Point-to-Point Protocol

Connecting to your ISP via modem,
your other computer via null-modem,
and whether PPP is DUN, or DUN is PPP.

David Morgan

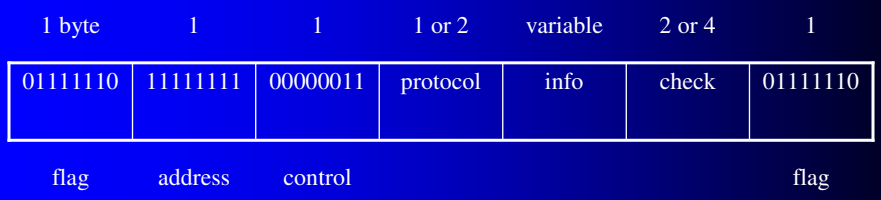
Setting the bar –

original design requirements

RFC1547

- packet framing
- transparency
- multiple network-layer protocols
- multiple types of links
- error detection
- connection liveness
- network-layer address negotiation

PPP data frame format



protocol — the protocol to which "info"s content belongs (numeric id)

info — a packet of the protocol identified in "protocol"

What protocol does a ppp frame carry?

- a protocol to establish the ppp connection itself or
- a protocol to establish the connection of some other protocol (which ppp will carry) or
- that other protocol

PPP numeric protocol id's

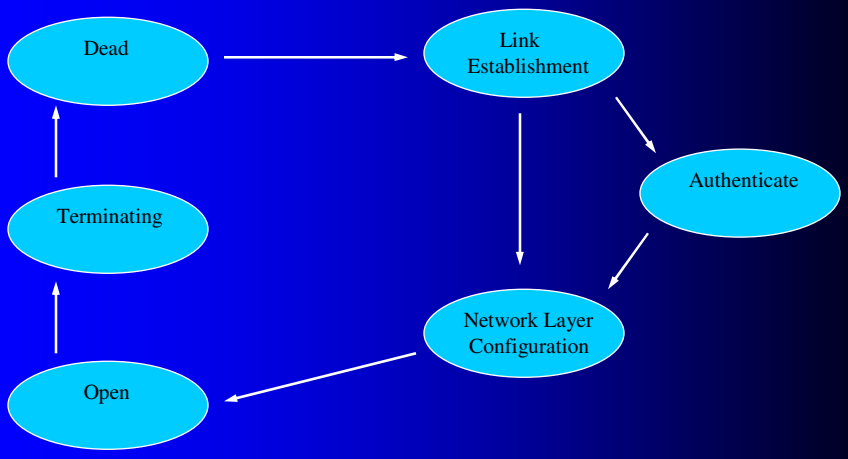
Set ppp connection	
C021	LCP
C023	PAP
C223	CHAP

Set other connection		Other protocol	
8021	IP control protocol	0021	IP
8026	Appletalk c.p.	0026	Appletalk
8029	IPX ctrl protocl	0029	IPX

For example…

- Link Control Protocol (LCP) or
- IP Control Protocol (IPCP) or
  - establishes endpoints' IP addresses specific to the PPP link
  - establishes whether to compress
- IP itself
  - IP packets, carrying which is PPP's purpose

LCP – state diagram



State diagrams – quick recap

- a set of states
- a set of inputs (might be events)
- a transition function f(s,i) of state, input
  - if you're in this state
  - and that input comes along
  - to which other state do you transition?

What can state diagrams represent?

- physical/engineering systems
- languages (natural and computer)
- protocols

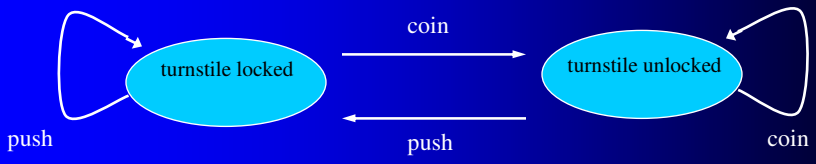
- related: finite state machines (FSMs), automata, finite automata, Turing machines

State diagram example of a physical system

State diagram example – a turnstile

- turnstile states
  - locked
  - unlocked
- inputs/events
  - customer pushes turnstile
  - customer inserts coin into turnstile
- transition function F(state, input)
  - turnstile locked, customer pushes → turnstile locked
  - turnstile locked, customer inserts → turnstile unlocked
  - turnstile unlocked, customer pushes → turnstile locked
  - turnstile unlocked, customer inserts → turnstile unlocked

