The IP Per Process Model: Bringing End-to-end Network Connectivity to Applications

(Spine Title: The IP Per Process Model)

(Thesis Format: Monograph)

by

Dan L. Siemon

Graduate Program in Computer Science

Submitted in Partial Fulfillment of the Requirements

for the Degree of

Master of Science

University of Western Ontario

London, Ontario

October 15, 2007

THE UNIVERSITY OF WESTERN ONTARIO

FACULTY OF GRADUATE STUDIES

CERTIFICATE OF EXAMINATION

Supervisor	Examiners
Hanan Lutfiyya	Dr. Mike Katchabaw
Supervisory Committee	Dr. Marc Moreno Maza
	Dr. Abdelkader Ouda

The thesis by

Dan L. Siemon

entitled

The IP Per Process Model: Bringing End-to-end Network Connectivity to Applications

is accepted in partial fulfillment of the
requirements for the degree of

Master of Science

Date	Chair of Thesis Examination Board

Abstract

The application of the end-to-end principle in the design of the Internet has been fundamental to its success. One of the most important advantages of the end-to-end principle is that it allows the introduction of new services and protocols into the network without requiring changes to intermediate nodes. This greatly reduces the difficulties associated with developing and deploying new transport layer protocols and network services.

Traditionally network protocol implementations are placed inside the operating system kernel. An alternative to this design found in the computing literature is user-level networking. User-level networking places the protocol implementation and processing inside the application. Among other advantages this design simplifies network stack development and deployment.

This thesis offers a network stack model based on user-level networking which has the primary goal of extending the benefits of the end-to-end principle to applications. This model is referred to as the IP per Process Model. A prototype of this model named Pnet/UNL has been built and evaluated against the Linux network stack. Performance evaluation shows this prototype to compare favorably against the Linux network stack on a 100 Mbps network but the performance gap widens at 1 Gbps.

Keywords: Internet, user-level networking, end-to-end principle, network stack, protocol implementation.

Acknowledgements

Karen, Hanan and the past and present residents of MC240.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Few things have changed the world as fast as the rise of the Internet over the last decade. This thesis investigates the end-to-end principle which has been critical to the success of the Internet and argues that this design principle can be used to guide the development of a new network stack model. This new network stack model extends the benefits of the end-to-end principle beyond the devices at the end of the network and into the applications executing on those devices.

1.1 A Look at Two Networks

At this point in time there are essentially two global networks: the PSTN and the Internet. Comparing the basic architecture of these two networks is very interesting because they embody very different ideas about how a network should function.

1.1.1 PSTN

The public switched telephone network (PSTN) has been in existence for over one hundred years [18]. It is the PSTN which carries phone conversations between people all around the world. Despite its age and the rise of the Internet the PSTN still effects the daily lives of millions of people. Quite likely even more so than the Internet. At a high level the PSTN consists of two major components, the telephone and PSTN switches. The telephone

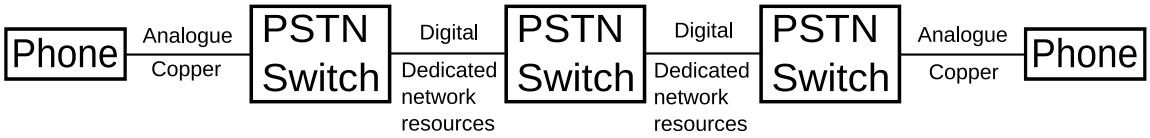

Figure 1.1: The PSTN

functions as the user attachment point or the end of the network. The interface between the PSTN switch and the telephone consists of a pair of copper wires and an analogue signal representing the audio. Such a simple interface limits the richness and complexity of the communication between the phone and the PSTN switch. The result is that while telephones now have features such as address books, call display and answering machines the fundamental simplicity of the phone has not changed. The carrier PSTN switches on the other hand are very large, costly and complex devices which have evolved extensively since their introduction. The role of the PSTN switch is to establish a connection between the source and destination phones. Originally this meant physically connecting the wires attached to the phones; this was a job done by human operators. Later the switching operation was automated but a dedicated copper path between the two phones was still established. As technology advanced the PSTN switch became responsible for digitizing the analogue voice signal for transmission to other switches. The final switch on the path was responsible for reconstituting the analog signal for transmission to the destination phone.

The PSTN follows a smart core, dumb ends network model. Consider the process of calling a remote party on the phone. The caller enters the phone number which is then communicated to the local carrier switch via a set of audible tones. These tones are carried on the same audio channel as the voice communication. After receiving the destination number the local switch establishes a call path through any necessary intermediate switches and eventually the destination phone. Along this call path network resources are dedicated and state information is maintained for the duration of the call. It is the responsibility of the network itself to maintain call quality and ensure no information is lost. The phone on the other hand, simply transmits its analog signal and plays no active role in assuring that the audio information reaches the destination phone. In fact, the two phones on a given connection are not actually communicating directly because the digital data transmitted through

the modern PSTN bears little resemblance to the analogue signal transmitted by the phone. The PSTN is a prominent example of the dumb end, smart core network model. While this model does have the advantage of making the end device very simple and thus a very low cost item, the negative side effects of this design choice effect the utility of the network and its ability to add new services.

