The New PPP Subsystem

As of linux-2.2.15 there were 3 different PPP implementations in the kernel. The asynchronous serial PPP driver, the synchronous PPP driver and the isdn_ppp module. Each PPP implementation is tailored for different situations, but there is a lot of duplicated code and common functionality.

The new generic PPP driver in linux-2.4 provides the functionality which is of use in any PPP implementation, including the networking interface, TCP/IP header compression, packet compression, the interface to the PPP daemon, the kernel facilities needed for demand dialing, and multilink bonding.

The generic PPP driver calls on the services of PPP ‘channels’ for sending and receiving frames. A PPP channel implementation can be arbitrarily complex internally but has a very simple interface with the generic PPP code: it merely has to be able to send PPP frames, receive PPP frames, and optionally handle ioctl requests. Currently there are PPP channel implementations for asynchronous serial ports, synchronous serial ports, and for PPP over ethernet.

The PPP channel interface deliberately does not include facilities for starting or stopping PPP channels. In general, each medium will already have a way of communicating with userland, which is used. Each medium will also have some actions to be performed at user level to initialize the medium for use as a PPP channel.

1. Notes

1.1. Original presentation

The original presentation of this talk occurred in room B of the Ottawa Linux Symposium, Ottawa Congress Centre, Ottawa, Ontario, Canada on the 21st of July, 2000 at 15:15 local time. This presentation was given by Paul Mackerras.

1.2. Presenter bio

Paul Mackerras is the maintainer of the Linux PPP implementation and the Linux port to the Power Macintosh. He works for Linuxcare, Inc. as a Senior Open Source Researcher, based in Canberra, Australia. Before joining Linuxcare, Mackerras was the leader of the CAP Research Program at the Australian National University, a program of research in parallel computing in collaboration with Fujitsu Laboratories, Japan. He has written numerous conference papers and reviewed journal articles.

Mackerras has contributed to several Opensource projects. As well as porting Linux to the Power Macintosh and rewriting and extending the Linux PPP implementation, he has contributed to the port of Linux to the AP1000+ distributed-memory multicomputer and to the development of the rsync algorithm and program. He has also ported NetBSD to a Motorola 68030-based system which he designed and built himself.

1.3. Presentation recording details

This transcript was created using the OLS-supplied recording of the original live presentation. This recording is available from ftp://ftp.linuxsymposium.org/ols2000/2000-07-21_16-54-48_B_64.mp3

The recording has a 64 kb/s bitrate, 32KHz sample rate, mono audio (due to the style of single microphone recording used) and has a file size of 20923200 bytes. The MD5 sum of this file is: 28569a4de4dfc65b3bd11393e974366f

1.4. Creation of this transcript

1.4.1. Request for corrections

This transcript was not created by a professional transcriptionist; it was created by someone with technical skills and an interest in the presented content. There may be errors found within this transcript; we ask that you report them to using the bug tracking interface described at http://olstrans.sourceforge.net/bugs.php3

1.4.2. Tools used in transcript creation

This transcription was made from the MP3 recording of the original presentation, using XMMS for playback and lyx (with docbook template) for the transcription.

1.4.3. Format of transcript files

The transcribed data should be available in a number of formats so as to provide more ready access to this data to a larger audience. The transcripts will be available in at least HTML, SGML and plain ASCII text formats; other formats may be provided.

1.4.4. Names of people involved with this transcription

This transcript was created by Jacob Moorman of the Marble Horse Free Software Group (whose pages live at http://www.marblehorse.org). He may be reached at roguemtl@marblehorse.org

The primary quality assurance for this document was performed by Stephanie Donovan. She may be reached at sdonovan@achilles.net

1.4.5. Notes related to the use of this document

1.4.6. Ownership of the content within this transcript

1.5. Markup used in this transcript

1.5.1. Time markers

At the end of each paragraph within the body of this transcript, a time offset is listed, corresponding to that point in the MP3 recording of the presentation. This time marker is emphasized (in document formats in which emphasis is supported) and is placed within brackets at the very end of each paragraph. For example, *05m, 30s* states that this paragraph ends at the five-minute, thirty-second mark in the MP3 recording.

1.5.2. Questions and comments from the audience

These recordings were created using a bud microphone attached to the speaker during their presentation. Due to the inherent range limitations of this type of microphone, some of the comments and questions from the audience are unintelligible. In cases where the speaker repeats the audience question, the question shall be omitted and a marker will be left in its place. Events which happen in the audience shall be bracketed, such as: The audience applauds.