State diagram - turnstile



- turquoise ovals are states
- wording on arrows are inputs/events
- arrows are transitions

State diagram example of a protocol

protocols are code

```
typedef enum {frame_arrival} event_type;
#include "protocol.h"
void sender1(void)
{
    frame        s;                      /* buffer for an outbound frame */
    packet       buffer;                 /* buffer for an outbound packet */
    while (true) {
         from_network_layer(&buffer);    /* go get something to send */
         s.info = buffer;                /* copy it into s for transmission */
         to_physical_layer(&s);          /* send it on its way */
    }
}
void receiver1(void)
{
    frame        r;
    event_type   event;                  /* filled in by wait, but not used here */
    while (true) {
         wait_for_event(&event);         /* only possibility is frame arrival */
         From_physical_layer(&r);        /* go get the inbound frame */
         To_network_layer(&r.info);      /* pass the data to the network
                                                 layer */
    }
}
```

State diagram example – 2-node net data link

data sender **data receiver**

- data sender sends frames holding data; data receiver sends frames holding acknowledgments
- sender numbers frames; acknowledgement is by frame number
- frame numbering is
  - from zero
  - circular (wraps back around to zero and continues)
  - goes up to maximum of one
- system is continuously active, sending something on channel

States of system's components

	represented by
• sender states	
– trying to get frame 0 handled (incl ack)	0
– trying to get frame 1 handled (incl ack)	1
• receiver states	
– expecting to receive frame 0	0
– expecting to receive frame 1	1
• channel states	
– carrying frame 0	0
– carrying frame 1	1
– carrying acknowledgement frame	A

12 possible states - system as a whole

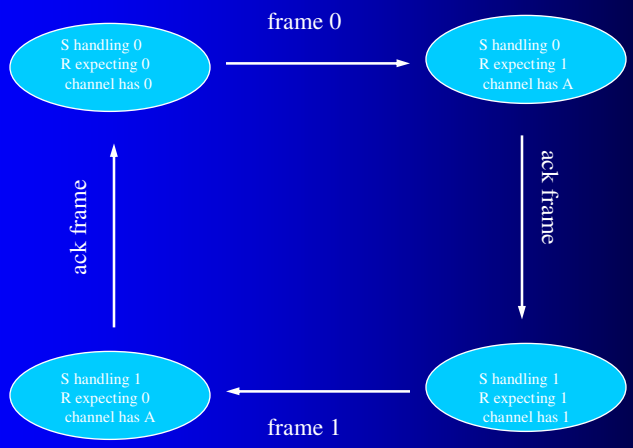
all combinations of components' states

sender	receiver	channel	
0	0	0	S handling 0, R expecting 0, 0 in channel
0	0	1	
0	0	A	
0	1	0	
0	1	1	
0	1	A	S handling 0, R expecting 1, A in channel
1	0	0	
1	0	1	
1	0	A	S handling 1, R expecting 0, A in channel
1	1	0	
1	1	1	S handling 1, R expecting 1, 1 in channel
1	1	A	

Events in the 2-node net

- frame 0 sent (from sender to receiver)
- acknowledgement frame sent (from receiver to sender)
- frame 1 sent (from sender to receiver)

State diagram – 2-node net



LCP – packet type codes

Code	Packet Type
1	Configure-Request
2	Configure-Ack
3	Configure-Nak
4	Configure-Reject
5	Terminate-Request
6	Terminate-Ack
7	Code-Reject

Network-layer config protocols

- not part of ppp itself
- a family of network control protocols
- network-layer specific to configure:
  - IP parameters for IP or
  - IPX parameters for IPX or
  - Appletalk parameters for Appletalk

