A PPPoE Implementation for Linux

David F. Skoll
Roaring Penguin Software Inc.
dfs@roaringpenguin.com, http://www.roaringpenguin.com

Abstract

Many DSL service providers use PPPoE for residential broadband Internet access. This paper briefly describes the PPPoE protocol, presents strategies for implementing it under Linux and describes in detail a user-space implementation of a PPPoE client.

1 Introduction

Many Internet service providers are using the Point-to-Point Protocol over Ethernet (PPPoE) to provide residential Digital Subscriber Link (DSL) broadband Internet access. Most ISP's do not support Linux and supply PPPoE clients only for Windows and Mac OS. This paper describes a PPPoE client for Linux.

The paper is organized as follows: Section 2 provides an introduction to the PPPoE protocol and a brief discussion of why it is used. Section 3 describes the various strategies which can be used to implement PPPoE under Linux. Section 4 describes `rp-pppoe`, a particular user-space implementation of PPPoE. Section 5 describes additional PPPoE-related tools and ports to non-Linux systems. Finally, Section 6 contains some concluding remarks.

2 The PPPoE Protocol

PPPoE is a protocol for encapsulating PPP frames in Ethernet frames[1]. PPP is a data-link-level protocol typically used to encapsulate network-level packets over an asynchronous serial line. This mode of usage is called *asynchronous PPP*.

2.1 Asynchronous PPP

Asynchronous PPP uses *byte stuffing* to mark frame boundaries[2]. The special byte 0x7E called a *flag sequence*. The start of a frame is marked by a flag sequence followed by bytes 0xFF and 0x03. Next, a two-byte *protocol* field identifies the network-layer protocol. Next, the network layer data is sent, followed by a two- or four-byte *frame check sequence.*

To make recognition of frame boundaries unambiguous, if the flag sequence appears inside a frame, it is *escaped* by transmitting the byte 0x7D (the *escape sequence*) followed by the original byte XOR'd with 0x20. Naturally, the escape sequence itself must be escaped, and is transmitted as 0x7D, 0x5D. Other byte values may be escaped.

The receiver discards the extra escape sequences to reconstruct the original PPP frame.

2.2 Synchronous PPP

Asynchronous serial links cannot inherently mark frame boundaries, so byte-stuffing (or some equivalent) is required. For data-link types which naturally mark frame boundaries, no byte-stuffing is needed. Since Ethernet has natural frame boundaries, PPP frames can be encapsulated in Ethernet frames without any byte stuffing.

The PPPoE Session Frame consists of a PPP frame inside an Ethernet frame, with six bytes of PPPoE information. The format of this PPPoE information will be described in Section 2.4.

2.3 PPPoE Discovery Phase

A *PPPoE session* consists of two PPP peers communicating over Ethernet[3]. Each peer knows the MAC address of the other peer. In addition, a *session number* is used to uniquely identify a particular PPPoE session between two peers.

While PPP is a peer-to-peer protocol, PPPoE is initially a client-server protocol. The client (usually a personal computer) searches for a PPPoE server (called an *access concentrator*) and obtains the access concentrator's MAC address and a session number. The process of setting up a PPPoE session is called *discovery*. PPPoE discovery uses special Ethernet frames with their own Ethernet frame type (0x8863).

To initiate discovery, the PPPoE client sends a PPPoE Active Discovery Initiation (PADI) frame.

This frame is sent to the broadcast Ethernet address (FF:FF:FF:FF:FF:FF) and may specify a particular "service name" which the client is interested in.

When an access concentrator receives a PADI frame, it responds with a PPPoE Active Discovery Offer (PADO) frame, if it is willing to set up a session with the client. The destination Ethernet address of the PADO frame is the unicast Ethernet address of the client who sent the PADI.

In general, there can be more than one access concentrator within broadcast range of the client. The client therefore collects PADO responses and picks one with which it would like to start a session. It sends a PPPoE Active Discovery Request (PADR) frame to the unicast Ethernet address of the access concentrator.

If the access concentrator agrees to set up a session with the client, it allocates resources to set up a PPP session and assigns a session number. It sends this number back to the client in a PPPoE Active Discovery Session-confirmation (PADS) frame. When the client receives the PADS frame, it knows the access concentrator's Ethernet address and the session number. It allocates resources to set up a PPP session.