In some sense the lack of new features and services introduced into the PSTN is surprising given the elapsed time and the importance of the PSTN to society. This stagnation can in large part be explained by the chosen network model, smart core and dumb ends. Imagine the process of adding a new unreliable, low-quality video multicast service to the PSTN. The first challenge the designers of this new service would encounter is that the PSTN does not offer an unreliable mode of operation. An unreliable data transmission mode is desirable because the designers would not want to burden the network or the source with data retransmission since video streams can tolerate lost packets. Adding an unreliable data transmission mode requires adding the necessary features to the PSTN switches. Unfortunately for the designers of this new service, data traveling through the PSTN must cross several different switches which are often manufactured by different companies and owned by different carriers. Now the task becomes not only designing the new service but convincing the PSTN switch manufacturers to implement the feature. It is unlikely that these companies would be willing to implement any new feature without demand from their customers, the carriers. The situation the video service designers find themselves in with the carriers is no better than found with the switch companies. The carriers have no interest in asking for switch support or even offering a new service unless it can be assured the investment will yield some return. Here again we find a catch-22. It is unlikely this new service will be profitable unless it is available to a large portion of the PSTN network users. As a result, unless either a switch company or service provider decides to take a large risk, this new service will never be deployed. This discussion has not even touched on the difficulties of upgrading the communications channel between the local switch and the phone and replacing the phone itself. There is a second aspect of the smart core, dumb ends network model that also needs to be discussed. That aspect is control. This network model places all control in the hands of the PSTN service providers. No new service or feature can be

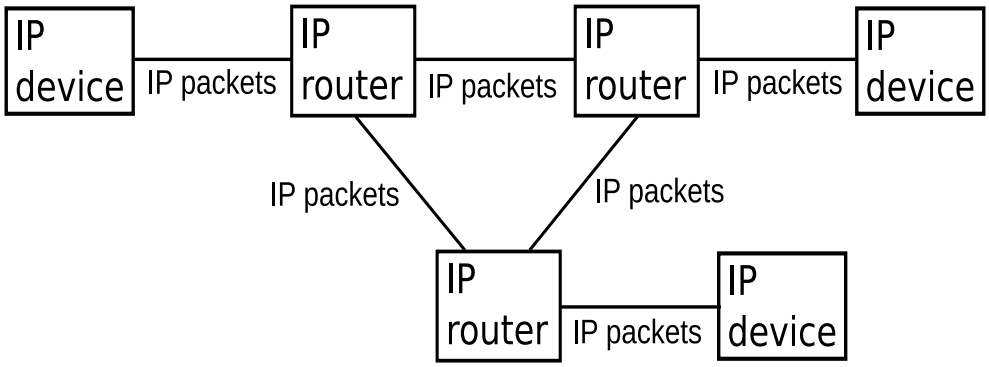

Figure 1.2: The Internet

offered on the PSTN without the permission of and implementation by the service provider. The deployment of new services which hurt existing revenue streams is unlikely as is the deployment of any service without a clear profit model. Worse, even if a single carrier could be found to implement a new service the ubiquitous nature of other PSTN services could not be achieved without the cooperation of every PSTN service provider in the country or even the world. In short, the smart core dumb ends network model forces an entire industry consisting of equipment manufacturers and service providers to agree upfront on any innovation. This model also places control of innovation in the hands of organizations who may have the most to lose from truly disruptive change.

1.1.2 Internet

The Internet functions on a model opposite to that of the PSTN. The PSTN is a smart core, dumb ends network. The Internet is a dumb core, smart ends network. This is somewhat counter-intuitive because the Internet appears to be a very intelligent and advanced network and there are certainly more services offered on the Internet than on the PSTN. The key aspect is not the overall intelligence or utility of the network but the placement of that intelligence.

Continuing the high level overview found in the PSTN section, the Internet can be considered to consist of two main components: end nodes and routers. End nodes are any device attached to the Internet which can send and receive data using the Internet Protocol (IP). Throughout most of the Internet's history end nodes consisted of devices that are

considered to be computers. As the Internet has become more ubiquitous the variation in the design and capabilities of Internet end nodes has increased to include mobile devices, cameras and other devices. For the purposes of this discussion the key aspect of all of these devices is their complexity. Any device which can send and receive IP traffic is much more complex than the PSTN phone. IP capable devices require the ability to break data into small units for transmission, buffer data and perform simple error detection. These are all capabilities well beyond the standard phone. The second component of the Internet in this high level overview is the Internet router. Internet routers play a role roughly analogous to PSTN switches. Here again the main difference is the level of complexity. The Internet appears reliable to most users but in reality the service offered by the Internet to its users is unreliable. There is no guarantee that the data sent by a node into the network will arrive at the destination at all let alone without errors or within certain time constraints. This simple model is accomplished by attaching a source and destination address to the data sent into the network. Routers look at the destination address and then forward the packet to the next router on the path to the destination. Notice that unlike the PSTN a path of dedicated resources is not created between the sending and receiving nodes. This allows routers to be conceptually simple devices whose main operations are a look-up of the destination address in a table to determine which outgoing port to transmit the packet on and the transmission of the packet.

