Pseudo-Network Drivers and Virtual Networks

*S.M. Bellovin**
smb@ulysses.att.com

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Many operating systems have long had *pseudo-teletypes*, inter-process communication channels that provide terminal semantics on one end, and a smart server program on the other. We describe an analogous concept, *pseudo-network* drivers. One end of the driver appears to be a real network device, with the appropriate interface and semantics; data written to it goes to a program, however, rather than to a physical medium. Using this and some auxiliary mechanisms, we present a variety of applications, including system test, network monitoring, dial-up TCP/IP, and ways to both improve and subvert network security. Most notably, we show how pseudo-network devices can be used to create *virtual networks* and to provide encrypted communications capability. We describe two implementations, one using a conventional driver for socket-based systems, and one using stream pipes for System V.

1. INTRODUCTION

Many operating systems have long had *pseudo-teletypes,* inter-process communication channels that provide terminal semantics on one end, and a smart server program on the other. In the same vein, we have implemented a *pseudo-network* driver. To the kernel, and in particular to IP, it appears to be a device; instead of transmitting the bits over a wire, the output packets are sent to a program. Similarly, packets written by the program are delivered to the network input handlers, exactly as if they were received over a real device. The general flow of control is shown in Figure 1.

IP (or another network protocol) hands packets to the bottom half of `Pnet`; the top half of the driver passes them to a server program, which can communicate with other servers. Similarly, the server can generate packets and pass them to the driver; these are in turn sent to IP.

There are two general implementation techniques available. For socket-based systems, such as SunOS and 4.3bsd, we have implemented a standard network device driver; a detailed description of the driver is given below. For *stream*[Ritc84] implementations of TCP/IP, a simple stream pipe may suffice, possibly with no kernel changes whatsoever; again, details are given below.

---

* Author's address: Steven M. Bellovin, Room 3C-536B, AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, New Jersey 07974.

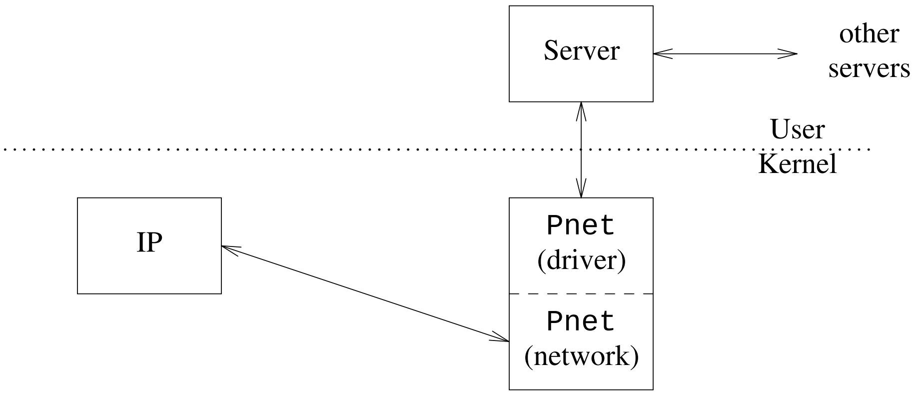

Figure 1. The Pseudo-Network Driver

Although the primary focus of the driver is TCP/IP,[Fein85, Come88] the socket version is actually quite general; it can handle any address families supported by the rest of the kernel. It has been tested on SunOS 4.0.1 and 4.0.3; with minor changes, it should run on 4.2bsd, 4.3bsd, and other related operating systems.

2. RE-INJECTION TECHNIQUES AND ISOLATED INTERFACES

A number of uses for `Pnet` involve *re-injecting* a transformed packet into the kernel for further processing. For example, the packet could be encrypted, repackaged with a new IP header and a protocol number indicating encryption, and sent on its way. Before discussing `Pnet` proper, it is worth examining possible mechanisms for re-injection; it is not trivial to implement, but is quite necessary.

The first, and most obvious way, is to build a new packet, and simply `write`() it to the `Pnet` device, under the assumption that IP will then forward it to the proper destination. However, many IP modules will *not* forward packets, either for security reasons or because forwarding packets is the business of gateways, not hosts.[Brad89]

For socket-based implementations, a second approach is to create a raw IP socket, and use it to re-inject the packets. Unfortunately, while that mechanism is suitable for transmitting the encrypted packets, it fails on decryption. Decrypted packets — received by a user-level process bound to that IP protocol number — should carry the IP source address of the original sender; the raw IP socket interface insists that packets carry authentic local source addresses. While it may be possible to kludge around this requirement, a cleaner solution can be obtained by implementing a new raw protocol in the Internet address family; this protocol would permit specification of an arbitrary IP header.[1]

We have opted to implement a variant of this mechanism. Rather than create a separate interface solely for packet re-injection, we have overloaded the address family field used by `pnetwrite`. As noted, these packets are passed directly to the IP output routine, rather than the input routine. This interface must be used with great care. Only minimal checks are done, to guard against kernel panics. No attempt is made to provide standard packet input processing, such as checksum validation, time-to-live counter decrementing, or option processing. More seriously, the packet is not checked to see if it is destined for this host. If it is, when the real driver receives the packet, it must pass it to IP's input routine. Of course, if the packet was destined for `Pnet`'s local address, it will be delivered again to the server, possibly causing a loop. `Pnet` broadcast packets are a particularly nasty case of this.

