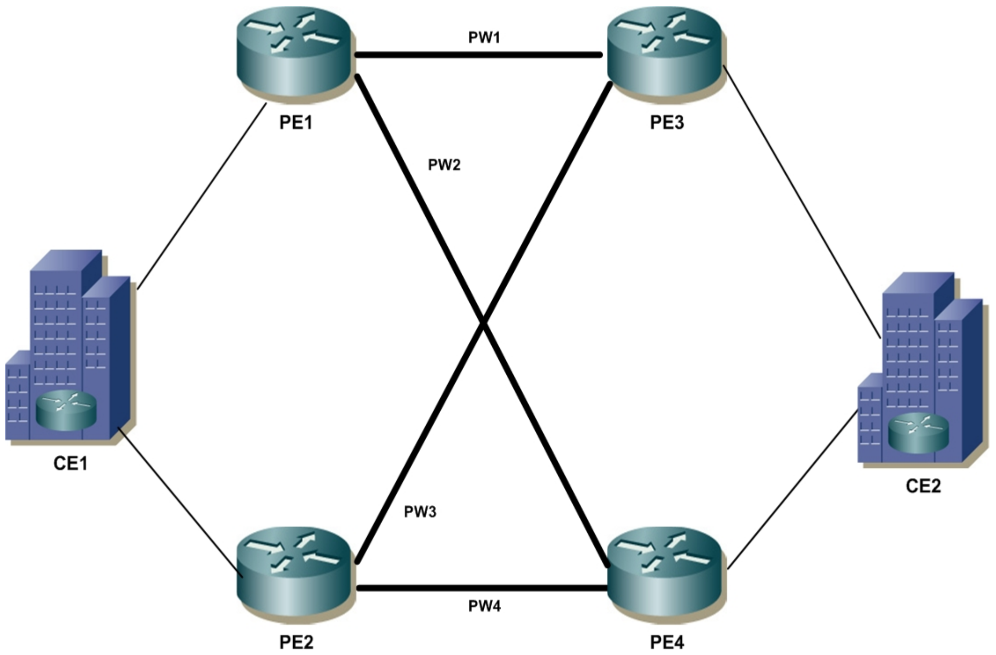


Figure 7-1: Scenario with Two CE and Four Pseudowires

Failure Scenario

If an AC failure takes place between CE1 and PE1, or a there is a node failure at PE1, ZebOS-XP provides a command that simulates the switchover required. The command syntax is:

```
(config-if)# vc-mode standby vlan <2-4096>
```

In this scenario, the command is executed as follows:

- On PE1, the `vc-mode standby` command is executed.
- On PE2, the `no vc-mode standby` command is executed.

When these two commands are executed, they set the Active/Standby modes for the pseudo-wires that bind to the port (port + VLAN). As a result, PE1 announces PW1 and PW2 as Standby, and PE2 announces PW3 and PW4 as Active.

Note: See the *Network Services Module Command Reference* for information about these commands.

1. If the PE1 node fails, it can no longer send control plane messages, but PE2 announces PW3 and PW4 as Active.
2. PW3 now has both endpoints announced as Active, so PE2 and PE3 use PW3 to forward, and PW1 is no longer used for forwarding.
3. In the event of a failure on PE1, PE3 and PE4 wait for the timeout to expire, then PW1 and PW2 are torn down.
4. In the event of an AC failure, PE1 announces PW1 and PW2 as Standby.

MTU-s with PW Redundancy

In the logic diagram that follows, MTU-s has redundant-spoke circuits for PW1 and PW2 with PE1 and PE2, respectively. MTU-s needs to configure which PW is Primary and which is Secondary.

Note: Only one secondary PW is supported at this time.

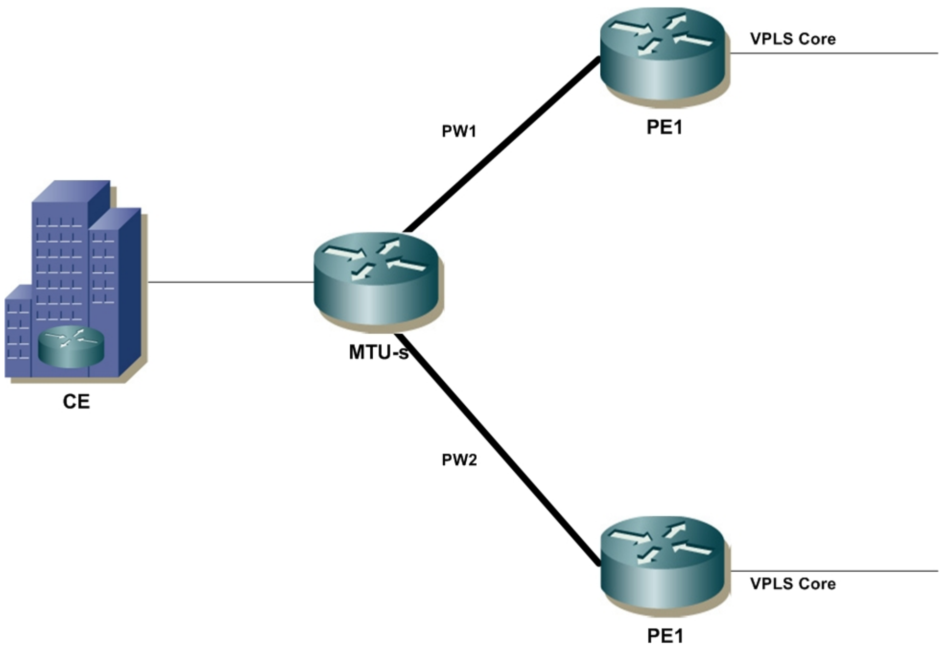


Figure 7-2: MTU-s with Pseudowire Redundancy

Scenario Description

Assume that PW1 is configured as Primary and PW2 is configured as Secondary. In this scenario, the following takes place:

1. MTU-s announces PW1 as Active and PW2 as Standby.
2. PE1 and PE2 announce both PW1 and PW2 as Active.
3. PW1 is used for forwarding, because both of its endpoints have been announced as Active.
4. If PW1 fails, MTU-s performs a switchover by announcing PW2 as Active and PW1 as Standby.
5. PW2 is now used for forwarding, because only it has had both endpoints announced as Active.
6. In Non-Revertive mode, when PW1 recovers from the failure, MTU-s does not perform another switchover. It keeps using PW2 for forwarding.
7. In Revertive mode, when PW1 recovers from the failure, MTU-s performs a switchover by announcing PW1 as Active and PW2 as Standby.
8. PW1 is again used for forwarding because both of its endpoints have now been announced as Active.

A command in ZebOS-XP configures Revertive mode. (The default mode is Non-Revertive.) The command syntax is:

```
    (config-if)# vc-mode (revertive) (vlan <2-4096>)
```

See the *Network Services Module Command Reference* for information about this command.

Note: The `pw-status-tlv` must be enabled in LDP to support PW redundancy. The behaviors described for both scenarios assume that the correct configurations have been made for all devices. ZebOS-XP LDP PW Redundancy provides protection against access circuit failure and PE node failure in both scenarios. See the *Multi-Protocol Label Switching Configuration Guide* for steps to configure PW redundancy.

Design Considerations

This section describes design considerations for the components related to PW Redundancy.

NSM

At most two Virtual Circuits are allowed to be bound to the same virtual port (port or port + VLAN). The two VCs are siblings. When they are bound to the interface, one is designated to be in the Primary mode, and and the other is designated to be in the Secondary mode. Siblings may be in Primary/Secondary or Primary/Primary combinations.

Note: Siblings using a secondary/secondary combination are not allowed.

NSM maintains the local `pw-status`. LDP informs NSM of the `remote-pw-status`. Only the VC for which both endpoints have been announced as Active is used for forwarding.

Primary/Secondary Combination

The Primary/Secondary combination is illustrated in the second logic diagram, above. A secondary VC backs up a primary VC, so when the primary VC goes down, the secondary VC is installed as the forwarder and does the forwarding. When the Primary VC comes back up, its behavior depends upon the mode it was originally set to, including the following:

- If the Primary VC was set to Revertive mode, the Secondary VC is removed from forwarder. The Primary VC is installed into forwarder and does the forwarding.
- If the Primary VC was set to Non-Revertive mode, the Secondary VC continues to do the forwarding.

Note: Non-Revertive is the default mode.

Primary/Primary Combination

The Primary/Primary combination is illustrated in the first logic diagram, above. When an AC or PE node failure is detected, the operator needs to manually give the command to simulate switchover of Active and Standby mode between PE nodes. Depending upon the local and the remote PW status, NSM decides which PW to use for forwarding.

NSM informs LDP of the local PW status when a VC is created and bound to an interface, and during runtime Active or Standby mode changes. NSM also keeps and updates remote PW status information received from LDP, and performs related activities.

Qualified Forwarder

When LDP sends VC FIB entries or PW-status messages to NSM, NSM installs the qualified entry as forwarder. A qualified entry must meet all of these conditions:

- Local `pw-status` does not have any fault and the standby bit is set.
- Remote `pw-status` does not have any fault and the standby bit is set.
- An LSP is available for the VC.

If the entry is qualified, the outcome depends upon the sibling combination:

- If the sibling combination is Primary/Primary and both VCs meet all conditions above, the most recent one is always used to replace the previous.
- If the sibling combination is Primary/Secondary and both VCs meet all conditions, the previous one is used in non-revertive mode, and the Primary is used in revertive mode.

Status Fault

When a primary VC local PW status fault is detected, or a remote PW status fault is received with the standby bit set, the actions taken depend on the sibling combination. If the sibling combination is primary, the following occurs:

1. If the Secondary VC only qualifies by clearing the local standby bit, NSM performs a switchover to use the Secondary VC to forward.
2. After the Primary VC is cleared of all faults or it receives a message with all fault-bit and standby-bit flags cleared,
    - If the VC was set up in Revertive mode, NSM performs a switchover to restore the Primary VC to forwarding.
    - If the VC was set up in Non-Revertive mode, NSM updates the local/remote PW status, but does not perform a switchover.

If the sibling combination is Primary/Primary, NSM updates the VC local and remote PW status, and removes the installed entry from forwarder.

LDP

LDP must support the `pw-status-tlv` and have it enabled to support PW redundancy functionality. A new `pw-status` code has been added, which is called the PW Standby Bit. LDP receives the PW status of each VC from NSM and sets it to the local `pw-status`. The `pw-status` is sent to the virtual circuit's peer via a label-mapping or notification message. When LDP receives a `pw-status` message from a peer, it saves it in the remote `pw-status`, then informs NSM using the VC_FIB_ADD message or PW status message.

VC Switchover

The user has to delete the primary PW or primary PW should operationally go down to switch between PWs. The switchover functionality is supported only in the Non-Revertive mode. When the user configures the vc-switchover CLI, the sibling PW should be operationally UP. The PW status is changed to Standby; this sends the modified PW status to peer. If the user configures the mode as Revertive after switchover, the primary PW is switched back if the primary PW status and sibling PW status is UP. If not, then secondary PW is used for forwarding.

nsm_mpls_api_pw_switchover

This call is used to manually switch between VCs.

Syntax

```
int
nsm_mpls_api_pw_switchover (u_int32_t vr_id, char *primary_vc, char *sec_vc)
```

Input Parameters

`vr_id`	VR identifier
`primary_vc`	Specify the Primary VC name
`sec_vc`	Specify the Secondary VC name

Output Parameters

None

Return Value

none

CHAPTER 8 Data Structures

This chapter lists the data structures used with LDP. It includes the following objects:

cli

This structure contains the CLI elements used with CLI functions. It is defined in the `lib/cli.h` file.

Member	Description
cel	CLI element
str	User input string
out_func	Output function to be used by cli_out()

Member	Description
out_val	Output function's first argument
line	Arbitrary information for line
auth	Authorization required
source	Input source
line_type	For line
index	Real CLI
index_sub	Real CLI
mode	Real CLI
status	Current CLI status
flags	Flags
self	Arbitrary information for self
privilege	Privilege level
ctree	CLI tree
zg	Global variable
vr	Global variable
lines	Terminal length
callback	Call back function
cleanup	Call back function
show_func	Call back function
type	Type of CLI
count	Total count
current	Arbitrary information about current node
arg	Look up argument
afi	Address family information
safi	Specific address family information
port_range	Layer 2 handling
cv	Vector used by IMI to encode a single CLI command string

Definition

```
struct cli
```

```
{
  /* CLI element.  */
  struct cli_element *cel;
  /* User input string.  */
  char *str;

  /* Output function to be used by cli_out().  */
  CLI_OUT_FUNC out_func;

  /* Output function's first argument.  */
  void *out_val;

  /* Alternate storage for cli_out() message */
  char *out_snoop_buf;

  /* Arbitrary information for line.  */
  void *line;

  /* Auth required.  */
  int auth;

  /* Input source.  */
  int source;
#define CLI_SOURCE_USER                 0
#define CLI_SOURCE_FILE                 1

  /* For "line". */
  int line_type;
  int min;
  int max;

  /* Real CLI.  */
  void *index;
  void *index_sub;
  int mode;
/* Current CLI status.  */
  enum {
    CLI_NORMAL,
    CLI_CLOSE,
    CLI_MORE,
    CLI_CONTINUE,
    CLI_MORE_CONTINUE,
    CLI_WAIT
  } status;

  /* Flags. */
  u_char flags;
#define CLI_FROM_PVR    (1 << 0)

  void *self;
  u_char privilege;
  struct cli_tree *ctree;

  /* Global variable.  */
  struct lib_globals *zg;
  struct ipi_vr *vr;

  /* Terminal length.  */
  int lines;

  /* Call back function.  */
  int (*callback) (struct cli *);
  int (*cleanup) (struct cli *);
  s_int32_t (*show_func) (struct cli *);
  int type;
  u_int32_t count;
  void *current;
  void *arg;
  afi_t afi;
  safi_t safi;

#ifdef HAVE_CUSTOM1
  /* L2 handling.  */
  u_int64_t port_range;
#endif /* HAVE_CUSTOM1 */

  /* Vector used by IMI to encode a single CLI command string.
   */
  cfg_vect_t *cv;

  /* Parse result saved for non-interactive shells */
  unsigned parse_result;
};
```

fec_matched_lpm_rib_table4

This table is used to store LPM matched FECs. Key is the matched RIB node. It is defined in the `ldpd/ldpd.h` file.

Definition

```
struct route_table *fec_matched_lpm_rib_table4;
#ifdef HAVE_IPV6
  struct route_table *fec_matched_lpm_rib_table6;
#endif /* HAVE_IPV6 */
#endif /* HAVE_LDP_INTER_AREA */
};
```

fec_matched_lpm_rib_table6

This table is used to store LPM matched FECs. Key is the matched RIB node. It is defined in the `ldpd/ldpd.h` file.

Definition

```
struct route_table *fec_matched_lpm_rib_table4;
#ifdef HAVE_IPV6
  struct route_table *fec_matched_lpm_rib_table6;
#endif /* HAVE_IPV6 */
#endif /* HAVE_LDP_INTER_AREA */
};
```

interface

This data structure contains interface-related configuration parameters. It is defined in the `lib/if.h` file.