Point-to-Point => Bilateral

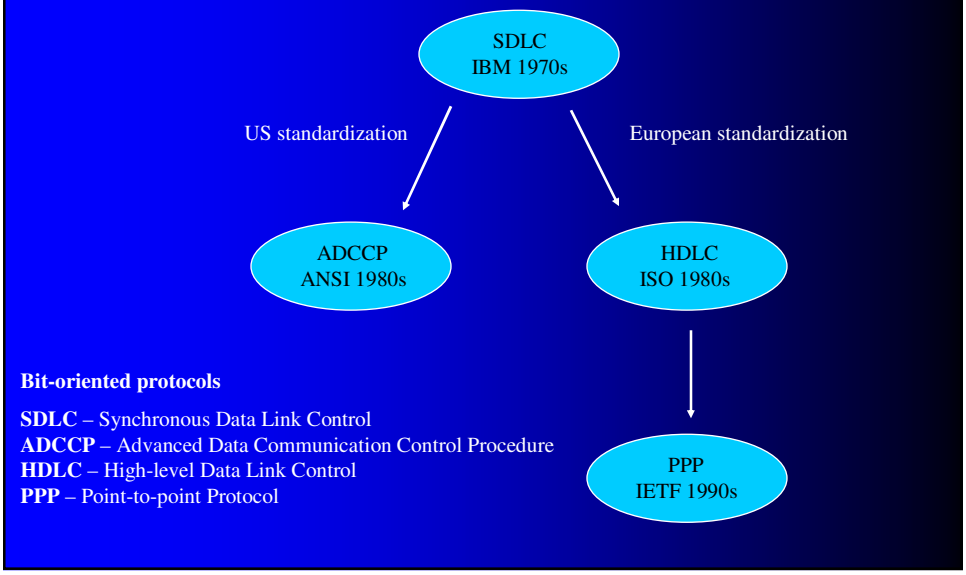
```
[root@EMACH1 /root]# ifconfig ppp0

ppp0      Link encap:Point-to-Point Protocol
          inet addr:206.170.218.155  P-t-P:209.233.193.30
          UP POINTOPOINT RUNNING NOARP MULTICAST  MTU:1524  Metric:1
          RX packets:7 errors:0 dropped:0 overruns:0 frame:0
          TX packets:7 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:10
```

Linux implementation of PPP

- in-kernel HDLC driver
  - composes/decomposes frames
  - interfaces with network layer (e.g., IP, IPX)
- user-space pppd executable
  - operates the various control protocols

PPP geneology

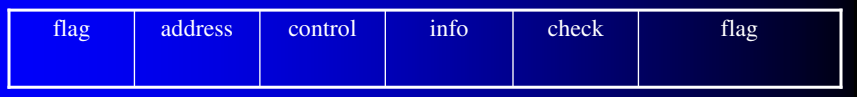


Bit-oriented protocols

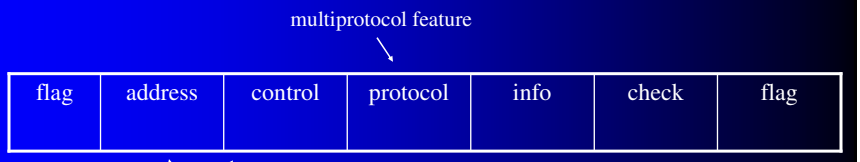
SDLC – Synchronous Data Link Control
ADCCP – Advanced Data Communication Control Procedure
HDLC – High-level Data Link Control
PPP – Point-to-point Protocol

HDLC and descendants

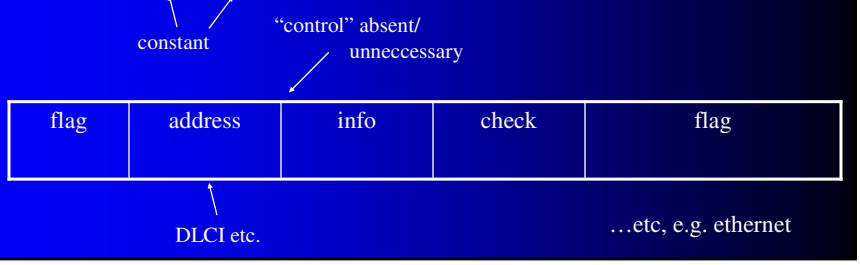
HDLC:



PPP:



Frame relay:



…etc, e.g. ethernet

Bit-oriented protocols

- Preceded historically by "character-oriented" protocols
- Seek to be "atomically arbitrary" at bit level
  - allow arbitrary number of bits as cargo
  - allow arbitrary bit settings/values
  - by starting and ending with flag pattern 01111110
- Achieved by bit "stuffing"
  - disambiguates in-data presence of reserved (e.g., flag's) bit patterns