Further, in cases where the audience comments or questions are not repeated by the speaker, they shall be included within this transcript and shall be enclosed within double quotes to delineate that the statements come from the audience, not from the speaker.

1.5.3. Editorial notes

The editor of this transcript, the transcriptionist (if you will), and the quality assurance resource who have examined this transcript may each include editorial notes within this transcript. These shall be placed within brackets.

1.5.4. Paragraph breaks

The paragraph breaks within this transcript are very much arbitrary; in many cases they represent pauses or breaks in the speech of the speaker. In other cases, they have been inserted to allow for enhanced clarity in the reading of this transcript.

1.5.5. Speech corrections by the speaker

During the course of the talk, the speaker may correct himself or herself. In these cases, the corrected speech will be placed in parenthesis. The reader of this transcript may usually ignore the parenthised sections as they represent corrected speech. For example: My aunt once had (a dog named Spot, sorry) a cat named Cleopatra.

1.5.6. Unintelligible speech

In sections where the speech of the author or audience has been deemed useful, but unintelligible by the transcriptionist or by the quality assurance resource, a marker will be inserted in their places, unintelligible. Several attempts will be made to correct words and phrases of this nature. In cases where the unintelligible words or phrases are clearly not of importance to the meaning and understanding of the sentence, they may be omitted without marker insertion.

2. Transcript

My name is Paul Mackerras. I am the PPP maintainer for Linux. I work for Linuxcare, Inc. I'm going to talk today about the new PPP driver architecture in the Linux 2.4 kernel. I've got a bad cold, so I can't speak very loud. So if you can't hear me, can you come forward? *00m, 23s*

Okay. So this is the outline of the talk. First of all, a quick overview of PPP. Then a quick look at how PPP is implemented in the 2.2 kernel, and what's wrong with that. Then a description of the new interface that we've designed for 2.4; the changes we've made to the Linux pppd. And lastly, just a little, quick thing about how you can use multilink PPP under Linux with 2.4. *01m, 01s*

PPP, I'm sure you all know, is used to establish network connections over all different kinds of point-to-point links, most commonly, dial-up modems. A lot of people use PPP over dial-up modems to their ISP; also, various kinds of synchronous serial links, ISDN links, and all sorts of other things, optical fibers, ATM, lots of things. PPP is really a collection of protocols defined by RFC documents. And the most important ones here, firstly the Link Control Protocol (LCP), used to establish the link between two peers and configure various aspects such as: Do you have to authenticate? Do you want to escape control characters? What's the largest packet size you can send? *01m, 53s*

Then, two authentication protocols, Password Authentication Protocol (PAP) and the Challenge Handshake Authentication Protocol (CHAP). They're used for one machine to prove to the other machine that it is actually who it says it is, and that it's allowed to make a connection, or whatever. *02m, 10s*

The Internet Protocol Control Protocol (IPCP), that's used used to configure things like what IP addresses you're going to use on the two ends of the link. The Compression Control Protocol (CCP); that's used to negotiate whether you're going to compress packets that are sent over the link. Now of course if one side is going to compress a packet and send it compressed, the other side needs to have a decompressor, so the CCP lets the two sides discover if they have any protocols in common that they can use for compressing packets. And lastly, the multilink protocol; that provides a way of using several physical links in parallel to make a single virtual PPP link, where you can actually split packets between the links, and so get reduced latency and increased bandwidth. *03m, 02s*

Okay. Now in the 2.2 kernel, the most notable thing about PPP is, that we have three or four separate PPP implementations that don't share any code and you use the implementation for your kind of medium. So if you've got an async tty, like a dial-up modem or something, you use the async PPP. If you've got an ISDN line, you use the ISDN PPP, which is a completely separate lump of code. *03m, 31s*

Or for some particular kinds of synchronous serial interfaces, you use the sync PPP implementation. And then of course, there are some smart Wide Area Network adapters that actually do PPP in the firmware on the board. Now I'm not going to talk about them, because that's not really a PPP implementation as far as the kernel is concerned. As far as the kernel is concerned, it just looks like an Ethernet interface or something, so we're not going to worry about them. *04m, 00s*