Member	Description
ifindex	Interface index
cindex	Interface attribute update flags
flags	Interface flags
status	ZebOS-XP internal interface status
metric	Interface metric
mtu	Interface MTU
duplex	Interface duplex status
autonego	Interface auto-negotiation
mdix	Medium Dependent Interface with crossover
arp_ageing_timeout	Interface ARP aging timeout
slot_id	Slot identifier
hw_type	Hardware address (type)
hw_addr	Hardware address
hw_addr_len	Hardware address length
bandwidth	Interface bandwidth, bytes per second
if_linktrap	Interface link up or link down traps
if_alias	Interface alias name
conf_flags	Indicates whether the bandwidth has been configured or read from the kernel
desc	Description of the interface
ifc_ipv4	Connected address list
ifc_ipv6	Connected IPv6 address list
unnumbered_ipv4	Unnumbered interface list

Member	Description
unnumbered_ipv6	Unnumbered IPv6 interface list
info	Daemon specific interface data pointer
vr	Pointer to virtual router context
vrf	Pointer to VRF context
struct pal_if_stats stats	Statistics fields
tunnel_if	Tunnel interface
struct label_space_data ls_data	Label space
admin_group	Administrative group to which this interface belongs to
max_resv_bw	Maximum amount of bandwidth that can be reserved (bytes)
bw_constraint	Bandwidth constraint per class type (bytes)
tecl_priority_bw	Available bandwidth at priority “P” (range 0 to 8)
bind	Bind information
num_dl	Number of data links. Based on this information, the system either uses the tree interface (datalink less than 1) or uses the pointer to the datalink structure. This is GMPLS information
gmpls_type	Type, which may include unknown, data, control, and data-control
gifindex	GMPLS interface index
phy_properties phy_prop	GMPLS interface common properties
dlink	Pointer to datalink
port	Port information
bridge_name	Bridge name
lacp_admin_key	LACP administration key
agg_param_update	Interface LACP aggregator update flag
lacp_agg_key	LACP aggregator key
lag_flags	Interface lag flags
agg_zif	Back pointer to hold the aggregator interface
config_channel_type	Interface configuration channel type
config_channel_id	Interface configuration channel id (key)
config_channel_mode	Interface configuration channel mode
bc_mode	Bandwidth constrain mode for each interface

Member	Description
vrx_flag	Flag for virtual router for CheckPoint VSX type
local_flag	Local source
vrx_if_info	Related VRX information
ifLastChange	Time for last status change
Pid	Process ID
type	Interface type L2/L3
config_duplex	Stores the configured duplex value
trust_state	QoS set trust state for port
vlan_classifier_group_id	VLAN classifier group ID
clean_pend_resp_list	Pending list of interface if not deleted properly
rule_group_list	APBF rule-group list an interface can be associated to multiple rule groups. Each node in the list is of type struct if_rule_group
rule_group	APBF rule-group structure, an interface can be associated to only one rule group
interface_cdr_ref	HA interface Checkpoint Abstraction Layer (CAL) created record reference value

Definition

```
/* Interface structure */
struct interface
{
  /* Interface name. */
  char name[INTERFACE_NAMSIZ + 1];
#ifdef HAVE_INTERFACE_NAME_MAPPING
  char orig[INTERFACE_NAMSIZ + 1];
#endif /* HAVE_INTERFACE_NAME_MAPPING */

  /* Interface index. */
  s_int32_t ifindex;

  /* Interface attribute update flags.  */
  u_int32_t cindex;

  /* Interface flags. */
  u_int32_t flags;
/* ZebOS-XP internal interface status */
  u_int32_t status;
#define NSM_INTERFACE_ACTIVE            (1 << 0)
#define NSM_INTERFACE_ARBITER           (1 << 1)
#define NSM_INTERFACE_MAPPED            (1 << 2)
#define NSM_INTERFACE_MANAGE            (1 << 3)
```

```
#define NSM_INTERFACE_DELETE            (1 << 4)
#define NSM_INTERFACE_IPV4_UNNUMBERED   (1 << 5)
#define NSM_INTERFACE_IPV6_UNNUMBERED   (1 << 6)
#ifdef HAVE_HA
#define HA_IF_STALE_FLAG                (1 << 7)
#endif /* HAVE_HA */
#define IF_HIDDEN_FLAG                  (1 << 8)
#define IF_NON_CONFIGURABLE_FLAG        (1 << 9)
#define IF_NON_LEARNING_FLAG            (1 << 10)

  /* Interface metric */
  s_int32_t metric;

  /* Interface MTU. */
  s_int32_t mtu;

  /* Interface DUPLEX status. */
  u_int32_t duplex;

  /* Interface AUTONEGO. */
  u_int32_t autonego;

/* Interface AUTONEGO. */
  u_int32_t autonego;

  /* Interface MDIX crossover. */
  u_int32_t mdix;

  /* Interface ARP AGEING TIMEOUT. */
  u_int32_t arp_ageing_timeout;

  /* Slot Id. */
  u_int32_t slot_id;

  /* Hardware address. */
  u_int16_t hw_type;
  u_int8_t hw_addr[INTERFACE_HWADDR_MAX];

  s_int32_t hw_addr_len;

  /* interface bandwidth, bytes/s */
  float64_t bandwidth;

  /*Interface link up or link down traps */
  s_int32_t if_linktrap;

  /* Interface alias name */
  char if_alias[INTERFACE_NAMSIZ + 1];
 /* Has the bandwidth been configured/read from kernel. */
  char conf_flags;
```

```
#define NSM_BANDWIDTH_CONFIGURED    (1 << 0)
#define NSM_MAX_RESV_BW_CONFIGURED  (1 << 1)
#define NSM_SWITCH_CAP_CONFIGURED   (1 << 2)
#define NSM_MIN_LSP_BW_CONFIGURED   (1 << 3)
#define NSM_MAX_LSP_SIZE_CONFIGURED (1 << 4)
  /* description of the interface. */
  char *desc;

  /* Connected address list. */
  struct connected *ifc_ipv4;
#ifdef HAVE_IPV6
  struct connected *ifc_ipv6;
#endif /* HAVE_IPV6 */

  /* Unnumbered interface list.  */
  struct list *unnumbered_ipv4;
#ifdef HAVE_IPV6
  struct list *unnumbered_ipv6;
#endif /* HAVE_IPV6 */

  /* Daemon specific interface data pointer. */
  void *info;

 /* Pointer to VR/VRF context. */
  struct ipi_vr *vr;
  struct ipi_vrf *vrf;

  /* Statistics fileds. */
  struct pal_if_stats stats;

#ifdef HAVE_TUNNEL
  /* Tunnel interface. */
  struct tunnel_if *tunnel_if;
#endif /* HAVE_TUNNEL */

#ifdef HAVE_MPLS
  /* Label space */
  struct label_space_data ls_data;
#endif /* HAVE_MPLS */

#ifdef HAVE_TE
  /* Administrative group that this if belongs to */
  u_int32_t admin_group;

  /* Maximum reservable bandwidth (bytes/s) */
  float32_t max_resv_bw;

#ifdef HAVE_DSTE
  /* Bandwidth constraint per class types (bytes/s) */
  float32_t bw_constraint[MAX_BW_CONST];
```

```
#endif /* HAVE_DSTE */

  /* Available bandwidth at priority p, 0 <= p < 8 */
  float32_t tecl_priority_bw [MAX_PRIORITIES];
#endif /* HAVE_TE */

  /* Bind information.  */
  u_char bind;
#define NSM_IF_BIND_VRF           (1 << 0)
#define NSM_IF_BIND_MPLS_VC       (1 << 1)
#define NSM_IF_BIND_MPLS_VC_VLAN  (1 << 2)
#define NSM_IF_BIND_VPLS          (1 << 3)
#define NSM_IF_BIND_VPLS_VLAN     (1 << 4)

#ifdef HAVE_GMPLS
  /* GMPLS information */
  /* Number of data links. Based on this information we will either use the
    tree if dl > 1 or use the pointer to the datalink structure */
  u_char num_dl;

 /* Type includes unknow/data/control/data-control */
  u_char gmpls_type;

  u_int32_t gifindex;

  /* GMPLS interface common properties */
  struct phy_properties phy_prop;

  /* Pointer to datalink */
  union
    {
      struct avl_tree *dltree;
      struct datalink *datalink;
    }dlink;
#endif /* HAVE_GMPLS */

#ifdef HAVE_L2
  void *port;
  char bridge_name[INTERFACE_NAMSIZ + 1 ];
#endif /* HAVE_L2 */

#ifdef HAVE_LACPD
  u_int16_t lacp_admin_key;
  u_int16_t agg_param_update;
  u_int32_t lacp_agg_key;
#endif /* HAVE_LACPD */

#ifdef HAVE_DSTE
  /* Bandwdith constrain mode for every interface. */
  bc_mode_t bc_mode;
```

```
#endif /* HAVE_DSTE */

#ifdef HAVE_VRX
  u_char vrx_flag;
#define IF_VRX_FLAG_NORMAL              0
#define IF_VRX_FLAG_WRPJ                1
#define IF_VRX_FLAG_WRP                 2

  /* Local src. */
  u_char local_flag;
#define IF_VRX_FLAG_LOCALSRC            1

  /* Related VRX information. */
  struct vrx_if_info *vrxif;
#endif /* HAVE_VRX */
  pal_time_t ifLastChange;

#ifdef HAVE_CUSTOM1
  int pid;
#endif /* HAVE_CUSTOM1 */
  u_char type; /* Interface type L2/L3 */
#define IF_TYPE_L3 0
#define IF_TYPE_L2 1

/* Maximum L2 MTUs */
#if defined (HAVE_VLAN_STACK) || defined (HAVE_PROVIDER_BRIDGE)
  #define IF_ETHER_L2_DEFAULT_MTU 1526
#elif defined (HAVE_VLAN)
  #define IF_ETHER_L2_DEFAULT_MTU 1522
#else
  #define IF_ETHER_L2_DEFAULT_MTU 1518
#endif

  /* To store the configured duplex value */
  u_char config_duplex;

#ifdef HAVE_QOS
  int trust_state;
#endif /* HAVE_QOS */

#ifdef HAVE_VLAN_CLASS
 u_int32_t vlan_classifier_group_id;
#endif /* HAVE_VLAN_CLASS */

 struct list *clean_pend_resp_list;

#ifdef HAVE_HA
 HA_CDR_REF interface_cdr_ref;
#ifdef HAVE_MPLS
 s_int32_t chkpt_info;
 HA_CDR_REF nsm_mpls_if_cdr_ref;
#endif /* HAVE_MPLS */
#endif /* HAVE_HA */
  struct list *rmap_if_match_cmd_list;
  /*LDP-IGP Sync */
  void *sync_params;

  /*Interface BW- Configured CIR/EIR sync*/
  struct nsm_band_width_profile *bw_profile;
};
```

ldp

This structure contains the configuration parameters used with the LDP master. It is defined in the `ldpd/ldpd.h` file.

Definition

```
struct ldp
{
  /* Router ID.  */
  u_int32_t router_id;
  u_int32_t router_id_nsm;

  /* Default index */
   s_int32_t ldp_default_entity_ix;

  /* LDP Entity Table */
  struct list *ldp_entities;

  /* fec table */
  struct route_table *ipv4_prefix_fec_table;
#ifdef HAVE_CR_LDP
  struct route_table *cr_lsp_fec_table;
#endif /* HAVE_CR_LDP */

  /* Table for LDP Hello adjacency */

  /* Remote addresses of ALL peers*/
 /* Table to store labelspace-to-address mapping. */
  struct route_table *ls_to_addr;

  struct route_table *conf_ipv4_fec;

  /* Table for LDP targeted Peers. */
  struct route_table *targeted_peers;

  /* Table for storing common labels per fec */
  struct route_table *fec_label_table;

#ifdef HAVE_MPLS_VC
  /* Virtual Circuit table. */
  struct route_table *vc_table;
  bool_t pw_status_mode;

#ifdef HAVE_MS_PW
  struct hash *ldp_ms_pw_hash_table;
#define LDP_MS_PW_HASH  lm->ldp->ldp_ms_pw_hash_table
#define LDP_MS_PW_MIN_IX   1
#define LDP_MS_PW_MAX_IX   65535
#define LDP_MS_PW_BUCKET_SIZE 1280
#endif /* HAVE_MS_PW */
#endif /* HAVE_MPLS_VC */

#ifdef HAVE_VPLS
  struct ptree *vpls_table;
#endif /* HAVE_VPLS */

  /* Thread to read accept sock */
  struct thread *t_accept;

  /* udp read thread */
  struct thread *t_read;

  /* Message ID. */
  u_int32_t message_id;

 /* Label id counter */
  u_int32_t label_id;

  /* no. of listeners for udp socket */
  u_int32_t udp_rcv_count;

  /* Accept Socket for TCP Connects */
  int accept_sock;

  /* Count of passive sessions waiting to accept */
  int accept_count;

  u_int16_t targeted_hello_interval;
  u_int16_t targeted_hold_time;
  u_int16_t targeted_hold_time_for_sync;
  u_int16_t hello_interval;
  u_int16_t hold_time;
  u_int16_t hold_time_for_sync;
  u_int16_t keepalive_timeout;
  u_int16_t keepalive_interval;
#ifdef HAVE_IPV6
  struct route_table *ipv6_prefix_fec_table;
  struct route_table *conf_ipv6_fec;
  /* udp read thread for IPV6 */
  struct thread *ipv6_t_read;
  /* Thread to read accept sock */
  struct thread *ipv6_t_accept;
  /* no. of listeners for udp IPV6 socket */
  u_int32_t udp_rcv_count6;
  /* Count of passive sessions waiting to accept */
  int accept_count6;
  /* udp recv socket for ipv6 */
  int u_sock6;
  /* ldp hello send socket for ipv6 */
  int s_sock6;
#endif /* HAVE_IPV6 */

 /* LDP configuration.  */
  u_int32_t config;
#define LDP_CONFIG_ROUTER_ID                 (1 << 0)
#define LDP_CONFIG_CONTROL_MODE              (1 << 1)
#define LDP_CONFIG_LABEL_MERGE               (1 << 2)
#define LDP_CONFIG_MIN_PDU_LEN               (1 << 3)
#define LDP_CONFIG_LOOP_DETECTION            (1 << 4)
#define LDP_CONFIG_REQUEST_RETRY             (1 << 5)
#define LDP_CONFIG_REQUEST_RETRY_COUNT       (1 << 6)
#define LDP_CONFIG_REQUEST_RETRY_TIMEOUT     (1 << 7)
#define LDP_CONFIG_MAXIMUM_LABEL_SPACE       (1 << 8)
#define LDP_CONFIG_GLOBAL_MERGE_CAP          (1 << 9)
#define LDP_CONFIG_IMPORT_BGP_ROUTES         (1 << 10)
#define LDP_CONFIG_EXPLICIT_NULL             (1 << 11)
#define LDP_CONFIG_NO_MULTICAST_HELLOS       (1 << 12)
#define LDP_CONFIG_RECONNECT_TIMEOUT         (1 << 13)
#define LDP_CONFIG_RECOVERY_TIME             (1 << 14)
#define LDP_CONFIG_LABEL_RETENTION_MODE      (1 << 15)

#ifdef HAVE_LDP_INTER_AREA
#define LDP_CONFIG_INTER_AREA_LSP            (1 << 16)
#define LDP_CONFIG_INTER_AREA_LSP_CONFIG_ONLY (1 << 17)
#endif /* HAVE_LDP_INTER_AREA */
#define LDP_CONFIG_LABEL_ADVERTISEMENT_MODE (1 << 18)
#define LDP_CONFIG_TARGETED_PEER_HELLO_INT  (1 << 19)
#define LDP_CONFIG_TARGETED_PEER_HOLD_TIME  (1 << 20)
#define LDP_CONFIG_TARGETED_PEER_HELLO_RECV (1 << 21)
#define LDP_INST_CONFIG_HOLD_TIME            (1 << 22)
#define LDP_INST_CONFIG_HELLO_INTERVAL       (1 << 23)
#define LDP_INST_CONFIG_KEEPALIVE_TIMEOUT    (1 << 24)
#define LDP_CONFIG_OPTIMIZATION              (1 << 25)
#define LDP_INST_CONFIG_KEEPALIVE_INTERVAL  (1 << 26)
#define LDP_INST_CONFIG_HOLD_TIME_SYNC       (1 << 27)

 /* LDP specific flags */
  u_int16_t flags;
#define LDP_FLAG_ROUTER_ID                   (1 << 0)
#define LDP_FLAG_ROUTER_ID_NSM               (1 << 1)
```

```
#define LDP_FLAG_TERMINATE                  (1 << 2)
#define LDP_FLAG_RESTART                    (1 << 3)
#define LDP_FLAG_PEER_RESTART               (1 << 4)
/* Flag used when LDP instance is deleted */
#define LDP_FLAG_INSTANCE_DEL               (1 << 5)

#ifdef HAVE_RESTART

#define GRACEFUL_RESTART_DISABLED           0
#define GRACEFUL_RESTART_ENABLED            1
  u_char graceful_restart;

#define GR_HELPER_MODE_DISABLE              0
#define GR_HELPER_MODE_ENABLE               1
  u_char helper_mode;
/* Restart reason for Graceful Restart. */
  u_char restart_reason;
#define LDP_RESTART_REASON_UNKNOWN                       0
#define LDP_RESTART_REASON_RESTART                       1
#define LDP_RESTART_REASON_UPGRADE                       2
#define LDP_RESTART_REASON_SWITCH_REDUNDANT              3

  int nbr_liveness_period;
  int max_recovery_period;

  /* Global Recovery Timer Thread */
  struct thread *t_recovery_time;

  /* Restarting session Count */
  int restart_count;

#endif /* HAVE_RESTART */

  /* LDP Version */
  u_int16_t version;

  /* Request retry time timeout */
  u_int16_t request_retry_timeout;

/* Loop detection enable/disable */
  #define LDP_LOOP_DETECTION_OFF                  0
  #define LDP_LOOP_DETECTION_ON                   1
  u_char loop_detection;

  /* Label merge capability */
#define LDP_GLOBAL_MERGE_CAPABLE               0
#define LDP_GLOBAL_NON_MERGE_CAPABLE           1
  u_char global_merge_cap;

  /* Label Distribution Control Mode.  */
  u_char control_mode;

  /* CLI configured Path Vector Limit */
  u_int32_t global_path_vec_limit;

 /* CLI Configured Hop count limit */
  u_int32_t global_hop_count_limit;

  /* Request retry enable/disable */
#define LDP_REQUEST_RETRY_OFF                   0
#define LDP_REQUEST_RETRY_ON                    1
  u_char request_retry;

  /* Propagate release flag */
  u_char propagate_release;

   /* Flag to keep track of ldp shutdown scenario */
  u_char shutdown;

  /* Retention mode (conservative/liberal) */
  u_char retention_mode;
 /* Advertisement mode */
  u_char adv_mode;

  /* udp recv socket */
  int u_sock;

  /* ldp hello send socket */
  int s_sock;

  struct avl_tree *ilm_add_queue;
  struct avl_tree *ftn_add_queue;

  /* time for entity last change */
  pal_time_t ldp_entity_last_change;

  /* time stamp of last peer change */
  pal_time_t last_peer_change;

  /* time for fec last change */
  pal_time_t fec_last_change;

  /* Time for Ldp Lsp Fec Last Change */
  pal_time_t ldp_lsp_fec_last_change;

#ifdef HAVE_RESTART
  struct thread *t_restart;     /* Restart State check timer. */
  struct ptree  *ipv4_fec_stale_table;
#ifdef HAVE_IPV6
  struct ptree  *ipv6_fec_stale_table;
```

```
#endif /* HAVE_IPV6 */
#endif /* HAVE_RESTART */

#ifdef HAVE_TCP_MD5SIG
  /* Table to store neighbor-to-password mapping. */
  struct route_table *nbr_to_passwd_table;
#endif /* HAVE_TCP_MD5SIG */

#ifdef HAVE_LDP_INTER_AREA
  /* Table for storing lpm matched fecs
     Key is the matched RIB node */
  struct route_table *fec_matched_lpm_rib_table4;
#ifdef HAVE_IPV6
  struct route_table *fec_matched_lpm_rib_table6;
#endif /* HAVE_IPV6 */
#endif /* HAVE_LDP_INTER_AREA */
};
```

ldp_adjacency

This data structure is defined in the `ldpd/ldpd.h` file.