Bit stuffing example

- sender stuffs 0 following any 5 consecutive 1s
- receiver deletes 0 following any 5 consecutive 1s
- applied to data but not the 2 flags

original data: 0110111111111111111110010

in-transit data: 01101111101111101111010010

recovered data: 0110111111111111111110010

PPP over a serial wire

"To use some serial line as a PPP link, you first establish the connection over your modem as usual, and subsequently ***convert the line to PPP*** mode. In this mode, all incoming data is passed to the PPP driver [pppd], which checks the incoming HDLC frames for validity…and unwraps and dispatches them."

The Linux Network Administrator's Guide
Olaf Kirsch

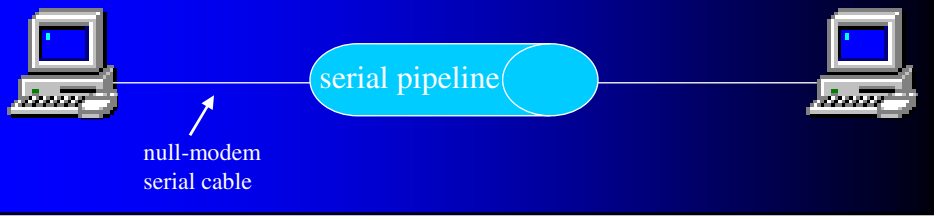
"Convert the [serial] line to PPP"

- Who does that?
  - pppd
- How? - establish pipe then pour
  - runs at ***both ends*** of a "pipe"
  - launch signature stream into pipe
  - handshake in the middle

Establish a pipe ("some serial line")

1) No link (pre-cabling)

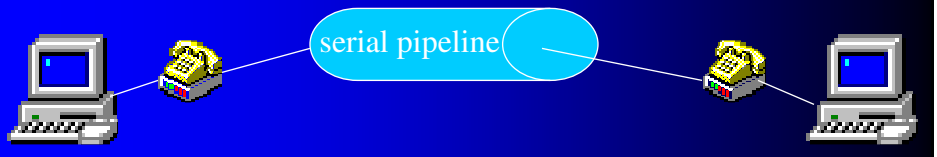
2) Serial link (post-cabling)



Equivalently…

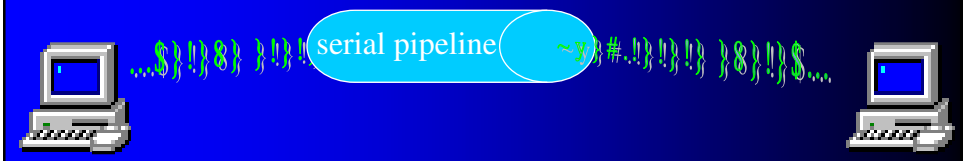
1) No link (pre-dial)

2) Serial link (post-dial)



Establish pipe then pour

3) Run pppd (twice…) to "pour"



PPP HowTo - "Testing your modem for dial out"

4) PPP link over serial link

PPP HowTo - "Setting up the PPP connection manually"

Initiating the "pour"

- Launch ppp software (side-by-side method)
  - at both ends
  - directed toward the cable-end
  - simultaneously
- Conduct specialized "login" (ISP method)
  - called side
    - runs login software continuously
    - launches ppp software against successful login (not shell)
  - calling side
    - logs in
    - then causes launch of ppp software

Meeting in the Middle

- Opposing ppp streams must travel thru *same device*
- If ISP, reliably launches/pours stream at you upon answer
- You must launch a counter-stream back at him

Connecting by hand

- PPP HowTo - "Setting up the PPP connection manually"
- Good for
  - Tutorial value
  - Groundwork to ensure successful automation

Automating connection setup

- Establishing the pipe - chat
- Pouring ppp stream into it – pppd
- pppd specifies chat in its ‘connect’ option
- HowTo embodies this in
  - /etc/ppp/ppp-on
    - invokes ppp-on-dialer
  - /etc/ppp/ppp-on-dialer
    - exec’s chat