Now, the kernel/user mode split. Basically the principle is that we have enough stuff down in the kernel that we can handle data packets completely, without going up to user space. So that includes the packet encapsulation of framing, compression, TCP/IP header compression, and if you're doing multilink, it includes the multilink fragmentation reassembly. Now the reason for doing this is basically for performance. If you've really only got a 14.4 Kbps modem, then it doesn't really matter if you're coming up to user mode for every packet, but if you've got an ADSL line that's going at a couple megabits, then it does make a big difference. *04m, 50s*

Then, up at user level, we have a daemon that handles (the control protocols) the PPP control protocols that I mentioned before. For the async PPP, you use the standard PPP daemon (pppd). For ISDN PPP, there's a derivative that's been forked off, called ipppd. *05m, 13s*

Now if we look at the async PPP, which is the one that I maintain, this is the structure. So the PPP module in the middle here has actually got two parts, two sort of personalities. One is that it looks like a network interface... You've got this part on the right that looks like a network interface; a ppp0, or whatever, network interface. And the part on the left looks like a tty line discipline. *05m, 51s*

A line discipline is a module of code that you can put in between a serial port or a pseudo-tty and the user level code that's reading and writing from that serial port. So for instance, when you're using a console, you've got a virtual terminal there and it's in a line discipline which gives you the line editing and echoing characteristics, so that when you type characters, they echo; when you press the delete button, it removes characters. That's done by the line discipline. *06m, 20s*

Now if we have a device which we want to use for PPP, we take off that standard, normal line discipline and we put on the PPP line discipline. What that means is that this code, this PPP line discipline code, gets firstly, to be able to send characters out, and secondly, to be able to process the received characters. And for data packets, it doesn't send them up to the ppp daemon, it shunts them sideways to the network interface. *06m, 46s*

Now this is fine for the normal sorts of situations where you have a dial-up modem, you're dialing up your ISP; it all works fine. Where it becomes restrictive is, firstly, if you want to do multilink, because if you want to handle another tty, then there's another one of these things and suddenly, you've got two network interfaces. And you don't want two network interfaces because you only want one virtual link. You only want one network interface, but its using the two devices. Secondly, it makes it harder to share code with, say, an ISDN port, or a synchronous serial port, or something. *07m, 32s*

So in doing the 2.3 series kernels, I basically threw out that code and started again, dividing the PPP stuff into a module that handles all of the common functions that are used for just about any PPP implementation on any medium that you could care to name. And then below that, we connect in modules that we call channels, which handle the specific details for different sorts of medium. *08m, 02s*

So, in the generic module, we have network interface unit, the ppp0 or whatever; we have the interface to the networking code and the interface to the PPP daemon. We have the packet compression and decompression, and the TCP/IP header compression and decompression. We have the code for detecting network traffic, so we can do idle timeouts and demand dialing. And we have the code for multilink PPP, the fragmentation and reassembly. Now all of those things are things that are independent of whether you've got a serial port, or an ISDN connection, or whatever, underneath. *08m, 46s*

Then, underneath we have different kinds of channels. These handle specific details (like); for instance, when you're doing PPP encapsulation on an async serial line, you have to do things like; if there are certain byte values, you have to escape them by putting in 7D byte, and then a mangled version of the byte; and you have to put in a 7E byte at the end and possibly the beginning as the frame delimiter. Now if you're doing PPP over an HDLC link, then those details of framing and transparency are handled, usually by the hardware, in lower levels; you do bit stuffing rather than byte stuffing and you have various bit combinations that are used for the frame delimitation and that sort of thing. *09m, 38s*

So we have a PPP async channel that knows how to deal with tty devices; a PPP sync tty channel for some kinds of synchronous serial ports; a PPPoE channel, that's used for PPP-over-ethernet. It's quite common these days for people to get an ADSL connection from their telco or whatever, that comes in and it looks like an Ethernet, but then you do PPP over that, so the provider can then do all of the access control and whatever. So then, in that case, we have PPP frames that are being transmitted over Ethernet. And that come in and then go in to the networking stack. *10m, 26s*

And finally, an interesting one is that someone has written an IrPPP channel type. What this does is, it takes the PPP frame that it's been given to transmit and it encapsulates it according to the IrDA specifications and sends it out the infrared port. So with that, if you have laptops or something that have IrDA ports, you can set up a PPP connection. Now you can, of course, do the tty emulation over the IrDA port and then do PPP over that, but that gives you multiple levels of framing that you don't really need. With IrPPP, you can just do it, more or less directly, in to the IrDA frame format. *11m, 13s*

