PPPoE and Linux

By David F. Skoll

Roaring Penguin Software Inc.

2 February, 2000

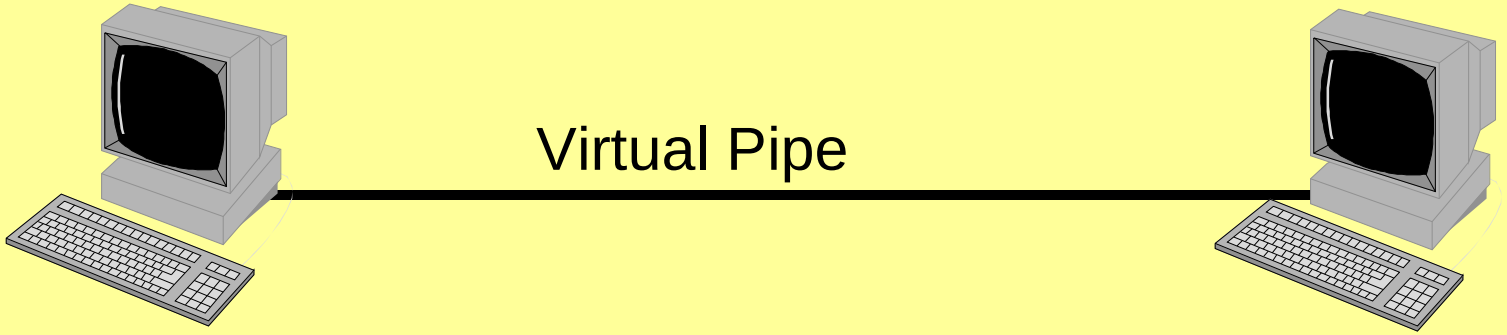
http://www.roaringpenguin.com
dfs@roaringpenguin.com

PPPoE and Linux: Overview

- TCP/IP Networking Basics
- PPP Basics
- PPPoE described
- Linux userland PPPoE implementation
- Some source code details
- Questions (and maybe some answers.)

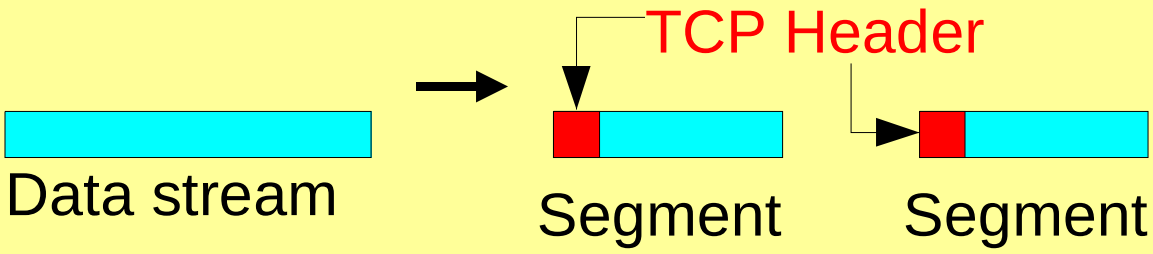
TCP/IP Networking Basics

- TCP/IP is a suite of *protocols* for network communication.
- Consider TCP: Applications appear to have a "pipe" connecting them. Whatever one application writes to the pipe, the other application reads.