PPP components

- In-kernel driver
  - HDLC frames
  - packet carrier
- Daemon ‘pppd’ in user space
  - LCP, negotiator of connection terms

pppd files

- /usr/sbin/pppd
- /etc/ppp/options

/var/log/messages

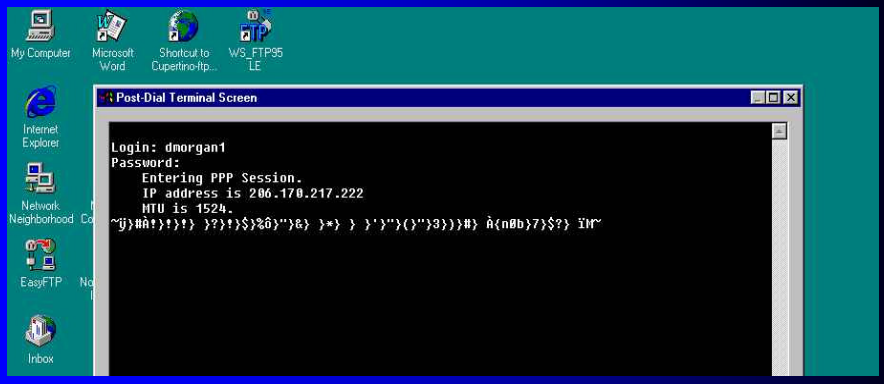
Sucessful connection:

```
Jan 22 05:08:09 localhost pppd[1118]: Serial connection established.
Jan 22 05:08:09 localhost pppd[1118]: Using interface ppp0
Jan 22 05:08:09 localhost pppd[1118]: Connect: ppp0 <--> /dev/ttyS0
Jan 22 05:08:10 localhost kernel: PPP BSD Compression module registered
Jan 22 05:08:10 localhost kernel: PPP Deflate Compression module registered
Jan 22 05:08:10 localhost pppd[1118]: local  IP address 206.170.217.235
Jan 22 05:08:10 localhost pppd[1118]: remote IP address 209.233.193.22
```

Signature garbage in linux

```
[root@vpnd ppp]# pppd notty
~ÿ}#À!}!}!} }4}"}&} } } } }%}&Ê]yç}'}"}(}"'>~
```

Signature garbage in Windows

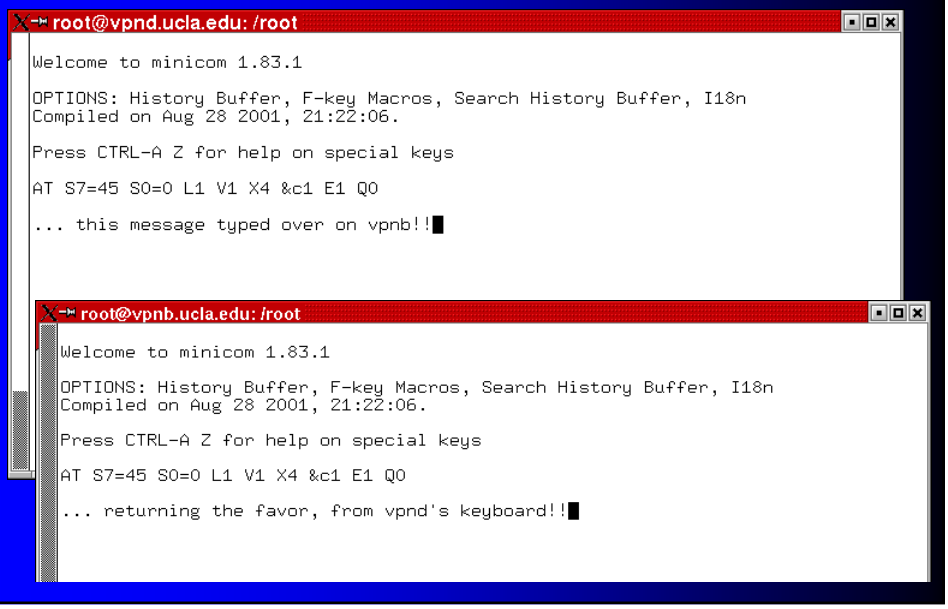


Test null-modem connection with minicom terminal emulator

A working minicom configuration (both sides):

```
[root@vpnb etc]# cat /etc/minirc.dfl
# Machine-generated file - use "minicom -s" to change parameters.
pr port             /dev/ttyS0
pu baudrate         115200
pu bits             8
pu parity           N
pu stopbits         1
```

