L2TP

Layer 2 Tunneling Protocol (L2TP) allows L2 frames to be tunneled over an IP network.

This document covers the kernel's L2TP subsystem. It documents kernel APIs for application developers who want to use the L2TP subsystem and it provides some technical details about the internal implementation which may be useful to kernel developers and maintainers.

Overview

The kernel's L2TP subsystem implements the datapath for L2TPv2 and L2TPv3. L2TPv2 is carried over UDP. L2TPv3 is carried over UDP or directly over IP (protocol 115).

The L2TP RFCs define two basic kinds of L2TP packets: control packets (the "control plane"), and data packets (the "data plane"). The kernel deals only with data packets. The more complex control packets are handled by user space.

An L2TP tunnel carries one or more L2TP sessions. Each tunnel is associated with a socket. Each session is associated with a virtual netdevice, e.g. `pppN`, `l2tpethN`, through which data frames pass to/from L2TP. Fields in the L2TP header identify the tunnel or session and whether it is a control or data packet. When tunnels and sessions are set up using the Linux kernel API, we're just setting up the L2TP data path. All aspects of the control protocol are to be handled by user space.

This split in responsibilities leads to a natural sequence of operations when establishing tunnels and sessions. The procedure looks like this:

1. Create a tunnel socket. Exchange L2TP control protocol messages with the peer over that socket in order to establish a tunnel.
2. Create a tunnel context in the kernel, using information obtained from the peer using the control protocol messages.
3. Exchange L2TP control protocol messages with the peer over the tunnel socket in order to establish a session.
4. Create a session context in the kernel using information obtained from the peer using the control protocol messages.

L2TP APIs

This section documents each userspace API of the L2TP subsystem.

Tunnel Sockets

L2TPv2 always uses UDP. L2TPv3 may use UDP or IP encapsulation.

To create a tunnel socket for use by L2TP, the standard POSIX socket API is used.

For example, for a tunnel using IPv4 addresses and UDP encapsulation:

```
int sockfd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
```

Or for a tunnel using IPv6 addresses and IP encapsulation:

```
int sockfd = socket(AF_INET6, SOCK_DGRAM, IPPROTO_L2TP);
```

UDP socket programming doesn't need to be covered here.

IPPROTO_L2TP is an IP protocol type implemented by the kernel's L2TP subsystem. The L2TPIP socket address is defined in struct sockaddr_l2tpip and struct sockaddr_l2tpip6 at include/uapi/linux/l2tp.h. The address includes the L2TP tunnel (connection) id. To use L2TP IP encapsulation, an L2TPv3 application should bind the L2TPIP socket using the locally assigned tunnel id. When the peer's tunnel id and IP address is known, a connect must be done.

If the L2TP application needs to handle L2TPv3 tunnel setup requests from peers using L2TPIP, it must open a dedicated L2TPIP socket to listen for those requests and bind the socket using tunnel id 0 since tunnel setup requests are addressed to tunnel id 0.

An L2TP tunnel and all of its sessions are automatically closed when its tunnel socket is closed.

Netlink API

L2TP applications use netlink to manage L2TP tunnel and session instances in the kernel. The L2TP netlink API is defined in include/uapi/linux/l2tp.h.

L2TP uses Generic Netlink (GENL). Several commands are defined: Create, Delete, Modify and Get for tunnel and session instances, e.g. `L2TP_CMD_TUNNEL_CREATE`. The API header lists the netlink attribute types that can be used with each command.

Tunnel and session instances are identified by a locally unique 32-bit id. L2TP tunnel ids are given by `L2TP_ATTR_CONN_ID` and `L2TP_ATTR_PEER_CONN_ID` attributes and L2TP session ids are given by `L2TP_ATTR_SESSION_ID` and

L2TP_ATTR_PEER_SESSION_ID attributes. If netlink is used to manage L2TPv2 tunnel and session instances, the L2TPv2 16-bit tunnel/session id is cast to a 32-bit value in these attributes.

