A User Level Networking Infrastructure for Linux

Andrew McRae

NetDevices Inc.

<amcrae@netd.com>

ABSTRACT

Currently, most network packet processing occurs within the confines of the kernel in the Linux system. This reflects the historic nature of a Linux/Unix kernel in that its primary role was as a network host or end-point e.g a web server, or desktop. However, considerable effort is being expended to provide a greater level of packet processing on Linux systems, as more features are required such as tunnelling, encryption, quality of service etc. Another factor is the growing use of Linux as a gateway or router. Both these factors rely on a implementing the bulk of the ever-more complex packet processing inside the kernel itself, mainly for performance reasons, though with the attendant issues of robustness, ease of programming, scalability etc.

This paper presents an alternative approach to implementing network services within the kernel itself, and describes a networking infrastructure that attempts to address the major issues with supporting a sophisticated and extensive packet processing environment on Linux without sacrificing performance or robustness. This infrastructure (termed 'NetIO') is implemented as a kernel module in Linux, but is designed to interact closely with user level processes in implementing the network services, bypassing the major issues of kernel limitations such as scalability, robustness, configuration and ease of programming, yet without suffering the performance limitations caused by kernel/user process interactions.

1. Introduction

For virtually the entire life of the Internet, standard CPU systems have been used as not only end points of communication (the whole point of the Internet, of course), but as forwarding and packet processing nodes. Before routers developed as separate devices with their own architectures and market, standard workstations were being pressed into service for communication gateways. Much of the popularity of the BSD releases of Unix were related to the TCP/IP support it provided, not only as a foundation for host-to-host communication using the socket paradigm, but also for acting as a IP packet forwarding gateway. With the emergence of Linux as a widely available Open Source Operating system compatible with (and often sharing code with) the other BSD based freely available Operating Systems, there has been much work and development associated with extending the networking packet processing facilities that are available as part of the Linux releases. There have even been projects dedicated to developing variations of Linux specifically designed to perform as routers (such as the Linux Router Project).

Other projects such as Zebra and XORP (eXtensible Open Router Project) have concentrated on the control and routing facilities, providing implementations for BGP, OSPF, RIP etc. Typically these systems are overlayed onto a separate forwarding abstraction, allowing them to operate on Linux kernels, or on more dedicated platforms.

Therein lies the major distinction between a dedicated router and a Linux kernel being used for packet processing, that typically a router has been designed with packet processing in mind, both from a hardware viewpoint, and from the software architecture viewpoint.

Of course, this has not prevented a large number of features and facilities that have been successfully developed and deployed using Linux and other freely available OS's.

However, there are still significant architectural barriers in attempting to implement many sophisticated packet processing features using Linux as a base platform.

This paper describes an architecture that uses Linux as a platform for a high performance network packet processing architecture that attempts to migrate packet handling out of the kernel and into user processes. In doing so, it addresses many of the issues that dog existing implementations of Linux based packet processing environments.

Method	*Pros*	*Cons*
Core kernel	• Highly integrated • High performance • Easily coupled with existing kernel facilities	• Difficult to write and debug • Hard to replace • Hard to interface to (requires kernel module) or control • Kernel bloat • Robustness
Loadable Kernel Module	• Only loaded when required • High performance • Easily couple with existing kernel facilities	• Robustness • Difficult to write/debug • Reliant on kernel APIs • Lack of portability
User Daemon	• Easy to write and debug • Portable • Easily controlled	• Poor performance • Not easily coupled with other features
Hybrid	• Good performance • Better control	• Requires both kernel and user level development • Requires good APIs for user level control

2. Existing Packet Processing Methods

Existing examples of packet processing features that are based on Linux (or other OS's) typically implement the facilities in a variety of ways:

- As an integral part of the networking facilities of the kernel e.g the core IPv4 packet processing providing INET socket facilities.
- Entirely as a loadable kernel module, usually tightly coupled with the existing kernel networking infrastructure. Examples are IPTables, for packet filtering and NAT processing, or Quality of Service processing. Often these facilities are programmed via user level utilities that interface to the kernel module via system calls, ioctls, or /proc access.
- As a combination of a kernel module performing most of the simple packet processing, with a user level daemon performing more sophisticated control functions or exception packet process. An example in common use is IPSec, where the encryption of packets is usually performed in the kernel module, but Key management and other control protocol processing is performed in user daemons that then interact with the kernel modules. Another example is PPP, where the PPP control processing can be performed in user space, but the main PPP encapsulation or handling is performed as part of a kernel module.
- Completely in user space, where typically a user daemon will employ a tunnel or tap driver or interface to redirect packets to the user space, where the packet processing can be performed and then the packet written back to the kernel. An example of this is the OpenVPN packet, providing a user level encrypted tunnel VPN that requires no specialised kernel modules beyond the standard tunnel interface driver.