Minicom test null-modem



Make manual serial connection

vpnd	pppd /dev/ttyS0 115200 \ 192.168.111.2:192.168.111.1 crtscts lock local
vpnb	pppd /dev/ttyS0 115200 \ crtscts lock local

Break manual serial connection

vpnd	killall pppd
vpnb	(or, above command here instead)

Two ways to diagnose

- "record"
  - use pppd's record option
    pppd … record <filename>
  - read resulting file with pppdump
    pppdump <filename>
- "debug"
  - use pppd's debug option, and optionally logfile option
    pppd … debug logfile <filename>
  - read ascii results in <filename> or system log

Typical "record" output

```
start Tue Apr 30 14:39:56 2002
time  0.2s
sent "~\ff}#\c0!}!}!} }4}\"}&} } } } }%}&`}(\bcF}'}\"}(}\"\fe}5~"
time  0.1s
rcvd "~\ff}#\c0!}\"}!} }4}\"}&} } } } }%}&`}(\bcF}'}\"}(}\"}5|~"
time  2.0s
rcvd "~\ff}#\c0!}!}!} }4}\"}&} } } } }%}&{\a5}6}#}'}\"}(}\"\811~"
sent "~\ff}#\c0!}\"}!} }4}\"}&} } } } }%}&{\a5}6}#}'}\"}(}\"j}%~\80!\01\01\00
      \1c\03\06\00\00\00\00\02\06\00-\0f\01\81\06\00\00\00\00\83\06\00\00\00
      \00w8~\80\fd\01\01\00\0f\1a\04x\00\18\04x\00\15\03/2\a3~"
rcvd "\80!\01\01\00\10\03\06\c0\a8o\02\02\06\00-\0f\01\86\fe~\80\fd\01\01\00
      \0f\1a\04x\00\18\04x\00\15\03/2\a3~\80!\04\01\00\10\81\06\00\00\00\00
      \83\06\00\00\00\00\a2q~\80\fd\02\01\00\0f\1a\04x\00\18\04x\00\15\03/L{~"
sent "\80!\02\01\00\10\03\06\c0\a8o\02\02\06\00-\0f\01\a7d~\80\fd\02\01\00\0f
      \1a\04x\00\18\04x\00\15\03/L{~\80!\01\02\00\10\03\06\00\00\00\00\02\06
      \00-\0f\01Y\cc~"
rcvd "\80!\03\02\00\0a\03\06\c0\a8o\01\fcM~"
sent "\80!\01\03\00\10\03\06\c0\a8o\01\02\06\00-\0f\01\bc\c6~"
rcvd "\80!\02\03\00\10\03\06\c0\a8o\01\02\06\00-\0f\01\9d\\~"
time  65.2s
rcvd "~\ff}#\c0!}%}\"} }0User requestS3~"
sent "~\ff}#\c0!}&}\"} }$\94}-~"
time  3.1s
end send
```

Typical “debug” output

```
using channel 17
Using interface ppp0
Connect: ppp0 <--> /dev/pts/0
sent [LCP ConfReq id=0x1 <asyncmap 0x0> <magic 0x75edf160> <pcomp> <accomp>]
rcvd [LCP ConfAck id=0x1 <asyncmap 0x0> <magic 0x75edf160> <pcomp> <accomp>]
rcvd [LCP ConfReq id=0x1 <asyncmap 0x0> <magic 0xda6ef3d8> <pcomp> <accomp>]
sent [LCP ConfAck id=0x1 <asyncmap 0x0> <magic 0xda6ef3d8> <pcomp> <accomp>]
sent [IPCP ConfReq id=0x1 <addr 0.0.0.0> <compress VJ 0f 01> <ms-dns1 0.0.0.0>
 <ms-dns3 0.0.0.0>]
sent [CCP ConfReq id=0x1 <deflate 15> <deflate(old#) 15> <bsd v1 15>]
rcvd [IPCP ConfReq id=0x1 <addr 192.168.111.2> <compress VJ 0f 01>]
sent [IPCP ConfAck id=0x1 <addr 192.168.111.2> <compress VJ 0f 01>]
rcvd [CCP ConfReq id=0x1 <deflate 15> <deflate(old#) 15> <bsd v1 15>]
sent [CCP ConfAck id=0x1 <deflate 15> <deflate(old#) 15> <bsd v1 15>]
rcvd [IPCP ConfRej id=0x1 <ms-dns1 0.0.0.0> <ms-dns3 0.0.0.0>]
sent [IPCP ConfReq id=0x2 <addr 0.0.0.0> <compress VJ 0f 01>]
rcvd [CCP ConfAck id=0x1 <deflate 15> <deflate(old#) 15> <bsd v1 15>]
Deflate (15) compression enabled
rcvd [IPCP ConfNak id=0x2 <addr 192.168.111.1>]
sent [IPCP ConfReq id=0x3 <addr 192.168.111.1> <compress VJ 0f 01>]
rcvd [IPCP ConfAck id=0x3 <addr 192.168.111.1> <compress VJ 0f 01>]
local  IP address 192.168.111.1
remote IP address 192.168.111.2
Script /etc/ppp/ip-up started (pid 3230)
Script /etc/ppp/ip-up finished (pid 3230), status = 0x0
```