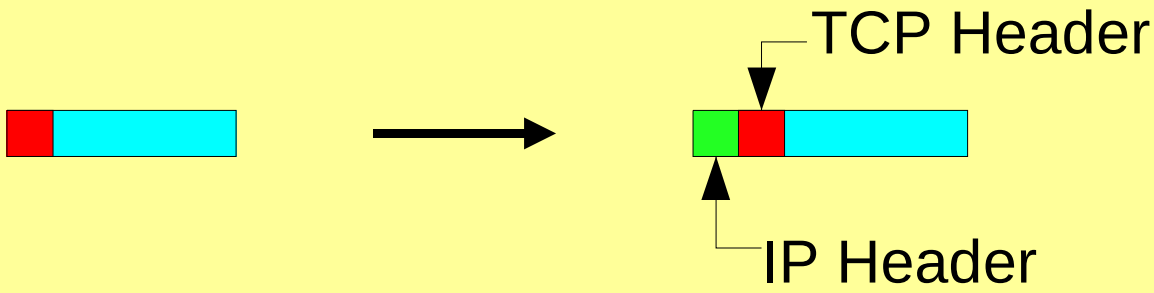
TCP/IP Networking Basics

- In fact, computer network transmit *packets* of information, not continuous streams.
- TCP breaks up the stream into chunks called *segments* and passes them to IP.
- The TCP Header specifies source and destination processes and other book−keeping information.



IP: The Network Layer

- IP is responsible for getting packets from the source machine to the destination machine.
- It is a *best−effort* protocol: It does not guarantee delivery of packets.
- TCP is responsible for detecting lost/duplicated packets and fixing things up.

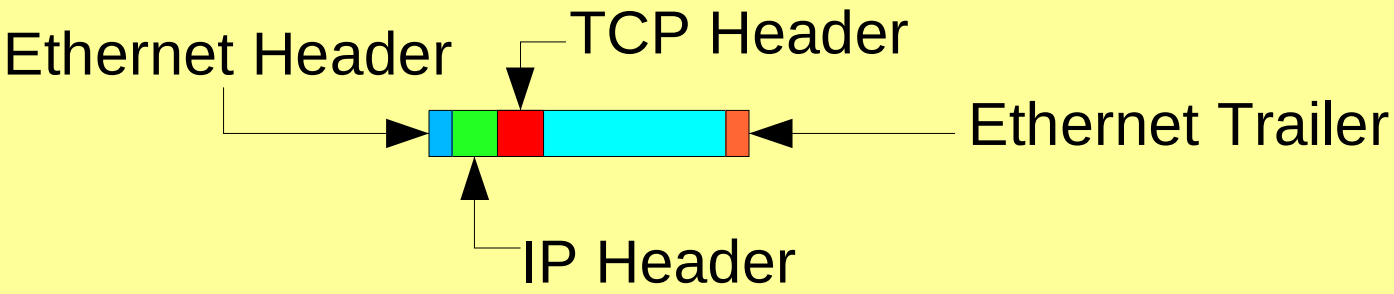


UDP: A simple transport layer

- UDP is a simple interface built on top of IP.
- It essentially adds source and destination ports only. Reliability, lost/duplicate packet detection, etc. are up to the application program.
- The IP layer does not care if a packet comes from UDP or TCP. Everything after the IP header is just data.

The Datalink Layer

- IP datagrams must be transmitted physically.
- The physical transmission means is called the *encapsulation*.
- Ethernet encapsulation: Adds a 14−byte Ethernet header and a 4−byte CRC trailer.

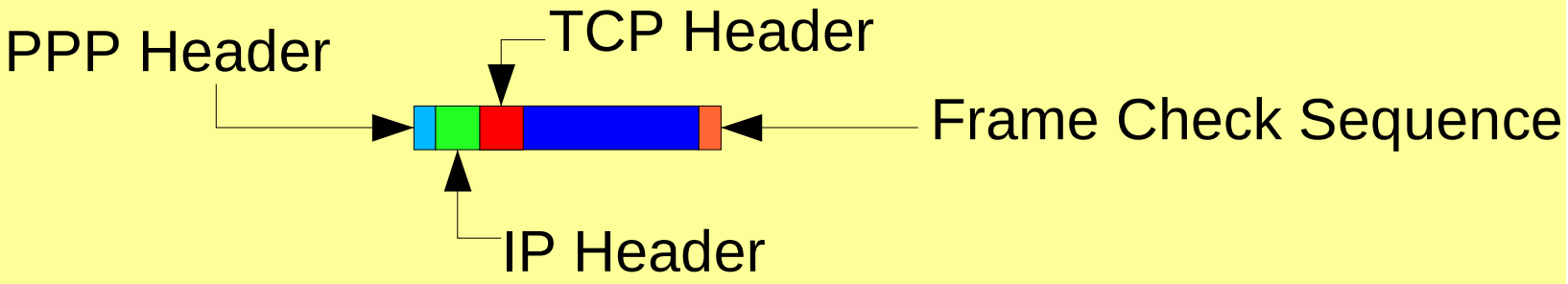


Header Contents

- Ethernet Header: Source and destination Ethernet addresses and frame type. The frame type field allows many different protocols such as IP, IPX, NetBEUI etc. be encapsulated in Ethernet.
- IP Header: Source and destination IP addresses and other info.
- TCP Header: Source and destination ports, sequence number and other info.