2.4 PPPoE Session Phase

Once each side knows the other's Ethernet address and the session number, the PPP session can begin. PPP frames are encapsulated in PPPoE session frames, which have Ethernet frame type 0x8864. A PPPoE session frame is shown in Figure 1.

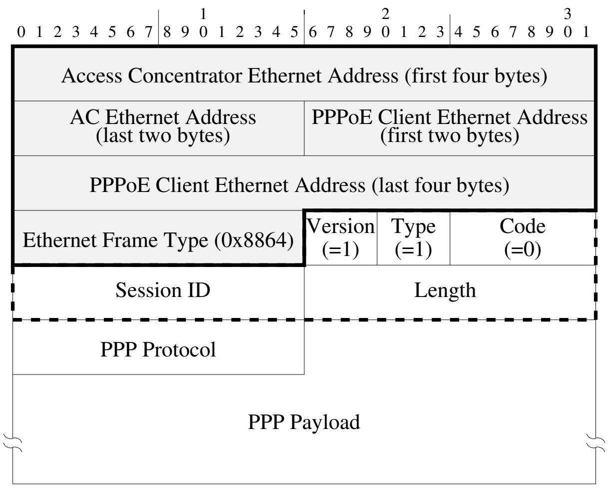

Figure 1: PPPoE Session Frame

In the session frame, the four-bit fields **Version** and **Type** are set to 1. The **Code** field is used in the discovery phase to identify the packet type, but is always set to zero in the session phase. The **Session ID** field is the session ID assigned during discovery. The **Length** field is the length of the PPP payload, not including the Ethernet or PPPoE headers.

The PPP data begins with the PPP protocol field. Note that no flag sequences are included. The PPP data is not byte-stuffed with the escape sequence, and does not include the final PPP frame-check sequence. (The FCS is omitted because Ethernet frames have their own frame check sequence, and there is no point in duplicating it.)

2.5 Why PPPoE?

PPPoE has many advantages for DSL service providers, and practically none for DSL consumers.

- Because PPPoE sessions are really just PPP sessions, IP addresses can be very dynamic. There is no possibility to hold on to a fixed IP addresses by renewing a DHCP lease frequently. Service providers can ensure that your assigned IP address is changed each time you connect.

- Because PPPoE creates the concept of a "session" over Ethernet, service providers can charge based on connect time. This allows them to discourage permanent connections and over-subscribe their IP address pool.

- Because PPP sessions almost always require authentication, DSL service providers can bill the correct client regardless of where he connects from (as dial-up ISP's can now.)

In theory, PPPoE offers the following advantages to end users. In practice, these advantages are either negligible or not implemented by the service provider.

- PPPoE can encapsulate non-IP protocols. Any protocol which can be encapsulated by PPP can be sent via PPPoE.

- Service providers can enter into agreements with large organizations to authenticate users and provide dedicated sessions behind the organizations' firewall (for employees who need remote access, for example.)

No ISP that I'm aware of supports non-IP protocols over PPPoE, and access through a firewall is better achieved with SSH or IPSec.

3 Implementing PPPoE under Linux

There are many possibilities for implementing a PPPoE client under Linux. The following are the three reasonable strategies:

1. Implement both the PPPoE discovery and session phases in a user-space program.

2. Implement PPPoE discovery in a user-space program and PPPoE session in the kernel.

3. Implement both the PPPoE discovery and session phases in the kernel.

We can dispose of the third strategy right away. PPPoE discovery is not speed-critical and consists of code which is seldom used. There's no point in bloating the kernel with discovery code.

The choice then boils down to including PPPoE session code in the kernel or in a user-space program. Again, the choice is clear: The kernel itself should handle the PPPoE session. Executing user code for each PPPoE frame is very inefficient.

As of this writing, PPPoE support exists in the experimental 2.3 kernels and is expected to be included in the final 2.4 kernels. In addition, there are kernel patches to add PPPoE support to 2.0 and 2.2 kernels.

Although the kernel is clearly the right place for PPPoE session support, I have implemented PPPoE in a user-space program. While this may be repugnant to kernel hackers, there are some advantages to a user-space program:

- A user-space program is easy to write and debug.