Implementing re-injection is harder for stream implementations. The only path into IP is the transport protocols' interface; for these, IP expects to fill in the source address, etc. Some sort of raw channel is needed; this might require changes to IP.

An alternative to packet re-injection is to implement *interface isolation*. If an interface is marked as *isolated* (presumably via `ifconfig`), packets from it are not forwarded. Thus, packets arriving via the `Pnet` driver could be forwarded, while packets arriving on an external link would not. Obviously, source routing would be disabled for isolated interfaces also. A final aspect of interface isolation, possibly controlled by a different bit, is to accept packets arriving on an isolated interface if and only if they are destined for the machine's IP address on that interface. That is, we do not permit the implicit forwarding to an alternate address associated with another network interface on the gateway.[2]

For some purposes, a simple isolation bit is insufficient; one would need isolation groups that define allowable forwarding patterns. Finally, one could escalate to full address screening,[Mogu89] though if encryption is univerally performed that is probably not necessary.

3. APPLICATIONS

The `Pnet` driver has many uses, ranging from the trivial to the complex. A few are discussed below. We have implemented some of these, and plan to implement others.

3.1 System Test

It is often difficult to test protocol implementations. The usual approach is to use sophisticated network monitors to observe the traffic and to create test packets. Such techniques, though, are expensive and often uncertain — fast hosts can easily overrun some network monitors. `Pnet`, though, makes life much easier — a host program can catch or generate all test packets.

Care must be taken when emulating a protocol in a program; some features are more difficult to emulate than others. During development of `Pnet`, we ran into trouble with fragmented ICMP `ECHO` packets; generating proper replies required receipt and reassembly of all

---

1. Some people may object that allowing a host process to impersonate an IP address is a security risk. First, this facility is only available to `root`; a rogue super-user has easier ways to spoof IP addresses. Second, the very existence of `Pnet` allows injection of packets with arbitrary addresses. Finally, as shown elsewhere,[Bell89] using an IP address for authorization is very unsafe in any event.

2. This does not necessarily provide enough security for the gateway machine. ICMP packets can have a global effect, regardless of the destination address used.

fragments.

3.2 Netspy

The `Pnet` driver can be used to monitor conversations. A routing entry can be constructed to direct traffic for a particular destination to the `Pnet` driver. After examination, the packet can be re-injected, adding IP Loose Source routing to carry the packet to the next hop.

The potential for abuse of this capability is, of course, obvious.

3.3 Non-IP Relays

In some environments, it is necessary to send IP packets over media for which IP drivers do not exist. Pnet provides a simple mechanism for accomplishing this; a program can retrieve the IP packets via the `Pnet` driver, encapsulate them for some other protocol, and transmit them to the far end.

This can be also be done in some situations where one side implements an IP driver directly. For example, some implementations of Datakit® VCS support contain an IP interface, while others do not. The latter can use `Pnet` to transmit packets to and from IP; at the far end of the Datakit VCS circuit, IP can handle them directly.

3.4 Replacing SLIP

The conventional mechanism for sending IP packets over tty lines — SLIP, or Serial Line IP[Romk88] — requires oddball code in the kernel. A line discipline is used for framing, which is reasonable enough; however, some implementations require a dummy process to linger to keep the line open, or some mechanism to prevent the normal close operations from taking place. Furthermore, dial-up SLIP operation is awkward, though it has been done.[Lanz89] All of that can be bypassed using `Pnet`. A single process can handle packets for all of the SLIP destinations; it can make calls as needed, transmit and receive data, etc. To be sure, a line discipline may be needed in any event, to buffer the incoming characters and avoid the need to wake up the SLIP daemon each time, but much of the complexity could be eliminated.

A `Pnet` implementation has the side-effect that all of the SLIP destinations would share the same IP network number. This is probably a good idea — using an entire network for each point-to-point link is wasteful, though presumably one could subnet a class C network and use it for 64 SLIP links[3] — but it requires good routing protocols to handle the point-to-point connections. The IP model normally requires that an interface driver be able to reach every connected host directly; this is often not the case with SLIP.

3.5 Bypassing Security Controls

The `Pnet` driver can also be used to implement a bypass for some common security controls. Assume, for example, a paranoid gateway that was configured to allow only electronic mail traffic; this would be configured to accept TCP packets with a source or destination port of 25, and to reject all others.[Mogu89] Two co-operating parties could set up a TCP circuit between `Pnet` servers, and simply assign one end to port 25. Assuming suitable routing information were exchanged, each end would have access to the other's IP networks.

---

3. At least 2 bits must be used for every subnet, as the host addresses 0 and -1 are still reserved.

Obviously, in this sort of situation two parties who merely wished to leak information could do so rather more simply. The point is that `Pnet` allows IP-level access, and is thus far more damaging.

3.6 More Reliable Datagrams

In the current congested Internet environment, datagram services are hard to use. Too many packets are dropped or delayed, leading to excessive retries and/or congestion.[Nowi89] If a TCP-based relay process is used with `Pnet`, application-level retry timers can be turned off, and advantage can be taken of recent TCP performance improvements.[Karn87, Jaco88] Similarly, if the underlying network is prone to data corruption, this mechanism is useful when using systems that turn off UDP checksumming.