Internet users take for granted the amazing variety of services available. It is not immediately obvious that it is the smart ends, dumb core design which allows these services to be developed and innovation to flourish on the Internet. Imagine adding the unreliable video service that was discussed in the PSTN section (1.1.1) to the Internet. The designers of this new service have an immeasurably easier time getting it deployed on the Internet than they did on the PSTN because no intermediate network nodes require modification. No router vendor needs to be convinced of the service's utility and no ISP needs to be convinced of the service's profitability. Instead the application can be written and deployed on only those network end nodes that wish to use the service.

A real world example of innovation on the Internet can be found in the World Wide Web (WWW). Most current Internet users use the terms Internet and WWW interchangeably.

The reality is that these two things are different. The Internet is the global IP network consisting of end nodes and routers. This network can carry any data capable of being layered on top of the IP protocol. The Internet is much older than the WWW and can be considered the medium on which the WWW is built. When Tim Berners-Lee came up with the idea of URLs to uniquely identify documents and HTML to allow links between those documents thus creating the WWW he was not required to get permission from router vendors to implement this new idea. Nor was he required to obtain permission from ISPs or the manufactures of the equipment that would run this new service. Instead he was able to write the first versions of these important technologies and deploy them on individual nodes around the Internet. A more recent example of this phenomenon can be found in the proliferation of Instant Messaging (IM) services on the Internet in recent years.

1.2 The End-to-end Principle

The end-to-end principle [44] is the guiding idea behind dumb core, smart end networks such as the Internet. In short, the end-to-end principle says that as much complexity as possible should be placed at the end or edge of the network and as little complexity as possible should be placed in the core or intermediate nodes. By making the core of the network simple little is assumed about the behavior of the applications which will use the network. This allows the network to support a wide variety of services without modification. Low complexity in the intermediate or core nodes of the network also greatly increases scalability because the fundamental operations of these devices are simple and do not require per connection or per node state information. Benefits also accrue from the placement of the complexity at the end of the network. It is only at the end of the network where major changes such as new protocols and services can be introduced quickly and easily because changes at the end of the network do not effect other network nodes.

An interesting way to conceptualize a true end-to-end network is as a sphere with all nodes of the network existing on the surface of the sphere with the distance between each of the ends being very close to zero [45]. That is, an end-to-end network provides a virtual direct

connection between all nodes in the network.

It is important to understand that being able to communicate with other nodes on an end-to-end network does not necessarily imply true end-to-end functionality. In fact a large portion of the nodes on the Internet today do not enjoy the benefits of the end-to-end principle. The reason for this is the proliferation of Network Address Translation (NAT). NAT was originally conceived as a short-term solution to the shortage of globally unique IPv4 addresses [7]. It was thought NAT would be only deployed until IPv6 or another protocol with a larger address space came into common use. Unfortunately, for many reasons including the delayed deployment of IPv6 and the use of address conservation methods such as NAT itself and Classless Inter Domain Routing (CIDR) [13], NAT has become a common part of the present Internet architecture.

NAT enables inter-node communication without requiring globally unique addresses by making use of higher layer protocol information to route packets towards their destination. When NAT is not used on the path between two network nodes the destination IP address is all the information that is required by intermediate routers to forward the packet towards its destination. However, when NAT is involved five pieces of information are required: the source and destination IP addresses, the transport protocol identifier and the transport layer source and destination port numbers (see figure 1.3). In order for the NAT process to occur all packets must pass through a device often referred to as a NAT gateway or NAT router. These devices have at least two network interfaces. One of these interfaces is attached to the global Internet and is assigned a globally unique IP address and the second interface is connected to the devices which do not have globally unique IP addresses.