- A user-space program is relatively portable. In fact, `rp-pppoe` has already been ported to NetBSD (by Geoff Mottram and Yannis Sismanis.)

- Many Linux users are novices, and the existing kernel patches are not easy for them to apply and configure. A simple RPM or DEB package with a helpful configuration shell script is much more palatable for new users.

- Many Linux users do not want to modify their kernels. If they ever need to upgrade the kernel for security reasons, they do not want to remember to have to compile in PPPoE support. (This reason will disappear once standard kernels include PPPoE.)

- Most residential DSL connections are slow (2.2Mb/s or less) and even a user-space client can keep up with this on all but the oldest hardware.

I therefore view the user-space client as a convenient but temporary measure until stable kernels include PPPoE support. Even after the standard kernel supports PPPoE, most of the user-space code involves the discovery phase, and can be re-used in newer kernels. The session code is very simple and there's no great loss in discarding it.

4 The `rp-pppoe` User-Space PPPoE Client

`rp-pppoe` is a free (under the GNU General Public License) user-space PPPoE implementation for Linux. Figure 2 illustrates the operation of `rp-pppoe`.

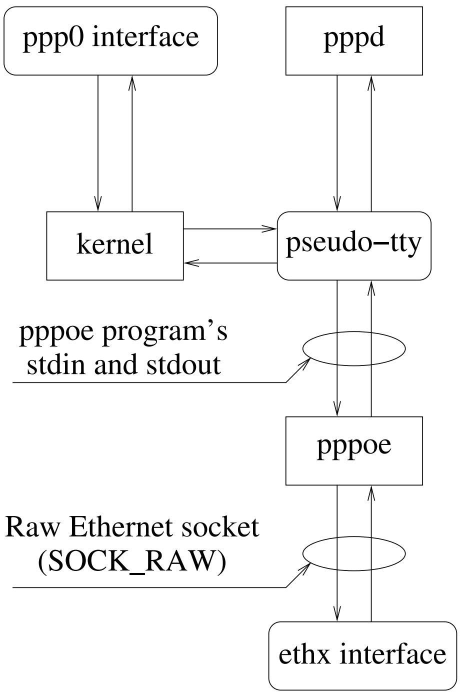

Figure 2: `rp-pppoe` Configuration

4.1 The Pseudo-TTY

The `pppd` program and the Linux kernel expect to transmit PPP frames over a TTY device. Luckily, UNIX (and Linux) support the concept of a *pseudo-tty*. This is a device which "looks" like a TTY, but instead of being connected to a physical terminal, it is connected to a UNIX process. Whenever something writes to the pseudo-tty, the data appears on the standard input of the back-end process. Whenever the back-end process writes to its standard output, the data may be read from the pseudo-tty.

Even more luckily, recent versions of `pppd` (2.3.7 and newer) support a `pty` option. This option automatically starts the back-end process and performs all the mundane operations required to connect it to a pseudo-tty.

So to start the PPPoE link, you start `pppd` with the appropriate `pty` option, which runs the `pppoe` executable connected to the pseudo-tty.

4.2 The Discovery Phase

Once `pppoe` begins executing, it starts PPPoE discovery. It creates a raw Ethernet socket. This special socket allows user-space programs to transmit and receive raw Ethernet frames.

`pppoe` constructs and transmits a PADI frame, and waits for PADO frames. When a PADO frame arrives (if it meets criteria specified on the `pppoe` command line), `pppoe` transmits a PADR frame. Once it receives a PADS frame, it records the session ID and moves to the session phase.

Note that Linux raw sockets perform only limited filtering on Ethernet frames. They are not nearly as flexible as the Berkeley Packet Filter found on BSD systems. Luckily, for PPPoE, filtering only on the Ethernet frame type is acceptable, and Linux raw sockets can perform this level of filtering. (We want to filter out non-PPPoE frames in the kernel; otherwise, non-PPPoE traffic could consume huge amounts of CPU time as `pppoe` is scheduled in to read the frame.)

4.3 The Session Phase

Once the session phase begins, `pppoe` reads asynchronously-framed PPP data on standard input, and writes it to standard output. Let's trace these operations.

When a frame is transmitted over the PPP interface, `pppoe`'s standard input becomes readable. `pppoe` reads from standard input and collects data until it has assembled an entire PPP frame.