Okay, so here's a diagram showing more or less how this works. We have the generic layer in the middle and we have an instance of it that is basically a network interface, ppp0 or whatever, and the TCP/IP stack, or whatever, talking to that. Then underneath, we have here, I've shown two channels linked in to the same PPP unit. And in this case, they're both PPP async channels talking to tty devices. So this is an example of how we might do multilink. We have packets coming in to the generic layer, are getting fragmented, sent out both of these channels, and are coming out both of these tty devices. Fragments coming in, come in through the channel, in to the generic layer, get reassembled, and get passed to the network stack. You see that the PPP async channel implements a line discipline. That is because that is the way to get hold of the data from the serial port. *12m, 22s*

One thing we can do here, of course, is we can do a kind of heterogenous multilink. We can say; perhaps this channel one is an ISDN connection. Or maybe your ISDN gives you two B channels, so you can have two. One example where this might be useful is if you have a situation where perhaps your local calls on an analog line are untimed, but your ISDN calls are subject to time charges. So what you can do is, you can have a dial-up modem which is basically on permanently, which gives you your basic connection, and then when you need more bandwidth, you can bring up your ISDN channels and just connect them in, and get the extra bandwidth. And when you no longer want them, you can disconnect those channels and go back to just using your dial-up modem. *13m, 14s*

Okay, here's the details of the interface between the generic layer and a channel. It's actually, really quite straight forward. All that the channel really has to do is, it has to be able to take a PPP frame in an skbuff structure and send that out on its channel; and it needs to be able to receive PPP frames on its channel, put them in an skbuff structure and pass them up to the generic layer; and that's about it. *13m, 46s*

So the channel has to provide two functions to the generic layer. One is the start_xmit() function which says; here is a PPP frame, send it. Now in fact, the channel is actually allowed to say when a start_xmit() function is called. It's allowed to say; sorry, I'm filled up at the moment, I can't take it. In that case, the generic layer will actually keep hold of the frame and send it again later. That's what this ppp_output_wakeup() function is for. That's for the channel to tell the generic layer; okay, I can take another frame now. And then, usually the generic layer will call the start transmit function again with the frame and say; send it now. And optionally the generic layer can provide an I/O control (ioctl) function which can then be called by pppd down through the generic layer in to the channel. *14m, 45s*

There are seven functions in the generic layer that the channel can call. The first, ppp_register_channel(); that's called when a new channel has sprung in to existence and the generic layer should know about it. Now the generic layer doesn't create channels; something else creates channels. So, for instance, in the PPP async case, a channel is created when a tty is set in to the PPP line discipline. *15m, 15s*

In the PPPoE case, I haven't looked at it in detail, but I would guess that a channel is created when some user level thing detects a UDP connection and packets that say; right, we want to set up a PPP connection here. So it's a general principle that whatever medium you have underlying the channel, there has to be a way to get at that and control that, independently of the PPP stuff. *15m, 50s*

So, think about a tty device. You can open the tty device; you can set its speed; you can then send modem commands; you can then do various things to get that thing ready to talk PPP. Then you make it a PPP channel and the PPP generic layer starts talking to it. Or for PPP over Ethernet, you've got your Ethernet there and what happens is, you start receiving packets from the other end that say; we want to set up the connection, or whatever. Or you run a user mode thing that starts it from your end. In any case, there's something that happens there and gets things ready to do PPP. And then, when its ready to do PPP, you connect it in to the PPP generic layer. *16m, 38s*

Okay, so ppp_register_channel() says; right, we're now ready to do PPP, here's the channel, go for it. ppp_unregister_channel() says; oops, sorry, can't use this channel any more, gone. So for instance, if you have a modem and then it hangs up, that would result in ppp_unregister_channel() being called and from then on, the generic layer won't try and use that any more. *17m, 07s*

ppp_output_wakeup(), I've already talked about. ppp_input() is very simple; it means the channel has a frame, here it is, go for it. ppp_input_error(); that's called if your channel has detected that you've dropped a frame. For instance, if the channel has got a frame and has checked the frame check sequence, the CRC at the end of the frame, and that's bad and it's dropped the frame, then it should let the generic layer know that its dropped the frame. *17m, 35s*