PPP

- PPP (Point−to−Point Protocol) is an encapsulation scheme for transmitting datagrams over serial links.
- PPP adds its own header and trailer and can carry IP and other traffic.
- PPP uses special *framing bytes* to mark the boundaries of frames.



PPP Framing (HDLC Framing)

- **0x7e**: *frame byte* marks frame boundaries.
- **0xff**: *frame address* marks start of frame (following 0x7d)
- **0x7d**: *escape byte* used to allow arbitrary data in the packet. The byte following 0x7d is Xor'd with 0x20. This allows arbitrary binary data in packets and ensures that framing and address bytes never appear within a packet.
- CPU−Intensive: Transmitter must examine *every byte* and escape them if necessary.

PPP Framing Example

- We wish to transmit:
  `0x00, 0x56, 0xff, 0x81, 0x7d, 0x7e`
- The bytes that go out are:
  `0x7d, 0x20, 0x56, 0x7d, 0xdf, 0x81, 0x7d, 0x5d, 0x7d, 0x5e`
- Receiver *un−escapes* the incoming stream to yield original set of bytes.
- By default, all bytes < 0x20 are escaped, as are 0x7d, 0x7e and 0xff.

PPPoE: PPP over Ethernet

- The ISP runs an *access concentrator* which accepts PPP connections.
- The client machine performs *discovery* to locate the access concentrator, and then switches to *session* mode once a connection is established.
- Discovery and session Ethernet frames have their own Ethernet protocol numbers, distinct from IP or any other protocol.

PPPoE Discovery: Step 1

- Client transmits a *PADI* frame (PPPoE Active Discovery Initiation)
- This frame is sent to the *broadcast* Ethernet address and basically says: "Hey, any access concentrators out there? I need you!"

PPPoE Discovery: Step 2

- Access concentrator(s) transmit a *PADO* frame (PPPoE Active Discovery Offer)
- This frame is sent to the client’s Ethernet address and says, "Hello, client. I am an access concentrator. Would you like to set up a session with me?"

PPPoE Discovery: Step 3

- The client picks an access concentrator (if more than one responded) and sends a *PADR* packet (PPPoE Active Discovery Request) to it’s Ethernet address.
- The PADR packet says, "Thank you, kind access concentrator, I would like to establish a session with you."

PPPoE Discovery: Step 4

- The access concentrator sends a *PADS* packet (PPPoE Active Discovery Session−confirmation.)
- This packet says, "Dear client, I am happy to establish a session with you. Here is your *session number*.
- This session number, combined with the source and destination Ethernet addresses, uniquely identifies a PPPoE session. This allows for multiple PPPoE sessions through a single ADSL modem.