Before discussing the negative effects of NAT it is important to acknowledge the benefits that are driving its adoption. The most obvious benefit is reduced address consumption. A potentially large number of end nodes can all be placed behind a single globally unique address. Related to address consumption is the fact that many Internet Service Providers (ISPs) charge for IP addresses. Many households and businesses now contain many IP enabled devices. NAT allows the use of many devices without increasing the service cost. NAT is also often incorrectly associated with increased security since it is impossible to

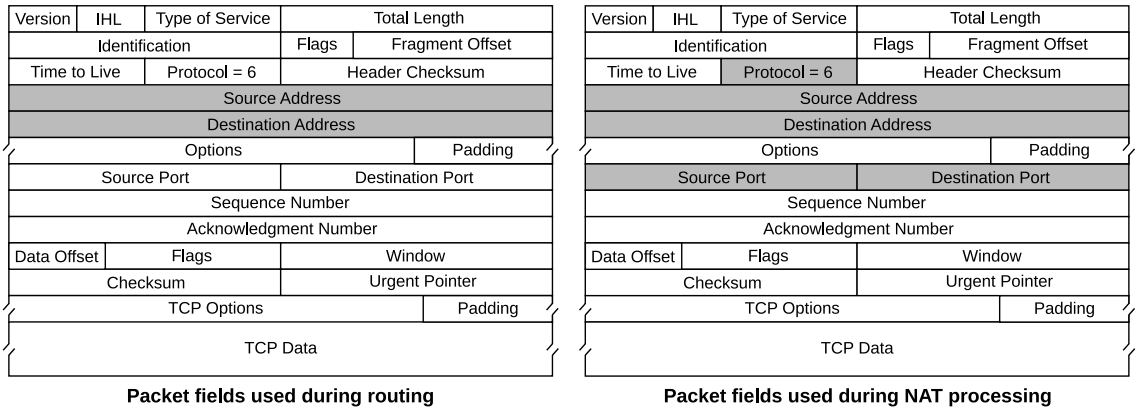

Figure 1.3: Packet Fields Used in Routing and NAT

address packets to network nodes behind a NAT gateway. This behavior is taken advantage of to provide a firewall which limits communication to internal nodes. In reality this type of security can be easily accomplished with a stateful firewall without the problems caused by introducing NAT into the network.

In order to understand the behavior of NAT there are two cases to examine. The first case is when a node behind the NAT gateway opens a connection to a node on the other side of the NAT gateway. From the standpoint of the initiating node this communication is no different than if NAT was not present on the network path. In reality the NAT device is capturing each transmitted packet and rewriting the source IP address and portions of the transport layer information. This operation requires rebuilding the protocol headers and recalculating the error detection checksum in both the IP and transport layer headers. These operations are more complicated than simple IP routing. The second situation to consider is a remote node wishing to initiate communication with a node behind a NAT gateway. This situation is much more complicated because the NAT gateway cannot move packets towards the destination node based on the IP destination address field alone. In fact, the destination IP address will be the globally unique IP address assigned to the NAT gateway.

The negative aspects of NAT can be summed up into one fact. NAT breaks the end-to-end model of the Internet. One of the key aspects of the end-to-end principle ignored by NAT is maintaining as little state information in the network as possible. Data communication occurring through a NAT device is completely reliant on the state information in the NAT

device. If the traffic flows through another path bypassing the NAT device or if the device loses its state information due to a power failure the communication channel is destroyed. In contrast when only routers are in the path any single router can fail without necessarily causing communication difficulties because packets may take another path to the destination. The centralization inherit in a NAT network can also be a performance problem. If the NAT device runs out of memory to maintain the state table or if it cannot perform the address translation at the required data rate then communication performance will be adversely effected. A second important aspect of the end-to-end principle as it is applied to the Internet is the ability to use any transport protocol or service on top of the IP protocol. This is possible because IP routers do not inspect any fields outside of the IP header. However, when NAT is involved higher layer protocol information is also required. This forces the NAT device to have some understanding of the higher layer protocol in order to obtain the information required to forward the packet. As a result nodes behind a NAT gateway are limited to using only those layer four protocols that the NAT gateway supports.

The above are only the most obvious problems caused by the loss of end-to-end connectivity between nodes when NAT is in use. Less obvious is the significant additional complexity forced onto application and operating system developers in order to work around problems introduced by NAT. Many applications and services on the Internet are now being augmented with real-time media capabilities. The addition of voice and video chat to instant messaging services such as AIM, Jabber/XMPP and MSN Messenger provide examples of this trend. These media streams can consume a significant amount of bandwidth and are latency sensitive. These features make it desirable for the media stream to be sent directly between the two communicating nodes avoiding the additional latency introduced by passing data through a third node. When both ends of the communication have globally unique IP addresses establishing a connection for direct media transmission is easy. The node initiating the media stream can open a listening socket and then send the remote application its IP address and a port number. The second node can then simply open a connection to this IP address and port number. When the node opening the listening socket is behind a NAT gateway the situation is much more difficult because the node's IP address is not globally unique. This makes the IP address sent to the second node useless. Even if the IP address

was useful the NAT gateway still would not have the required port mapping to direct the received packets to the appropriate internal node. As a result of these problems many NAT devices contain special support for protocols that pass IP addresses between hosts in order to rewrite the IP addresses as the data passes through the NAT gateway.

Working around the problems introduced by NAT has become very important to both application and operating system developers. As a result many network applications now rely on technologies such as UPnP [55], STUN [41], ICE [43] and TURN [42] to successfully communicate when NAT is in use on the network path. While it is true that these techniques can be implemented by the operating system or as shared libraries for use by application developers they still add a significant amount of complexity that is simply not necessary when true end-to-end connectivity is in place.