The last two are quite simple. ppp_channel_index(); each channel has got an index assigned by the generic layer when ppp_register_channel() is done and pppd needs to know that index, so this is the way that you get that index, pass it back to pppd and that way, then, pppd can actually connect in to it. ppp_unit_number(); that just says, if the channel is connected to a PPP unit, like a network device, what's the index of that. *18m, 14s*

Multilink becomes relatively straight forward when we do things this way because we can connect more than one channel to one PPP unit. So then the ppp unit, like ppp0, that represents what's called a bundle in multilink terms. A bundle is like the virtual PPP link. And the PPP channel; each channel represents a single physical link. The generic layer handles the details of fragmenting and recombining the PPP frames. And of course, your channels can be of different types. *18m, 50s*

Okay. One of the issues in designing and implementing the PPP generic layer was to handle the details of really trying to keep the amount of transmit buffering to a minimum. It's best to keep the amount of buffering to a minimum to minimize latency (and to make the); if the high-level networking code is making decisions about prioritization and reordering of packets based on priority, then you don't want to have a lot of buffering below that, because that means the reordering becomes less effective. *19m, 28s*

So starting from the top, when a packet is transmitted, we have a transmit queue for the PPP unit, that usually contains, at most, one packet. It can, under some circumstances, contain more than one packet. The (start_transmit() function) the hard_start_transmit() function of that network interface device always accepts the packet that the high-level networking code gives to it, but then there are flow control things there; there's the netif_stop_queue and the netif_start_queue primitives in the networking layer that the PPP generic layer uses to flow control stuff in to its queue. *20m, 08s*

Coming down from the transmit queue, we then do the packet compression. The important thing here is that after you've done packet compression, you can't re-order the packets anymore. You have to send them out in the same order. Also, it becomes highly undesirable to drop packets, because what happens with compressed packets is that each packet is compressed based on the history of the preceding packets. So that if the receiver receives a packet and you know, receives packet n, doesn't receive packet n+1; then when it gets packet n+2, it basically can't handle it, it can't process it; it has to drop it, because to decompress it, it needs the history from the previous packets. So in that case, there is a mechanism for resetting and recovering, but that involves round trips and that gets very slow and your performance goes to zero. *21m, 00s*

Okay, so after this point, we can't re-order and we really, really don't want to drop packets. Once a packet's been compressed, we try to send it out on one or more channels. Now if we're not doing multilink, basically we give it to the channel. If we're not doing multilink, we only have one channel, so we give it to the channel and if it takes it, good; if it doesn't take it, we just save it for later. If we are doing multilink, then at this point, we fragment the packet and we have a channel transmit queue for each channel, which, once again, usually contains, at most, one packet. And that's if the channel refuses the packet fragment; that's where we store the fragment for later. *21m, 49s*

So really there should be, at most, one or two packets buffered up on the transmit side of the generic stuff at any time. This is all designed so the channel, in fact, really only needs to store one packet, which is the packet it's sending. This is why I made it so the channel can refuse to take a packet in the start_xmit() routine, so that you don't really need any sort of a queue in the channel; you just need a place to store the one you're working on. And if you're not working on one, that's fine; and if you are working on one, then you refuse any more at that stage. There's no logic for timing out and retransmitting packets in the generic layer; that's all handled higher up in the core net code. *22m, 45s*

The generic layer has been designed fairly carefully and has been tested to be SMP-safe. Obviously, we use spinlocks to ensure the integrity of our data structures. In order to achieve this SMP safety, we have to ask the channels to provide some amount of synchronization and mutual exclusion as well. And the generic layer can, in turn, provide some guarantees about how things work. So the channel is required to guarantee; when you call ppp_register_channel(), the channel provides a PPP channel object, which represents that channel. Now the channel is required to guarantee that that actually stays in existence; that that memory doesn't get freed from the time you call ppp_register_channel() until after ppp_unregister_channel() returns. Now on an SMP system, where you could have multiple threads, this might require a spinlock. *23m, 56s*

The channel is also required to guarantee for us that it's not going to be calling ppp_input() in one thread, at the same time that it's calling ppp_unregister_channel() in another thread. Now there are reasons, if you think about it, you can't guarantee the existence of an object with a lock inside the object; that's why I didn't put a lock inside the object to guarantee this. But that sort of thing will typically require a spinlock, or some sort of guarantee from the stuff calling the channel code, to achieve this. *24m, 37s*