Clearly, there are significant advantages to providing networking applications in user space, such as:

- Ease of development, where user level debugging and tracing can be used.
- Robustness, where software error or faults will not cause kernel crashes or unstable behaviour.
- Portability, since the kernel-user APIs are more well known and stable then internal kernel APIs.

As any kernel programmer can tell you, trying to debug or develop kernel code as opposed to user level code can be an order of magnitude more difficult. So why doesn't all networking processing occur at user space? Essentially, the problem is one of performance. Any packet processing that occurs in user space generally involves a copy of data from the kernel space to user space for the user process to get the packet data, and vice versa for delivering the process packet back to the kernel. This is fine when the data is being as an end point e.g via a socket interface, and the user process is performing some action such as acting as a web server, or a database transaction server, but if the action is not end-point orientated, but instead some kind of network packet processing, copying the packet data is a high cost operation. Most network packet processing simply involves header manipulation rather than operating on the entire packet, so there is significant benefit to avoiding a complete copy of the packet into and out of user space.

There is also a question of interaction with other network features. On a typical router, there are many features that can be applied to a packet as it is being processed through a device, such as filtering, header manipulation (NAT etc), classification etc. If each feature were implemented as a separate user process, the packet may undergo many kernel-user copies as it was being processed, significantly degrading the performance. There is a strong advantage to tightly coupling associated network packet processing features, so that the features can share processing context, and provide a low cost of handoff to each feature.

So a popular model that has emerged is a hybrid model, where some level of control plane or exception processing is performed at user space, but the core of the packet processing occurs in kernel modules, to gain performance. The cost, however, is that more and more packet processing features are finding their way into the kernel, causing bloat or robustness issues. A significant problem here is that the demand for more complex network features is very strong, adding to an already long list of desirable features.

3. Kernel Packet Processing.

A question then arises: Is the kernel the appropriate place for performing network packet processing? Clearly, for the original intent where the host is a end-point in the network, it is entirely appropriate (and desirable) that the kernel provide the communications infrastructure for user processes i.e the socket paradigm for sending and receiving network data. And the kernel is highly suited and tuned to perform this operation in an efficient and flexible manner. The significant question that is raised is rather, Is the kernel a good router? Or is it even desirable that the kernel perform the level of packet processing and feature handling that most routers have?

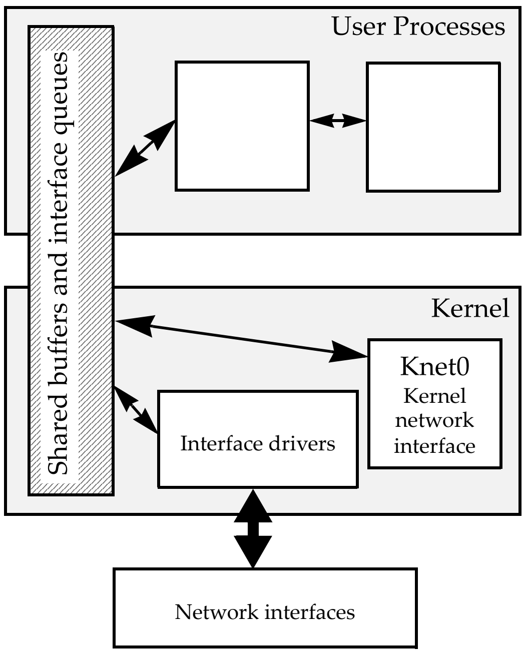
The premise of this paper is that it is essentially an undesirable approach to attempt to make the Linux kernel an efficient and flexible platform for routing network packets. An alternative is proposed, that for network packet processing as a gateway/router requirement, it is much more desirable to migrate this processing to user space, and allow the kernel to perform the role it is best suited for, to act as a supporting OS infrastructure for the user space applications. There are a number of reasons why it is desirable to migrate the packet switching functions out of the kernel and into user space:

- Scalability - the current kernel infrastructure is not designed to deal with more than a few tens of network interfaces at the most (a limit is currently set at 255 interfaces). The scalability goals of routers generally need to take into account highly channelised interfaces such as multiple T1s and E1s, and ATM virtual circuits, as well as potentially large numbers of virtual interfaces for aggregation and tunneling. Routers are usually designed to allow scalability to tens of thousands of interfaces, several orders of magnitude higher then typical kernels.
- Performance - the kernel packet switching code is not designed to perform at router-like speed, but more designed to operate at lower speeds for high touch functionality. A typical switching performance goal of mid-range routers is in the 1-2 million packet-per-second range, several times greater than can be achieved using a typical kernel.
- Robustness – kernel code is inherently more prone to undergoing catastrophic failure in the event of software error or faults (often only rectified through a complete reboot), whereas a user process can be restarted after it crashes without any system reboot.
- Modularity – whilst kernel modules can be loaded and unloaded easily, any code that is loaded into the kernel has to deal with the fact that it is being linked into a relatively large and complex image, and consequently has greater restrictions on the available APIs etc.
- Licensing – the legal status in terms of the GPL is less clear for kernel modules, since they are closely tied to an existing body of code under the GPL. User level code is more readily free of specific GPL conditions, allowing the use of a wider range of available code modules.
- Ease of programming – it is considerable more difficult to debug and code a kernel module rather than a user module, and user level code can take advantage of the kernel facilities (such as system calls) to build cleaner APIs, and also take advantage of the large range of user level libraries available.

Of course, the challenge is to provide an infrastructure for user level packet processing that has the performance advantages of kernel packet processing, but avoiding the problems discussed above.

4. NetIO

NetDevices Inc. has developed an alternative approach to performing network packet processing, termed the NetIO system.



The NetIO system comprises of a number of components that interact together to form a framework for migrating network packet processing to user space. The aim of this framework is to provide a alternative (parallel) packet processing infrastructure, essentially using the kernel as a foundation for implementing this framework as an application on the kernel.

The main components of NetIO are:

1. A memory area that is shared between the user and kernel space.
2. Network device drivers for interfacing to physical devices.
3. A psuedo network interface to the kernel

5. User/Kernel Shared Memory Area

Central to the NetIO system is a shared kernel/user memory area that is used for packet buffers and communication queues between kernel and user space entities.

The shared memory structure is allocated in kernel virtual memory space using the kernel vmalloc call, and the user process accesses this structure by calling mmap on the character device filename under which the NetIO system is installed.

The memory structure contains the following items:

- An array of fixed length buffers used to store packet data. Each packet buffer (called a '*nbuf*') is 2Kbytes in length, with a small portion of reserved space at the start for some housekeeping information.
- An array of data structures holding transmission queue lengths for interfaces, designed for communicating interface queue lengths to the user process so that Quality of Service congestion control can be implemented.
- An array of data words implementing a FIFO ring buffer, used to pass commands/ data from the kernel to the user process. This ring buffer is known as the **In Queue** or **InQ** for short.
- An array of data words implementing a FIFO ring buffer, passing commands/data from the user process to the kernel. This ring is known as the **Out Queue** or **OutQ** for short.
- A small number of variables used in the management of the InQ and OutQ.

This shared memory structure is the primary mechanism for communication with the user process. In addition to this, supporting code in the NetIO module also provides other facilities such as:

- Initialisation code for the memory structure elements.
- Kernel device code for representing the Kswitch module as a character device in the */ dev* namespace (called */dev/netio*), so that the user process can access it via *mmap* and *ioctl* system calls.
- A driver based page mapper for allocation of the pages for the shared memory area.
- Code to interface with the network device drivers.
- *Ioctl* handlers for performing various housekeeping operations and device control operations.