1.3 The Kernel Network Stack

Network protocol implementations are usually referred to as network stacks in reference to the layered model used to describe many protocol architectures. See figure 1.4 for the Open Systems Interconnection (OSI) and Internet network layers [33, 26]. Conceptually each layer encompasses a certain type of functionality. The physical layer defines the electrical or optical signaling on a communication medium such as copper wire or optical fiber. The data link layer is used to describe protocols such as Ethernet which allow the communication between two hosts on the same network. Layered on top of the data link layer is the network layer. This layer describes protocols which allow hosts on separate networks to communicate. The IP protocol is perhaps the most most famous example of a network layer protocol because it is now used to connect millions of smaller networks around the globe.

While the layered network protocol model provides a useful basis for the discussion of network protocols not all protocols fit nicely into a single layer. TCP for example encompasses both the session and transport layers of the OSI model. Also note that the Internet defines the protocol layers differently than the commonly mentioned seven layer OSI model (see

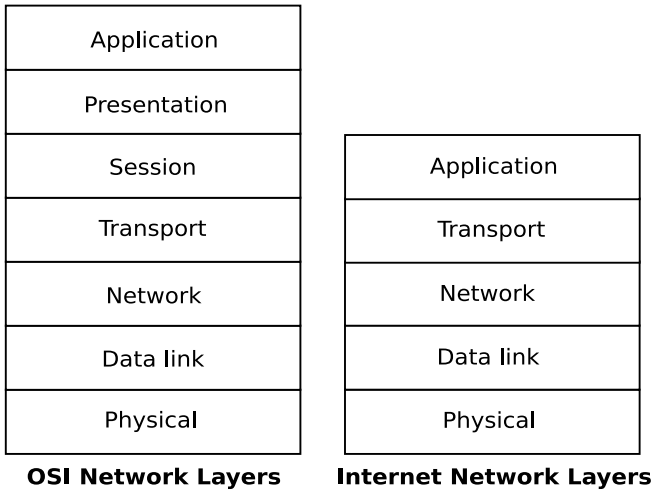

Figure 1.4: The OSI and Internet Network Layers

figure 1.4).

By far the most commonly deployed network stack design is the in-kernel design. This design places the network protocol implementations within the operating system kernel. The in-kernel network stack design consists of three main components: the network interface (NI) driver, the implementation of the network protocols and the application itself (see figure 1.5). The first two of these components are part of the operating system kernel. The NI driver is generally a protocol independent component which is responsible for communicating directly with the NI hardware. When a data frame is received by the network interface it will raise a hardware interrupt which results in the execution of the device driver's interrupt handler. The interrupt handler then either copies the data from the NI into main memory or arranges for a direct memory access (DMA) transfer to move the received data frame into memory. Once the received packet has been moved from the network interface into memory protocol processing can occur. Older network stacks perform protocol processing within the interrupt handler immediately after the packet has been moved into memory. This design however leads to a major performance problem called livelock [39]. Livelock occurs because interrupts are usually the highest priority task within the operating system kernel and thus performing complex processing in interrupt context can block the execution of other important tasks. More modern in-kernel network stacks perform protocol processing within a lower priority process to avoid this problem. The primary task of

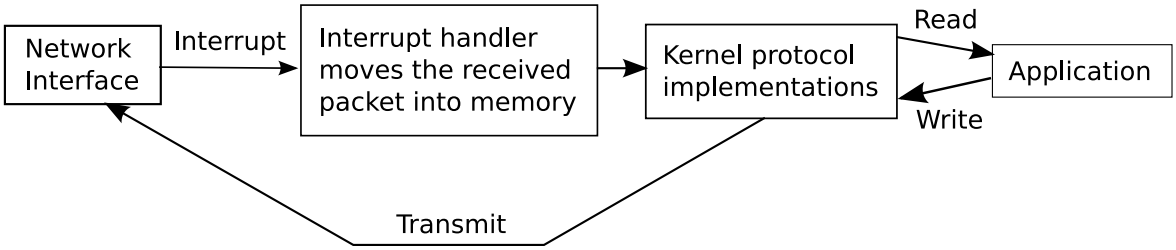

Figure 1.5: Kernel Network Stack

the protocol implementation is to determine which application the data is destined for and after performing the protocol processing, placing the data into a buffer where it can be read by the application process.

The most common interface to the kernel network protocol stack is the sockets application programming interface (API). The sockets API presents the application with an interface to the network which is very similar to file I/O. When a packet is received the kernel retrieves the data from the network interface and performs the necessary protocol processing. The application is then able to obtain the received data by executing the read system call on the socket. A similar process happens when an application wishes to transmit data. The application writes the data to a socket buffer via the write system call. The kernel looks after packaging the data into protocol data units (PDUs) for transmission to the network. Notice that the sockets API and the in-kernel network stack severely limit how the application interacts with the network. By only allowing the transport layer payload to pass between the application and the kernel network stack it becomes impossible for applications to use any network protocols not supported by the kernel network stack.