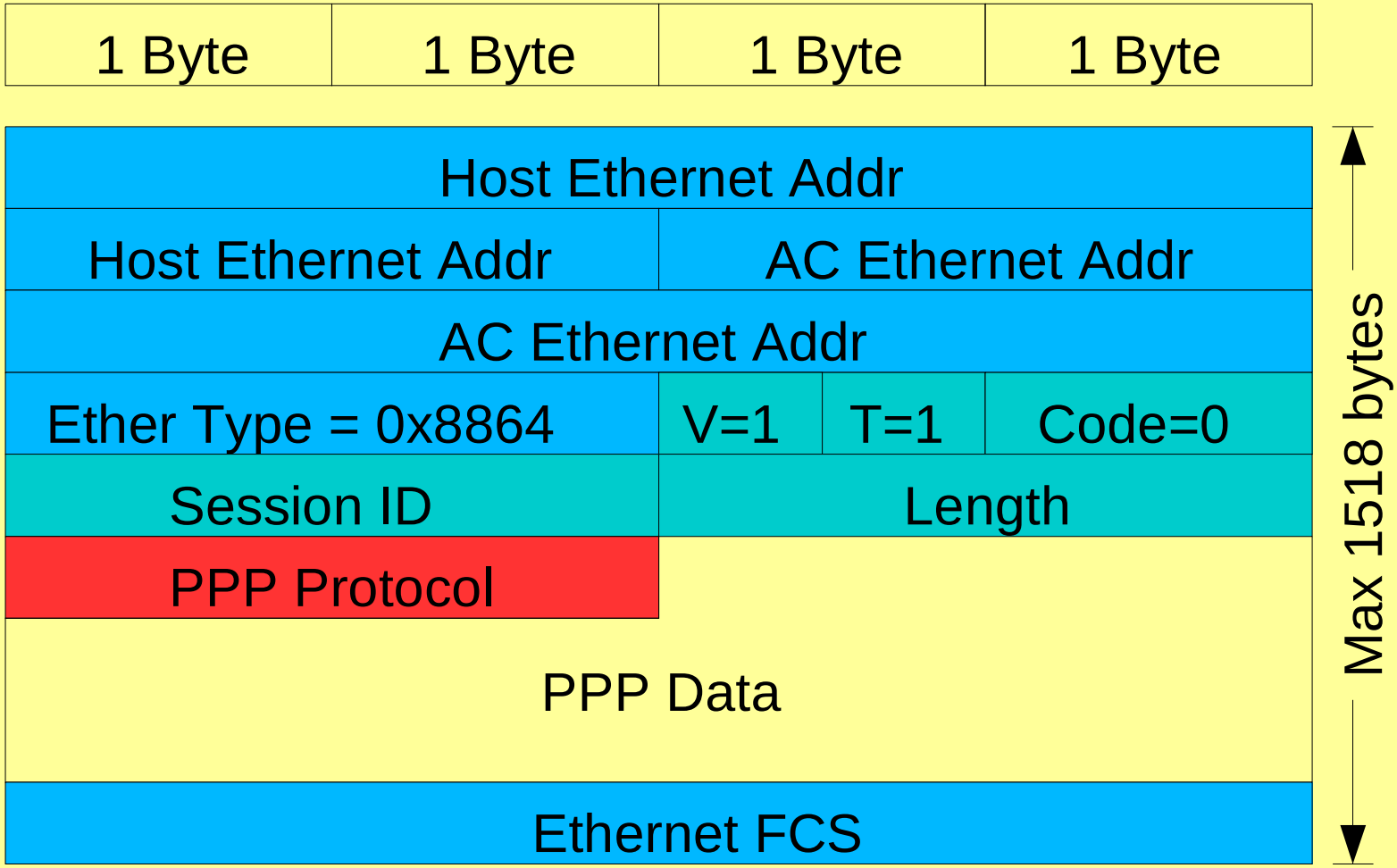
PPPoE Session Phase

- This is a normal PPP session. However:
  - Packets are transmitted over Ethernet instead of a serial link.
  - No escape bytes are required because frame boundaries are explicit in Ethernet encapsulation.
  - 6 bytes of overhead are added in addition to the Ethernet header.
  - No PPP FCS is required because Ethernet has its own CRC.

Why a Userland Solution?

- The *best place* for PPPoE is in the Linux kernel. However:
  - I do not want to patch my kernel. I do not want my customers to patch theirs, either −− they are typically not Linux−savvy.
  - It’s much easier to write user−mode software. It’s also more likely to be portable to *BSD, Solaris, etc.
  - I wanted a simple solution that people can install and get running in 10 minutes.