In the L2TP_CMD_TUNNEL_CREATE command, L2TP_ATTR_FD tells the kernel the tunnel socket fd being used. If not specified, the kernel creates a kernel socket for the tunnel, using IP parameters set in L2TP_ATTR_IP[6]_SADDR, L2TP_ATTR_IP[6]_DADDR, L2TP_ATTR_UDP_SPORT, L2TP_ATTR_UDP_DPORT attributes. Kernel sockets are used to implement unmanaged L2TPv3 tunnels (iproute2's "ip l2tp" commands). If L2TP_ATTR_FD is given, it must be a socket fd that is already bound and connected. There is more information about unmanaged tunnels later in this document.

L2TP_CMD_TUNNEL_CREATE attributes:-

Attribute	Required	Use
CONN_ID	Y	Sets the tunnel (connection) id.
PEER_CONN_ID	Y	Sets the peer tunnel (connection) id.
PROTO_VERSION	Y	Protocol version. 2 or 3.
ENCAP_TYPE	Y	Encapsulation type: UDP or IP.
FD	N	Tunnel socket file descriptor.
UDP_CSUM	N	Enable IPv4 UDP checksums. Used only if FD is not set.
UDP_ZERO_CSUM6_TX	N	Zero IPv6 UDP checksum on transmit. Used only if FD is not set.
UDP_ZERO_CSUM6_RX	N	Zero IPv6 UDP checksum on receive. Used only if FD is not set.
IP_SADDR	N	IPv4 source address. Used only if FD is not set.
IP_DADDR	N	IPv4 destination address. Used only if FD is not set.
UDP_SPORT	N	UDP source port. Used only if FD is not set.
UDP_DPORT	N	UDP destination port. Used only if FD is not set.
IP6_SADDR	N	IPv6 source address. Used only if FD is not set.
IP6_DADDR	N	IPv6 destination address. Used only if FD is not set.
DEBUG	N	Debug flags.

L2TP_CMD_TUNNEL_DESTROY attributes:-

Attribute	Required	Use
CONN_ID	Y	Identifies the tunnel id to be destroyed.

L2TP_CMD_TUNNEL_MODIFY attributes:-

Attribute	Required	Use
CONN_ID	Y	Identifies the tunnel id to be modified.
DEBUG	N	Debug flags.

L2TP_CMD_TUNNEL_GET attributes:-

Attribute	Required	Use
CONN_ID	N	Identifies the tunnel id to be queried. Ignored in DUMP requests.

L2TP_CMD_SESSION_CREATE attributes:-

Attribute	Required	Use
CONN_ID	Y	The parent tunnel id.
SESSION_ID	Y	Sets the session id.
PEER_SESSION_ID	Y	Sets the parent session id.
PW_TYPE	Y	Sets the pseudowire type.
DEBUG	N	Debug flags.
RECV_SEQ	N	Enable rx data sequence numbers.
SEND_SEQ	N	Enable tx data sequence numbers.
LNS_MODE	N	Enable LNS mode (auto-enable data sequence numbers).
RECV_TIMEOUT	N	Timeout to wait when reordering received packets.
L2SPEC_TYPE	N	Sets layer2-specific-sublayer type (L2TPv3 only).
COOKIE	N	Sets optional cookie (L2TPv3 only).
PEER_COOKIE	N	Sets optional peer cookie (L2TPv3 only).
IFNAME	N	Sets interface name (L2TPv3 only).

For Ethernet session types, this will create an l2tpeth virtual interface which can then be configured as required. For PPP session types, a PPPoL2TP socket must also be opened and connected, mapping it onto the new session. This is covered in "PPPoL2TP Sockets" later.

L2TP_CMD_SESSION_DESTROY attributes:-

Attribute	Required	Use
CONN_ID	Y	Identifies the parent tunnel id of the session to be destroyed.
SESSION_ID	Y	Identifies the session id to be destroyed.

Attribute	Required	Use
IFNAME	N	Identifies the session by interface name. If set, this overrides any CONN_ID and SESSION_ID attributes. Currently supported for L2TPv3 Ethernet sessions only.

`L2TP_CMD_SESSION_MODIFY` attributes:-

Attribute	Required	Use
CONN_ID	Y	Identifies the parent tunnel id of the session to be modified.
SESSION_ID	Y	Identifies the session id to be modified.
IFNAME	N	Identifies the session by interface name. If set, this overrides any CONN_ID and SESSION_ID attributes. Currently supported for L2TPv3 Ethernet sessions only.
DEBUG	N	Debug flags.
RECV_SEQ	N	Enable rx data sequence numbers.
SEND_SEQ	N	Enable tx data sequence numbers.
LNS_MODE	N	Enable LNS mode (auto-enable data sequence numbers).
RECV_TIMEOUT	N	Timeout to wait when reordering received packets.