Imagine the development of an application that would see benefits from utilizing a custom network stack. The benefits could come from a new transport protocol, a slightly modified transport protocol or even just a new API to interact with existing protocols. Using the in-kernel model the developer is forced to make these network stack modifications to the operating system kernel. Since the required source code is not available for some commercial operating systems modifying the network stack in any significant way may be impossible. Even when the required source code is available the task is still very difficult. One of the reasons for this difficulty is the lack of development tools equivalent to the ones

available for user-space development. Depending on the operating system tools such as debuggers, memory validators and profilers are more difficult to use or are non-existent when developing in kernel-space. Development time is also significantly increased due to the requirement that the kernel be restarted to test new code. This requires a complete system reboot or at minimum a virtual machine restart. Finally, reliability is also a major concern for in-kernel network stacks. Any fault has the potential to bring down the entire system not just the application using the new features.

Even if the application developer is willing to go through all of the work required to create a custom network stack or a new protocol implementation the developer is now limited to deploying the application on a single operating system because network stack implementations are not easily moved between operating systems. The reason for this is the lack of a standard API analogous to the POSIX APIs [20] which would make a protocol implementation portable across kernels. Worse, some operating system kernels provide no guarantee of API or binary stability between versions of the same kernel. Linux is perhaps the strongest example of this [24]. It is not uncommon for the function signatures and data structures within the Linux kernel to change between minor releases. This rapid change within the operating system kernel forces the custom network stack developer to keep pace with the rest of the kernel development in order for the new features to continue to work on future operating system releases. There has been some work on making protocol implementations portable across operating systems. In [25] the authors investigate the difficulties associated with this problem. One solution to this problem is the Base system presented in [15]. Base provides a virtual environment for the execution of network protocol code. This allows protocol code to be written once and then easily executed on another operating system as long as the virtualized environment is present. Despite the work in this area these features are not found in current commodity operating systems.

According to [22] the centralized in-kernel network stack also introduces performance problems on symmetric multiprocessing (SMP) and multicore systems. On traditional computer systems with only one CPU the fact that all protocol processing goes through one central kernel subsystem does not effect performance because only one task can be executing at a time. However, the current trend is towards SMP and multicore systems. In order

to manage concurrency and obtain the performance benefits of the larger number of processing units network stacks must introduce complicated data structures and fine grained locking. From Amdahl's law [1] we know that systems do not scale linearly when some portion of the system is not parallelizable. As a result it is difficult for in-kernel protocol implementations to scale with the number of processors. This combined with the cost of locking in terms of negative cache effects makes the centralized nature of the in-kernel stack a performance problem.

Despite the difficulties involved in creating a custom network stack or a new protocol implementation and the fact that the implementation will be tied to a particular operating system kernel it is possible for the application developer to deploy new protocols if desired. However, the difficulty for both the developer and the user of the new software makes this unlikely to happen. Very few applications intended for mass distribution can require modification of the underlying operating system. The next section of this chapter introduces two standardized transport layer protocols which offer many advantages to application developers but see little use on the Internet. These protocols may be deployed more widely if the development and deployment of transport protocols was easier.

1.4 Alternative Transport Protocols

In the five layer Internet protocol architecture the transport layer is the layer on top of which application data is added. The overwhelming majority of transport layer traffic on the Internet is either transmission control protocol (TCP) or user datagram protocol (UDP). TCP provides a reliable byte-stream oriented service which guarantees the in-order delivery of data. As the only reliable transport layer protocol in common use TCP is used for many services including HTTP, SMTP and most instant messaging services. UDP provides an unreliable datagram service that does not guarantee data delivery in order or otherwise. UDP is most often used where either the loss or reordering of data is acceptable or for very short transactions such as DNS requests where the overhead associated with establishing a TCP connection may be prohibitive.

In terms of Internet time both TCP and UDP were defined a long time ago. TCP was defined in RFC 793 in 1981 [36] and UDP in RFC 768 in 1980 [37]. Since that time there has been a lot of work on the TCP congestion control algorithm but the protocol itself has seen very little change besides the addition of a few protocol options [21]. UDP has seen even less change but given the simplicity of the protocol this is hardly surprising. From the lack of change in TCP and UDP and the limited use of other transport protocols one might assume that there is no desire for significant change at the transport layer. Upon further investigation is it is clear that this is not the case. In fact, there are several transport layer protocols which have been defined in RFCs similarly to UDP and TCP but which have not seen wide deployment.

