Reprinted from the

Proceedings of the Linux Symposium

Volume Two

July 21th–24th, 2004
Ottawa, Ontario
Canada

TCPfying the Poor Cousins

Arnaldo Carvalho de Melo
Conectiva S.A.
`acme@conectiva.com.br`
`http://advogato.org/person/acme`
`http://www.conectiva.com.br`

Abstract

In this paper I will describe the work I am doing on the Linux networking infrastructure, with emphasis on cleaning the code, but with important "side effects" like reduction of core structures already saving over 600 bytes on UDP sockets all over the net in 2.5/2.6 (tcp, etc.), elimination of data dependencies, reduction of the non-mainstream network families maintenance cost by making them use code that now is in `net/ipv4` but can be moved to `net/core`, leaving only the really ipv4-specific code and making LLC use it as a proof of concept (work done in my net-exp tree, pending submission).

TCP code becomes used by the poor cousins, they appreciate that!

1 How This Started

Making IPX uptodate with regards to advances in the core networking infrastructure, to kill `deliver_to_old_ones`, i.e., special cases in the core kernel for protocols that hasn't been converted to shared skbs and multithreading.

In the process I noticed several areas where code was replicated or used a different, older framework, due to the evolution of the core networking infrastructure.

Also de experience of porting the NetBEUI and LLC code released as GPL by Procom Inc. from the 2.0 Linux kernel networking infrastructure to 2.4 and then to 2.5/6, working on a BSD sockets API for `PF_LLC`, initially contributed by Jay Schullist was instrumental in realising the existing similarities in the infrastructure needs required by several protocol families.

2 TCP/IP Evolves Faster

Most of the attention is given, of course, to TCP/IP, and in the process new infrastructure is created, with TCP/IP using it at first and sometimes leaving things like the `deliver_to_old_ones` function to simulate the previously existing big networking lock and the `SOCKOPS_WRAPPED` macro, to allow the other protocol families to continue working, hoping that their maintainers do the necessary work, but this sometimes doesn't happen for a long time.

In other cases code is added to TCP/IP that, upon further inspection, could be moved to `net/core` and be useful for the other protocol families.

Doing this factorization will help make these improvements to TCP/IP be taken advantage of by the other protocol families and will help in realising the ultimate goal of keep the protocol families code with just what is completely specific.

3 Trimming `struct sock`

In 2.4, `struct sock` has a big fat union that has most of the private data for each protocol family, so when any change had to be done to a specific protocol family the layout of `struct sock` would change, generating unnecessary recompilation of most of the network related code in the kernel.

In 2.6 this has changed and `struct sock` nowadays is mostly free of details specific to network protocol families.

In the process two ways were devised to store the network protocol private area, one for protocols that have stringent performance requirements, like TCP/IP, using per-protocol slab caches and another one, simpler, that allows protocol families to just allocate a chunk of memory and store its pointer in the `struct sock` member `sk_protinfo`. As most stacks now use helper macros to access its private area, the eventual switch to the slab cache approach is easily done.

With this in place the footprint of the `struct sock`, that was of about 1280 bytes on a UP machine in 2.4 to 308 bytes for the generic sock slabcache in 2.6, with the `tcp_sock` slabcache using 1004 bytes, `udp_sock` slabcache using 484 bytes and finally the `unix_sock` (`PF_UNIX` sockets) using just 356 bytes.

This changes also resulted in a performance gain in the establishment of connections, as was verified with the `lmbench` tool.

Another related change was to diminish the data dependency among `struct sock` and `struct tcp_tw_bucket`, that is a "mini socket" used to represent TCP connections in the `TIME_WAIT` state. To accomplish this, `struct sock_common` was introduced, that is the minimal required set of members common to these structs. With this data layout we will certainly avoid bugs introduced when changing only one of the structs, like has happened at least once to my knowledge.

4 Using `list.h` in the Networking Code

With the advent of the hashed lists (`struct hlist_node`) it turned out to be useful to make the networking code follow the general kernel trend of using the `linux/list.h` macros, replacing the ad-hoc lists present in the networking code.

The work consisted of introducing a set of helper macros to handle `struct sock` list handling, namely `sk_add_node` and `sk_del_node_init`, and bind list variants.

These functions also bump the reference count for the socket, something that was not being done by some protocols, that have since been converted to use this new set of helper macros, thus fixing some bugs in the process.

It should also be noted that `hlist` started using prefetch as part of the process of convincing David Miller, the Linux Networking maintainer, to accept such changes. Performance gains are an important technique in getting code-cleaning patches accepted.

5 Socket Timers Manipulation Helpers

Another area that received attention was the socket timers manipulation routines, that in some protocols aren't always bumping the reference count as they should and do in the Bluetooth and TCP/IP code.

To abstract this handling the `sk_reset_timer` and `sk_stop_timer` functions were introduced recently to do the `timer_list` handling and deal with `struct sock` reference counting.

6 Factorization of net/ipv4 Code

In the past Alan Cox worked on having datagram code that could be shared among several network families shared at the `net/core/datagram.c` file, moving chunks of code out of the UDP implementation.

Now with this work I'm trying to do the same with the stream code, now moving chunks of TCP code to the core infrastructure.

Initial steps are just moving code around, like `tcp_eat_skb`, that became `sk_eat_skb`; `tcp_data_wait` became `sk_wait_data`; and here we see something interesting, namely the fact that this function correctly sets the `SOCK_ASYNC_WAITDATA` bits in the `struct socket` flags member, something that some protocols aren't doing now but will as soon as they start using `sk_wait_data`.

In my `net-experimental` tree I have introduced some new members to the `struct sock` member `sk_prot`, allowing both TCP and LLC to use common `stream_sendmsg` and `stream_sendpage` functions, that are generalizations of `tcp_sendmsg` and `tcp_sendpage`. Further work is needed to fully determine the performance implications of such changes, but no noticeable performance drop or stability problems have been verified in using this patched kernel in my main machine for over a month.

7 BSD Sockets Layer

There is some duplication of work at the BSD sockets level among the network protocol families implementation. Trying to reduce the code required to implement a protocol family is being investigated, with some proofs-of-concept already implemented, where the functions now used for TCP/IP are being shared with LLC.

The idea here is to to reduce the protocol-specific implementation to just that, i.e., what is absolutely specific to each protocol.

Perhaps this will make it easier to stack protocols, allowing combinations that are possible in other kernels but not on Linux right now.

The extra function pointers in `sk->sk_prot` probably won't be a problem because they will make it possible to eliminate `sock->proto_ops` by calling directly the `sk->sk_prot` functions.

8 Future Developments

With this newly common infrastructure, it may be possible to add features like network async I/O to all protocols. More sharing will be investigated, trying to avoid pitfalls that appeared in similar work done in other kernel subsystems.

9 Conclusion

Looking every other year at how core infrastructures evolve and how the implementations of subsystems attached to those infrastructure evolve is something that should be done, paying off in terms of code clarity, reduction of the cost of maintaining code that has come out of mainstream but are still used in lots of legacy setups.

Another eventual benefit gained is the performance one, as making the code clear and more general is not incompatible with having fast code.

Reuse the code, Luke.

10 Acknowledgments

I'd like to thank David S. Miller for all the support he gives me in continuing this work, reviewing my patches, and providing much valued words of wisdom. And my respect for Andi Kleen, for helping me out in my childhood as a Linux networking wannabe hacker, Alan Cox for throwing me the Procom NetBEUI stack, that was fun! And nasty as well. As well as all the fine kernel networking hackers that spare some of their time to comment on my ideas.