If this strategy is adopted, great care must be taken if application-level retry timers are still used. TCP segments can be delayed or lost as easily as UDP packets; however, since TCP will retransmit on its own, it is highly undesirable for the application to do so as well. Application-level retransmissions will simply generate extra load; they will not provide better service.

4. VIRTUAL NETWORKS

There are a number of protocols, typically broadcast-based ones, that operate properly only within a single IP network. If the machines that wish to run such a protocol are geographically dispersed, it may not be feasible to connect them to the same net. Using `Pnet`, though, this can be accomplished reasonably easily: a server could declare the interface to be a broadcast network, and transmit broadcast packets to all appropriate destinations. This is an example of a *virtual network*. While there is an obvious efficiency loss in broadcast virtual networks, the gain in functionality may make it worthwhile for some applications.

Virtual networks have other uses as well. For example, consider the case of a large corporation with many internal TCP/IP networks, and a single gateway to the Internet. It may be desirable to allow a very few selected hosts access to the Internet through the gateway; most, though, would be blocked for security reasons. The selected hosts and the gateway could form a virtual network; only its address would be advertised to the outside world.

A more general way to phrase this is that virtual networks allow for routing and control independent of the physical topology. This can be used to implement many different useful schemes, including ‘‘roamer hosts’’.

Virtual network packets may be carried by TCP, UDP, or IP. If UDP is used, checksumming should be turned off for that connection; it represents needless expense, as the encapsulated packet will undergo further validity checking when delivered to its ultimate destination.

4.1 An Encrypted Virtual Network

Perhaps the most interesting use of `Pnet` is to implement encryption, access control, and authentication mechanisms. We shall spend some time on a detailed description of just such a system; it is currently under development. Since ours is loosely modeled on the *Blacker Front End*[BFE, Mund87] but is much less secure, we dub it *Greyer*. There are two principal uses for `Greyer`: providing end-to-end encryption between a pair of hosts communicating over an insecure network, and providing network-level encryption between a pair of gateways, each of which is protecting a group of naive hosts. We will consider each design in turn.

At first blush, providing end-to-end encryption is simple. Create a virtual network, as described above. When a host wishes to make a secure call to a destination, it uses the destination network's address on the virtual network. All of the packets are thus delivered to the `Pnet` server, which encrypts them and sends them along. These servers have addresses on the insecure physical network. The destination server receives the packet, decrypts it, and writes it on the `Pnet` device; in the kernel, the packet is recognized as destined for the local host, and is delivered to the application in the usual fashion.

There is a catch, however. One of the benefits of encryption is the implied authentication it provides. Applications which believe they are conversing over the secure virtual net may quite reasonably extend much greater trust. Unfortunately, packets with a virtual net destination address may be delivered to a host over its physical network interface; these packets have not been validated in any way. They will nevertheless be accepted.

The easiest solution is the interface isolation mechanism described earlier. Note that we must isolate the physical network, not the virtual one. That is, we will accept packets over the virtual interface for either address; we do not wish to accept packets for the presumed-secure virtual address over the physical link. If the host has more than one physical address, this solution is too simplistic; it may be necessary to use isolation groups.

Some may object that this is not a real problem. After all, even though forged packets to the protected address may be sent via the physical network, replies will be sent via the virtual network, and hence will be encrypted. Unfortunately, there are ways to attack hosts that rely on IP addresses for authentication, even if responses are not heard.[Bell89, Morr85] More simply, IP source routing could be used, thereby forcing the target host to reply via the same insecure path.

Gateways and `Greyer`

Gateways using `Greyer` may require interface isolation as well. For inbound traffic, the rationale is simple: we do not wish unauthorized packets to enter the protected subnet. If the interface were not isolated, an enemy could simply use the physical network address of a target host.

Outbound traffic may need to be restricted also. In a classified environment, for example, individual users may not select the data transmission mode; that is up to the administrator. It is thus necessary to guard against internal traffic being routed directly to the external interface. On the other hand, we cannot simply turn off packet-forwarding, or we would have no way to deliver outbound packets to the `Pnet` server.

Our model, then is this: hosts behind the `Greyer` gateway forward their packets to it to reach a remote secure host. On the gateway, routing table entries specify that the next hop is on the virtual net; this forces the packets to be delivered to the `Pnet` server for encryption. The packets are encrypted and encapsulated, and transmitted over the insecure network to a remote server. It must then decrypt them and hand them back to IP. We may use re-injection; if the host will permit packet-forwarding from the `Pnet` interface, a `write`() over the `Pnet` device will serve.

As noted, encryption provides implied authentication. It also provides authorization: the key distribution center may, at its option, decline to issue a key for a conversation deemed administratively prohibited. In fact, the `Greyer` mechanisms could simply be used for authorization without bothering with transmitting the encrypted text at all, as in the *Visa* protocols.[Estr89] There are obvious risks of address forgery here, of course.