The generic layer functions shouldn't be called from hard interrupt level; they can be called from soft interrupt or bottom half level, or from main line, but not from hard interrupt level. The generic layer gives a guarantee to the channels that it won't re-enter the start_xmit() routine or the ppp_ioctl() routine, and it won't call one while the other is executing. So inside the start_xmit() routine, you can presume there's only one thread in there. You're not going to have two threads, both simultaneously, in your start_xmit() routine, two CPUs. The generic layer also guarantees that, by the time ppp_unregister_channel() returns, it won't be calling the start_xmit() or ppp_ioctl() routine on another CPU. *25m, 36s*

Now on the top side, where the generic layer talks to the PPP daemon, the way it does that is through /dev/ppp. This is a character device and there's only one. There's only one /dev/ppp. The way we use this to talk to multiple units and channels is each time you open /dev/ppp, you get a separate instance; it's a bit like the Solaris clone open type thing. You get a separate instance of /dev/ppp, which you can then attach to a unit or to a channel. *26m, 13s*

So you can open /dev/ppp and say; I want to attach this to ppp2 and then you can do ioctls and send and receive frames and things, using that file descriptor that you got from the open, and that will talk directly to ppp2. You can also attach it to a channel, so once you know a channel number. Ahh, 27... That might be, you've just taken ttyS0 and put it in to PPP line discipline, that made a new channel, so that's channel 27. You can open /dev/ppp, attach it to channel 27 and then control that channel through that file descriptor. Obviously you can send PPP frames by writing to /dev/ppp and you can receive them by reading from it. *27m, 00s*

There are also a number of ioctls; you can create a new PPP unit, do the attach like I said. You can attach to a channel and then say; connect this to ppp2, so you can take your file descriptor that you attached to channel 27 and say; right, connect yourself to

PPP unit 2. That way, that gives you the connection and gives the generic layer the way to actually send and receive frames. And then you can use ioctls to set various types of parameters, like control character escaping and maximum frame size, and that sort of thing. *27m, 37s*

So, the ppp daemon does all this; you don't really need to worry about all of this unless you're writing a new PPP daemon. And there are some people who want to do exactly that, and that's great. So there it is. *27m, 55s*

I'm up to ppp-2.4.0b4 now and beta 5 is on its way, but from beta 2 and later, they will support the new generic layer, the new structure in 2.4. At the moment, we have the PPP async and PPP sync tty channels and the PPPoE stuff is actually in the kernel. pppd doesn't have the PPPoE stuff in it yet; there's a bit more restructuring that's going to need to be done to get that in cleanly. (I think you have patches for the moment, for b4. b2? b3? b3 didn't last very long; it had a bad bug.) *28m, 41s*

pppd has this plug-in architecture where you can add new code that's packaged up as a shared library, stick that in to the pppd executable and add new functionality that way. I'm currently hacking in to pppd to make it possible to support a new type of channel, using a plug-in. So then what you'd do is you'd have your new channel type and perhaps you'd have that as a kernel module; you'd load that kernel module, you run pppd with the corresponding plug-in and you can talk to your new type of channel. *29m, 20s*

What this plug-in... the sorts of things this plug-in would have to do are firstly, talk to your magic newt. Okay, suppose you've got some new packet radio interface which has got some particular characteristics like, you can't receive and send at the same time, and there are some other funny quirks that you need to cope with. Okay, so you write the kernel channel module that knows about all that; does the encapsulation and framing, does the flow control, does the alternation of transmitting and receiving, and so forth. Then at the user level in pppd, what you need is firstly, something to get your new channel ready to do PPP. *30m, 04s*

So you need something to, for instance, turn on the radio transmitter and do some fancy control stuff to actually find someone out there on the other side of the world who wants to talk with you on your packet radio interface. So then you're ready to do PPP, so that's fine, the rest of the core pppd code can take over and do PPP stuff. And then at the end, your channel plug-in needs to do things like, it needs to know how to stop doing PPP and say goodbye to the other guy and turn off your radio, or something. *30m, 48s*

The multilink support; the basic functionality is there and working, but it still needs more development. The way it works at the moment is that we have one PPP process per link and we use a TDB, Tridge's Trivial Database, which he wrote for Samba. That's used to match up links and bundles. At the moment, the first pppd that you run will set up the first link and also control the bundle and then you run another pppd for the second link and that will join on to the first one. Now, if you kill the second one, that's fine; that will take down the second link and you'll revert to just using one. But if you take down the first link, at the moment, it will take down the whole connection. That is undesirable. So that is something that needs to be fixed. *31m, 40s*

