This document describes how to use the kernel's L2TP drivers to provide L2TP functionality. L2TP is a protocol that tunnels one or more sessions over an IP tunnel. It is commonly used for VPNs (L2TP/IPSec) and by ISPs to tunnel subscriber PPP sessions over an IP network infrastructure. With L2TPv3, it is also useful as a Layer-2 tunneling infrastructure.

Features

L2TPv2 (PPP over L2TP (UDP tunnels)).
L2TPv3 ethernet pseudowires.
L2TPv3 PPP pseudowires.
L2TPv3 IP encapsulation.
Netlink sockets for L2TPv3 configuration management.

History

The original pppol2tp driver was introduced in 2.6.23 and provided L2TPv2 functionality (rfc2661). L2TPv2 is used to tunnel one or more PPP sessions over a UDP tunnel.

L2TPv3 (rfc3931) changes the protocol to allow different frame types to be passed over an L2TP tunnel by moving the PPP-specific parts of the protocol out of the core L2TP packet headers. Each frame type is known as a pseudowire type. Ethernet, PPP, HDLC, Frame Relay and ATM pseudowires for L2TP are defined in separate RFC standards. Another change for L2TPv3 is that it can be carried directly over IP with no UDP header (UDP is optional). It is also possible to create static unmanaged L2TPv3 tunnels manually without a control protocol (userspace daemon) to manage them.

To support L2TPv3, the original pppol2tp driver was split up to separate the L2TP and PPP functionality. Existing L2TPv2 userspace apps should be unaffected as the original pppol2tp sockets API is retained. L2TPv3, however, uses netlink to manage L2TPv3 tunnels and sessions.

Design

The L2TP protocol separates control and data frames. The L2TP kernel drivers handle only L2TP data frames; control frames are always handled by userspace. L2TP control frames carry messages between L2TP clients/servers and are used to setup / teardown tunnels and sessions. An L2TP client or server is implemented in userspace.

Each L2TP tunnel is implemented using a UDP or L2TPIP socket; L2TPIP provides L2TPv3 IP encapsulation (no UDP) and is implemented using a new l2tpip socket family. The tunnel socket is typically created by userspace, though for unmanaged L2TPv3 tunnels, the socket can also be created by the kernel. Each L2TP session (pseudowire) gets a network interface instance. In the case of PPP, these interfaces are created indirectly by pppd using a pppol2tp socket. In the case of ethernet, the netdevice is created upon a netlink request to create an L2TPv3

ethernet pseudowire.

For PPP, the PPPoL2TP driver, net/l2tp/l2tp_ppp.c, provides a mechanism by which PPP frames carried through an L2TP session are passed through the kernel's PPP subsystem. The standard PPP daemon, pppd, handles all PPP interaction with the peer. PPP network interfaces are created for each local PPP endpoint. The kernel's PPP subsystem arranges for PPP control frames to be delivered to pppd, while data frames are forwarded as usual.

For ethernet, the L2TPETH driver, net/l2tp/l2tp_eth.c, implements a netdevice driver, managing virtual ethernet devices, one per pseudowire. These interfaces can be managed using standard Linux tools such as "ip" and "ifconfig". If only IP frames are passed over the tunnel, the interface can be given an IP addresses of itself and its peer. If non-IP frames are to be passed over the tunnel, the interface can be added to a bridge using brctl. All L2TP datapath protocol functions are handled by the L2TP core driver.

Each tunnel and session within a tunnel is assigned a unique tunnel_id and session_id. These ids are carried in the L2TP header of every control and data packet. (Actually, in L2TPv3, the tunnel_id isn't present in data frames - it is inferred from the IP connection on which the packet was received.) The L2TP driver uses the ids to lookup internal tunnel and/or session contexts to determine how to handle the packet. Zero tunnel / session ids are treated specially - zero ids are never assigned to tunnels or sessions in the network. In the driver, the tunnel context keeps a reference to the tunnel UDP or L2TPIP socket. The session context holds data that lets the driver interface to the kernel's network frame type subsystems, i.e. PPP, ethernet.