Definition

```
struct ldp_adjacency
{
  struct ldp_session *session;

  /* My interface. */
  struct ldp_interface *ldpif;

  /* Pointer back to it's list/table node */
  void * node;

  /* Peer address. */
  union
  {
    struct pal_in4_addr peer4;
#ifdef HAVE_IPV6
   struct pal_in6_addr peer6;
#endif /* HAVE_IPV6 */
  }peer;

  /* Peer address which we use to form tcp connection */
  union
  {
    struct pal_in4_addr peer_tcp4;
#ifdef HAVE_IPV6
    struct pal_in6_addr peer_tcp6;
#endif /* HAVE_IPV6 */
  }peer_tcp;
#ifdef HAVE_TCP_MD5SIG
  u_int8_t passwd_type;
  u_int8_t * passwd;
  /* LDP Peer Password flag */
#define LDP_PEER_PASSWORD_SET   (1 << 0)
#define LDP_PEER_PASSWORD_PENDING (1 << 1)
  u_char passwd_flag;
#endif /* HAVE_TCP_MD5SIG */

  /* LDP ID. */
  struct ldp_id id;

  /* Holdtime thread for this adjacency */
  struct thread *t_holdtime;

#ifdef HAVE_SNMP
  /* Timestamp when last hello packet was received */
  pal_time_t hello_received;
#endif /* HAVE_SNMP */

  /* Hold time */
  u_int16_t holdtime;

  /* Flag for track whether adjacency is with a targeted peer */
  u_char targeted_peer;

  /* LDP adjacency family */
  u_char family;
};
```

ldp_adv_list_master

This data structure supports LDP advert-list master. It is defined in the `ldpd/ldpd.h` file.

Definition

```
struct ldp_adv_list_master
{
  struct ldp_adv_list *head;
  struct ldp_adv_list *tail;
};
```

ldp_entity

This data structure is defined in the `ldpd/ldpd.h` file.

Definition

```
struct ldp_entity
{
  /* Ldp Id */
  struct ldp_id *id;

  /* Transport address corresponding to entity */
  struct pal_in4_addr trans_addr4;
#ifdef HAVE_IPV6
  struct pal_in6_addr trans_addr6;
#endif /*HAVE_IPV6*/

  /* Table for LDP Hello adjacency */
  struct route_table *hello_adjacencies;

  /* Targeted Peer - Pointer to the tagreted peer corresponding
     to this ldp enity*/
  struct ldp_targeted_peer *targeted_peer;

  /* Remote addresses of ALL peers*/
  struct route_table *remote_addresses;

 /* List of sessions */
  struct list *sessions;

   /* Entity Index - newly added member */
  u_int32_t index;

  /* Source of LDP Entity Creation - CLI or SNMP */
  #define       SRC_CLI         1
  #define       SRC_SNMP        2

  int src;

  u_int8_t trans_addr_kind;

  /* Flag to keep track of ldp shutdown scenario */
  u_char shutdown;

  /* Label Advertisement Mode.  */
  u_char adv_mode;

  /* Label Retention Mode.  */
  u_char retention_mode;

  /* Label Distribution Control Mode.  */
  u_char control_mode;
  #define ORDERED_CONTROL_MODE               0
  #define INDEPENDENT_CONTROL_MODE           1

  /* Timer configuration.  */
  u_int16_t hold_time;
  u_int16_t keepalive_timeout;
  u_int16_t hello_interval;
  u_int16_t targeted_hello_interval;
  u_int16_t targeted_hold_time;
  u_int16_t targeted_hold_time_for_sync;
  u_int16_t keepalive_interval;

  /* Max PDU length - newly added member */
  u_int32_t pdu_len;

 /* LDP Entity configuration.  */
  u_int32_t config;
  #define LDP_CONFIG_HOLD_TIME                (1 << 1)
  #define LDP_CONFIG_KEEPALIVE_TIMEOUT        (1 << 2)
  #define LDP_CONFIG_ADV_MODE                 (1 << 3)
  #define LDP_ENTITY_CONFIG_LABEL_RETENTION_MODE (1 << 4)
  #define LDP_CONFIG_ENTITY_LOOP_DETECTION    (1 << 5)
  #define LDP_CONFIG_HELLO_INTERVAL           (1 << 6)
  #define LDP_CONFIG_KEEPALIVE_INTERVAL       (1 << 7)
  #define LDP_CONFIG_TARGETED_HELLO_INTERVAL  (1 << 8)
  #define LDP_CONFIG_TARGETED_HOLD_TIME       (1 << 9)
  #define LDP_CONFIG_TARGETED_HELLO_RECV      (1 << 10)
  #define LDP_CONFIG_TARGETED_HOLD_TIME_SYNC  (1 << 11)
  #define LDP_CONFIG_MAX_PDU_LENGTH           (1 << 12)

  /* LDP Entity loop detection status */
  u_char loop_detect_status;
  #define LDP_ENTITY_LOOP_DETECTION_ON      1
  #define LDP_ENTITY_LOOP_DETECTION_OFF     0

 /* Timer configuration.  */
  u_int32_t reconnect_timeout;

  /* Init Session threshold - - newly added member */
  u_int32_t init_sess_threshold;

  /* loop_detection_count should be replaced by the following variables */

  /* Path Vector Limit - - newly added member */
  u_int32_t path_vec_limit;

  /* Hop count limit -- newly added member */
  u_int32_t hop_count_limit;

  /* Entity statistics */
  struct ldp_stats stats;

  /*Entity row status */
   mpls_row_status_t row_status;

 /* Entity Admn Status */
   u_int32_t admn_status;
  /* Targeted Hello receipt global setting */
  #define LDP_TARGETED_HELLO_RECV_DISABLED        0
  #define LDP_TARGETED_HELLO_RECV_ENABLED         1
  /* statistics. */
  u_int32_t notification_send;
  u_int32_t notification_recv;
  u_int32_t hello_send;
  u_int32_t hello_recv;
  u_int32_t initialization_send;
  u_int32_t initialization_recv;
  u_int32_t keepalive_send;
  u_int32_t keepalive_recv;
  u_int32_t address_send;
  u_int32_t address_recv;
  u_int32_t address_withdraw_send;
  u_int32_t address_withdraw_recv;
  u_int32_t label_mapping_send;
  u_int32_t label_mapping_recv;
  u_int32_t label_request_send;
  u_int32_t label_request_recv;
  u_int32_t label_withdraw_send;
  u_int32_t label_withdraw_recv;
  u_int32_t label_release_send;
  u_int32_t label_release_recv;
  u_int32_t request_abort_send;
  u_int32_t request_abort_recv;
};
```

ldp_fec

This data structure is defined in the `ldpd/ldpd.h` file.

Definition

```
struct ldp_fec
{
  u_int32_t row_status;
  u_int32_t stor_type;

  union {
    struct prefix prefix; /* fec prefix or host address */
#ifdef HAVE_CR_LDP
    struct ldp_lspid lspid;
#endif
#ifdef HAVE_MPLS_VC
    struct ldp_vcid vc;
#endif /* HAVE_MPLS_VC */
  } u;
  u_char fec_type; /* host, prefix or cr-lsp */
};
```

ldp_fec_cb

This data structure is for the FEC control block. It defined in the `ldpd/ldpd.h` file.

Definition

```
struct ldp_fec_cb
{
  struct ldp_fec fec;

  u_int8_t family;

/* Pointer to ldp route table*/
  struct route_node *ldp_rt;

#ifdef HAVE_CR_LDP
  /* initial attributes for LSP setup */
  struct ldp_cr_attr *lsp_attr;
#endif /* HAVE_CR_LDP */

#ifdef HAVE_SNMP
  u_int32_t snmp_index;
  u_int32_t snmp_lastchange;
#endif
```

ldp_id

This data structure is defined in the `ldpd/ldpd.h` file.

Definition

```
struct ldp_id id;

  /* Holdtime thread for this adjacency */
  struct thread *t_holdtime;

#ifdef HAVE_SNMP
  /* Timestamp when last hello packet was received */
  pal_time_t hello_received;
#endif /* HAVE_SNMP */

  /* Hold time */
  u_int16_t holdtime;

  /* Flag for track whether adjacency is with a targeted peer */
  u_char targeted_peer;

  /* LDP adjacency family */
  u_char family;
```

```
};
```

ldp_interface

This data structure is defined in the `ldpd/ldp_interface.h` file.

Definition

```
struct ldp_interface
{
  /* Primary IP address.  */
  struct pal_in4_addr primary_addr;

#ifdef HAVE_IPV6
  /* Primary IPv6 address.  */
  struct pal_in6_addr primary_addr6;
#endif /* HAVE_IPV6 */

  /* Interface structure.  */
  struct interface *ifp;

  u_int32_t config_seq_num;

  /* Accept Socket for TCP Connects */
  int accept_sock;
  /* Thread to read accept sock */
  struct thread *t_accept;
  /* Count of passive sessions waiting to accept */
  int accept_count;

  /* Timer configuration.  */
  u_int32_t reconnect_timeout;

  /* PDU Length. */
  u_int32_t pdu_length;


  /* LDP configuration.  */
  u_int32_t config;
#define LDP_IF_CONFIG_HELLO_INTERVAL          (1 << 0)
#define LDP_IF_CONFIG_HOLD_TIME               (1 << 1)
#define LDP_IF_CONFIG_KEEPALIVE_INTERVAL      (1 << 2)
#define LDP_IF_CONFIG_KEEPALIVE_TIMEOUT       (1 << 3)
#define LDP_IF_CONFIG_ADV_MODE                (1 << 6)
#define LDP_IF_CONFIG_LABEL_RETENTION_MODE    (1 << 7)
#define LDP_IF_CONFIG_LABEL_TYPE              (1 << 8)
#define LDP_IF_CONFIG_NO_MULTICAST_HELLOS     (1 << 9)
#define LDP_IF_CONFIG_RECONNECT_TIMEOUT       (1 << 10)
#define LDP_IF_CONFIG_RECOVERY_TIME           (1 << 11)
#define LDP_IF_CONFIG_HOLD_TIME_SYNC          (1 << 12)
#define LDP_IF_CONFIG_MAX_PDU_LENGTH          (1 << 13)
```

```
#define LDP_IF_CONFIG_SYNC_DELAY_INTERVAL     (1 << 14)

  /* Type of interface */
  u_int16_t interface_type;

  /* Timer configuration.  */
  u_int16_t hello_interval;
  u_int16_t hold_time;
  u_int16_t hold_time_for_sync;
  u_int16_t keepalive_interval;
  u_int16_t keepalive_timeout;
  u_int16_t sync_delay_interval;

  /* Threads */
  struct thread *t_hello;
#ifdef HAVE_IPV6
  struct thread *ipv6_t_hello;
#endif /* HAVE_IPV6 */

  /* Pointer to labelspace-to-address object. */
  struct ls_to_addr *l_addr;

 /*
   * Storage for label space data.
   *
   * When we receive an update for an interface, we dont know
   * which interface will be affected until it's too late.
   * Therefore, we need to store a backup copy of label space
   * data in the protocol specific interface structure.
   */
  struct label_space_data ls_data;

  /* Flags. */
  u_char flags;
#define LDP_IF_FLAG_NO_MULTICAST_HELLOS    (1 << 0)
#define LDP_IF_FLAG_BIND_MPLS_VC           (1 << 1)
#define LDP_IF_FLAG_BIND_VPLS              (1 << 2)
#define LDP_IF_FLAG_ADDRS_ADVERTISED       (1 << 3)
#define LDP_IF_FLAG_SHUTDOWN               (1 << 4)

  /* Interface Activation Status */
  u_char if_active;

  /* Advertisement Mode */
  u_char adv_mode;

  /* Label Retention Mode */
  u_char retention_mode;

  /* Label merge capability */
  u_char label_merge_cap;

  /* flag to test for primary address */
  u_char addrflag;
#ifdef HAVE_IPV6
  u_char addrflag6;
#endif /* HAVE_IPV6 */

  /* Flag to tell whether interface is enabled by the user.  */
  u_char enable_interface;
#define LDP_IPV4_ENABLE     (1 << 0)
#define LDP_IPV6_ENABLE     (1 << 1)
#define LDP_IGP_SYNC_ENABLE (1 << 2)

 /* Flag to tell whether IPV4 or IPV6 has been enabled by the user */
  u_char enable_family;
#define LDP_IF_IPV4_ENABLE     (1 << 0)
#define LDP_IF_IPV6_ENABLE     (1 << 1)
#define LDP_IF_IGP_SYNC_ENABLE (1 << 2)

};
```

ldp_ip_nh

This data structure is defined in the ldpd/ldp_nsm.h file.

Definition

```
struct ldp_ip_nh
{
  int type;
  u_int8_t family;
  union
  {
    struct pal_in4_addr nexthop4;
#ifdef HAVE_IPV6
    struct pal_in6_addr nexthop6;
#endif /* HAVE_IPV6 */
  }u;
  u_int32_t ifindex;
};
```

ldp_session

This data structure is defined in the ldpd/ldpd.h file.

Definition

```
struct ldp_session
{

  /* Back pointer to ldp entity */
  struct ldp_entity *entity;

  /* List of adjacencies */
  struct list *adjacencies;

  /* Peer LDP-ID */
  struct ldp_id id;

  /* My LDP-ID */
  struct ldp_id my_id;

  u_int8_t family;

  /* Peer address. */
  union
  {
    struct pal_in4_addr peer4;
#ifdef HAVE_IPV6
   struct pal_in6_addr peer6;
#endif /* HAVE_IPV6 */
  }u;

  /* Session specific timers */
  u_int32_t reconnect_timeout;
  u_int32_t recovery_time;

  /* TCP socket for this LDP session.  */
  int sock;

   /* Types of session peer */
  #define LDP_SESS_MULTICAST_PEER    (1<<0)
  #define LDP_SESS_TARGETED_PEER     (1<<1)
  #define LDP_SESS_VC_PEER           (1<<2)
  u_int32_t peer_type;

#ifdef HAVE_SNMP
  /* time stamp of last state change */
  pal_time_t last_state_change;

  /* statistics */
  u_int32_t unknown_mess_recv;
  u_int32_t unknown_tlv_recv;
#endif /* HAVE_SNMP */

  /* Session specific timers */
  u_int16_t keepalive_timeout;

   /* Max PDU length */
  u_int32_t pdu_len;
  /* State of current session */
  u_char state;
  u_char old_state;

  /* Active / Passive role */
#define SESSION_ROLE_PASSIVE              0
#define SESSION_ROLE_ACTIVE               1
  u_char role;

  /* Peer label merge capability */
  u_char peer_label_merge_cap;

#ifdef HAVE_RESTART
  bool_t nsm_stale_entries_mark_sent;

  /* Peer Graceful Restart capability */
#define GRACEFUL_RESTART_NOT_CAPABLE      0
#define GRACEFUL_RESTART_CAPABLE          1
  u_char graceful_restart_cap;

#define GRACEFUL_RESTART_PEER_LSR               (1 << 0)
#define GRACEFUL_RESTART_RECONNECT_TIMEOUT      (1 << 1)
#define GRACEFUL_RESTART_RECOVERY_TIMEOUT       (1 << 2)
#define GRACEFUL_RESTART_SESSION_PRESERVE       (1 << 3)
#define GRACEFUL_RESTART_FSM_STARTED            (1 << 4)
#define GRACEFUL_RESTART_RESET                  (1 << 5)
#define GRACEFUL_RESTART_ADJ_HOLD_TIMER_EXPIRED (1 << 6)
  u_char grflags;
#endif /* HAVE RESTART */

  /* Label advertisement discipline
  * 0 == Downstream Unsolicited
  * 1 == Downstream on Demand
  *
  * If two LSRs cannot agree on discipline type, use DOD for
  * ATM and FR and DU for everything else.
  */
  u_char adv_mode;

  /* Copy of retention mode from ldpif or ldp, whichever one applies */
  u_char retention_mode;

  /* Flag to track whether all FECs need to be advertised to peer */
  u_char fec_adv_enabled;

  /* Flag to check whether tcp is established or not */
  u_char tcp_established;

  /* status of peer session */
#define LDP_SESSION_OK_TO_SEND_REQUEST    (1 << 0)
```

```
#define LDP_SESSION_NO_LABEL_NOTIF_SENT   (1 << 1)
  u_char status_record;

  /* Delete flag */
  u_char delete_flag;

 u_char shutdown;

  /* My interface id */
  struct ldp_interface *ldpif;

  /* Graceful Restart Timer Threads */
  struct thread *t_reconnect_time;
  struct thread *t_recovery_time;

  /* TCP reconnect timer */
  struct thread *t_tcp_reconnect;

  /* Threads */
  struct thread *t_read;
  struct thread *t_write;
  struct thread *t_event;

  /* LDP IGP Sync delay timer thread */
  struct thread *t_sync_delay_timer;

  /* Message "queue" for write thread */
  /* This will consist of write_queue_node
     structs defined above */
  struct list *write_queue;

  /* Keepalive threads */
  struct thread *t_keepalive_timeout;
  struct thread *t_keepalive_interval;

  /* Re-connect timeout parameters */
#define LDP_SESSION_MIN_RECONNECT_TIME       15
#define LDP_SESSION_MAX_RECONNECT_TIME       120
  int reconnect_interval;
  struct thread *t_reconnect;

  /* Local copy of addresses received */
  struct route_table *remote_addresses;

  /* Input data buffer */
  struct ldp_session_buffer recv_buf;

  /* Upstream Control Block list */
  struct ldp_upstream *ucb_list;
  /* Downstream Control Block list */
  struct list *dcb_list;

  /* Initial fec advertisement delay for a session */
  struct thread *t_fec_adv_delay;

#ifdef HAVE_MPLS_VC
  struct list *svl_list;
#endif /* HAVE_MPLS_VC */
};
```