Encapsulation for `Greyer`

There are two issues to consider when deciding how to encapsulate `Greyer` packets for transmission over the insecure network: how should session key information be distributed, and what transport mechanism should be used? The two questions are related.

First, we assume that the `Greyer` server will not have keys for each possible destination; rather, it will use something like Needham-Schroeder[Need78, Denn81, Need87] or Kerberos[Stei88] to obtain a session key. It is therefore necessary to transmit this session key to the remote `Greyer` server. If TCP is used as the transport mechanism, the solution is obvious: send the session key at the start of each connection. If a key expires, the connection may be torn down and a new one constructed.

If, on the other hand, a datagram mechanism is used (either UDP or a new IP protocol type), the problem is a bit harder. One possibility is to send a special packet containing the key to the remote `Greyer` server; depending on the reliability of the underlying network, it may be desirable to await an acknowledgement before transmitting any packets that use the key. More likely, we will use the SP3 protocol from SDNS.[SP3]

A final possibility is to include the encrypted key in each packet. This preserves the stateless nature of IP gateways, at the obvious cost in bandwidth. The exact choice depends heavily on the characteristics of the physical network; we will address this question further when `Greyer` is implemented.

5. SOCKET IMPLEMENTATION DETAILS AND ALTERNATIVES

The socket `Pnet` driver consists of two distinct halves, a network driver and a character device driver. Each contains the usual entry points: attach, output, and ioctl for the network driver, and open, close, read, write, ioctl, and select for the character driver. We describe each half in turn.

The network output routine (`pnoutput`) is quite straight-forward. If the character half of the driver is not open, packets are rejected with code `ENETDOWN`. Otherwise, the packet is queued for the server program. A header containing the destination address is prepended to the packet, in the form of a single `struct sockaddr`. It is important that the server program use this address to determine the header, rather than looking at the packet header; to do otherwise would require that it duplicate most of the functions of IP. If the program's input queue is full, the packet is discarded and `ENOBUFS` is returned to the caller. No attempt is made to loop back packets destined for a local address; that is left to the server.

The rest of the network driver half is comparatively trivial. One, perhaps incorrect, decision: if the interface is turned off via `SIOCSIFFLAGS`, the server program is sent an EOF message.

The character driver is a bit more complex. `Pnetread` blocks until data has been enqueued by `pnoutput`; if `FASYNC` mode has been selected, it returns an error code instead if the queue is empty.

`Pnetwrite` is more problematic for several reasons. First, it cannot accept just a raw packet; it needs address family information in order to route the packet to the proper protocol. While a simple `short` would suffice, the current driver requires a full `struct sockaddr`; this simplifies use of the same data structures for the input and output halves of the program. The other fields in this structure are currently unused, though that may change in the future.

A second complication is the need to *re-inject* packets into the system, as described above. If the high-order bit of the address family is on, the packet is passed to the *output* routine of that protocol, rather than the input routine. Currently, only `AF_INET` is supported for this option.

Finally, it is not obvious how to block if the protocol input queue is full. It is easy enough for the server process to sleep; however, there is no ‘‘interrupt routine’’ to awaken it when the queue drains. Accordingly, a timer routine is used to poll the queue status.

`Pnetselect` has a similar problem; additionally, since it lacks information on which protocol input queue is desired, it cannot assert definitively that space is available. As a heuristic, it queries the status of the last queue to which a `write`() was attempted.

`Pnetioctl` permits the server to set the `IFF_BROADCAST` and `IFF_POINTOPOINT` flags for the interface; since the driver has no way of knowing the intended use of the interface, it cannot make a default choice. Additionally, the server can set and reset `IFF_UP`; while this flag can be set via `SIOCSIFFLAGS`, use of that `ioctl`() is restricted to `root`.

It is also possible for the server to change the maximum transmission unit (MTU) allowed for the interface. If another network medium is used to relay `Pnet` packets, the MTU for the `Pnet` interface should be set to the MTU of the medium minus any required headers, to avoid fragmentation.

Rejected Alternatives

An alternative implementation technique would have been to replace the character driver with a new `socket` address family. That would have allowed use of `sendto`() and `recvfrom`() system calls to pass the auxiliary address, rather than requiring a prepended header. Similarly, much of the existing code for `socket` input/output could be used, rather than writing new routines. This approach turned out to be infeasible for several reasons.

The first is simply a question of packaging. As some systems are distributed, it is much easier to add new device drivers than to add new address families. There is no accessible table to configure the `domain` structure for a new address family, nor are there vacant entries in the address family name space. While some dormant entry could be reused, this seemed unwise. Nor is it possible to add additional entries to a binary system; there are several routines, and one table, that ‘‘know’’ how many address families there are.[4]

A second reason is the permission structure. It was useful to permit non-`root` users to access this facility, at least during testing; this is very easily accomplished via the file system’s permission mechanisms. Doing the same for a `socket` family would have been awkward.

Finally, the device driver interface is much more standardized across releases than is the `socket` interface, and much more documentation exists for it.

It should be noted parenthetically that although modifying distributed source code is not *a priori* a bad idea, it is often infeasible. Source code distributions are sometimes not as current as binary-only distributions, and not everyone is licensed to receive source code.

---

4. Amusingly enough, the SunOS 4.0 distribution does not have `AF_MAX` set high enough for all of the address families named in `socket.h`.