The stream transmission control protocol (SCTP) [51] is one example of a new transport protocol which could replace TCP in many situations. SCTP is a reliable transport protocol which was originally conceived for use as a signaling protocol within telecommunications networks. SCTP offers several features not found in TCP. The most obvious of these features is a records based architecture. TCP is a byte-stream protocol. That is, TCP ensures a simple byte-stream is transmitted to the remote end but it does not see any structure within that byte-stream. Data is delivered to the receiving process when the number of bytes received reaches a set threshold. This complicates application development because the size of reads on the receive side does not necessary match the size of writes on the send side. As a result the receiving application may receive partial records when reading and would then be forced to store the partial read for use when the rest of the data arrives. SCTP on the other hand allows the application to delineate message boundaries within the stream. The receiving protocol implementation is then able to pass these individual messages to the application when they have been completely received. Another interesting feature provided by SCTP is virtual connections within a single connection. This feature allows two communicating peers to essentially mark parts of the communication as being separate logical communication channels. For instance a single SCTP connection may be used to transmit both signaling and media data and the receiving application can choose to read from either virtual connection. Although virtual connections could be accomplished using a single TCP connection it would require the application to delineate portions of the data stream.

However, the byte stream nature of TCP makes it impossible for the receiving application to read from a virtual connection of its choice as data is only passed to the application in the same order it was sent. SCTP is also capable of multihomed operation which allows two nodes with more than one communication path between them to use the second path for redundancy. Although SCTP offers many features that benefit both the user and the application developer applications which make use of SCTP are hard to find.

The datagram congestion control protocol (DCCP) is another example of an interesting transport layer protocol which is seeing little use at the present time. DCCP is similar to UDP in that it is not connection oriented and it is an unreliable protocol. What DCCP adds to datagram based delivery is congestion control that is TCP compatible. Many latency sensitive applications such as streaming media and gaming make use of UDP instead of TCP. The reason for this is that while UDP packets do not travel any faster through the Internet than TCP packets, latency sensitive streams see performance benefits using UDP because UDP does not enforce in order delivery like TCP does. This allows lost packets to be ignored instead of adding latency to the data delivery as the missing piece of data is retransmitted. The rising amount of UDP data poses a problem for the Internet in that TCP data flows are nice to the network in the presence of congestion while this is not true of UDP packet flows. TCP responds to data loss by reducing the transmission rate. This model generally assumes that packet loss is a result of network congestion. By backing off the transmission rate, congestion and the resulting packet loss can be avoided. Most UDP applications on the other hand ignore packet loss and continue to transmit packets at the desired data rate. This can be a problem since many media streams have a relatively high and constant data rate. The fear is that a large number of UDP packet streams which do not have congestion control behavior could cause a congestion collapse of the network [11]. By adding TCP compatible congestion control DCCP allows datagram based applications to also respond to network congestion. DCCP is defined in RFC 4340 [23] dated March 2006.

The existence and standardization of SCTP and DCCP shows that there is room for improvement in the Internet's transport layer protocols. Given the end-to-end design of the Internet one might expect that it is easy to deploy new protocols such as these on the Inter-

net. Unfortunately, there are at least two common situations where the end-to-end nature of the Internet is broken making the deployment of new transport protocols a much harder problem than it would be on a truly end-to-end network. The first such situation comes from the existence of NAT in the network. The problems associated with NAT have already been discussed and are not the focus of this thesis. The second situation comes from difficulties associated with developing and deploying new protocols and custom network stack software using the in-kernel network stack model.

1.5 The Application as the End of the Network

The end-to-end principle is a powerful design principle that brings simplicity, scalability and the ability to easily deploy new applications and services to a network architecture such as the Internet. The ends of the Internet are usually considered to be the devices which do not forward packets for other network nodes. By this definition the normal laptop computer, server and mobile device are all considered end nodes while a router obviously is not. This definition of an end node encompasses the device hardware, the operating system and any applications executing on the operating system. Given the in-kernel network stack architecture used by most Internet end nodes a more accurate definition of the ends of the Internet may be the operating system kernel itself. It is the in-kernel protocol implementation that is the consumer and originator of all network packets. This is made more clear by the fact that applications can only interact with the network via the very limited sockets interface. This restricts most applications to simply being a source and sink for data transferred via the network not an actual component of the network itself.

The strange thing about considering the operating system kernel the end of the network is that the kernel is very seldom the true source or destination of any network communication. The kernel exists to manage the hardware resources for its hosted applications. By itself the kernel does not offer useful services to external entities. Consider managing a server through the Secure Shell (SSH) protocol. The destination of the network communication is the SSH protocol daemon not the operating system kernel. It is the SSH daemon that

executes the command shell process that allows the administrator to manage the system. The commands executed within this command shell may request a service from the kernel, perhaps loading a file from disk but this has little to do with the actual SSH network protocol connection. Similarly when a web browser requests a file from an HTTP server it is not interested in communicating with the remote kernel. Rather it is asking the HTTP server process for a service. This HTTP server process may itself request that the kernel load a file or perform some other hardware related task but this is irrelevant to the web browser requesting the file.