`L2TP_CMD_SESSION_GET` attributes:-

Attribute	Required	Use
CONN_ID	N	Identifies the tunnel id to be queried. Ignored for DUMP requests.
SESSION_ID	N	Identifies the session id to be queried. Ignored for DUMP requests.
IFNAME	N	Identifies the session by interface name. If set, this overrides any CONN_ID and SESSION_ID attributes. Ignored for DUMP requests. Currently supported for L2TPv3 Ethernet sessions only.

Application developers should refer to include/uapi/linux/l2tp.h for netlink command and attribute definitions.

Sample userspace code using libmnl:

- Open L2TP netlink socket:

```
struct nl_sock *nl_sock;
int l2tp_nl_family_id;

nl_sock = nl_socket_alloc();
genl_connect(nl_sock);
genl_id = genl_ctrl_resolve(nl_sock, L2TP_GENL_NAME);
```

- Create a tunnel:

```
struct nlmsghdr *nlh;
struct genlmsghdr *gnlh;

nlh = mnl_nlmsg_put_header(buf);
nlh->nlmsg_type = genl_id; /* assigned to genl socket */
nlh->nlmsg_flags = NLM_F_REQUEST | NLM_F_ACK;
nlh->nlmsg_seq = seq;

gnlh = mnl_nlmsg_put_extra_header(nlh, sizeof(*gnlh));
gnlh->cmd = L2TP_CMD_TUNNEL_CREATE;
gnlh->version = L2TP_GENL_VERSION;
gnlh->reserved = 0;

mnl_attr_put_u32(nlh, L2TP_ATTR_FD, tunl_sock_fd);
mnl_attr_put_u32(nlh, L2TP_ATTR_CONN_ID, tid);
mnl_attr_put_u32(nlh, L2TP_ATTR_PEER_CONN_ID, peer_tid);
mnl_attr_put_u8(nlh, L2TP_ATTR_PROTO_VERSION, protocol_version);
mnl_attr_put_u16(nlh, L2TP_ATTR_ENCAP_TYPE, encap);
```

- Create a session:

```
struct nlmsghdr *nlh;
struct genlmsghdr *gnlh;

nlh = mnl_nlmsg_put_header(buf);
nlh->nlmsg_type = genl_id; /* assigned to genl socket */
nlh->nlmsg_flags = NLM_F_REQUEST | NLM_F_ACK;
nlh->nlmsg_seq = seq;

gnlh = mnl_nlmsg_put_extra_header(nlh, sizeof(*gnlh));
gnlh->cmd = L2TP_CMD_SESSION_CREATE;
gnlh->version = L2TP_GENL_VERSION;
gnlh->reserved = 0;

mnl_attr_put_u32(nlh, L2TP_ATTR_CONN_ID, tid);
mnl_attr_put_u32(nlh, L2TP_ATTR_PEER_CONN_ID, peer_tid);
mnl_attr_put_u32(nlh, L2TP_ATTR_SESSION_ID, sid);
```

```
mnl_attr_put_u32(nlh, L2TP_ATTR_PEER_SESSION_ID, peer_sid);
mnl_attr_put_u16(nlh, L2TP_ATTR_PW_TYPE, pwtype);
/* there are other session options which can be set using netlink
 * attributes during session creation -- see l2tp.h
 */
```

- Delete a session:

```
struct nlmsghdr *nlh;
struct genlmsghdr *gnlh;

nlh = mnl_nlmsg_put_header(buf);
nlh->nlmsg_type = genl_id; /* assigned to genl socket */
nlh->nlmsg_flags = NLM_F_REQUEST | NLM_F_ACK;
nlh->nlmsg_seq = seq;

gnlh = mnl_nlmsg_put_extra_header(nlh, sizeof(*gnlh));
gnlh->cmd = L2TP_CMD_SESSION_DELETE;
gnlh->version = L2TP_GENL_VERSION;
gnlh->reserved = 0;

mnl_attr_put_u32(nlh, L2TP_ATTR_CONN_ID, tid);
mnl_attr_put_u32(nlh, L2TP_ATTR_SESSION_ID, sid);
```