ldp_downstream

```
 struct ldp_downstream
    {
      struct ldp_downstream *next;
      struct ldp_downstream *prev;
      struct ldp_downstream *fec_dcb_next;
      struct ldp_session *session;               /* backpointer to ldp session */
      struct ldp_label label;                    /* label sent to the peer */

    /* List of nh_info structures */
     + struct  list  * nexthops;
   }
      struct nh_info
      {
         struct ldp_ip_nh *next_hop;
         u_int32_t nhlfe_ix;  };
```

ldp_upstream

This data structure is defined in the `ldpd/ldp_fec.h` file.

Definition

```
struct ldp_upstream;
struct ldp_downstream;

ldp_fec_install_ilm (struct ldp_upstream *ucb, struct ldp_downstream *dcb)
void ldp_fec_remove_ilm (struct ldp_upstream *ucb, struct ldp_downstream *dcb)
void ldp_fec_install_ilm_all (struct ldp_downstream *dcb, bool_t unset)
void ldp_fec_install_ftn (struct ldp_downstream *dcb, struct list *list)
void ldp_fec_remove_ftn (struct ldp_downstream *dcb,struct list *nh_list)
void ldp_fec_remove_ilm (struct ldp_upstream *);
void ldp_fec_install_ftn (struct ldp_downstream *);
void ldp_fec_remove_ftn (struct ldp_downstream *, struct pal_in4_addr *);
void ldp_fec_install_ilm_all (struct ldp_downstream *);
void ldp_fec_remove_ilm_all (struct ldp_downstream *);
int ldp_fec_downstream_add (struct ldp_fec_cb *, struct ldp_downstream *);
int ldp_fec_downstream_remove (struct ldp_fec_cb *, struct ldp_downstream *);
s_int32_t ldp_fec_ftn_add_queue_cmp (void *, void *);
```

```
s_int32_t ldp_fec_ilm_add_queue_cmp (void *, void *);


#ifdef HAVE_LDP_INTER_AREA
struct route_node *ldp_fec_matched_lpm_rib_table_add (struct route_table *,
                                                      struct prefix *,
                                                      struct ldp_fec_cb *);
void ldp_fec_matched_lpm_rib_table_clear (struct route_table *);
void ldp_update_fec_cb_lpm_entries_new_nexthop (struct prefix *,
                                               struct ldp_ip_nh *);
void ldp_update_fec_cb_lpm_entries_new_prefix (struct prefix *,
                                               struct ldp_ip_nh *);
void ldp_update_fec_cb_lpm_entries_prefix_deleted (struct route_table *,
                                                   struct prefix *);
#endif /* HAVE_LDP_INTER_AREA */

#ifdef HAVE_RESTART
int ldp_fec_stale_add_msg_process (struct nsm_msg_stale_info *);
struct ldp_fec_stale_info * ldp_fec_stale_new (struct nsm_msg_stale_info *, int *);
```

listnode

This data structure is defined in the `ha/common/libs/linklist.h` file.

Definition

```
struct listnode
{
  struct listnode *next;
  struct listnode *prev;
  void *data;
};
```

pal_in4_addr

This data structure helps manage IPv4 address functions. It is defined in the `pal/dummy/pal_types.h` file.

Definition

```
struct pal_in4_addr {
  u_int32_t s_addr; // IPv4 address in 32-byte format
};
```

prefix

This data structure holds information about an IPv4 address. It is defined in the `lib/prefix.h` file.

Member	Description
family	Prefix’s family member
prefixlen	Prefix length
pad1	Prefix address one
pad2	Prefix address two

Definition

```
struct prefix
{
  u_int8_t family;
  u_int8_t prefixlen;
  u_int8_t prefix_style;
  u_int8_t pad1;
  union
  {
    u_int8_t prefix;
    struct pal_in4_addr prefix4;
#ifdef HAVE_IPV6
    struct pal_in6_addr prefix6;
#endif /* HAVE_IPV6 */
    struct
    {
      struct pal_in4_addr id;
      struct pal_in4_addr adv_router;
    } lp;
    u_int8_t val[9];
  } u;
};
```

route_node

This data structure is for each routing entry. It is defined in the `lib/table.h` file.

Definition

```
struct route_node
{
  /* DO NOT MOVE the first 2 pointers. They are used for memory
     manager as well */
  struct route_node *link[2];
#define l_left   link[0]
#define l_right  link[1]

  /* Actual prefix of this radix. */
  struct prefix p;

  /* Tree link. */
  struct route_table *table;
  struct route_node *parent;

  /* Lock of this radix */
  u_int32_t lock;

  /* Each node of route. */
  void *info;

  /* Aggregation. */
  void *aggregate;
};
```

route_table

This data structure is the Routing table top structure. It is defined in the `lib/table.h` file.

Definition

```
struct route_table
{
  struct route_node *top;

  /* Table identifier. */
  u_int32_t id;
};
```

snmp_mplsFecEntry_T

This data structure is defined in the `ldpd/ldp_api.h` file.

Definition

```
struct snmp_mplsFecEntry_T
{
  u_int32_t mplsFecIndex;
  s_int32_t mplsFecType;
  u_int32_t  mplsFecAddrLength;
  s_int32_t mplsFecAddrFamily; /* SNMP_AG_ADDR_TYPE_XXX from ~/lib/snmp_misc.h */
  s_int32_t mplsFecRowStatus; /* SNMP_AG_ROW_XXX from ~/lib/snmp_misc.h */
  s_int32_t mplsFecStorType;  /* SNMP_AG_STOR_XXX from ~/lib/snmp_misc.h */
  /*unsigned char mplsFecAddr[65]; */
  union
   {
     struct pal_in4_addr mplsFecAddr;
#ifdef HAVE_IPV6
     struct pal_in6_addr mplsFecAddr6;
#endif /* HAVE_IPV6 */
   }u;

#define FEC_TYPE_FLAG           (1 << 0)
#define FEC_ADDR_FLAG           (1 << 1)
#define FEC_ADDR_LEN_FLAG        (1 << 2)

  u_char flags;
};
```

CHAPTER 9 LDP API

This chapter contains the API for LDP. It includes the following functions:

ldp_api_get_id

This call gets the Router ID of the LDP.

Syntax

```
void
ldp_api_get_id (u_char id_ptr[4])
```

Input Parameters

None

Output Parameters

`id_ptr`	Contains the Router ID of the LDP entity if LDP is active, zero value otherwise.

Return Value

None

ldp_api_get_loop_detection

This call gets the loop-detection configuration of LDP.

Syntax

```
int
ldp_api_get_loop_detection (void)
```

Input Parameters

None

Output Parameters

None

Return Value

`LDP_TRUE` if LDP is configured to perform loop-detection

`LDP_FALSE`, otherwise.

ldp_api_set_inter_area_lsp

This API is called by the `inter-area ldp` command and is used to configure inter-area LSPs.

Syntax

```
void
ldp_api_set_inter_area_lsp (struct ldp *ldp, bool_t no_restart)
```

Input Parameters

`ldp`	LDP instance
`no_restart`	No restart.

Output Parameters

None

Return Values

None

ldp_api_set_loop_detection

This call sets the loop detection flag for the LDP instance.

Syntax

```
int
ldp_api_set_loop_detection (struct ldp *ldp, u_char loop_detect)
```

Input Parameters

`ldp`	LDP instance
`loop_detect`	Loop Detect flag. The valid values for this flags include the following:

`LDP_LOOP_DETECTION_ON (1)`

Enables loop detection capability of LDP.

`LDP_LOOP_DETECTION_OFF (0)`

Disables loop detection capability of LDP.

Output Parameters

None

Return Values

LDP_TRUE

LDP_FALSE

ldp_api_get_loop_detect_hop_limit

This call returns the loop-detect hop-count limit for the given entity, if the loop detection is enabled.

Syntax

```
u_char
ldp_api_get_loop_detect_hop_limit (struct ldp *ldp,
                                   struct ldp_entity *entity)
```

Input Parameters

`ldp`	LDP instance
`entity`	LDP entity

Output Parameters

None

Return Values

Hop count limit

LDP_ERROR

ldp_api_set_loop_detect_hop_limit

This call sets the LDP loop-detect hop limit for all LDP entities.

Syntax

```
int
ldp_api_set_loop_detect_hop_limit (struct ldp *ldp,
                                   u_int32_t val, u_char count)
```

Input Parameters

ldp	LDP instance
val	LDP hop-count limit
count	LDP loop-detection count flag. The allowed values include the following:
	LDP_LOOP_DETECTION_COUNT_SET(1)
	LDP_LOOP_DETECTION_COUNT_UNSET(0)

Output Parameters

None

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_api_set_entity_loop_detect_hop_limit

This call sets the LDP entity loop-detect hop limit.

Syntax

```
void
ldp_api_set_entity_loop_detect_hop_limit (struct ldp_entity *entity,
                                          u_int32_t val, u_char count,
                                           bool_t entity_config_set)
```

Input Parameters

*entity	LDP entity
count	LDP loop-detection count flag.
	LDP_LOOP_DETECTION_COUNT_SET(1)
	LDP_LOOP_DETECTION_COUNT_UNSET(0)
entity_config_set	
	LDP entity configuration set.

Output Parameters

None

Return Values

None

ldp_api_get_entity_last_change

This call gets the time of the last change of the entity.

Syntax

```
s_int32_t
ldp_api_get_entity_last_change (struct ldp *ldp)
```

Input Parameters

`ldp`	LDP instance

Output Parameters

None

Return Values

LDP entity last change time

ldp_api_get_entity_index_next

This call gets the next available entity index.

Syntax

```
u_int32_t
ldp_api_get_entity_index_next (struct ldp *ldp)
```

Input Parameters

`ldp`	LDP instance

Output Parameters

None

Return Values

LDP entity

NULL

ldp_api_entity_lookup_by_id

This function returns the LDP entity based on the given LDP ID.

Syntax

```
struct ldp_entity *
ldp_api_entity_lookup_by_id (struct ldp *ldp, struct ldp_id *ldp_id)
```

Input Parameters

`ldp`	LDP instance
`ldp_id`	LDP ID

Output Parameters

None

Return Values

LDP entity

NULL

ldp_api_entity_lookup_by_index

This function returns the entity based on the LDP index.

Syntax

```
struct ldp_entity *
ldp_api_entity_lookup_by_index (struct ldp *ldp, struct ldp_id *id,
                                u_int32_t *index)
```

Input Parameters

`ldp`	LDP instance
`*id`	LDP ID
`*index`	LDP index

Output Parameters

None

Return Values

LDP entity or NULL

ldp_api_entity_lookup_next_by_index

This function returns the next entity based on the LDP index.

Syntax

```
struct ldp_entity *
ldp_api_entity_lookup_next_by_index (struct ldp *ldp, struct ldp_id *id,
                                      u_int32_t *index)
```

Input Parameters

`ldp`	LDP instance
`*id`	LDP ID
`*index`	LDP index

Output Parameters

index   The next LDP index

Return Values

LDP entity

NULL

ldp_api_get_proto_version

This call gets the protocol version.

Syntax

```
u_int32_t
ldp_api_get_proto_version (struct ldp *ldp)
```

Input Parameters

`ldp`	LDP instance

Output Parameters

None

Return Values

Protocol version

ldp_api_set_admn_status

This call sets the admin status for the entity.

Syntax

```
int
ldp_api_set_admn_status (struct ldp_entity *entity, s_int32_t intval)
```

Input Parameters

entity	LDP entity
intval	Admin status. The valid values included either: SNMP_AG_ADMIN_ENABLE(1) or SNMP_AG_ADMIN_DISABLE(2).

Output Parameters

None

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_api_get_tcp_port

This call gets the Transmission Control Protocol (TCP) port used by LDP.

Syntax

```
u_int16_t
ldp_api_get_tcp_port ()
```

Input Parameters

None

Output Parameters

None

Return Value

LDP_DEFAULT_PORT_TCP: TCP port used by LDP. Only supports default TCP port.

ldp_snmp_api_set_entity_tcp_port

This call sets the TCP port number used by the LDP entity.

Syntax

```
int
ldp_snmp_api_set_entity_tcp_port (struct ldp_entity *entity, s_int32_t intval)
```

Input Parameters

`entity`	LDP entity
`intval`	TCP port number used by the LDP entity. Only the default value, LDP_DEFAULT_PORT_TCP (646), is allowed.

Output Parameters

`entity`	LDP entity

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_api_get_udp_port

This call gets the User Datagram Protocol (UDP) port used by LDP.

Syntax

```
u_int16_t
ldp_api_get_udp_port ()
```

Input Parameters

None

Output Parameters

None

Return Value

LDP_DEFAULT_PORT_UDP: UDP port used by LDP.

ldp_api_get_max_pdu_length

This call gets the maximum PDU length.

Syntax

```
u_int16_t
ldp_api_get_max_pdu_length (struct ldp *ldp, struct ldp_entity *entity)
```

Input Parameters

`ldp`	LDP instance
`entity`	LDP entity

Output Parameters

None

Return Values

LDP_PDU_MAX_SIZE

ldp_api_get_keepalive_timer

This call gets the keepalive timer value for the given entity.

Syntax

```
u_int32_t
ldp_api_get_keepalive_timer (struct ldp *ldp, struct ldp_entity *entity)
```

Input Parameters

`ldp`	LDP instance
`entity`	LDP entity

Output Parameters

None

Return Values

Null

Keepalive timeout value

ldp_api_set_keepalive_timer

This call sets the keepalive timer for all LDP entities.

Syntax

```
int
ldp_api_set_keepalive_timer (struct ldp *ldp, u_int16_t keepalive_timeout,
                             u_char timeout)
```

Input Parameters

`ldp`	LDP instance
`keepalive_timeout`	Value of the keepalive timer, the permitted range is (1 - 65535)
`timeout`	Keepalive timer flag. The valid values are LDP_KEEPALIVE_TIMEOUT_SET (1) and LDP_KEEPALIVE_TIMEOUT_UNSET(0)

Output Parameters

None

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_api_set_entity_keepalive_timer

This call sets the keepalive timer for the entity.

Syntax

```
int ldp_api_set_entity_keepalive_timer (struct ldp_entity *entity,
                                         u_int16_t keepalive_timeout,
                                         u_char timeout)
```

Input Parameters

`*entity` LDP entity

`keepalive_timeout`
Value of the keepalive timer, the permitted range is (1 - 65535)

`timeout` Keepalive timer flag. The valid values are LDP_KEEPALIVE_TIMEOUT_SET (1) and LDP_KEEPALIVE_TIMEOUT_UNSET(0)

Output Parameters

None

Return Values

LDP_ERR_INVALID_KEEPALIVE_TIMEOUT

LDP_SUCCESS

ldp_api_if_set_keepalive_timer

This call sets the keepalive timeout value for the LDP interface.