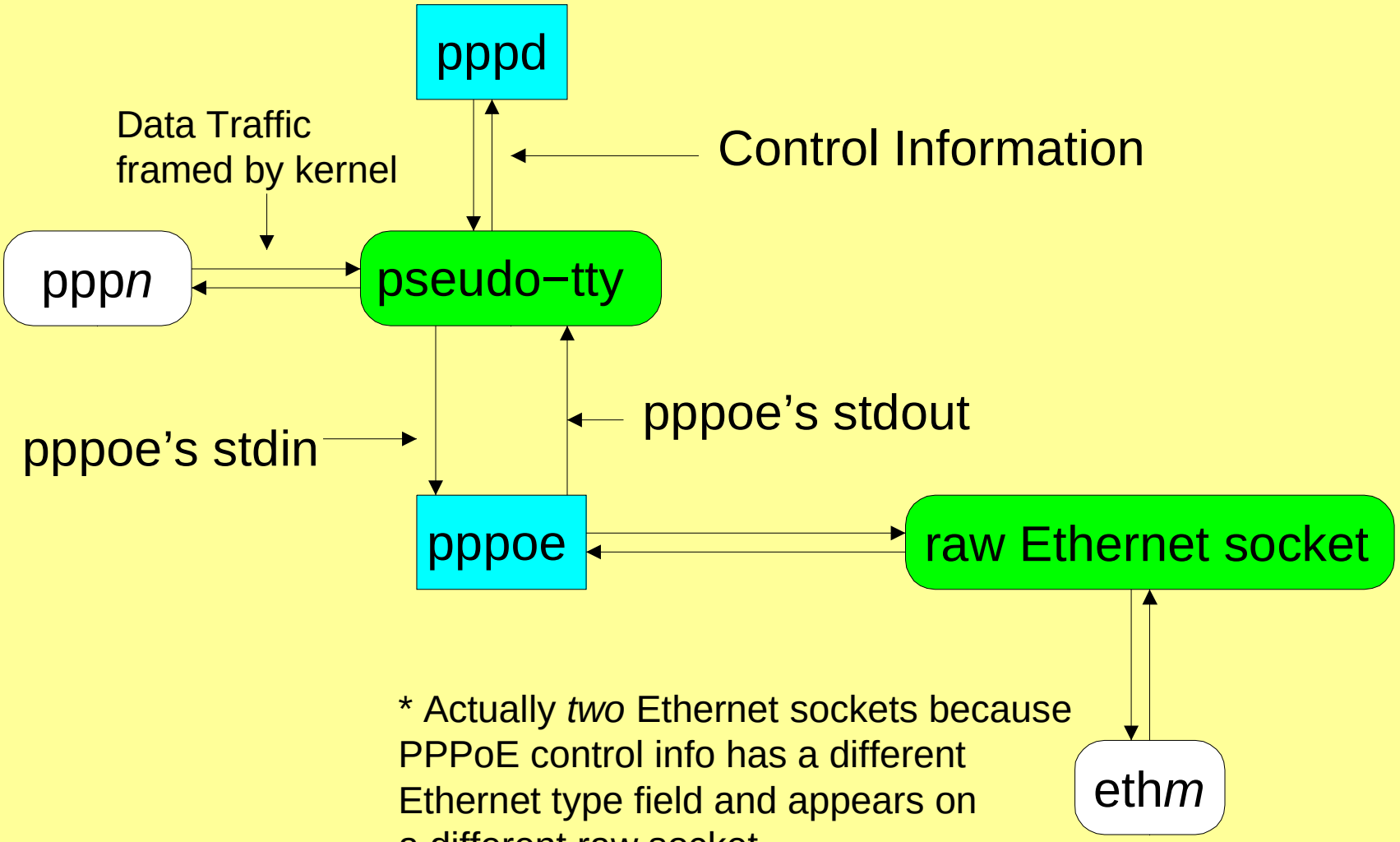
PPPoE Session Packets



pppd Operation

- pppd creates a network device (ppp*n*) and "binds" it to a tty (typically, a serial port.)
- Packets sent to ppp*n* are framed using normal PPP framing and transmitted over the tty.
- Data received from the tty are "un−framed" and made to appear to come from ppp*n*.
- The PPPoE trick: The tty used by pppd can be a *pseudo−tty* instead of a real serial port.

pppd Operation, continued



* Actually *two* Ethernet sockets because PPPoE control info has a different Ethernet type field and appears on a different raw socket.

pppoe’s Point of View

- HDLC−framed data appears on stdin. This must be "un−framed" and sent over the raw Ethernet socket.
- Incoming pppoe packets appear on the raw Ethernet socket and must be framed and written to stdout.
- PPPoE control packets may appear on a second raw Ethernet socket.