- Delete a tunnel and all of its sessions (if any):

```
struct nlmsghdr *nlh;
struct genlmsghdr *gnlh;

nlh = mnl_nlmsg_put_header(buf);
nlh->nlmsg_type = genl_id; /* assigned to genl socket */
nlh->nlmsg_flags = NLM_F_REQUEST | NLM_F_ACK;
nlh->nlmsg_seq = seq;

gnlh = mnl_nlmsg_put_extra_header(nlh, sizeof(*gnlh));
gnlh->cmd = L2TP_CMD_TUNNEL_DELETE;
gnlh->version = L2TP_GENL_VERSION;
gnlh->reserved = 0;

mnl_attr_put_u32(nlh, L2TP_ATTR_CONN_ID, tid);
```

PPPoL2TP Session Socket API

For PPP session types, a PPPoL2TP socket must be opened and connected to the L2TP session.

When creating PPPoL2TP sockets, the application provides information to the kernel about the tunnel and session in a socket connect() call. Source and destination tunnel and session ids are provided, as well as the file descriptor of a UDP or L2TPIP socket. See struct pppol2tp_addr in include/linux/if_pppol2tp.h. For historical reasons, there are unfortunately slightly different address structures for L2TPv2/L2TPv3 IPv4/IPv6 tunnels and userspace must use the appropriate structure that matches the tunnel socket type.

Userspace may control behavior of the tunnel or session using setsockopt and ioctl on the PPPoX socket. The following socket options are supported:-

DEBUG	bitmask of debug message categories. See below.
SENDSEQ	• 0 => don't send packets with sequence numbers • 1 => send packets with sequence numbers
RECVSEQ	• 0 => receive packet sequence numbers are optional • 1 => drop receive packets without sequence numbers
LNSMODE	• 0 => act as LAC. • 1 => act as LNS.
REORDERTO	reorder timeout (in millisecs). If 0, don't try to reorder.

In addition to the standard PPP ioctls, a PPPIOCGL2TPSTATS is provided to retrieve tunnel and session statistics from the kernel using the PPPoX socket of the appropriate tunnel or session.

Sample userspace code:

- Create session PPPoX data socket:

```
struct sockaddr_pppol2tp sax;
int fd;

/* Note, the tunnel socket must be bound already, else it
 * will not be ready
 */
```

```
sax.sa_family = AF_PPPOX;
sax.sa_protocol = PX_PROTO_OL2TP;
sax.pppol2tp.fd = tunnel_fd;
sax.pppol2tp.addr.sin_addr.s_addr = addr->sin_addr.s_addr;
sax.pppol2tp.addr.sin_port = addr->sin_port;
sax.pppol2tp.addr.sin_family = AF_INET;
sax.pppol2tp.s_tunnel  = tunnel_id;
sax.pppol2tp.s_session = session_id;
sax.pppol2tp.d_tunnel  = peer_tunnel_id;
sax.pppol2tp.d_session = peer_session_id;

/* session_fd is the fd of the session's PPPoL2TP socket.
 * tunnel_fd is the fd of the tunnel UDP / L2TPIP socket.
 */
fd = connect(session_fd, (struct sockaddr *)&sax, sizeof(sax));
if (fd < 0 ) {
        return -errno;
}
return 0;
```

Old L2TPv2-only API

When L2TP was first added to the Linux kernel in 2.6.23, it implemented only L2TPv2 and did not include a netlink API. Instead, tunnel and session instances in the kernel were managed directly using only PPPoL2TP sockets. The PPPoL2TP socket is used as described in section "PPPoL2TP Session Socket API" but tunnel and session instances are automatically created on a connect() of the socket instead of being created by a separate netlink request:

- Tunnels are managed using a tunnel management socket which is a dedicated PPPoL2TP socket, connected to (invalid) session id 0. The L2TP tunnel instance is created when the PPPoL2TP tunnel management socket is connected and is destroyed when the socket is closed.
- Session instances are created in the kernel when a PPPoL2TP socket is connected to a non-zero session id. Session parameters are set using setsockopt. The L2TP session instance is destroyed when the socket is closed.