Syntax

```
int
ldp_api_if_set_keepalive_timer (u_int16_t keepalive_timeout,
                                struct interface* ifp,
                                u_char if_timeout)
```

Input Parameters

`ifp` Interface pointer

`keepalive_timeout`
Keepalive timeout

`if_timeout` Keepalive interval flag. The permitted values are LDP_IF_KEEPALIVE_TIMEOUT_SET(1) and LDP_IF_KEEPALIVE_TIMEOUT_SET(0).

Output Parameters

None

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_api_set_keepalive_int

This call sets the keepalive interval for all entities.

Syntax

```
int
ldp_api_set_keepalive_int (struct ldp *ldp,
                          u_int16_t keepalive_interval,
                          u_char interval)
```

Input Parameters

`ldp`	LDP instance
`keepalive_interval`	Keepalive interval, the permitted range is (1 - 65535)
`interval`	Keepalive interval flag. The permitted values are LDP_KEEPALIVE_INTERVAL_SET(1) and LDP_KEEPALIVE_INTERVAL_UNSET(0)

Output Parameters

None

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_api_if_set_keepalive_int

This call sets the keepalive interval for the interface.

Syntax

```
int
ldp_api_if_set_keepalive_int (u_int16_t keepalive_interval,
                              struct interface* ifp,
                              u_char if_int)
```

Input Parameters

`ifp`	Interface pointer
`keepalive_interval`	
	Keepalive interval; the allowed range is (1 - 65535).
`if_int`	Keepalive interval flag. The permitted values are LDP_IF_KEEPALIVE_INTERVALL_SET(1) and LDP_IF_KEEPALIVE_INTERVAL_UNSET(0)

Output Parameters

None

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_api_set_entity_keepalive_int

This call sets the keepalive interval for an entity.

Syntax

```
int ldp_api_set_entity_keepalive_int (struct ldp_entity *entity,
                                       u_int16_t keepalive_interval,
                                       u_char interval)
```

Input Parameters

`entity`	LDP entity
`keepalive_interval`	
	Keepalive interval; the permitted range is (1 - 65535).
`interval`	Keepalive interval flag. The permitted values are LDP_KEEPALIVE_INTERVAL_SET(1) and LDP_KEEPALIVE_INTERVAL_UNSET(0)

Output Parameters

None

Return Values

LDP_SUCCESS when the keepalive interval is successfully set

LDP_ERROR when the keepalive interval is not set

ldp_api_get_hold_timer

This call gets the hold time of an entity.

Syntax

```
u_int32_t
ldp_api_get_hold_timer (struct ldp *ldp, struct ldp_entity *entity)
```

Input Parameters

`ldp`	LDP instance
`entity`	LDP entity

Output Parameters

None

Return Values

Hold time

LDP_ERROR

ldp_api_set_hold_timer

This call sets the hold time for all entities.

Syntax

```
int
ldp_api_set_hold_timer (struct ldp *ldp,
                        u_int16_t hold_timer, u_char flag)
```

Input Parameters

`ldp`	LDP instance
`hold_timer`	Hold timer flag. Valid values are LDP_HOLD_TIME_SET (1) and LDP_HOLD_TIME_UNSET (0)
`flag`	Hold timer flag. Valid values are LDP_HOLD_TIME_SET (1) and LDP_HOLD_TIME_UNSET (0)

Output Parameters

None

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_api_set_entity_hold_timer

This call sets the entity hold timer.

Syntax

```
int ldp_api_set_entity_hold_timer (struct ldp_entity *entity,
                                    u_int16_t hold_timer,
                                    u_char flag)
```

Input Parameters

entity	LDP entity
hold_timer	Value of the hold timer
flag	Hold timer flag. Valid values are LDP_HOLD_TIME_SET (1) and LDP_HOLD_TIME_UNSET (0)

Output Parameters

None

Return Values

LDP_ERR_INVALID_HOLD_TIME

LDP_SUCCESS

LDP_ERROR

ldp_api_set_targeted_hold_timer

This call sets the targeted hold time.

Syntax

```
int
ldp_api_set_targeted_hold_timer (struct ldp *ldp,
                                 u_int16_t targeted_hold_timer,
                                 u_char targeted_time)
```

Input Parameters

ldp	LDP instance
targeted_hold_timer	Targeted hold time, the range of values is (1 - 65535)
targeted_time	Targeted hold time flag. The valid values are LDP_TARGETED_HOLD_TIME_SET (1) and LDP_TARGETED_HOLD_TIME_UNSET (0)

Output Parameters

None

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_api_if_set_hold_timer

This call sets the hold timer for the given LDP interface.

Syntax

```
int
ldp_api_if_set_hold_timer (u_int16_t hold_time, struct interface* ifp,
                          u_char if_time)
```

Input Parameters

`hold_time`	Hold time value (1-65535)
`ifp`	Interface pointer
`if_time`	Hold time flag. The valid values are LDP_IF_HOLD_TIME_SET (1) and LDP_IF_HOLD_TIME_UNSET (0)

Output Parameters

None

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_api_get_session_threshold

This call gets the init session threshold value for the given entity.

Syntax

```
int
ldp_api_get_session_threshold (struct ldp *ldp, struct ldp_entity * entity)
```

Input Parameters

`ldp`	LDP instance
`entity`	LDP entity

Output Parameters

None

Return Values

The init session threshold value.

ldp_api_get_retention_mode

This call gets the retention mode of the LDP entity.

Syntax

```
u_char ldp_api_get_retention_mode (struct ldp *ldp, struct ldp_entity *entity)
```

Input Parameters

`ldp`	LDP instance
`entity`	LDP entity

Output Parameters

None

Return Values

The value of the retention mode

ldp_api_set_retention_mode

This call sets the retention mode for all LDP entities.

Syntax

```
int
ldp_api_set_retention_mode (struct ldp *ldp,
                            s_int32_t intval, u_char retention)
```

Input Parameters

`ldp`	LDP instance
`intval`	Value of session threshold
`retention`	Retention flag. The allowed values are LDP_LABEL_RETENTION_MODE_SET (1) and LDP_LABEL_RETENTION_MODE_UNSET (0)

Output Parameters

None

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_api_set_entity_retention_mode

This call sets the entity retention mode.

Syntax

```
void ldp_api_set_entity_retention_mode (struct ldp_entity *entity, s_int32_t intval,
                                        u_char retention)
```

Input Parameters

entity	LDP entity
intval	Value of session threshold
retention	Retention flag. The allowed values are LDP_LABEL_RETENTION_MODE_SET (1) and LDP_LABEL_RETENTION_MODE_UNSET (0)

Output Parameters

None

Return Values

None

ldp_api_if_set_retention_mode

This call sets the interface retention mode.

Syntax

```
int
ldp_api_if_set_retention_mode (u_char is_liberal,
                               struct interface *ifp,
                               u_char if_retention)
```

Input Parameters

is_liberal	Retention mode
ifp	Interface pointer
if_retention	Retention flag. The allowed values are LDP_IF_LABEL_RETENTION_MODE_SET (1) and LDP_IF_LABEL_RETENTION_MODE_UNSET (0)

Output Parameters

None

Return Values

LDP_SUCCESS

LDP_FAILURE

ldp_api_get_default_label_retention_mode

This call gets the default label retention mode.

Syntax

```
int
ldp_api_get_default_label_retention_mode (struct ldp *ldp, int mode)
```

Input Parameters

`ldp`	LDP instance
`mode`	Value of the label retention mode

Output Parameters

None

Return Values

LIBERAL_RETENTION_MODE if the retention mode is liberal

CONSERVATIVE_RETENTION_MODE if the retention mode is conservative

LDP_ERROR if the LDP instance is not found

ldp_api_get_transport_addr_kind

This call gets the transport address kind of the given entity.

Syntax

```
int
ldp_api_get_transport_addr_kind (struct ldp *ldp, struct ldp_entity *entity)
```

Input Parameters

`ldp`	LDP instance
`entity`	LDP entity

Output Parameters

None

Return Values

Transport address kind value in the entity

ldp_api_get_target_peer_recv

This call checks whether or not receiving targeted hello messages is enabled in the LDP entity.

Syntax

```
u_char
ldp_api_get_target_peer_recv (struct ldp *ldp, struct ldp_entity *entity)
```

Input Parameters

`ldp`	LDP instance
`entity`	LDP entity

Output Parameters

None

Return Values

LDP_TRUE if receiving targeted hello messages is enabled in the LDP entity

LDP_FALSE if not

ldp_api_set_target_peer_recv

This call sets the target hello receive flag for all entities in an LDP instance.

Syntax

```
int
ldp_api_set_target_peer_recv (struct ldp *ldp, int flag)
```

Input Parameters

`ldp`	LDP instance
`flag`	Targeted hello receive flag, including:
`LDP_TRUE`	Sets the targeted hello receive flag
`LDP_FALSE`	Unsets the targeted hello receive flag

Output Parameters

None

Return Values

LDP_SUCCESS

LDP_API_SET_ERROR

ldp_api_targeted_peer_add

This call creates a targeted peer.

Syntax

```
int
ldp_api_targeted_peer_add (struct ldp *ldp, struct prefix *p,
                           struct ldp_entity *entity, int src)
```

Input Parameters

`ldp`	LDP instance
`p`	Prefix
`entity`	LDP entity
`src`	Source, either CLI or SNMP

Output Parameters

None

Return Values

LDP_SUCCESS	If the targeted peer is successfully added
LDP_ERR_TARGETED_PEER_DEFINED	If the targeted peer is already defined
LDP_ERR_MEM_ALLOC_FAILURE	Memory allocation failure while creating a new targeted peer

ldp_api_targeted_peer_del

This call deletes a targeted peer.

Syntax

```
int
ldp_api_targeted_peer_del (struct ldp *ldp, struct prefix *p)
```

Input Parameters

`ldp`	LDP instance
`p`	Prefix

Output Parameters

None

Return Values

LDP_ERR_TARGETED_NOT_FOUND if the specified targeted peer is not found

Zero if the targeted peer is successfully deleted

ldp_api_get_entity_row_status

This call gets the LDP entity row status.

Syntax

```
int
ldp_api_get_entity_row_status (struct ldp *ldp, struct ldp_entity *entity)
```

Input Parameters

`ldp`	LDP instance
`entity`	LDP entity

Output Parameters

None

Return Values

Row status of the LDP entity

ldp_api_get_discontinuity_time

This call gets the last time a discontinuity occurred in any of the counters associated with this LDP entity.

Syntax

```
pal_time_t
ldp_api_get_discontinuity_time ()
```

Input Parameters

None

Output Parameters

None

Return Value

System time at which the last discontinuity occurred in any of the counters associated with this LDP entity.

0

ldp_api_get_fec_last_change

This call gets the last change time of the FEC.

Syntax

```
s_int32_t
ldp_api_get_fec_last_change (struct ldp *ldp)
```

Input Parameters

`ldp`	LDP instance

Output Parameters

None

Return Values

FEC last change time

ldp_api_get_path_vector_limit

This call gets the path vector limit value of the given entity, if the loop detection is enabled.

Syntax

```
u_int32_t
ldp_api_get_path_vector_limit (struct ldp *ldp, struct ldp_entity *entity)
```

Input Parameters

ldp	LDP instance
entity	LDP entity

Output Parameters

None

Return Values

The path vector limit value.

0

ldp_api_get_target_peer_addr_type

This call gets the target peer address type.

Syntax

```
int
ldp_api_get_target_peer_addr_type (struct ldp *ldp, struct ldp_entity *entity)
```

Input Parameters

ldp	LDP instance
entity	LDP entity

Output Parameters

None

Return Values

SNMP_AG_ADDR_TYPE_ipv4

SNMP_AG_ADDR_TYPE_ipv6

LDP_ERROR

ldp_api_get_entity_label_type

This call gets the entity label type.

Syntax

```
int
ldp_api_get_entity_label_type (struct ldp *ldp, struct ldp_entity *entity)
```

Input Parameters

`ldp`	LDP instance
`entity`	LDP entity

Output Parameters

None

Return Values

SNMP_AG_LBL_TYPE_generic

LDP_ERROR

ldp_api_get_last_peer_change

This call gets the last time at which an addition or deletion to the peer table or session table occurred.

Syntax

```
pal_time_t
ldp_api_get_last_peer_change (struct ldp *ldp)
```

Input Parameters

`ldp`	LDP instance

Output Parameters

None

Return Value

System time at which the last addition or deletion to the peer table or session table occurred.

ldp_api_get_interface_by_label_space

This call returns the first valid interface for the given label-space.

Syntax

```
struct interface *
ldp_api_get_interface_by_label_space (int label_space)
```

Input Parameters

`label_space`	ID of the label-space.

Output Parameters

None

Return Value

A pointer to the first valid interface associated with this label-space.

NULL

ldp_api_getnext_interface_by_label_space

This call returns the interface of the label-space that is next after the given label-space.

Syntax

```
struct interface *
ldp_api_getnext_interface_by_label_space (int label_space, int first)
```

Input Parameters

`label_space`	ID of the label-space.
`first`	A boolean value designating whether to return the interface with the minimal label-space, or the interface with the minimal label-space greater than label_space.

Output Parameters

None

Return Value

A pointer to the requested interface.

NULL

None

ldp_get_mplsLdpLspFecEntry

This call gets the LDP FEC instance parameters in the SNMP FEC structure.

Syntax

```
int
ldp_get_mplsLdpLspFecEntry (struct ldp* ldp,
                             snmp_mplsLdpLspFecEntry_T * entry_ptr
```

Input Parameters

`ldp`	LDP instance

Output Parameters

`entry_ptr`	SNMP LDP LSP entry

Return Values

LDP_SUCCESS

LDP_FALSE

ldp_getnext_mplsLdpLspFecEntry

This call gets the next LDP FEC instance parameters in the SNMP FEC structure.

Syntax

```
int
ldp_getnext_mplsLdpLspFecEntry (struct ldp *ldp,
                                snmp_mplsLdpLspFecEntry_T * entry_ptr)
```

Input Parameters

`ldp`	LDP instance

Output Parameters

`entry_ptr`	SNMP LDP LSP entry

Return Values

LDP_SUCCESS

LDP_FALSE

ldp_api_create_ldp_entity

This call creates an LDP entity with the given entity index and LDP ID.

Syntax

```
ldp_api_create_ldp_entity (s_int32_t idx, struct ldp_id ldpId, int src)
```

Input Parameters

`idx`	LDP entity index
`ldpId`	LDP ID; combination of LSR ID and label space
`src`	Source. This parameter can take following values:
`SRC_CLI`	Denotes that this entity is created through CLI.
`SRC_SNMP`	Denotes that this entity is created through SNMP.

Output Parameters

None

Return Values

LDP_TRUE

LDP_FALSE

ldp_api_delete_ldp_entity

This call deletes the LDP entity.

Syntax

```
int
ldp_api_delete_ldp_entity (struct ldp_entity *entity)
```

Input Parameters

`entity`	LDP entity

Output Parameters

None

Return Values

-1 if the entity is not deleted

0 if the entity is successfully deleted

ldp_api_get_default_control_mode

This call gets the default control mode.

Syntax

```
int
ldp_api_get_default_control_mode (struct ldp_entity *ldp_entity)
```

Input Parameters

`ldp_entity`	LDP entity

Output Parameters

None

Return Values

ORDERED_CONTROL_MODE

INDEPENDENT_CONTROL_MODE

ldp_api_set_control_mode

This call sets the control mode.

Syntax

```
int
ldp_api_set_control_mode (struct ldp *ldp, u_char is_ordered, u_char mode)
```

Input Parameters

`ldp`	LDP instance
`is_order`	IS order, including ORDERED_CONTROL_MODE or INDEPENDENT_CONTROL_MODE
`mode`	Control mode flag

Output Parameters

None

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_api_get_adv_mode

This call gets the advertisement mode used by the given entity.

Syntax

```
u_char
ldp_api_get_adv_mode (struct ldp *ldp, struct ldp_entity *entity)
```

Input Parameters

`ldp`	LDP instance
`entity`	LDP entity

Output Parameters

None

Return Values

Advertisement mode

ldp_api_set_adv_mode

This call sets the advertisement mode for all entities.

Syntax

```
int
ldp_api_set_adv_mode (struct ldp *ldp,
                      u_char is_downstream_on_demand, u_char advertise)
```

Input Parameters

`ldp` LDP instance

`is_downstream_on_demand`

LDP downstream on-demand flag. The valid values are LDP_TRUE (0) and LDP_FALSE(1) corresponding to DOWNSTREAM_ON_DEMAND_MODE and DOWNSTREAM_UNSOLICITED_MODE, respectively.

`advertise` Advertisement flag. The valid values are LDP_ADVERTISE_MODE_SET (1) and LDP_ADVERTISE_MODE_UNSET (0).

Output Parameters

None

Return Values

LDP_SUCCESS

LDP_FALSE

ldp_api_if_set_adv_mode

This call sets the advertisement mode for the LDP interface.

Syntax

```
int
ldp_api_if_set_adv_mode (u_char is_on_demand,
                         struct interface *ifp,
                         u_char if_advertise)
```

Input Parameters

`ifp` Interface pointer

`is_on_demand` LDP downstream on-demand flag. The valid values are LDP_TRUE (0) and LDP_FALSE(1) corresponding to DOWNSTREAM_ON_DEMAND_MODE and DOWNSTREAM_UNSOLICITED_MODE, respectively.