5.1 Performance of Socket-Based `Pnet`

Obviously, performance is a concern when packets are copied to and from user level an extra time before being transmitted. To measure the performance of the socket implementation, we employed a modified version of `ping`(8). This version transmitted a new ICMP `ECHO` packet immediately upon receipt of the response to the previous packet; it also printed the total elapsed time for the packet sequence. Employing this technique, rather than measuring the per-packet round-trip time, allowed us to avoid problems with the coarse granularity of the system clock. In the actual test, between a Sun 3/60 and a Sun 3/75, we measured performance at user-data lengths ranging from 0 to 1300 bytes. Each measurement consisted of 100 ICMP packets; we repeated each test 100 times The goal was to acount for both the per-packet and per-byte overhead. UDP was used as the transport mechanism, with checksumming turned off (the SunOS default). Each test was repeated several times. Note that the timing represents four copy operations on each byte: when sending the `ECHO` packet, when it is received on the target machine, when the response packet is sent, and when it is received.

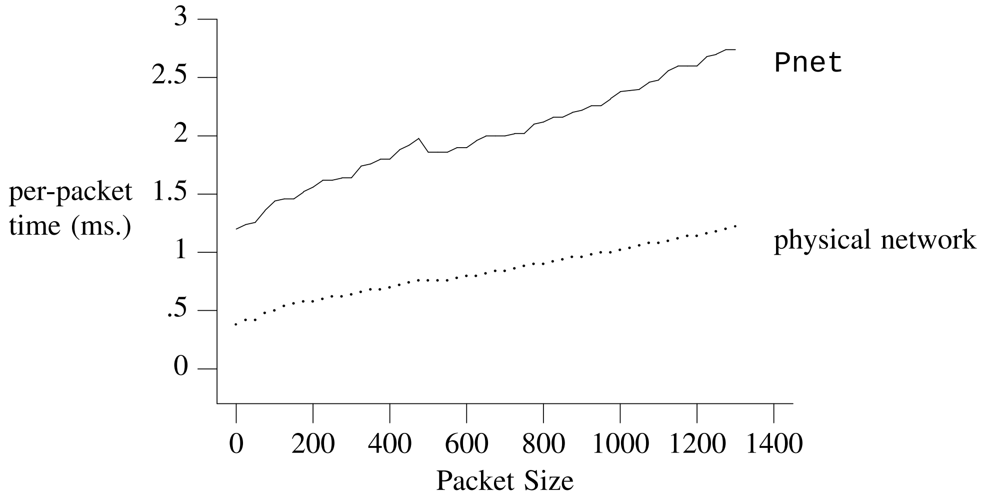

Figure 2. Median ICMP `ECHO` Time

Figure 2 shows the median times in milliseconds for each packet size. There is a glitch in the graph at around 500 bytes; this is most likely due to buffer allocation strategies. Packets of more than 512 bytes — counting the IP and ICMP headers, in this case — are copied into a single *mbuf cluster*, rather than a chain of *mbuf*s.

A second graph, Figure 3, shows the ratio of the times. It appeared that `Pnet` performed at one half to one third the speed of the raw underlying network. To validate this, we used `ftp`(1) to copy a large file to `/dev/null`, after ensuring that the entire file was in the

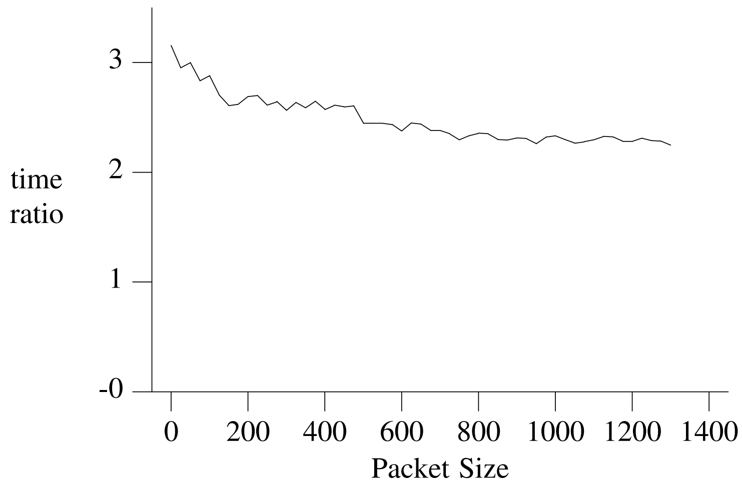

Figure 3. Ratio of Median Times

sender's buffer cache. The speed ratio was noticeably worse than might be expected from the previous measurements, 3.4 to 1. We attributed this difference primarily to CPU time consumption. Copying the data to and from the `Pnet` driver is CPU-intensive; thus, `Pnet` is competing with `ftp` and TCP itself for processor time. The `ttcp` throughput benchmark developed by Mike Muuss yielded similar results. Visual observation of `perfmeter` during `ttcp` runs displays indicated that the receiving host sustained additional CPU load; the transmitting host actually had more idle time.