This API is still supported but its use is discouraged. Instead, new L2TPv2 applications should use netlink to first create the tunnel and session, then create a PPPoL2TP socket for the session.

Unmanaged L2TPv3 tunnels

The kernel L2TP subsystem also supports static (unmanaged) L2TPv3 tunnels. Unmanaged tunnels have no userspace tunnel socket, and exchange no control messages with the peer to set up the tunnel; the tunnel is configured manually at each end of the tunnel. All configuration is done using netlink. There is no need for an L2TP userspace application in this case -- the tunnel socket is created by the kernel and configured using parameters sent in the `L2TP_CMD_TUNNEL_CREATE` netlink request. The `ip` utility of `iproute2` has commands for managing static L2TPv3 tunnels; do `ip l2tp help` for more information.

Debugging

The L2TP subsystem offers a range of debugging interfaces through the debugfs filesystem.

To access these interfaces, the debugfs filesystem must first be mounted:

```
# mount -t debugfs debugfs /debug
```

Files under the l2tp directory can then be accessed, providing a summary of the current population of tunnel and session contexts existing in the kernel:

```
# cat /debug/l2tp/tunnels
```

The debugfs files should not be used by applications to obtain L2TP state information because the file format is subject to change. It is implemented to provide extra debug information to help diagnose problems. Applications should instead use the netlink API.

In addition the L2TP subsystem implements tracepoints using the standard kernel event tracing API. The available L2TP events can be reviewed as follows:

```
# find /debug/tracing/events/l2tp
```

Finally, /proc/net/pppol2tp is also provided for backwards compatibility with the original pppol2tp code. It lists information about L2TPv2 tunnels and sessions only. Its use is discouraged.

Internal Implementation

This section is for kernel developers and maintainers.

Sockets

UDP sockets are implemented by the networking core. When an L2TP tunnel is created using a UDP socket, the socket is set up as

an encapsulated UDP socket by setting encap_rcv and encap_destroy callbacks on the UDP socket. l2tp_udp_encap_recv is called when packets are received on the socket. l2tp_udp_encap_destroy is called when userspace closes the socket.

L2TPIP sockets are implemented in net/l2tp/l2tp_ip.c and net/l2tp/l2tp_ip6.c.

Tunnels

The kernel keeps a struct l2tp_tunnel context per L2TP tunnel. The l2tp_tunnel is always associated with a UDP or L2TP/IP socket and keeps a list of sessions in the tunnel. When a tunnel is first registered with L2TP core, the reference count on the socket is increased. This ensures that the socket cannot be removed while L2TP's data structures reference it.

Tunnels are identified by a unique tunnel id. The id is 16-bit for L2TPv2 and 32-bit for L2TPv3. Internally, the id is stored as a 32-bit value.

Tunnels are kept in a per-net list, indexed by tunnel id. The tunnel id namespace is shared by L2TPv2 and L2TPv3. The tunnel context can be derived from the socket's sk_user_data.

Handling tunnel socket close is perhaps the most tricky part of the L2TP implementation. If userspace closes a tunnel socket, the L2TP tunnel and all of its sessions must be closed and destroyed. Since the tunnel context holds a ref on the tunnel socket, the socket's sk_destruct won't be called until the tunnel sock_put's its socket. For UDP sockets, when userspace closes the tunnel socket, the socket's encap_destroy handler is invoked, which L2TP uses to initiate its tunnel close actions. For L2TPIP sockets, the socket's close handler initiates the same tunnel close actions. All sessions are first closed. Each session drops its tunnel ref. When the tunnel ref reaches zero, the tunnel puts its socket ref. When the socket is eventually destroyed, it's sk_destruct finally frees the L2TP tunnel context.

Sessions

The kernel keeps a struct l2tp_session context for each session. Each session has private data which is used for data specific to the session type. With L2TPv2, the session always carries PPP traffic. With L2TPv3, the session can carry Ethernet frames (Ethernet pseudowire) or other data types such as PPP, ATM, HDLC or Frame Relay. Linux currently implements only Ethernet and PPP session types.