Userspace Programming
=====================

For L2TPv2, there are a number of requirements on the userspace L2TP daemon in order to use the pppol2tp driver.

1. Use a UDP socket per tunnel.

2. Create a single PPPoL2TP socket per tunnel bound to a special null session id. This is used only for communicating with the driver but must remain open while the tunnel is active. Opening this tunnel management socket causes the driver to mark the tunnel socket as an L2TP UDP encapsulation socket and flags it for use by the referenced tunnel id. This hooks up the UDP receive path via udp_encap_rcv() in net/ipv4/udp.c. PPP data frames are never passed in this special PPPoX socket.

3. Create a PPPoL2TP socket per L2TP session. This is typically done by starting pppd with the pppol2tp plugin and appropriate arguments. A PPPoL2TP tunnel management socket (Step 2) must be created before the first PPPoL2TP session socket is created.

When creating PPPoL2TP sockets, the application provides information to the driver about the socket in a socket connect() call. Source and destination tunnel and session ids are provided, as well as the file

descriptor of a UDP socket. See struct pppol2tp_addr in include/linux/if_ppp.h. Note that zero tunnel / session ids are treated specially. When creating the per-tunnel PPPoL2TP management socket in Step 2 above, zero source and destination session ids are specified, which tells the driver to prepare the supplied UDP file descriptor for use as an L2TP tunnel socket.

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.
```

Only the DEBUG option is supported by the special tunnel management PPPoX socket.

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.

For L2TPv3, userspace must use the netlink API defined in include/linux/l2tp.h to manage tunnel and session contexts. The general procedure to create a new L2TP tunnel with one session is:-

1. Open a GENL socket using L2TP_GENL_NAME for configuring the kernel using netlink.

2. Create a UDP or L2TPIP socket for the tunnel.

3. Create a new L2TP tunnel using a L2TP_CMD_TUNNEL_CREATE request. Set attributes according to desired tunnel parameters, referencing the UDP or L2TPIP socket created in the previous step.

4. Create a new L2TP session in the tunnel using a L2TP_CMD_SESSION_CREATE request.

The tunnel and all of its sessions are closed when the tunnel socket is closed. The netlink API may also be used to delete sessions and tunnels. Configuration and status info may be set or read using netlink.

The L2TP driver also supports static (unmanaged) L2TPv3 tunnels. These are where there is no L2TP control message exchange with the peer to setup the tunnel; the tunnel is configured manually at each end of the tunnel. 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 driver supports a flexible debug scheme where kernel trace
messages may be optionally enabled per tunnel and per session. Care is
needed when debugging a live system since the messages are not
rate-limited and a busy system could be swamped. Userspace uses
setsockopt on the PPPoX socket to set a debug mask.

The following debug mask bits are available:

```
PPPOL2TP_MSG_DEBUG    verbose debug (if compiled in)
PPPOL2TP_MSG_CONTROL  userspace - kernel interface
PPPOL2TP_MSG_SEQ      sequence numbers handling
PPPOL2TP_MSG_DATA     data packets
```

If enabled, files under a l2tp debugfs directory can be used to dump
kernel state about L2TP tunnels and sessions. To access it, the
debugfs filesystem must first be mounted.

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

Files under the l2tp directory can then be accessed.

```
# 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.) Users should use the netlink API.

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

Unmanaged L2TPv3 Tunnels
========================

Some commercial L2TP products support unmanaged L2TPv3 ethernet
tunnels, where there is no L2TP control protocol; tunnels are
configured at each side manually. New commands are available in
iproute2's ip utility to support this.

To create an L2TPv3 ethernet pseudowire between local host 192.168.1.1
and peer 192.168.1.2, using IP addresses 10.5.1.1 and 10.5.1.2 for the
tunnel endpoints:-

```
# modprobe l2tp_eth
# modprobe l2tp_netlink

# ip l2tp add tunnel tunnel_id 1 peer_tunnel_id 1 udp_sport 5000 \
  udp_dport 5000 encap udp local 192.168.1.1 remote 192.168.1.2
# ip l2tp add session tunnel_id 1 session_id 1 peer_session_id 1
# ifconfig -a
# ip addr add 10.5.1.2/32 peer 10.5.1.1/32 dev l2tpeth0
# ifconfig l2tpeth0 up
```

Choose IP addresses to be the address of a local IP interface and that of the remote system. The IP addresses of the l2tpeth0 interface can be anything suitable.