Note that `pppoe` must keep a small state machine to record where it is in the PPP frame assembly. There's no guarantee that it will read an entire PPP frame in one chunk, or that it won't read more than one frame. This is because UNIX write operations do not preserve write boundaries; if you write one byte to a pseudo-tty followed by three bytes, the back-end process may read all four bytes at once, depending on scheduling.

During PPP frame assembly, `pppoe` removes escape sequences and "de-stuffs" the frame. This converts the asynchronous PPP framing into a synchronous PPP frame.

Once the PPP frame is assembled, PPPoE headers are added and the frame is transmitted over the raw socket. (Actually, most of the PPPoE headers are constant for a given session, so the PPP frame is simply assembled into a buffer right after the constant PPPoE header portions.)

When an incoming PPPoE frame is received by the Ethernet card, `pppoe`'s raw socket becomes readable. In this case, we are guaranteed that a `read` operation will return one (and only one) frame, and will return the entire frame if the buffer is big enough. `pppoe` reads the frame into a buffer. It then adds asynchronous byte-stuffing to the data and computes the PPP frame-check sequence. (Recall that the PPP FCS is not transmitted over PPPoE.) Finally, it writes the result to standard-output, where it is picked up by the kernel or `pppd`.

4.4 Synchronous PPP

You can immediately see two gross inefficiencies: User-space code is executed for every PPPoE frame, and byte-stuffing and de-stuffing is done twice. For outgoing frames, the kernel carefully performs byte stuffing, which is undone by `pppoe`. For incoming frames, `pppoe` stuffs them and the kernel de-stuffs them.

There is an option to `pppd` and `pppoe` which enables synchronous PPP. In this case, no byte-stuffing is performed. However, correct operation in this mode relies on `pppoe` reading exactly one frame at a time from standard input. (There are no frame boundary markers, so `pppoe` assumes that it gets a complete frame for each `read` system call.) While this seems to work on fast machines, it is not recommended, because delays in scheduling in `pppoe` can cause serious problems.

The kernel-mode implementation of PPPoE has no problem guaranteeing frame boundaries (because kernel code is invoked for each `read` and `write` call), so the kernel-mode implementation uses synchronous PPP without worries.

4.5 The MTU

PPPoE introduces a real and annoying problem. The maximum Ethernet frame is 1518 bytes long. 14 bytes are consumed by the header, and 4 by the frame-check sequence, leaving 1500 bytes for the payload. For this reason, the Maximum Transmission Unit (MTU) of an Ethernet interface is usually 1500 bytes. This is the largest IP datagram which can be transmitted over the interface without fragmentation.

PPPoE adds another six bytes of overhead, and the PPP protocol field consumes two bytes, leaving 1492 bytes for the IP datagram. The MTU of PPPoE interfaces is therefore 1492 bytes.

When a TCP connection is initiated, each side can optionally specify the Maximum Segment Size (MSS). TCP chops a stream of data into segments, and MSS specifies the largest segment each side will accept. By default, the MSS is chosen as the MTU of the outgoing interface minus the usual size of the TCP and IP headers (40 bytes), which results in an MSS of 1460 bytes for an Ethernet interface.

TCP stacks try to avoid fragmentation, so they use an MSS which will not cause fragmentation on their outgoing interface. Unfortunately, there may be intermediate links with lower MTU's which will cause fragmentation. Good TCP stacks perform *path MTU discovery*.

In path MTU discovery, a TCP stack sets a special Don't Fragment (DF) bit in the IP datagrams. Routers which cannot forward the datagram without fragmenting it are supposed to drop it and send an ICMP "Fragmentation-Required" datagram to the originating host. The originating host then tries a lower MTU value.

Unfortunately, many routers are anti-social and do not generate the fragmentation-required datagrams. Many firewalls are equally anti-social and drop all ICMP datagrams.

Now consider a client workstation on an Ethernet LAN connected to a PPPoE gateway. It opens a TCP connection to a web server. Because the Ethernet MTU is 1500, it suggests an MSS of 1460. The web server is also on an Ethernet and also suggests an MSS of 1460. The client then requests a web page. This request is typically small and reaches the web server. The server responds with many TCP segments, most of which are 1460 bytes long.