Some L2TP session types also have a socket (PPP pseudowires) while others do not (Ethernet pseudowires). We can't therefore use the socket reference count as the reference count for session contexts. The L2TP implementation therefore has its own internal reference counts on the session contexts.

Like tunnels, L2TP sessions are identified by a unique session id. Just as with tunnel ids, the session id is 16-bit for L2TPv2 and 32-bit for L2TPv3. Internally, the id is stored as a 32-bit value.

Sessions hold a ref on their parent tunnel to ensure that the tunnel stays extant while one or more sessions references it.

Sessions are kept in a per-tunnel list, indexed by session id. L2TPv3 sessions are also kept in a per-net list indexed by session id, because L2TPv3 session ids are unique across all tunnels and L2TPv3 data packets do not contain a tunnel id in the header. This list is therefore needed to find the session context associated with a received data packet when the tunnel context cannot be derived from the tunnel socket.

Although the L2TPv3 RFC specifies that L2TPv3 session ids are not scoped by the tunnel, the kernel does not police this for L2TPv3 UDP tunnels and does not add sessions of L2TPv3 UDP tunnels into the per-net session list. In the UDP receive code, we must trust that the tunnel can be identified using the tunnel socket's sk_user_data and lookup the session in the tunnel's session list instead of the per-net session list.

PPP

net/l2tp/l2tp_ppp.c implements the PPPoL2TP socket family. Each PPP session has a PPPoL2TP socket.

The PPPoL2TP socket's sk_user_data references the l2tp_session.

Userspace sends and receives PPP packets over L2TP using a PPPoL2TP socket. Only PPP control frames pass over this socket: PPP data packets are handled entirely by the kernel, passing between the L2TP session and its associated `pppN` netdev through the PPP channel interface of the kernel PPP subsystem.

The L2TP PPP implementation handles the closing of a PPPoL2TP socket by closing its corresponding L2TP session. This is complicated because it must consider racing with netlink session create/destroy requests and pppol2tp_connect trying to reconnect with a session that is in the process of being closed. Unlike tunnels, PPP sessions do not hold a ref on their associated socket, so code must be careful to sock_hold the socket where necessary. For all the details, see commit 3d609342cc04129ff7568e19316ce3d7451a27e8.

Ethernet

net/l2tp/l2tp_eth.c implements L2TPv3 Ethernet pseudowires. It manages a netdev for each session.

L2TP Ethernet sessions are created and destroyed by netlink request, or are destroyed when the tunnel is destroyed. Unlike PPP sessions, Ethernet sessions do not have an associated socket.

Miscellaneous

RFCs

The kernel code implements the datapath features specified in the following RFCs:

RFC2661	L2TPv2	https://tools.ietf.org/html/rfc2661
RFC3931	L2TPv3	https://tools.ietf.org/html/rfc3931
RFC4719	L2TPv3 Ethernet	https://tools.ietf.org/html/rfc4719

Implementations

A number of open source applications use the L2TP kernel subsystem:

iproute2	https://github.com/shemminger/iproute2
go-l2tp	https://github.com/katalix/go-l2tp
tunneldigger	https://github.com/wlanslovenija/tunneldigger
xl2tpd	https://github.com/xelerance/xl2tpd

Limitations

The current implementation has a number of limitations:

1. Multiple UDP sockets with the same 5-tuple address cannot be used. The kernel's tunnel context is identified using private data associated with the socket so it is important that each socket is uniquely identified by its address.
2. Interfacing with openvswitch is not yet implemented. It may be useful to map OVS Ethernet and VLAN ports into L2TPv3 tunnels.
3. VLAN pseudowires are implemented using an `l2tpethN` interface configured with a VLAN sub-interface. Since L2TPv3 VLAN pseudowires carry one and only one VLAN, it may be better to use a single netdevice rather than an `l2tpethN` and `l2tpethN`:M pair per VLAN session. The netlink attribute `L2TP_ATTR_VLAN_ID` was added for this, but it was never implemented.

Testing

Unmanaged L2TPv3 Ethernet features are tested by the kernel's built-in selftests. See tools/testing/selftests/net/l2tp.sh.

Another test suite, l2tp-ktest, covers all of the L2TP APIs and tunnel/session types. This may be integrated into the kernel's built-in L2TP selftests in the future.