`if_advertise` Advertisement flag. The valid values are LDP_IF_ADVERTISE_MODE_SET (1) and LDP_IF_ADVERTISE_MODE_UNSET (0)

Output Parameters

None

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_api_set_entity_adv_mode

This call sets the entity advertisement mode.

Syntax

```
void ldp_api_set_entity_adv_mode (struct ldp_entity *entity,
                             u_char is_downstream_on_demand,
                             u_char advertise)
```

Input Parameters

`entity` LDP entity

`is_downstream_on_demand`

LDP downstream on-demand flag. The valid values are LDP_TRUE (0) and LDP_FALSE(1) corresponding to DOWNSTREAM_ON_DEMAND_MODE and DOWNSTREAM_UNSOLICITED_MODE, respectively.

`advertise` Advertisement flag. The valid values are LDP_ADVERTISE_MODE_SET (1) and LDP_ADVERTISE_MODE_UNSET (0)

Output Parameters

None

Return Values

LDP_ERROR

LDP_SUCCESS

ldp_api_set_request_retry_timeout

This call sets the request retry timeout value for all entities.

Syntax

```
int
ldp_api_set_request_retry_timeout (u_int16_t request_retry_timeout,
                                   struct ldp *ldp, u_char request)
```

Input Parameters

`request_retry_timeout`

Request retry timer value with a range of (1 - 65535)

`ldp` LDP instance

`request` request retry timer flag. The allowed values are LDP_REQUEST_RETRY_TIMEOUT_SET(1) and LDP_REQUEST_RETRY_TIMEOUT_UNSET(0)

Output Parameters

None

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_api_set_global_merge_capability

This call sets global merge capability for the LDP instance.

Syntax

```
int
ldp_api_set_global_merge_capability  (u_char enable, struct ldp *ldp,
                                       u_char merge_capability)
```

Input Parameters

`enable`	Flag to enable or disable merge capability; permitted values are LDP_TRUE, LDP_FALSE
`ldp`	LDP instance
`merge_capability`	Global merge capability flag; valid values are LDP_GLOBAL_MERGE_CAPABILITY_SET(1) and LDP_GLOBAL_MERGE_CAPABILITY_UNSET(0)

Output Parameters

None

Return Values

LDP_FALSE

LDP_TRUE

ldp_api_get_hello_int

This call gets the hello interval value for the given entity.

API CAll

```
u_int32_t
ldp_api_get_hello_int (struct ldp *ldp, struct ldp_entity *entity)
```

Input Parameters

`ldp`	LDP instance
`entity`	LDP entity

Output Parameters

None

Return Values

Hello interval

ldp_api_set_hello_int

This call sets the hello interval.

Syntax

```
int
ldp_api_set_hello_int (struct ldp *ldp, u_int16_t hello_interval,
                       u_char interval)
```

Input Parameters

`ldp`	LDP instance
`hello_interval`	LDP hello interval value
`interval`	hello interval flag. Valid values are LDP_HELLO_INTERVAL_SET(1) and LDP_HELLO_INTERVAL_UNSET(0)

Output Parameters

None

Return Values

`LDP_SUCCESS` if set was successful

`LDP_FALSE` if set was not successful

ldp_api_set_entity_hello_int

This call sets the entity hello interval for the LDP entity.

Syntax

```
int
ldp_api_set_entity_hello_int (struct ldp_entity *entity,
                              u_int16_t hello_interval , u_char interval)
```

Input Parameters

`entity`	LDP entity
`hello_interval`	LDP hello interval value
`interval`	Hello interval flag. Valid values are LDP_HELLO_INTERVAL_SET(1) and LDP_HELLO_INTERVAL_UNSET(0)

Output Parameters

None

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_api_if_set_hello_int

This call sets the hello interval for the LDP interface.

Syntax

```
int
ldp_api_if_set_hello_int (u_int16_t hello_interval,
                          struct interface *ifp, u_char if_interval)
```

Input Parameters

`hello_interval`	LDP hello interval value
`ifp`	Interface pointer
`if_interval`	Interface interval flag.

Output Parameters

None

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_api_set_tar_peer_hello_interval

This call sets the targeted peer hello interval for all entities.

Syntax

```
int
ldp_api_set_tar_peer_hello_interval (struct ldp *ldp,
                                     u_int16_t hello_interval,
                                     u_char target_hello_int)
```

Input Parameters

`ldp`	LDP instance
`hello_interval`	Hello interval
`target_hello_int`	Hello interval flag. The permitted values are LDP_TARGETED_PEER_HELLO_INTERVAL_SET (0) and LDP_TARGETED_PEER_HELLO_INTERVAL_SET (1).

Output Parameters

None

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_api_get_fec_from_rt

This call gets the FEC from the route table.

Syntax

```
int
ldp_api_get_fec_from_rt (s_int32_t indx, struct prefix *prefix,
                         struct ldp_fec **fec,
                         struct route_table *rt,
                         struct prefix **prefixptr)
```

Input Parameters

`indx`	FEC index
`prefix`	IP address in prefix format
`rt`	Route table

Output Parameters

`fec`	LDP FEC structure
`prefixptr`	FEC prefix pointer

Return Values

LDP_TRUE

LDP_ERROR

ldp_api_session_clean_all

This call cleans up sessions specific to the interface.

Syntax

```
u_char
ldp_api_session_clean_all (struct ldp_interface *ldpif, u_char explicit_clean)
```

Input Parameters

`ldpif`	LDP interface
`explicit_clean`	Flag for explicit session cleanup, including LDP_FALSE and LDP_TRUE.

Output Parameters

None

Return Values

LDP_FALSE If sessions are not cleaned

LDP_TRUE If sessions are cleaned

ldp_api_session_restart

This call restarts the LDP session.

Syntax

```
int
ldp_api_session_restart (struct ldp_session *session, u_char notify,
                         bool_t restart_gracefully)
```

Input Parameters

`ldp_session` LDP session

`notify` Flag to indicate if notifications are to be sent

`restart_gracefully`

Restart gracefully

Output Parameters

None

Return Values

0 if session is restarted successfully

ldp_api_statistics_reset

This call resets LDP statistics.

Syntax

```
void
ldp_api_statistics_reset (struct ldp_entity *entity)
```

Input Parameters

`entity`	LDP entity

Output Parameters

None

Return Values

None

ldp_api_get_target_peer_addr

This call gets the peer address used for targeted discovery in the specified label space.

Syntax

```
u_int32_t
ldp_api_get_target_peer_addr (struct ldp *ldp, u_int32_t label_space)
```

Input Parameters

`ldp`	LDP instance
`label_space`	Relevant label space

Output Parameters

None

Return Values

Peer address used for extended discovery.

0

ldp_api_get_adjacency

This call gets the LDP adjacency.

Syntax

```
int
ldp_api_get_adjacency (struct ldp *ldp,
                        struct ldp_id *peer_id,
                        struct listnode **ses_node,
                        struct ldp_adjacency **adj)
```

Input Parameters

ldp	LDP instance
peer_id	LDP peer ID

Output Parameters

ses_node	LDP session
adj	LDP adjacency

Return Values

LDP_SUCCESS

LDP_FALSE

ldp_api_clear_adjacency

This call clears adjacencies.

Syntax

```
int
ldp_api_clear_adjacency (int clear_all, struct prefix *p)
```

Input Parameters

clear_all	Flag indicating if all adjacencies are to be cleared or not
p	Prefix

Output Parameters

None

Return Values

LDP_SUCCESS if the adjacency is successfully cleared

LDP_ERROR if the adjacency is not cleared

ldp_api_set_advert_list

This call sets the advertise list.

Syntax

```
int
ldp_api_set_advert_list (struct cli *cli, afi_t afi,
                         struct ldp_adv_list_master *master,
                         char *prefix_acl, char *peer_acl)
```

Input Parameters

`cli`	Pointer to the CLI structure
`afi`	Address family (IPv4/IPv6)
`master`	Master data structure
`prefix_acl`	Prefix access control list name
`peer_acl`	Peer access control list name

Output Parameters

None

Return Values

`CLI_SUCCESS` if set was successful

`CLI_ERROR` if set was not successful

ldp_api_unset_advert_list

This call unsets the advertise list.

Syntax

```
int
ldp_api_unset_advert_list (struct cli *cli, afi_t afi,
                           struct ldp_adv_list_master *master,
                           char *prefix_acl, char *peer_acl)
```

Input Parameters

`cli`	Pointer to the CLI structure
`afi`	Address family (IP/IP6)
`master`	Master data structure
`prefix_acl`	Prefix access control list name
`peer_acl`	Peer access control list name

Output Parameters

None

Return Values

`CLI_SUCCESS` if successful; `CLI_ERROR` if not successful

ldp_api_clear_advert_list

This call clears the advertise list.

Syntax

```
int
ldp_api_clear_advert_list (struct cli *cli,
                           struct ldp_adv_list_master *master,
                           char *prefix_acl, char *peer_acl)
```

Input Parameters

`cli`	Pointer to the CLI structure
`master`	Master data structure
`prefix_acl`	Prefix access control list name
`peer_acl`	Peer access control list name

Output Parameters

None

Return Values

CLI_SUCCESS if clear was successful

CLI_ERROR if clear was not successful

ldp_api_create_ldp_instance

This call creates an LDP instance.

Syntax

```
nt ldp_api_create_ldp_instance ()
```

Input Parameters

None

Output Parameters

None

Return Values

LDP_SUCCESS   if instance is successfully created

LDP_ERROR   if instance is not created

ldp_api_delete_ldp_instance

This call deletes an LDP instance.

Syntax

```
int ldp_api_delete_ldp_instance ()
```

Input Parameters

None

Output Parameters

None

Return Value

LDP_SUCCESS	if instance is successfully deleted

ldp_api_router_id_set

This call sets the router ID.

Syntax

```
int
ldp_api_router_id_set (struct ldp *ldp, struct pal_in4_addr *id)
```

Input Parameters

`ldp`	LDP instance
`id`	Router ID

Output Parameters

None

Return Values

0 if the router ID is successfully set

ldp_api_router_id_unset

This call unsets the router ID.

Syntax

```
void ldp_api_router_id_unset (struct ldp *ldp)
```

Input Parameters

`ldp`	LDP instance

Output Parameters

None

Return Values

None

ldp_api_set_multicast_hellos

This call sets the multicast hellos.

Syntax

```
void
ldp_api_set_multicast_hellos (struct ldp *ldp)
```

Input Parameters

ldp	LDP instance

Output Parameters

None

Return Values

None

ldp_api_unset_multicast_hellos

This call unsets the multicast hellos.

Syntax

```
void
ldp_api_unset_multicast_hellos (struct ldp *ldp)
```

Input Parameters

ldp	LDP instance

Output Parameters

None

Return Values

None

ldp_api_ls_to_addr_update_by_addr

This call updates the label space address by IP address.

Syntax

```
int
ldp_api_ls_to_addr_update_by_addr (struct ldp *ldp, struct pal_in4_addr *addr)
```

Input Parameters

ldp	LDP instance
addr	IP address

Output Parameters

None

Return Values

LDP_ERR_NOT_FOUND when a match is not found

LDP_SUCCESS if successfully updated

LDP_FALSE if not updated

ldp_api_ls_to_addr_update_by_val

This call updates the label space address by label space value.

Syntax

```
int
ldp_api_ls_to_addr_update_by_val (struct ldp *ldp, u_int16_t label_space,
                                  struct pal_in4_addr *addr)
```

Input Parameters

`ldp`	LDP instance
`label_space`	Label space value
`addr`	IP address

Output Parameters

None

Return Values

LDP_ERR_NOT_FOUND when a match is not found

LDP_SUCCESS if successfully updated

LDP_FALSE if not updated

ldp_api_ls_to_addr_get

This call gets label-space to address mapping instance.

Syntax

```
int ldp_api_ls_to_addr_get (struct ldp *ldp, u_int16_t lspace,
                            struct pal_in4_addr *addr)
```

Input Parameters

`ldp`	LDP instance
`lspace`	Label space value
`addr`	IP address

Output Parameters

None

Return Values

None

ldp_api_nsm_redistribute

This call selectively redistributes or flushes routes.

Syntax

```
void
ldp_api_nsm_redistribute (u_char type, u_char add)
```

Input Parameters

`type`	Route type: IPI_ROUTE_DEFAULT IPI_ROUTE_KERNEL IPI_ROUTE_CONNECT IPI_ROUTE_STATIC IPI_ROUTE_RIP IPI_ROUTE_RIPNG IPI_ROUTE_OSPF IPI_ROUTE_OSPF6 IPI_ROUTE_BGP IPI_ROUTE_ISIS
`add`	Redistributes or flushes

Output Parameters

None

Return Values

None

ldp_api_activate_interface

This call activates the interface.

Syntax

```
int
ldp_api_activate_interface (struct ldp_interface *ldpif, afi_t afi)
```

Input Parameters

`ldpif`	LDP interface
`afi`	Address family (IP/IP6)

Output Parameters

None

Return Values

None

ldp_api_deactivate_interface

This call deactivates the interface.

Syntax

```
void
ldp_api_deactivate_interface (struct ldp_interface *ldpif, u_int16_t lspace)
```

Input Parameters

`ldpif`	LDP interface
`lspace`	Label space

Output Parameters

None

Return Values

None

ldp_api_interface_enable_multicast_hellos

This call enables multicast hellos for the specified interface.

Syntax

```
void
ldp_api_interface_enable_multicast_hellos (struct ldp_interface *ldpif)
```

Input Parameters

`ldpif`	LDP interface

Output Parameters

None

Return Values

None

ldp_api_interface_disable_multicast_hellos

This call disables multicast hellos for the specified interface.

Syntax

```
void
```

```
ldp_api_interface_disable_multicast_hellos (struct ldp_interface *ldpif)
```

Input Parameters

`ldpif`	LDP interface

Output Parameters

None

Return Values

None

ldp_update_fec_cb_lpm_entries_new_prefix

This call disables multicast hellos for the specified interface.

Syntax

```
void
ldp_update_fec_cb_lpm_entries_new_prefix (struct prefix *p,
                                           struct ldp_ip_nh *nh)
```

Input Parameters

`p`	Prefix
`*nh`	

Output Parameters

None

Return Values

None

route_node_lookup

Lookup same prefix node. Return NULL when we can't find route. */

struct route_node *

route_node_lookup (struct route_table *table, struct prefix *p)

Syntax

```
void
ldp_update_fec_cb_lpm_entries_new_prefix (struct prefix *p,
                                           struct ldp_ip_nh *nh)
```

Input Parameters

`p`	Prefix
`*nh`	

Output Parameters

None

Return Values

None

CHAPTER 10 LDP Traps

This chapter contains the traps for LDP. It includes the following traps:

- ldpTrapSessionUp on page 121
- ldpTrapSessionDown on page 121
- ldpTrapEntityInitSesThreshold on page 122

ldpTrapSessionUp

This call sends a trap when an LDP session becomes operational.

Syntax

```
void
ldpTrapSessionUp (struct ldp_session *session)
```

Input Parameters

`session`	Relevant session.

Output Parameters

None

Return Value

None

ldpTrapSessionDown

This call sends a trap when an LDP session is not operational.

Syntax

```
void
ldpTrapSessionDown (struct ldp_session *session)
```

Input Parameters

`session`	Relevant session.

Output Parameters

None

Return Value

None

ldpTrapEntityInitSesThreshold

This call sends a trap indicating that some LDP entity retried unsuccessfully to establish a session an excessive number of times.

Syntax

```
void
ldpTrapEntityInitSesThreshold (struct ldp_entity  *entity)
```

Input Parameters

`entity`	LDP entity.

Output Parameters

None

Return Value

None

CHAPTER 11 LDP MIBs API

This chapter describes Management Information Base (MIB) support.

Supported Tables

The LDP MIB is implemented in accordance with these standards:

- RFC 3815: Label Distribution Protocol (LDP) MIB.