So to set up multilink PPP, you need a recent 2.4 kernel. You need a recent pppd. And the basic idea is very simple; you just connect to the remote machine twice, or three times, or however many times. The separate invocations of pppd are, actually, virtually identical. Usually, it will just be the device name that differs, so you're using a different serial port or whatever. You have to use the multilink option, of course, to tell pppd that you want to do multilink. *32m, 22s*

Now what happens is that there's this thing called an end-point discriminator; this is a magic blob of data that is hopefully unique between different systems. So a good one to use, and the one that pppd uses by default, is actually the MAC address on your first ethernet, provided you've got an ethernet. And during the multilink negotiation, the two ends exchange their own endpoint discriminator. *32m, 50s*

Then what pppd does is (it looks for) it says; okay the other guy has got this discriminator, he's perhaps authenticated himself as this machine, or user, or whatever. And it uses that information and matches that in the database to see; have we already made a connection to that same system? If we haven't already made a connection to that system, we just set up the first link; we just set up a new bundle. If we do find that we have already set up a connection to that system, then we say; okay, this is the second or subsequent link in a multilink bundle, and we'll just join our link on to that existing bundle. *33m, 28s*

Now it's possible, I guess, that there would be some cases where the discriminator is pretty weak or the other end doesn't authenticate, and so what you can do is you can specify a third thing, a bundle name, with an option to pppd, that becomes the third part of the key that's used to look up in the database. So you can actually use that to establish two separate bundles to the same system, or to make sure that when you are connecting to different systems that are using the same end-point discriminator that they're actually different bundles. *34m, 06s*

So when pppd detects that the other system is the same system as one we've already connected to, it doesn't create a new pppd unit. The first one would create, say, a ppp0; the second doesn't create a ppp1, it just joins its on to ppp0. *34m, 37s*

There are some pppd options related to multilink, new options: multilink, no-multilink; bundle, that's the one I just talked about that lets you discriminate between bundles; mrru, (that's) you know the mru option for pppd is basically the maximum packet size, this is the maximum packet size for the whole bundle, which can be different from the maximum packet size for the individual links. Note: Most pppd options are not preceded by - or –, as is commonly done with most programs. *35m, 06s*

In the multilink specification, there's two different sizes of sequence numbers that you can use, 24-bits or 12-bits, and corresponding to that, the multilink header either takes either 4 or 2 bytes. If you use the mpshortseq option, that will tell pppd that it's allowed to use 12-bit sequence numbers and to ask the other end if that's okay too. And the endpoint option lets you set what the local end-point discriminator is going to be. If you don't use that option, then it will look for eth0 and take the MAC address on that. *35m, 39s*

And as I said, there's now this /var/run/pppd.tdb; that's a database of connections that contains more information than just the things I was talking about before. It actually contains details like what the process ID of pppd is and what device it's using, and all sorts of things. That's for all pppd's that are running, so it's possible now to write utilities that query that database and print out a list of connections; print out a list of users that have got connections in to your system, perhaps, and what IP addresses they're using, and various kinds of things like that. *36m, 21s*

Future work; the first one reminds me of the verse in the Lord of Rings about one ring to bind them all, one ring to bring them. The speaker laughs. I'd like to bring all of the pppd, all of the PPP implementations together to use the generic layer, so I need to talk to the ISDN guys and also, perhaps, the syncppp maintainer if such person exists, and see if we can't restructure their code to use the generic layer and if there's changes needed to the generic layer, that's cool. *37m, 08s*

As I said before, in the process of restructuring pppd to support diverse channel types, so that you can write your channel module for your packet radio and your plug-in for pppd and that works fine. And last thing, one thing I've wanted to do for a while is to actually provide a way you can make a socket connection to pppd, and connect to it from a GUI and query it for status information and perhaps give it commands to take the link up or down, or sideways, or whatever. Okay, any questions? *37m, 43s*

The speaker calls on an audience member for a question. The question is omitted as it is repeated by the speaker.