Repeat the above at the peer, with ports, tunnel/session ids and IP addresses reversed. The tunnel and session IDs can be any non-zero 32-bit number, but the values must be reversed at the peer.

Host 1	Host2
udp_sport=5000	udp_sport=5001
udp_dport=5001	udp_dport=5000
tunnel_id=42	tunnel_id=45
peer_tunnel_id=45	peer_tunnel_id=42
session_id=128	session_id=5196755
peer_session_id=5196755	peer_session_id=128

When done at both ends of the tunnel, it should be possible to send data over the network. e.g.

```
# ping 10.5.1.1
```

Sample Userspace Code
=====================

1. Create tunnel management PPPoX socket

```
        kernel_fd = socket(AF_PPPOX, SOCK_DGRAM, PX_PROTO_OL2TP);
        if (kernel_fd >= 0) {
                struct sockaddr_pppol2tp sax;
                struct sockaddr_in const *peer_addr;

                peer_addr = l2tp_tunnel_get_peer_addr(tunnel);
                memset(&sax, 0, sizeof(sax));
                sax.sa_family = AF_PPPOX;
                sax.sa_protocol = PX_PROTO_OL2TP;
                sax.pppol2tp.fd = udp_fd;       /* fd of tunnel UDP socket */
                sax.pppol2tp.addr.sin_addr.s_addr = peer_addr->sin_addr.s_addr;
                sax.pppol2tp.addr.sin_port = peer_addr->sin_port;
                sax.pppol2tp.addr.sin_family = AF_INET;
                sax.pppol2tp.s_tunnel = tunnel_id;
                sax.pppol2tp.s_session = 0;     /* special case: mgmt socket */
                sax.pppol2tp.d_tunnel = 0;
                sax.pppol2tp.d_session = 0;     /* special case: mgmt socket */

                if(connect(kernel_fd, (struct sockaddr *)&sax, sizeof(sax) ) < 0
) {
                        perror("connect failed");
                        result = -errno;
                        goto err;
                }
        }
```

2. Create session PPPoX data socket

```
        struct sockaddr_pppol2tp sax;
        int fd;

        /* Note, the target 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 socket.
         */
        fd = connect(session_fd, (struct sockaddr *)&sax, sizeof(sax));
        if (fd < 0 )    {
                return -errno;
        }
        return 0;
```

Internal Implementation
=======================

The driver keeps a struct l2tp_tunnel context per L2TP tunnel and a struct l2tp_session context for each session. The l2tp_tunnel is always associated with a UDP or L2TP/IP socket and keeps a list of sessions in the tunnel. The l2tp_session context keeps kernel state about the session. It has private data which is used for data specific to the session type. With L2TPv2, the session always carried PPP traffic. With L2TPv3, the session can also carry ethernet frames (ethernet pseudowire) or other data types such as ATM, HDLC or Frame Relay.

When a tunnel is first opened, the reference count on the socket is increased using sock_hold(). This ensures that the kernel socket cannot be removed while L2TP's data structures reference it.

Some L2TP sessions also have a socket (PPP pseudowires) while others do not (ethernet pseudowires). We can't 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.

To Do
=====

Add L2TP tunnel switching support. This would route tunneled traffic from one L2TP tunnel into another. Specified in http://tools.ietf.org/html/draft-ietf-l2tpext-tunnel-switching-08

Add L2TPv3 VLAN pseudowire support.

Add L2TPv3 IP pseudowire support.

Add L2TPv3 ATM pseudowire support.

Miscellaneous
==========

The L2TP drivers were developed as part of the OpenL2TP project by
Katalix Systems Ltd. OpenL2TP is a full-featured L2TP client / server,
designed from the ground up to have the L2TP datapath in the
kernel. The project also implemented the pppol2tp plugin for pppd
which allows pppd to use the kernel driver. Details can be found at
http://www.openl2tp.org.