The MIB tables and variables listed in the sections below are supported in ZebOS-XP. The tables contain cross-references to the function for each Object Type.

mplsLDPEntityTable

Object Type	Syntax	Access	Functions
mplsLdpEntityProtocolVersion		read-create	ldp_snmp_api_set_protocol_version
mplsLdpEntityAdminStatus		read-create	ldp_api_set_admn_status
mplsLdpEntityTcpPort		read-create	ldp_snmp_api_set_entity_tcp_port
mplsLdpEntityUdpDscPort		read-create	ldp_snmp_api_set_entity_tcp_port
mplsLdpEntityMaxPduLength		read-create	ldp_snmp_api_set_max_pdu_length
mplsLdpEntityKeepAliveHoldTimer		read-create	ldp_snmp_api_set_keepalive_timer
mplsLdpEntityHelloHoldTimer		read-create	ldp_snmp_api_set_hello_hold_timer
mplsLdpEntityInitSessionThreshold		read-create	ldp_snmp_api_set_init_session_threshold
mplsLdpEntityLabelDistMethod		read-create	ldp_snmp_api_set_adv_mode
mplsLdpEntityLabelRetentionMode		read-create	ldp_snmp_api_set_label_retention_mode
mplsLdpEntityPathVectorLimit		read-create	ldp_snmp_api_set_path_vector_limit
mplsLdpEntityHopCountLimit		read-create	ldp_snmp_api_set_hop_count_limit
mplsLdpEntityTransportAddrKind		read-create	ldp_snmp_api_set_transport_addr_kind
mplsLdpEntityTargetPeer		read-create	ldp_snmp_api_set_target_peer_recv
mplsLdpEntityTargetPeerAddrType		read-create	ldp_snmp_api_set_target_peer_addr_type

mplsLdpEntityTargetPeerAddr		read-create	ldp_snmp_api_set_target_peer_recv
mplsLdpEntityLabelType		read-create	ldp_snmp_api_set_label_type
mplsLdpEntityStorageType		read-create	ldp_snmp_api_set_fec_stor_type
mplsLdpEntityRowStatus		read-create	ldp_snmp_api_set_entity_row_status

mplsFecTable

Object Type	Syntax	Access	Functions
mplsFecType		read-write	ldp_snmp_api_get_fec
mplsFecAddrType		read-write	ldp_snmp_api_set_fec_addr_type
mplsFecAddr		read-write	ldp_snmp_api_set_fec_addr
mplsFecAddrPrefixLength		read-write	ldp_snmp_api_set_fec_addr_prefix_length
mplsFecStorageType		read-write	ldp_snmp_api_set_fec_stor_type
mplsFecRowStatus		read-write	ldp_snmp_api_set_fec_row_status

APIs

Functions	Description
ldp_snmp_init	This call initializes the LDP SNMP support code
ldp_snmp_stop	This call stops the LDP SNMP support code
ldp_snmp_api_set_entity_udp_port	This call sets the UDP port number used by the LDP entity
ldp_snmp_api_set_max_pdu_length	This call sets the maximum PDU length of the LDP entity
ldp_snmp_api_set_keepalive_timer	This call sets the keepalive time of the LDP entity
ldp_snmp_api_set_init_session_threshold	This call sets the initial session threshold for the entity
ldp_snmp_api_set_target_peer_recv	This call sets the targeted hello receive flag for the LDP entity
ldp_snmp_api_set_entity_row_status	This call sets the row status of the LDP entity
ldp_snmp_api_get_peer_trans_addr	This call gets the peer transport address
ldp_snmp_api_get_inseg_ldp_lsp	This call gets the values of the in segment LSP parameters for the given index
ldp_snmp_api_getnext_inseg_ldp_lsp	This call gets the values of the next in segment LSP parameters for the next index

Functions	Description
ldp_snmp_api_get_outseg_ldp_lsp	This call gets the values of the outsegment LSP parameters for the given index
ldp_snmp_shadow_entry_lookup	This function returns the SNMP FEC shadow entry based on the FEC index
ldp_snmp_api_set_fec_type	This call sets the FEC type for the given FEC index
ldp_snmp_api_set_fec_addr_prefix_length	This call sets the FEC address prefix length for the given FEC prefix
ldp_snmp_api_set_fec_addr_type	This call sets the FEC address type for the SNMP FEC entry.
ldp_snmp_api_set_fec_row_status	This call sets the row status of the FEC table
ldp_snmp_api_set_path_vector_limit	This call sets the path vector limit of the entity
ldp_snmp_api_set_fec_stor_type	This call sets the FEC storage type for the SNMP FEC entry
ldp_snmp_fec_entry_add	This call adds the specified shadow table entry to the FEC table
ldp_snmp_api_set_adv_mode	This call sets the advertisement mode of the LDP entity
ldp_snmp_api_get_fec	This call gets the FEC for the given index
ldp_snmp_api_getnext_fec	This call gets the next FEC structure in the route table
ldp_snmp_api_del_fec	This call deletes the LDP FEC with the given FEC index
ldp_snmp_api_conv_ldp_2_snmp	This call converts an LDP FEC entry to an FEC SNMP entry
snmp_convert_ldp_to_entry	This call converts the LDP entries to SNMP entries
ldp_snmp_api_get_session_node	This call gets the LDP session node for the given peer LDP ID
ldp_snmp_get_session_node	This call gets the LDP session which matches the given entity ID and peer LDP ID
ldp_snmp_getnext_session_node	This call gets the next available LDP session
ldp_snmp_extract_session_data	This call gets the session-related data
snmp_get_entity_staticstics	This call gets the LDP entity statistics
ldp_snmp_api_getnext_adjacency	This call gets the parameters of the next adjacency
ldp_snmp_api_get_peer	This call gets the peer parameters
ldp_snmp_api_getnext_peer	This call gets the next peer parameters
ldp_snmp_api_getnext_peer_addr	This call gets the next LDP peer address
ldp_snmp_api_get_adjacency_node	This call gets the adjacency node

ldp_snmp_init

This call initializes the LDP SNMP support code.

Syntax

```
void
ldp_snmp_init (void)
```

Input Parameters

None

Output Parameters

None

Return Value

None

ldp_snmp_stop

This call stops the LDP SNMP support code.

Syntax

```
void
ldp_snmp_stop (void)
```

Input Parameters

None

Output Parameters

None

Return Value

None

ldp_snmp_api_set_protocol_version

This call sets the protocol version of the LDP entity.

Syntax

```
int
ldp_snmp_api_set_protocol_version (struct ldp_entity *entity, s_int32_t intval)
```

Input Parameters

`entity`	LDP entity
`intval`	Protocol version

Output Parameters

`entity`	LDP entity

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_snmp_api_set_entity_udp_port

This call sets the UDP port number used by the LDP entity.

Syntax

```
int
ldp_snmp_api_set_entity_udp_port (struct ldp_entity *entity, s_int32_t intval)
```

Input Parameters

`entity`	LDP entity
`intval`	TCP port number used by the LDP entity. Only the default value, LDP_DEFAULT_PORT_TCP (646), is allowed.

Output Parameters

`entity`	LDP entity

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_snmp_api_set_max_pdu_length

This call sets the maximum PDU length of the LDP entity.

Syntax

```
int
ldp_snmp_api_set_max_pdu_length (struct ldp_entity *entity, s_int32_t intval)
```

Input Parameters

`entity`	LDP entity
`intval`	Maximum PDU length

Output Parameters

`entity`	LDP entity

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_snmp_api_set_keepalive_timer

This call sets the keepalive time of the LDP entity.

Syntax

```
int
ldp_snmp_api_set_keepalive_timer (struct ldp_entity *entity, s_int32_t intval)
```

Input Parameters

`entity`	LDP entity
`intval`	Value of keepalive time

Output Parameters

`entity`	LDP entity

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_snmp_api_set_hello_hold_timer

This call sets the hello hold timer of the LDP entity.

Syntax

```
int
ldp_snmp_api_set_hello_hold_timer (struct ldp_entity *entity, s_int32_t intval)
```

Input Parameters

`entity`	LDP entity
`intval`	Hello time value

Output Parameters

`entity`	LDP entity

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_snmp_api_set_init_session_threshold

This call sets the initial session threshold for the entity.

Syntax

```
int
ldp_snmp_api_set_init_session_threshold (struct ldp_entity *entity,
                                         s_int32_t intval)
```

Input Parameters

`entity`	LDP entity
`intval`	Value of session threshold

Output Parameters

`entity`	LDP entity

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_snmp_api_set_label_retention_mode

This call sets the label retention mode of the LDP entity

Syntax

```
int
ldp_snmp_api_set_label_retention_mode (struct ldp_entity *entity, s_int32_t intval)
```

Input Parameters

`entity`	LDP entity
`intval`	Value of label retention mode

Output Parameters

`entity`	LDP entity

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_snmp_api_set_transport_addr_kind

This call sets the transport address kind used by the LDP entity.

Syntax

```
int
ldp_snmp_api_set_transport_addr_kind (struct ldp_entity *entity, s_int32_t intval)
```

Input Parameters

`entity`	LDP entity
`intval`	Value of label retention mode

Output Parameters

`entity`	LDP entity

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_snmp_api_set_target_peer_recv

This call sets the targeted hello receive flag for the LDP entity.

Syntax

```
int
ldp_snmp_api_set_target_peer_recv (struct ldp_entity *entity, s_int32_t enable)
```

Input Parameters

`entity`	LDP entity
`enable`	Targeted hello receive flag, including:
`LDP_TRUE`	Sets the targeted hello receive flag
`LDP_FALSE`	Unsets the targeted hello receive flag

Output Parameters

None

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_snmp_api_set_entity_row_status

This call sets the row status of the LDP entity.

Syntax

```
int
ldp_snmp_api_set_entity_row_status (struct ldp_entity **org_entity,
                                     s_int32_t row_status,
                                     struct ldp_id *ldpId,
                                     u_int32_t *idx)
```

Input Parameters

`org_entity`	LDP ORG entity
`row_status`	Value of the row status; the permitted values include the following: SNMP_AG_ROW_createAndGo SNMP_AG_ROW_createAndWait SNMP_AG_ROW_active SNMP_AG_ROW_notReady SNMP_AG_ROW_notInService

	SNMP_AG_ROW_destroy
*ldpId	LDP ID
idx	LDP index

Output Parameters

entity	LDP entity

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_snmp_api_get_peer_trans_addr

This call gets the peer transport address.

Syntax

```
int
ldp_snmp_api_get_peer_trans_addr (struct ldp *ldp,
                                  struct ldp_entity *entity,
                                  struct ldp_id *peer_id,
                                  struct pal_in4_addr *addr)
```

Input Parameters

ldp	LDP instance
entity	LDP entity ID
peer_id	LDP peer ID

Output Parameters

addr	Transport address of the peer

Return Values

LDP_SUCCESS

LDP_FALSE

ldp_snmp_api_get_inseg_ldp_lsp

This call gets the values of the in segment LSP parameters for the given index.

Syntax

```
int ldp_snmp_api_get_inseg_ldp_lsp (struct ldp *ldp,
                                    struct ldp_id *ldp_id,
                                    u_int32_t ldp_index,
                                    struct ldp_id *peer_id,
                                    s_int32_t *lspIndex,
                                    s_int32_t *label_type,
                                    s_int32_t *lsp_type)
```

Input Parameters

`ldp`	LDP instance
`ldp_id`	LDP entity ID
`ldp_index`	LDP entity index
`peer_id`	LDP peer ID

Output Parameters

`lspIndex`	LSP index
`label_type`	LSP label type
`lsp_type`	LSP type

Return Values

LDP_SUCCESS

LDP_FALSE

ldp_snmp_api_getnext_inseg_ldp_lsp

This call gets the values of the next in segment LSP parameters for the next index.

Syntax

```
snmp_mplsInSegLdpLspEntry_T *
ldp_snmp_api_getnext_inseg_ldp_lsp (struct ldp *ldp,
                             struct ldp_id *ldp_id,
                             u_int32_t *entity_indx,
                             struct ldp_id *peer_id,
                             s_int32_t *lspIndex,
                             snmp_mplsInSegLdpLspEntry_T **best_entry)
```

Input Parameters

`ldp`	LDP instance
`ldp_id`	LDP entity ID
`entity_indx`	LDP entity index
`peer_id`	LDP peer ID

Output Parameters

`lspIndex`	LSP index
`**best_entry`	Best entry

Return Values

LDP_SUCCESS

LDP_FALSE

ldp_snmp_api_get_outseg_ldp_lsp

This call gets the values of the outsegment LSP parameters for the given index.

Syntax

```
int ldp_snmp_api_get_outseg_ldp_lsp (struct ldp *ldp,
                                      struct ldp_id *ldp_id,
                                      u_int32_t ldp_index,
                                      struct ldp_id *peer_id,
                                      s_int32_t *lspIndex,
                                      s_int32_t *label_type,
                                      s_int32_t *lsp_type)
```

Input Parameters

`ldp`	LDP instance
`ldp_id`	LDP entity ID
`ldp_index`	LDP entity index
`peer_id`	LDP peer ID

Output Parameters

`lspIndex`	LSP index
`label_type`	LSP label type
`lsp_type`	LSP type

Return Values

LDP_SUCCESS

LDP_FALSE

ldp_snmp_api_getnext_outseg_ldp_lsp

This call gets the values of the next outsegment LSP parameters for given index.

Syntax

```
snmp_mplsOutSegLdpLspEntry_T *
ldp_snmp_api_getnext_outseg_ldp_lsp (struct ldp *ldp,
                                      struct ldp_id *ldp_id,
                                      u_int32_t *ldp_index,
                                      struct ldp_id *peer_id,
                                      s_int32_t *lspIndex,
                                      snmp_mplsOutSegLdpLspEntry_T **best_entry)
```

Input Parameters

`ldp`	LDP instance
`ldp_id`	LDP entity ID
`ldp_index`	LDP entity index
`peer_id`	LDP peer ID

Output Parameters

`lspIndex`	LSP index
`**best_entry`	Best entry

Return Values

LDP_SUCCESS

LDP_FALSE

ldp_snmp_api_set_hop_count_limit

This call sets the hop-count limit used by the LDP entity.

Syntax

```
int
ldp_snmp_api_set_hop_count_limit (struct ldp_entity *entity, s_int32_t intval)
```

Input Parameters

`entity`	LDP entity
`intval`	Value of label retention mode

Output Parameters

`entity`	LDP entity

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_snmp_shadow_entry_lookup

This function returns the SNMP FEC shadow entry based on the FEC index.

Syntax

```
struct snmp_mplsFecEntry_T *ldp_snmp_shadow_entry_lookup (u_int32_t index,
                                                           bool_t exact)
```

Input Parameters

`index`	FEC index
`exact`	Type of lookup (get or get next)

Output Parameters

None

Return Values

NULL

select_entry

ldp_snmp_api_set_fec_type

This call sets the FEC type for the given FEC index. The FEC could be present either in the FEC route table or the FEC shadow entry table.

Syntax

```
int
ldp_snmp_api_set_fec_type (struct ldp *ldp, s_int32_t fecType,
                           struct snmp_mplsFecEntry_T *shadow_entry,
                           struct ldp_fec *fec)
```

Input Parameters

`ldp`	LDP instance
`fecType`	FEC type
`shadow_entry`	SNMP FEC shadow entry. This value can be NULL.
`fec`	Pointer to FEC

Output Parameters

None

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_snmp_api_set_fec_addr_prefix_length

This call sets the FEC address prefix length for the given FEC prefix. The FEC could be present either in the FEC route table or the FEC shadow entry table.

Syntax

```
int
ldp_snmp_api_set_fec_addr_prefix_length (s_int32_t addrLength, struct prefix *p,
                                          struct snmp_mplsFecEntry_T
                                          *shadow_entry)
```

Input Parameters

`addrLength`	Value of address length
`prefix`	FEC prefix
`shadow_entry`	SNMP FEC shadow entry. This value can be NULL.

Output Parameters

None

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_snmp_api_set_fec_addr_type

This call sets the FEC address type for the SNMP FEC entry.

Syntax

```
int
ldp_snmp_api_set_fec_addr_type (s_int32_t addrType,
                                struct snmp_mplsFecEntry_T *shadow_entry)
```

Input Parameters

`addrType`	Address type
`shadow_entry`	SNMP FEC entry

Output Parameters

`shadow_entry`	SNMP FEC entry

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_snmp_api_set_fec_row_status

This call sets the row status of the FEC table. The FEC could be present either in the FEC route table or the FEC shadow entry table.

Syntax

```
int
ldp_snmp_api_set_fec_row_status (struct ldp *ldp, s_int32_t index, s_int32_t rowStatus,
                                 bool_t fec_present,
                                 struct snmp_mplsFecEntry_T *shadow_entry,
                                 struct ldp_fec *fec, struct prefix *prefix)
```

Input Parameters

`ldp`	LDP instance
`index`	LDP index
`rowStatus`	Value of row status; the permitted values are
	SNMP_AG_ROW_createAndGo,
	SNMP_AG_ROW_createAndWait,
	SNMP_AG_ROW_active,
	SNMP_AG_ROW_notInService,
	SNMP_AG_ROW_notReady
`fec_present`	Flag to notify whether or not the FEC is present in the FEC route table. Values include:
`LDP_TRUE`	FEC is present in the FEC route table
`LDP_FALSE`	FEC is present in the SNMP shadow entry table.
`shadow_entry`	SNMP FEC shadow entry. This value can be NULL.
`fec`	Pointer to the FEC.
`prefix`	Pointer to the prefix.

Output Parameters

None

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_snmp_api_set_path_vector_limit

This call sets the path vector limit of the entity.

Syntax

```
int
ldp_snmp_api_set_path_vector_limit (struct ldp_entity *entity, s_int32_t intval)
```

Input Parameters

`entity`	LDP entity
`intval`	Value of path vector limit

Output Parameters

`entity`	LDP entity

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_snmp_api_set_target_peer_addr_type

This call sets the targeted peer address type.

Syntax

```
int
ldp_snmp_api_set_target_peer_addr_type (struct ldp_entity *entity, s_int32_t intval)
```

Input Parameters

`entity`	LDP entity
`intval`	Value of path vector limit

Output Parameters

`entity`	LDP entity

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_snmp_api_set_target_peer_addr

This call sets the targeted peer address used by the LDP entity.