Automate one side (vpnd)

“Running pppd as a server is just a matter of configuring a serial tty device to invoke pppd with appropriate options when an incoming data call has been received.”

Linux Network Administrators Guide, section 8.10.1

striving for ever greater precision:

“…when a login is conducted across a connection created through an incoming call”

Strategy – on "server"

- extend login capability across serial wire
- create user with a custom shell (shell = program auto-run upon user's login)
- write that shell to launch pppd into the serial wire

- then, pppd can be provoked on *this* machine from *that* one by logging into here, from there

Extend across wire login ability

vpnd	agetty ttyS0 115200 Problem: "If getty run from command line: Programs get stopped" Text-Terminal-HOWTO sec. 14.1 So do it from /etc/inittab, next slide
vpnb	

Extend login ability across wire

vpnd	edit /etc/inittab: `7:2345:respawn:agetty ttyS0 115200` init q
vpnb	

Create user w/custom shell

vpnd	useradd ppp edit /etc/passwd: `ppp:x:503:507::/home/ppp:`**`/etc/ppp/ppplogin`** or useradd –s /etc/ppp/ppplogin ppp
vpnb	

custom shell

Make shell launch pppd into wire

vpnd	edit /etc/ppp/ppplogin:
	`#!/bin/sh`
	`mesg n`
	`stty -echo`
	`exec /usr/sbin/pppd -detach /dev/ttyS0 115200 10.2.2.2:10.1.1.1 crtscts local silent`

vpnb	

Run pppd: by far side login

vpnd	
vpnb	minicom login to vpnd as ppp exit (crtl-A Q Enter) pppd /dev/ttyS0 115200 local

Check for new ppp interface

- ifconfig
- tail /var/log/messages

Automate remaining side (vpnb)

"A set of scripts automates the log in and PPP startup so all you have to do… is issue a single command to fire up your connection."

PPP HowTo

Strategy – on "client"

- automate "login:" and "password:" reponses we gave manually in minicom
- put them in a "chat" script
- run pppd to call the chat script (no more minicom)

Script local pppd invocation

vpnd	

vpnb	edit /etc/ppp/ppp-on-nullmodem:
	`pppd /dev/ttyS0 115200 local \`
	`connect /etc/ppp/ppp-on-login-nullmodem`

Script farside login

vpnd	

vpnb	edit /etc/ppp/ppp-on-login-nullmodem:
	`/usr/sbin/chat -v \`
	`'' '\r\r' \`
	`ogin: ppp \`
	`assword: vpn`

Run pppd invocator

vpnd	
vpnb	`[root@vpnb ppp]# ./ppp-on-nullmodem`

Check for new ppp interface

- ifconfig
- tail /var/log/messages

Biblio

- http://www.theory.physics.ubc.ca/ppp-linux.html
- comp.protocols.ppp newsgroup
- RFCs
  - 1547 requirements
  - 1661 PPP
  - 1471 LCP
  - 1332 IPCP
- http://www.ibiblio.org/pub/Linux/docs/HOWTO/PPP-HOWTO
- man page

Answer to the quiz

DUN is PPP, PPP is not DUN.