If CPU capacity is really the limiting factor, the performance difference would not be seen if there was extra CPU capacity available. Looked at another way, we can make available more CPU time per packet by slowing down the interarrival rate. This was most easily accomplished by running similar tests across a long-haul link, in this case between Murray Hill, New Jersey, and Allentown, Pennsylvania. IP access between the two sites is via a 1.344M bps point-to-point link; additionally, several other local area networks and gateways intervene at each end.

The raw throughput graph is shown in Figure 4; the speed ratio is shown in Figure 5. Performance is a bit more variable, due to the vagaries of the shared link; as can be seen, though, the shapes of the two sets of graphs correspond nicely. The `Pnet` link is only about 1.1 times slower than the direct link in this case. A complication arose because of packet loss on the link; given the design of the test program, each dropped packet caused a one-second timeout before the next ICMP packet was sent. We adjusted for this by subtracting one second from the total time for each such packet.

The overhead of `Pnet` should be relatively constant for a given packet size, regardless of the link speed. Figure 6 shows the difference in throughput for both sets of tests; as can be seen,

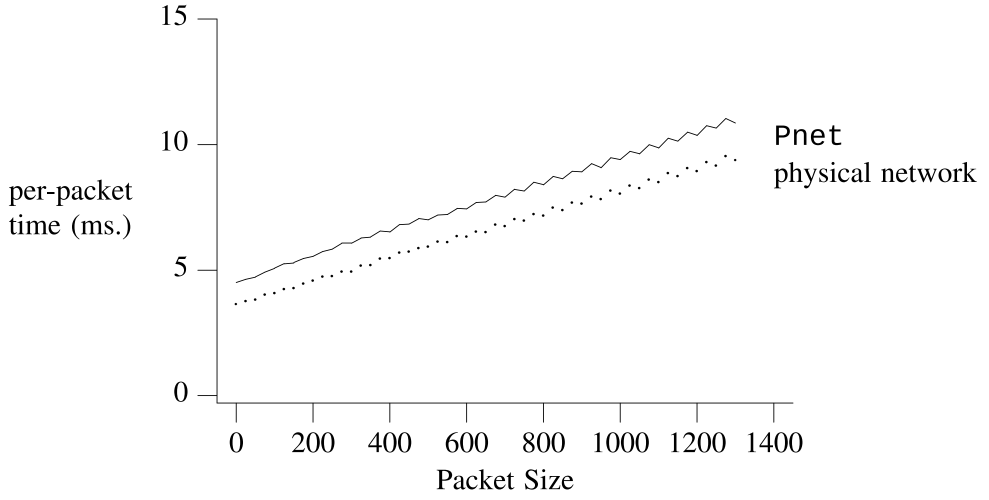

Figure 4. Median ICMP `ECHO` Time — Long Haul

the two graphs are quite similar.

Finally, the same `ftp` and `ttcp` tests were run; throughput for `Pnet` was essentially the same as on the physical network.

6. A STREAM VERSION OF `Pnet`

On a system with a good stream implementation of TCP/IP, there is no need for a `Pnet` driver. The native drivers can be used instead, for all of the applications described above. The mechanism is quite simple: create a stream pipe, and link one end of it to IP. Issue the appropriate configuration `ioctl`() calls (i.e., to inform IP of the network number and IP address), and the stream will be treated the same as any other device driver. For conventional devices, this configuration process is typically table-driven. Since `Pnet` devices are dynamically created, a table is not usable; instead, the `Pnet` server must handle the process manually.

Shutting down a pipe-based `Pnet` driver is often difficult. Shutdown may be disorderly; one or both ends of the pipe may be closed before IP's `close` routine is called. It is therefore vital to detect `M_HANGUP` messages traveling upstream. Another crucial detail is whether IP is prepared to delete the interface control structures. In some versions of stream TCP/IP, much of the rest of the networking code is unprepared to deal with the possibility of such deletions. For example, route table entries often point to the per-interface structure; if these are not cleaned up, problems can occur. In fact, some early implementations of SLIP for 4.2bsd were known to crash when the interface was deleted.

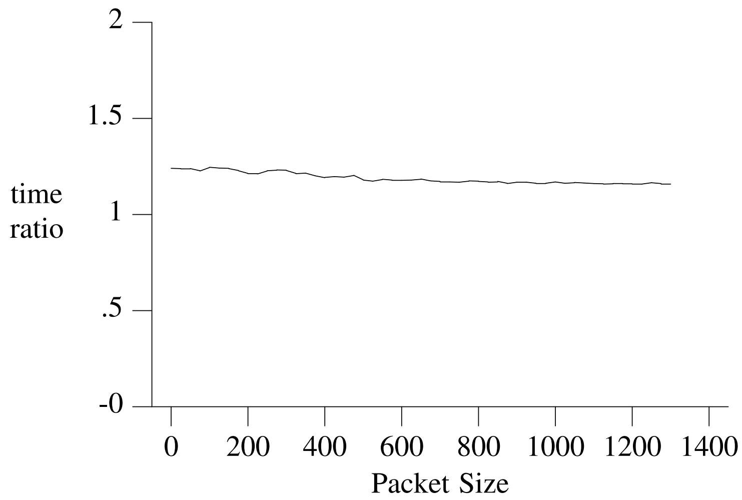

Figure 5. Ratio of Median Times — Long Haul

The 9th Edition implementation of stream TCP/IP deals well enough with shutdowns; however, the IP destination address is not passed downstream along with the packet unless ARP[Plum82] is in use. The implementation is thus able to deal only with Ethernet[5] networks and point-to-point links, for which the concept of destination address is not relevant. This is obviously easy to fix.