Syntax

```
int
ldp_snmp_api_set_target_peer_addr (struct ldp_entity *entity,
                                   struct pal_in4_addr *addr)
```

Input Parameters

`entity` LDP entity

`addr` Targeted peer address used by the LDP entity

Output Parameters

`entity` LDP entity

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_snmp_api_set_label_type

This call sets the label type used by the LDP entity.

Syntax

```
int
ldp_snmp_api_set_label_type (struct ldp_entity *entity, s_int32_t intval)
```

Input Parameters

`entity` LDP entity

`intval` Label type used by the LDP entity. Only the default value SNMP_AG_LBL_TYPE_generic (1) is allowed.

Output Parameters

`entity` LDP entity

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_snmp_api_set_entity_stor_type

This call sets the storage type of the LDP entity.

Syntax

```
int
ldp_snmp_api_set_label_type (struct ldp_entity *entity, s_int32_t intval)
```

Input Parameters

`entity`	LDP entity
`intval`	Storage type used by the LDP entity. Only the default value, SNMP_AG_STOR_volatile (2), is allowed.

Output Parameters

`entity`	LDP entity

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_snmp_api_set_fec_addr

This call sets the FEC address either in the SNMP shadow entry or given prefix.

Syntax

```
int
ldp_snmp_api_set_fec_addr_prefix_length (s_int32_t addrLength, struct prefix *p,
                                          struct snmp_mplsFecEntry_T
                                          *shadow_entry)
```

Input Parameters

`addrLength`	FEC address length
`p`	FEC prefix
`shadow_entry`	SNMP FEC shadow entry

Output Parameters

None

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_snmp_api_set_fec_stor_type

This call sets the FEC storage type for the SNMP FEC entry.

Syntax

```
int
ldp_snmp_api_set_fec_stor_type (s_int32_t storType,
                                struct snmp_mplsFecEntry_T *shadow_entry)
```

Input Parameters

`storType`	Value of the storage type
`shadow_entry`	SNMP FEC entry

Output Parameters

`shadow_entry`	SNMP FEC entry

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_snmp_fec_entry_add

This call adds the specified shadow table entry to the FEC table.

Syntax

```
int ldp_snmp_fec_entry_add (u_int32_t index,
                            struct snmp_mplsFecEntry_T *shadow_entry)
```

Input Parameters

`index`	FEC index
`shadow_entry`	SNMP FEC shadow entry.

Output Parameters

None

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_snmp_api_fill_lsp_fec_entry

This call fills the SNMP LSP FEC entry parameters.

Syntax

```
void
ldp_snmp_api_fill_lsp_fec_entry (snmp_mplsLdpLspFecEntry_T *entry_ptr,
                                 struct ldp_upstream *ucb)
```

Input Parameters

`ucb`	LDP upstream values

Output Parameters

`entry_ptr`	SNMP LDP LSP entry

Return Values

ldp_snmp_api_set_adv_mode

This call sets the advertisement mode of the LDP entity.

Syntax

```
int
ldp_snmp_api_set_adv_mode (struct ldp_entity *entity, s_int32_t intval)
```

Input Parameters

`entity`	LDP entity
`intval`	Advertisement mode

Output Parameters

`entity`	LDP entity

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_snmp_api_get_fec

This call gets the FEC for the given index.

Syntax

```
int
ldp_snmp_api_get_fec (struct ldp *ldp, s_int32_t indx, struct ldp_fec **fec,
                      struct prefix *prefix, int *fec_type,
                      struct prefix **prefixptr)
```

Input Parameters

`ldp`	LDP instance
`indx`	FEC index

Output Parameters

`fec`	LDP FEC structure
`fec_type`	FEC type; the permitted values are 1 and 2.
`prefix`	IP address in prefix format
`prefixptr`	FEC prefix pointer

Return Values

LDP_SUCCESS

LDP_FALSE

ldp_snmp_api_getnext_fec

This call gets the next FEC structure in the route table.

Syntax

```
struct ldp_fec *
ldp_snmp_api_getnext_fec (struct ldp *ldp, s_int32_t* indx,
                                 struct prefix* prefix, int *fec_type)
```

Input Parameters

`ldp`	LDP instance
`indx`	FEC index

Output Parameters

`indx`	Next FEC index
`prefix`	IP address in prefix format
`fec_type`	FEC type; the permitted values are 1 and 2

Return Values

LDP FEC

NULL

ldp_snmp_api_del_fec

This call deletes the LDP FEC with the given FEC index.

Syntax

```
int
ldp_snmp_api_del_fec (s_int32_t fec_indx)
```

Input Parameters

`fec_indx`	FEC index

Output Parameters

None

Return Values

LDP_TRUE

LDP_ERROR

ldp_snmp_api_conv_ldp_2_snmp

This call converts an LDP FEC entry to an FEC SNMP entry.

Syntax

```
void
ldp_snmp_api_conv_ldp_2_snmp (s_int32_t indx, struct prefix *prefix, int fec_type,
                              struct snmp_mplsFecEntry_T * entry_ptr,
                              struct ldp_fec *fec)
```

Input Parameters

`indx`	FEC index
`prefix`	FEC prefix

`fec_type`	FEC type; the permitted values are 1 and 2
`entry_ptr`	SNMP FEC entry
`fec`	FEC entry

Output Parameters

None

Return Values

None

snmp_convert_ldp_to_entry

This call converts the LDP entries to SNMP entries.

Syntax

```
int
snmp_convert_ldp_to_entry (struct ldp *ldp,
                           struct ldp_entity *entity,
                           struct ldp_id *EntityLdpId,
                           s_int32_t EntityIndex,
                           snmp_mplsLdpEntityEntry_T *entry_ptr)
```

Input Parameters

`ldp`	LDP instance
`entity`	LDP entity
`EntityLdpId`	LDP ID
`EntityIndex`	LDP entity index
`entry_ptr`	SNMP entity table

Output Parameters

`entity`	LDP entity

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_snmp_api_get_session_node

This call gets the LDP session node for the given peer LDP ID.

Syntax

```
struct listnode *
ldp_snmp_api_get_session_node (struct ldp *ldp, struct ldp_id *peer_id)
```

Input Parameters

`ldp`	LDP instance

`peer_id`	Peer LDP ID

Output Parameters

None

Return Values

LDP session node

NULL

ldp_snmp_get_session_node

This call gets the LDP session which matches the given entity ID and peer LDP ID.

Syntax

```
struct listnode *
ldp_snmp_get_session_node (struct ldp *ldp,
                           struct ldp_id *entity_id,
                           s_int32_t* entity_indx,
                           struct ldp_id *peer_id)
```

Input Parameters

`ldp`	LDP instance
`entity_id`	Entity LDP ID
`entity_indx`	LDP entity index
`peer_id`	Peer LDP ID

Output Parameters

None

Return Values

LDP session node

NULL

ldp_snmp_getnext_session_node

This call gets the next available LDP session.

Syntax

```
struct listnode*
ldp_snmp_getnext_session_node (struct ldp *ldp,
                               struct ldp_id *entity_id,
                               s_int32_t *entity_indx,
                               struct ldp_id *peer_id)
```

Input Parameters

`ldp`	LDP instance

`entity_id`	Entity LDP ID
`entity_indx`	LDP entity index
`peer_id`	Peer LDP ID

Output Parameters

None

Return Values

LDP session node

NULL

ldp_snmp_extract_session_data

This call gets the session-related data.

Syntax

```
int
ldp_snmp_extract_session_data (struct ldp *ldp,
                               struct ldp_session *session,
                               struct listnode *ln,
                               u_int32_t *state,
                               u_int32_t *role,
                               u_int32_t *proto_version,
                               u_int32_t *hold_time_rem,
                               u_int32_t *keep_alive_time,
                               u_int32_t *max_pdu_len,
                               u_int32_t *discont_time,
                               u_int32_t *last_state_change)
```

Input Parameters

`ldp`	LDP instance
`ldp`	LDP instance
`session`	LDP session
`ln`	Session adjacency

Output Parameters

`state`	State of the session
`role`	Role of the session
`proto_version`	Protocol version of the session
`hold_time_rem`	Remaining hold time value of the session
`keep_alive_time`	Keepalive time of the session
`max_pdu_len`	Maximum PDU length of the session
`discont_time`	Discontinue timer
`last_state_change`	Time when the last state change occurred

Return Values

LDP_SUCCESS

LDP_ERROR

ldp_snmp_extract_session_stat

This call gets the session statistics.

Syntax

```
void
ldp_snmp_extract_session_stat (struct ldp_session *session,
                               struct listnode *ln,
                               u_int32_t *mplsLdpSesStatsUnkMesTypeErrors,
                               u_int32_t *mplsLdpSesStatsUnkTlvErrors)
```

Input Parameters

`session`	LDP session
`ln`	Session adjacency

Output Parameters

`mplsLdpSesStatsUnkMesTypeErrors`

Number of unknown message types received

`mplsLdpSesStatsUnkTlvErrors`

Number of unknown TLVs received

Return Values

None

snmp_get_entity_staticstics

This call gets the LDP entity statistics.

Syntax

```
void snmp_get_entity_staticstics (struct ldp_entity * entity,
                             struct ldp_id * EntityLdpId,
                             s_int32_t EntityIndex,
                             snmp_mplsLdpEntityStatsEntry_T * entry_ptr);
```

Input Parameters

`entity`	LDP entity
`EntityLdpId`	Entity LDP ID
`EntityIndex`	Entity index

Output Parameters

`entry_ptr`	SNMP entity statistics table

Return Values

None

ldp_snmp_api_get_adjacency

This call gets the adjacency.

Syntax

```
static int
ldp_snmp_api_get_adjacency_node (struct ldp *ldp,
                                 struct ldp_id *peer_id,
                                 u_int32_t indx,
                                 struct listnode **ses_node,
                                 struct listnode **adj_node)
```

Input Parameters

`ldp`	LDP instance
`*peer_id`	LDP peer ID
`indx`	LDP index

Output Parameters

`ses_node`	LDP session
`adj_type`	Adjacent session type; it can be either Leaf_mplsLdpHelloAdjType_link or Leaf_mplsLdpHelloAdjType_targeted

Return Values

LDP_SUCCESS

LDP_FALSE

ldp_snmp_api_getnext_adjacency

This call gets the parameters of the next adjacency.

Syntax

```
int
ldp_snmp_api_getnext_adjacency (struct ldp *ldp,
                                struct ldp_id *entity_id, s_int32_t *entity_indx,
                                struct ldp_id *peer_id, u_int32_t *indx,
                                u_int32_t *hold_rem, s_int32_t *hold_time,
                                u_int32_t *adj_type)
```

Input Parameters

`ldp`	LDP instance
`entity_id`	LDP entity ID
`entity_indx`	LDP entity index
`*peer_id`	LDP peer ID

`indx`	LDP index

Output Parameters

`hold_rem`	Remaining hold time of the adjacent session
`hold_time`	Hold time of the adjacent session
`adj_type`	Adjacent session type; it can be either Leaf_mplsLdpHelloAdjType_link or Leaf_mplsLdpHelloAdjType_targeted

Return Values

LDP_SUCCESS

LDP_FALSE

ldp_snmp_api_get_peer

This call gets the peer parameters.

Syntax

```
int
ldp_snmp_api_get_peer (struct ldp *ldp,
                       struct ldp_id *entity_id,
                       s_int32_t entity_indx,
                       struct ldp_id *peer_id,
                       u_int32_t *adv_mode,
                       u_int32_t *path_vector_limit,
                       s_int32_t *peerTransAddrType,
                       struct pal_in4_addr *peerTransAddr)
```

Input Parameters

`ldp`	LDP instance
`entity_id`	LDP entity ID
`entity_indx`	LDP entity index
`peer_id`	LDP peer ID

Output Parameters

`adv_mode`	Advertisement mode of the peer. The value used is either Leaf_mplsLdpPeerLabelDistMethod_downstreamOnDemand or Leaf_mplsLdpPeerLabelDistMethod_downstreamUnsolicited
`path_vector_limit`	Path vector limit of the peer
`peerTransAddrType`	Transport address type of the peer; the value used is SNMP_AG_ADDR_TYPE_ipv4
`peerTransAddr`	Transport address of the peer

Return Values

LDP_SUCCESS

LDP_FALSE

ldp_snmp_api_getnext_peer

This call gets the next peer parameters.

Syntax

```
int
ldp_snmp_api_getnext_peer (struct ldp *ldp,
                           struct ldp_id *entity_id,
                           s_int32_t *entity_indx,
                           struct ldp_id *peer_id,
                           u_int32_t *adv_mode,
                           u_int32_t *path_vector_limit,
                           s_int32_t *peerTransAddrType,
                           struct pal_in4_addr *peerTransAddr)
```

Input Parameters

`ldp`	LDP instance

Output Parameters

`entity_id`	LDP entity ID
`entity_indx`	LDP entity index
`peer_id`	LDP peer ID
`adv_mode`	Advertisement mode of the peer; the values used is either Leaf_mplsLdpPeerLabelDistMethod_downstreamOnDemand or Leaf_mplsLdpPeerLabelDistMethod_downstreamUnsolicited
`path_vector_limit`	Path vector limit of the peer
`peerTransAddrType`	Transport address type of the peer; the value used is SNMP_AG_ADDR_TYPE_ipv4
`peerTransAddr`	Transport address of the peer

Return Values

LDP_SUCCESS

LDP_FALSE

ldp_snmp_api_get_peer_addr

This call gets the LDP peer address.

Syntax

```
int
ldp_snmp_api_get_peer_addr (struct ldp *ldp,
                            struct ldp_id *entity_id,
                            s_int32_t *entity_indx,
                            struct ldp_id *peer_id,
                            s_int32_t *peer_addr_indx,
                            struct pal_in4_addr *peer_adr)
```

Input Parameters

ldp	LDP instance
entity_id	LDP entity ID
entity_indx	LDP entity index
peer_id	LDP peer ID
peer_addr_indx	LDP peer address index

Output Parameters

peer_adr	LDP peer address

Return Values

LDP_SUCCESS

LDP_FALSE

ldp_snmp_api_getnext_peer_addr

This call gets the next LDP peer address.

Syntax

```
int
ldp_snmp_api_getnext_peer_addr (struct ldp *ldp,
                                struct ldp_id *entity_id,
                                s_int32_t *entity_indx,
                                struct ldp_id *peer_id,
                                s_int32_t *peer_addr_indx,
                                struct pal_in4_addr *peer_adr)
```

Input Parameters

ldp	LDP instance
entity_id	LDP entity ID
entity_indx	LDP entity index
peer_id	LDP peer ID
peer_addr_indx	LDP peer address index

Output Parameters

entity_id	LDP entity ID
entity_indx	LDP entity index
peer_id	LDP peer ID
peer_adr	LDP peer address

Return Values

LDP_SUCCESS

LDP_FALSE

ldp_snmp_api_get_adjacency_node

This call gets the adjacency node.

Syntax

```
static int
ldp_snmp_api_get_adjacency_node (struct ldp *ldp,
                                  struct ldp_id *peer_id,
                                  u_int32_t indx,
                                  struct listnode **ses_node,
                                  struct listnode **adj_node)
```

Input Parameters

`ldp`	LDP instance
`peer_id`	LDP peer ID
`indx`	LDP index

Output Parameters

`ses_node`	Session node
`adj_node`	Adjacency node

Return Values

LDP_SUCCESS

LDP_FALSE

ldp_api_set_admn_status

This call gets administrative status of this LDP Entity.

Syntax

```
static int
ldp_api_set_admn_status (entity, intval)
```

Input Parameters

`entity`	LDP entity
`intval`	XXXXX

Output Parameters

`entity`	LDP entity

Return Values

LDP_SUCCESS

LDP_FALSE

ldp_snmp_api_set_entity_tcp_port

This call gets TCP Port for LDP.

Syntax

```
static int
ldp_snmp_api_set_entity_tcp_port (entity, intval)
```

Input Parameters

`entity`	LDP entity
`intval`	TCP port number used by the LDP entity.

Output Parameters

`entity`	LDP entity

Return Values

LDP_SUCCESS

LDP_FALSE

ldp_snmp_api_set_entity_tcp_port

This call the UDP Discovery Port for LDP.

Syntax

```
static int
lldp_snmp_api_set_entity_tcp_port (entity, intval)
```

Input Parameters

`entity`	LDP entity
`intval`	TCP port number used by the LDP entity.

Output Parameters

`sentity`	LDP entity

Return Values

LDP_SUCCESS

LDP_FALSE

ldp_snmp_api_set_fec_addr

This call gets the value of this object is interpreted based on the value of the 'mplsFecAddrType' object.

Syntax

```
static int
ldp_snmp_api_set_fec_addr (&mplsFecAddr, shadow_entry, prefix);
```

Input Parameters

`fec`	Pointer to FEC
`shadow_entry`	SNMP FEC shadow entry. This value can be NULL.
`prefix`	FEC prefix

Output Parameters

`XXXX`	XXXXX

Return Values

LDP_SUCCESS

LDP_FALSE

Index