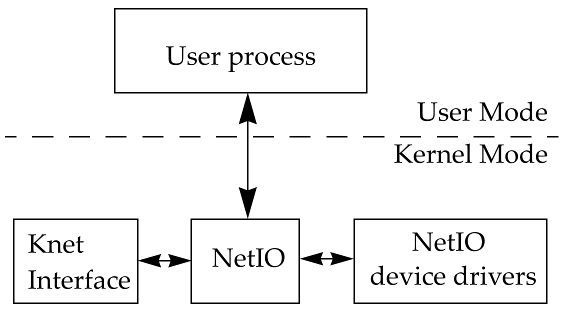
6. Network Device Drivers

Another part of the NetIO system are the actual device drivers. This are like any other device driver, in that they are designed to interface to physical devices, and manage the transfer of data to and from the devices. The major difference is that the devices are specific to the NetIO system, and do not appear as network interfaces in the kernel themselves. The device drivers interface to the packet buffer pool to obtain network buffers for receiving packets, and place message entries on the NetIO queues to the user process.

Existing network device drivers can be easily converted to use the NetIO driver interface, by modifying the allocation of buffers to use the NetIO buffers rather than the skbuff routines etc.

7. Kernel Network Interface

The question may well be asked, If the NetIO system is entirely separate from the traditional kernel network infrastructure, how does it communicate with this framework? The answer is that a psuedo network interface is used to provide a communications path between the kernel and NetIO:



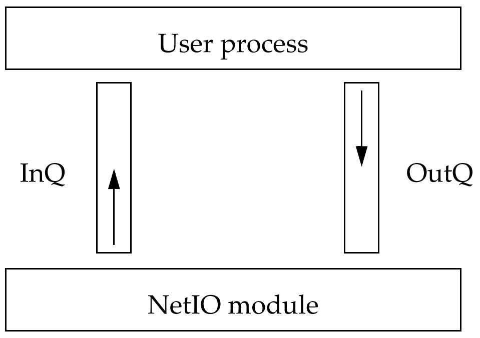
The intent of the *knet* interface is to provide a network interface that then delivers packets to the NetIO framework from the kernel. Processes that use a standard socket interface to send packets will have these packets delivered into the NetIO framework, where the user applications that are processing the packets can then forward the packet onto the appropriate interface. In a similar fashion, packets being received in the NetIO framework that are to be handled locally by any application can be delivered to the knet interface, and the packet processed just as if it had been received via a normal kernel network interface. This allows any application to transparently connect to the networking framework provided by the NetIO subsystem. One way of considering the NetIO framework is to view it as a *virtual router* that connects to the kernel via a single network interface.

8. User/Kernel Interface

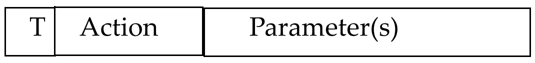
User applications interface to the NetIO kernel subsystem through a standard device file that is used to *mmap* the shared memory are into the process's address space.

Apart from the the packet buffer pool, the main form of communication is via two FIFO rings, called the In Queue (InQ) and the Out Queue (OutQ). Each of these queues are write-only from one side, and read-only from the other. The queues wrap around, and a toggle bit is used to flag the end of the queue (the toggle is flipped as the queue wraps, so that the reader does not have to clear any OWN bits before moving on).

The queues act as FIFOs between the user process and kernel code:



The format of each 32 bit entry in the queues is:



The Toggle value is 1 bit, the Action value is 7 bits, with the parameters for each action being 24 bits. The size of each queue is calculate so that overflow is a rare, if not impossible, situation.

The InQ is used to pass received packets to the user process, QoS notifications, device status etc. The OutQ is used to pass packets to be transmitted or buffer free commands. Each queue reader is responsible for ensuring that the queue is serviced regularly. The user process does this by checking the toggle bit in the InQ, and if no entry is ready, then the user process can perform a poll or select on the NetIO device until a new entry has been placed on the InQ.

Within the kernel, a Linux kernel tasklet is used to service the OutQ, and the tasklet is initiated as a result of device driver interrupts, or as deadman requests from the user process.

9. Conclusion

The NetIO system is successfully being used to implement an alternative framework for high performance, robust packet processing.

One end result of the use of FIFO queues for user/kernel interface is to minimise the number of process scheduling context switches and kernel system calls that need to be made under high load conditions. In a high load, steady state situation, with a single user process performing the packet processing, no system calls need be made for I/O rendezvous, and no process context switches will occur (only device driver interrupts occurring), thus the maximum CPU utilisation is achieved. This, with the combination of no packet data copying for packet processing, ensures that maximum packet processing throughput can be obtained, yet within the safety of a user process.

There are significant advantages for using a separate infrastructure for packet processing, such as robustness and performance, yet without forsaking the performance that a kernel may provide.