System V versions typically use the *Data Link Provider Interface* (DLPI)[McGr89] protocol between IP and the device driver. The `Pnet` server must implement its half of this protocol, a non-trivial matter. DLPI does provide for the destination address to be passed along. Unfortunately, it also introduces another complication at shutdown: the protocol requires that a link be unbound at connection tear-down, via a `DL_UNBIND_REQ` message and acknowledgement. This is not difficult for a resident device driver, but is problematic when the ‘‘device’’ is a pipe. Shutdown can occur when a server program has exited; there is obviously no way for the server to receive or send any more messages.

We worked with a pre-release version of the System V Release 4 streams TCP/IP, based on the Lachman/Convergent code; for it, some of these concerns were minimized. For example, although the drivers do acknowledge IP’s `DL_UNBIND_REQ` message, the acknowledgement is silently ignored; thus, its absence is not missed. Similarly, while some implementation-specific details — for example, associating the stream with a statistics structure, and actually keeping

5. Ethernet is a registered trademark of Xerox Corporation.

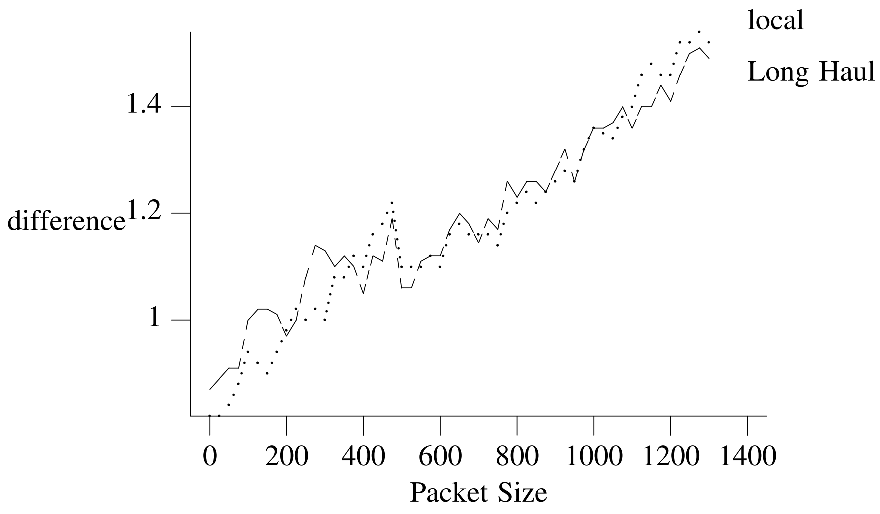

Figure 6. Time Differences

counts in that structure — are messy, the existing drivers in our version ignored them, so we ignored them as well.

Given that, we must implement the following aspects of the protocol:

- Respond to `DL_BIND_REQ` with a `DL_BIND_ACK` message. Since both of these messages are transmitted as `M_PROTO` streams messages, they could be sent and received easily enough via `putmsg`() and `getmsg`().
- Respond to a `DL_INFO_REQ` message with a `DL_INFO_ACK` message. Again, this requires no kernel code.
- Accept and send data via `DL_UNITDATA_REQ` and `DL_UNITDATA_IND`.
- Accept a few `ioctl`() calls. This version of IP requires that the socket `ioctl`() calls, notably `SIOCSIFFLAGS`, `SIOCSIFADDR`, and `SIOCSIFNAME` (to set the interface structure name) be fielded by the driver (or a convergence module), and an `M_IOCACK` message sent back upstream. This one is more difficult, since there is no way to process `M_IOCTL` messages at the stream head, or to generate responses.

We could have implemented this via a special-purpose module. Indeed, if a module were needed anyway, to handle `DL_UNBIND_REQ`, we would probably have opted for that solution. Given that everything else could be handled at user level, though, we provided a general alternative, the `mesg/rmesg` module pair used in 9th Edition systems. These modules encapsulate all stream messages, regardless of type, as an `M_DATA` message preceeded by an `M_PROTO` header. In the reverse direction, a user-generated header is examined to produce an arbitrary-type message from the data portion written via putmsg().[6] A consequence of this is that even the DLPI messages are encapsulated this way; thus, the user process is slightly more complex than might otherwise be the case.

A few minor changes were needed to the implementation of IP. Most important, IP needs to recognize the M_HANGUP message, to indicate that the pipe has been closed. The proper response to this is to delete the data structure identifying a stream, and to delete any routing table entries pointing to it. The routing table adjustments should also be made when an I_UNLINK message is received for any stream; the lack of such could be considered a bug in IP regardless of of the presence of Pnet.