The maximum-sized segments result in 1500-byte IP datagrams and make their way to the DSL provider. The DSL provider cannot transmit a 1500-byte IP datagram over a PPPoE link, so it drops it (assume for now that the DF bit is set.) Furthermore, being anti-social, the DSL provider does not send an ICMP message to the web server.

The net result is that packets are silently dropped. The web client hangs waiting for data, and the web server keeps retransmitting until it finally gives up, or the connection is closed by the user aborting the web client.

One way around this is to artificially set an MSS for the default route on all LAN hosts behind the PPPoE gateway. This is annoying, as it requires changes on each host.

Instead, `rp-pppoe` "listens in" on the MSS negotiation and modifies the MSS if it is too big. (This was inspired by `mssclampfw` by Marc Boucher.)

`rp-pppoe` can be configured to intercept all TCP packets with the SYN bit set and silently adjust any advertised MSS options so they will be appropriate for the PPPoE link. Although the MSS option can appear in any TCP packet, in practice, most implementations send it only with SYN and SYN-ACK packets.

Adjusting the MSS is a gross hack. It breaks the concept of the transport-layer being end-to-end. It will not work with IPSec, because IPSec will not let you damage IP packets (they will fail to authenticate.) Nevertheless, it is a fairly effective solution to an ugly real-world problem, and is used by default in `rp-pppoe`.

5 Additional PPPoE Tools and Ports

In addition to the PPPoE client, the `rp-pppoe` package includes a couple of other useful tools: `pppoe-server` implements a PPPoE server, and `pppoe-sniff` examines frames from non-Linux systems to determine if any special run-time options are required to establish a connection.

5.1 The PPPoE Server

The PPPoE server is a very simple program. It listens for PPPoE discovery frames. When a PADI frame is received, it constructs a cookie by hashing the peer Ethernet address, the server's Ethernet address and a random number. This cookie is sent back in a PADO frame.

The cookie is designed to stop a simple-minded denial-of-service attack. In this attack, a malicious client sends many PADR frames with fake source Ethernet addresses. If the server responded to the PADR with a PADS, it would have to allocate resources for a PPP connection. Flooding the server with many PADR frames could quickly exhaust its resources. For this reason, requiring the client to return the cookie in its PADR frame ensures that the PADI was received from a valid Ethernet address.

The server can take further measures to limit denial-of-service attacks (such as limiting the number of PPP sessions per Ethernet address), but the current implementation of `pppoe-server` does not do that.

Once `pppoe-server` has received a valid PADR frame, it responds with a PADS and simply forks and execs `pppd` to handle the PPP connection. The new `pppd` process uses the `pty` option along with a special flag to `pppoe` informing it of the session number assigned by the server and the peer's Ethernet address.

Since each PPP session creates a `pppd` and `pppoe` process, `pppoe-server` is not suitable for production use as a heavy-traffic PPPoE server. It is meant to test PPPoE client ports and validate RFC-compliance of PPPoE clients.

5.2 PPPoE Sniffing

Some Internet service providers require special data in the PADI and PADR frames. For example, some providers require a `Service-Name` tag with a specific value. Unfortunately, since most providers support only Windows and Mac OS, and most ISP help-desk personnel haven't a clue about the inner workings of PPPoE, obtaining the information required to establish a PPPoE session under Linux is sometimes difficult.

The `rp-pppoe` package includes a program called `pppoe-sniff`. Figure 3 illustrates how to operate `pppoe-sniff`.