The question was, do I do load balancing over multilink? Yes, it does balance the load over the channels with multilink. The way it works is that when it's got a packet to transmit, it looks to see which channels would be available to transmit a packet; and the way it does that is, it looks at the transmit queue for the channel, and if it's empty, then it says; okay, we can give that channel a fragment. And then it divides the packet between all of the channels that are free at the moment. *38m, 18s*

Now, what this tends to do is that, under conditions of light load, usually most of the channels will be empty, so you'll tend to fragment the packet over all the channels. And that's good because it gives you reduced latency, because the fragment is shorter on each channel. Then when you get under heavy load, very often, there'll only be one channel free and that tends to give you less fragmentation. So you're tending to put whole PPP packets out each channel and using them in turn. Now if one channel is a lot slower than others, then it will take a lot longer before it becomes free again and you'll be tending to put packets out the other channels. So that's how the balancing is done. *39m, 04s*

The speaker calls on an audience member for a question. The question is omitted as it is repeated by the speaker.

The question was, does the multilink stuff have the provision for transmitting packets without a multilink header when using multilink? Okay. Yes. The answer is basically yes. Not data packets, but control packets, yes. On data packets, no. You can transmit packets by having an instance of /dev/ppp that's attached to the channel. If you write a packet to that channel, it will go directly to that channel without any processing at all; without any compression or multilink header; it will just go to the channel as-is. So that's how you can transmit a packet without a multilink header, but not data packets. Why would you want to do that? *40m, 04s*

The audience member who asked the question responds.

I'll talk to you about it later.

The speaker calls on an audience member for a question. The question is omitted as it is repeated by the speaker.

The question was, is there a risk that the multilink PPP will reorder packets? If you read the multilink spec, (RFC1717, no that's the old one) RFC1990, the multilink header contains a sequence number and that sequence number is incremented for each fragment that goes out. And part of the reassembly process is actually to make sure that the fragments are processed in increasing sequence number order. So that makes sure that, even if a packet arrives faster on one channel than the other, (that) the first fragment or packet gets processed first at the receiver. *40m, 53s*

The speaker calls on an audience member for a question.

So basically if you have one very slow link and one very fast link, then the fast link will need to wait for the slow link because they have to process packets.

Ahh. The processing of the packets on the fast link would need to wait until the packet had fully come through on the slow link; that's true. And, in fact, there are situations where if you've got a fast link and a slow link, with TCP, perhaps sometimes the total bandwidth you will see will just be twice the bandwidth of the slow link; that can happen. *41m, 28s*

But it should still be possible to use the bandwidth of the faster link so that several packets will go through on the faster link and they'll just get queued up and then the one on the slow link eventually gets through; it will get processed and then all of the ones on the faster link will get through. So, it's possible to get the full bandwidth, but then you get interactions with TCP round trip time (RTT) estimation and retransmission, and all that sort of TCP nastiness. *42m, 04s*

The speaker calls on an audience member for a question. The question is omitted as it is repeated by the speaker.

The question was, how difficult would it be to get PPP working over a USB connection. Now do you mean a USB serial port? Okay. I think the USB serial stuff actually creates a tty device, doesn't it? I'm pretty sure that's right, that when you have a USB serial port, that the USB code actually creates a tty device and then you can just run the standard PPP stuff on that serial device. *42m, 41s*

Some people have asked in the past about connecting USB to two computers and using that as a kind of network between them. Now, in fact you need, because USB is strictly a master/slave sort of bus, the host is the master and controls everything on the bus, you can't just connect a USB cable between two computers. But you can get boxes that have got a bit of RAM in them and two USB slave connections so that one computer can write some stuff in to the RAM and another can read it. If you had that sort of thing, you could do PPP over that, by creating a channel type that knew how to send the USB commands to actually read and write to that RAM. *43m, 26s*

Questions? Okay. Thank you.

The audience applauds. The talk concludes. *43m, 35s*

3. Additional Resources

3.1. pppd

The latest versions of pppd for Linux are available from ftp://ftp.linuxcare.com.au/pub/ppp/

3.2. RFCs Related to PPP

RFCs are available from a number of places, including ftp://nis.nsf.net/internet/documents/rfc

Of particular relevance to PPP are: RFC1661 (The Point-to-Point Protocol), RFC1990 (The PPP Multilink Protocol), RFC1962 (The PPP Compression Control Protocol), RFC2153 (PPP Vendor Extensions), RFC1332 (The PPP Internet Protocol Control Protocol), and RFC1994 (PPP Challenge Handshake Authentication Protocol).