Code Overview

- main: Read command−line options and set internal variables.
- discovery: Perform PPPoE discovery
- session: Perform PPPoE session phase

Discovery Function pseudocode

- while (!maxAttempts)
  - send PADI
  - wait for PADO with timeout *t*
  - double timeout *t*; return error if maxAttempts
- select access concentrator
- while (!maxAttempts)
  - send PADR
  - wait for PADS with timeout *t*
  - double timeout *t*, return error if maxAttempts

Session Function pseudocode

- Wait for one of: data on stdin, incoming PPPoE session packet, incoming PPPoE control packet or timeout.
- If timeout:
  - Log an error message and quit
- Timeout works in conjunction with lcp−echo−interval to detect abruptly−dropped links.

Session Function part 2

- If data on stdin:
  - Un−frame the data
  - If we have collected a complete frame, transmit it over Ethernet
- Data from stdin can arrive in unpredicably−sized chunks. We need a *state machine* to keep track of where we are in the HDLC framing scheme.

Session Function part 3

- If PPPoE session packet:
  - Frame the packet using HDLC framing
  - Write to stdout
- Incoming Ethernet packets are always read in their entirety, one packet at a time. No need for a state machine to keep track of framing.

Session Function part 4

- If PPPoE control packet:
  - Ignore all but PADT packets.
  - If we receive a PADT packet, log a message and exit.

I/O Multiplexing: The good, the bad and the ugly.

- Data from stdin, the Ethernet sockets and the timeouts can happen unpredictably. How to we monitor them all and react when any event occurs?
- The ugly way: Multiple processes (using fork) which each monitor one event source. Luke Stras's client works this way.
- The bad way: Multiple threads (using pthread_create) which each monitor one event source. Bell's Enternet client works this way.

I/O Multiplexing

- The good (i.e., traditional UNIX) way: The *select* system call.
- select lets you specify:
  - A set of file descriptors to watch for "readability"
  - A set of file descriptors to watch for "writability"
  - A set of file descriptors to watch for error conditions
  - A timeout
- When *any* event happens, select returns and you can find out what happened.

Actual Code Fragment

```
if (optInactivityTimeout > 0) {
    tv.tv_sec = optInactivityTimeout;
    tv.tv_usec = 0;
    tvp = &tv;
}
FD_ZERO(&readable);
FD_SET(0, &readable);     /* ppp packets come from stdin */
FD_SET(DiscoverySocket, &readable);
FD_SET(SessionSocket, &readable);
r = select(maxFD, &readable, NULL, NULL, tvp);
```

- Everything before select sets up parameters
- select waits for an event to happen
- pppoe runs in a single process and a single thread. This minimizes use of resources.
- Multiple threads or processes are overkill for this case.

Datalink Access

- *BSD: Berkeley Packet Filter
- System V: DLPI
- Linux: SOCK_PACKET or SOCK_RAW
- The BSD and SysV implementations are much better than the Linux implementation:
  - They perform filtering in the kernel.
  - They buffer packets if requested to do so, minimizing the number of read system calls.
- Linux does no (well, primitive) filtering and no buffering. Luckily, for 1MB/s ADSL, it’s OK.

Summary

- User−mode PPPoE client less efficient than kernel−mode, but easier to set up and independent of kernel updates.
- Client runs in a single thread and is relatively efficient. Fully RFC−compliant.
- Client is GPL'd. Get it from:

  http://www.roaringpenguin.com/pppoe.html

- Kernel−mode client at:

  http://www.davin.ottawa.on.ca/pppoe/

Code Overview (If we have time)