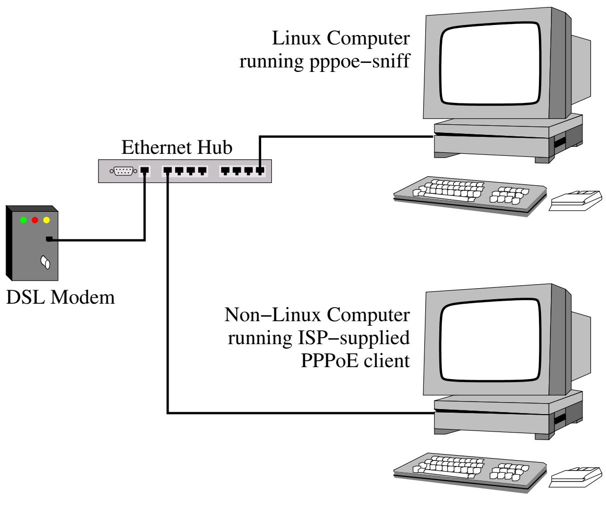

Figure 3: Using `pppoe-sniff`

Connect a Linux computer to an Ethernet hub. Connect another computer running the ISP-supported PPPoE client to the hub. Finally, connect the DSL modem either to the uplink port of the hub with a straight-through cable, or to a normal port of the hub with a crossover cable. It is important to use a hub and not a switch, because the Linux computer must be able to see all traffic between the DSL modem and the other computer. Also, the two computers and the modem should be the only devices connected to the hub, because some DSL modems react badly if they see Ethernet frames from more than two other MAC addresses.

Once the setup is complete, start `pppoe-sniff` on the Linux machine and make a connection to the ISP using the ISP-supplied client.

Because one person using `rp-pppoe` reported that his ISP used non-standard frame types for PPPoE discovery and session frames (0x3c12 and 0x3c13 instead of 0x8863 and 0x8864), `pppoe-sniff` listens for all Ethernet frames and picks out likely-looking PPPoE frames based on the type, version and code fields. It is conceivable that non-PPPoE traffic could confuse (or crash!) `pppoe-sniff`, so do not run such traffic while `pppoe-sniff` is operating.

`pppoe-sniff` looks for a likely-looking PADR frame and a likely-looking session frame. From these frames, it gleans the frame types used by the ISP, as well as any Service-Name tag which may be required during discovery. It prints its findings out in the form of command-line options to supply to `pppoe`. In this way, you can usually determine any special options required by your ISP without having to go through technical support personnel.

5.3 NetBSD Port

In addition to Linux, `rp-pppoe` has been ported to NetBSD. The port was done by Geoff Mottram and Yannis Sismanis. It involved substituting calls to NetBSD's BPF API instead of Linux's SOCK_PACKET. The port was quite easy and affected only about 10% of the total code in `rp-pppoe`. It should be easy to port `rp-pppoe` to SVR4-derived UNIXes which use DLPI, or even to write a `libpcap`- and `libnet`-based version, which will be portable to all systems to which these libraries have been ported.

(Actually, while `rp-pppoe` is quite portable, it requires the `pty` option of `pppd`, which apparently is no longer maintained on FreeBSD or OpenBSD. Anyone wanting to port `rp-pppoe` to a new UNIX will have to ensure that `pppd` is ported, too.)

6 Summary

PPPoE is not a wonderful protocol for end-users. Unfortunately, it is here to stay, and the Linux community will have to live with it.

The user-space `rp-pppoe` is a convenient method of connecting to a PPPoE provider, but it should be viewed as a temporary measure until stable kernels support PPPoE natively. Once this happens, the `rp-pppoe` software will be used only for the discovery phase, and will offload the session phase to the kernel.

Bugs in many routers prevent the correct operation of path MTU discovery. `rp-pppoe` has a gross hack to work around this for TCP connections. I would love to eliminate this hack, but it requires that all router vendors fix their software. This is not very likely to happen.

Some ISP's use non-standard frame types and/or require special Service-Name tags during the discovery phase. The `pppoe-sniff` program attempts to extract this information by listening to an ISP-supplied PPPoE client's connection.

`rp-pppoe` is available at the following URL: `http://www.roaringpenguin.com/pppoe/`. A kernel-mode PPPoE implementation is available at `http://www.davin.ottawa.on.ca/pppoe/`.

7 Acknowledgements

I'd like to thank all the people who have downloaded, tested and used `rp-pppoe`. In particular, Geoff Mottram, Yannis Sismanis, Heiko Shlittermann, Gary Cameron, Julian Gorfajn, Geoff Kuenning, Patrick Smith and Jason Lassaline submitted patches, ports and bug reports.

References

[1] RFC 1661, "The Point-to-Point Protocol (PPP)", W. Simpson, Editor, July 1994.

[2] RFC 1662, "PPP in HDLC-like Framing", W. Simpson, Editor, July 1994.

[3] RFC 2516, "A Method for Transmitting PPP Over Ethernet (PPPoE)", L. Mamakos et al., February 1999.