Finally, although the current code permits a stream to be attached via either I_LINK or I_PLINK, the latter is inappropriate for a pipe. If the owning process dies, the user end of the pipe will be closed, thus generating an M_HANGUP and disabling the stream. The IP end, though, will be permanently attached; no process is likely to come along and issue the appropriate I_PUNLINK. Nor is there any significant benefit to the user process in being able to do a persistent link. Consequently, IP should reject I_PLINK calls for pipes. Unfortunately, that is not easy to do; the check is very implementation-dependent. Consequently, we have omitted it in this prototype.

7. CONCLUSIONS

We have demonstrated how one simple piece of code can be used to create a variety of powerful mechanisms. Given comparatively minor changes to the stream versions of IP, it woas simpler yet. We have implemented some of the applications described above; work on others is in progress, notably Greyer.

REFERENCES

[BFE] ‘‘Blacker Front End Interface Control Document,’’ pp. 1-25−1-40 in *DDN Protocol Handbook*, ed. E.J. Feinler, O.J. Jacobsen, M.K. Stahl, and C.A. Ward.

[Bell89] S.M. Bellovin, ‘‘Security Problems in the TCP/IP Protocol Suite,’’ *Computer Communications Review* **19**(2), pp. 32-48 (April, 1989).

[Brad89] R.T. Braden,ed., ‘‘Requirements for Internet hosts - communication layers.,’’ RFC 1122 (October 1989).

[Come88] D. Comer, *Internetworking with TCP/IP : Principles, Protocols, and Architecture,* Prentice-Hall, Inc. (1988).

[Denn81] D.E. Denning and G.M. Sacco, ‘‘Timestamps in Key Distribution Protocols,’’ *Communications of the ACM* **24**(8), pp. 533-536, ACM (August 1981).

---

6. In practice, life is a bit more complex; M_FLUSH messages must be processed both in the kernel and sent to the user process. Furthermore, security considerations dictate that use of mesg/rmesg be restricted to the superuser.

[Estr89] D. Estrin, J.C. Mogul, and G. Tsudik, ‘‘Visa Protocols for Controlling Inter-Organization Datagram Flow,’’ *IEEE Journal on Selected Areas in Communications* **7**(4), pp. 486-498, (Special Issue on Secure Communications) (May 1989).

[Fein85] E.J. Feinler, O.J. Jacobsen, M.K. Stahl, and C.A. Ward, *DDN Protocol Handbook,* DDN Network Information Center, SRI International (1985).

[Jaco88] V. Jacobson, ‘‘Congestion Avoidance and Control,’’ pp. 314-329 in *Proceedings of SIGCOMM ’88* (August 1988).

[Karn87] P. Karn and C. Partridge, ‘‘Improving Round-Trip Estimates in Reliable Transport Protocols,’’ pp. 2-7 in *Proceedings of SIGCOMM ’87* (August 1987).

[Lanz89] L. Lanzillo and C. Partridge, ‘‘Implementation of Dial-Up IP for UNIX Systems,’’ in *Proc. Winter USENIX Conference*, San Diego, California (January, 1989).

[McGr89] G.J. McGrath, ‘‘DPLI Interface Specifications.,’’ AT&T White Paper (February 1989).

[Mogu89] J. Mogul, ‘‘Simple and Flexible Datagram Access Controls for UNIX-based Gateways,’’ in *Proc. Summer USENIX Conference*, Baltimore, Maryland (June, 1989).

[Morr85] R.T. Morris, ‘‘A Weakness in the 4.2BSD UNIX TCP/IP Software,’’ Computing Science Technical Report No. 117, AT&T Bell Laboratories, Murray Hill, New Jersey (February 1985).

[Mund87] G.R. Mundy and R.W. Shirey, ‘‘Defense Data Network Security Architecture,’’ in *Proc. MILCOM ’87*, IEEE, Washington, D.C. (1987).

[Need78] R.M. Needham and M. Schroeder, ‘‘Using Encryption for Authentication in Large Networks of Computers,’’ *Communications of the ACM* **21**(12), pp. 993-999, ACM (December, 1978).

[Need87] R.M. Needham and M. Schroeder, ‘‘Authentication Revisited,’’ *Operating Systems Review* **21**(1), p. 7 (January 1987).

[Nowi89] B. Nowicki, ‘‘Transport Issues in the Network File System,’’ *Computer Communications Review* **19**(2), pp. 16-20 (April, 1989).

[Plum82] D.C. Plummer, ‘‘Ethernet Address Resolution Protocol: Or converting network protocol addresses to 48.bit Ethernet address for transmission on Ethernet hardware.,’’ RFC 826 (November 1982).

[Ritc84] D.M. Ritchie, ‘‘A Stream Input-Output System,’’ *AT&T Bell Laboratories Technical Journal* **63**(8, part 2), pp. 1897-1910 (October 1984).

[Romk88] J.L. Romkey, ‘‘Nonstandard for transmission of IP datagrams over serial lines: SLIP.,’’ RFC 1055 (June 1988).

[SP3] SDNS Protocol and Signalling Working Group, SP3 Sub-Group, ‘‘SDNS Secure Data Networking System Security Protocol 3 (SP3),’’ SDN.301 (July 12, 1988).

[Stei88] J. Steiner, C. Neuman, and J.I. Schiller, ‘‘Kerberos: An Authentication Service for Open Network Systems,’’ in *Proc. Winter USENIX Conference*, Dallas (1988).