Since applications are the true consumers and originators of data transmitted through the network it is the applications themselves which should be considered the ends of the network not the operating system kernels upon which they are executed. This insight may seem strange but it is not new [14]. When the application is considered the end of the network the common in-kernel network stack architecture becomes a major impediment to end-to-end connectivity. The reason for this is that the in-kernel network stack isolates the application behind the in-kernel protocol implementations thereby making it impossible for the application to deploy custom network protocols or even a customized network stack. One way to conceptualize this idea is to consider the kernel a node in the network which allows only select data to pass through. This has the effect of destroying end-to-end connectivity to the applications at the end of the network.

1.6 Thesis Focus

Although the widely deployed in-kernel network stack architecture does not give applications end-to-end connectivity this thesis takes the view that applications should also enjoy the benefits of end-to-end connectivity. This thesis offers an alternative network stack architecture which reduces the operating system kernel's interaction with network traffic to that of an IP router. At the same time the application is given the ability to transmit and receive entire IP packets making it a true part of the network and thereby creating true end-to-end connectivity for applications. This new network stack architecture is referred to as

the *IP per process model*.

The remainder of this thesis is organized as follows. Chapter 2 introduces the concept of user-level networking and describes previous work in this area that is related to this thesis. Chapter 3 describes the IP per process network stack architecture which brings end-to-end connectivity to applications. This chapter also discusses the advantages and consequences of this new architecture. Chapter 4 introduces a prototype implementation of the IP per process model. This prototype consists of two parts: a Linux kernel module named Pnet and a user-level network protocol library named UNL. Chapter 5 presents a performance evaluation of the Pnet/UNL prototype. Finally, chapter 6 presents conclusions and ideas for future work in this area.

Chapter 2

Related work

In order for the application to be a true part of the network it must be capable of processing and generating its own network packets. The technique of placing network protocol processing within the application process is usually referred to as *user-level networking*. User-level networking has existed in the computing literature since 1993 [53]. Since user-level networking is key to the network stack design presented in this thesis the many user-level networking projects discussed in the computing literature form the work most closely related to this thesis.

User-level network stacks can be implemented as a shared library to which the application can link against at runtime. This architecture allows the reuse of the protocol implementation by several applications just as a single GUI toolkit library may be used by several applications on a single system. Although there are many possible reasons to use user-level

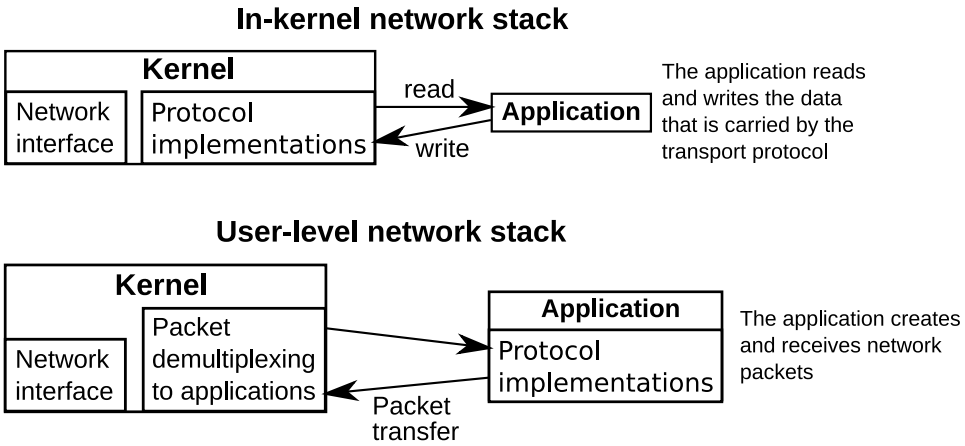

Figure 2.1: User-level Networking

networking there appears to be four primary goals which drive the development of most user-level networking implementations. These four goals are: performance, maintenance, customizability and ease of development.

2.1 User-level Networking for Performance

There is an enormous disparity between the relative performance of CPUs and the memory subsystems in commodity computers [30]. In the context of network stacks this disparity means that it can take longer to load the packet data into the CPU to be operated on than the time required for the actual protocol operations. As a result large performance benefits can be realized if the required data is still in the cache hierarchy when it is needed. This fact makes the avoidance of cache misses incredibly important to any network stack design.

Traditional in-kernel network stacks use three separate contexts. The first moves the data from the network interface into the main memory. The second performs protocol processing and queues the data into the application's socket buffers. The third context is the application itself (see figure 1.5). Between each of these contexts the operating system kernel makes a scheduling decision. If the scheduled task is the next step in the protocol processing, the packet data will still be in the cache hierarchy thereby saving the relatively large amount of time required to load it from main memory. If the next scheduled task is not the next step in the protocol processing or if the task is scheduled on a different CPU it is much more likely that the data will be flushed from the cache as the subsequent tasks may use the same cache lines. For example, an application which is not scheduled immediately after the protocol processing is complete may be forced to load the packet data from main memory again even though the kernel performed the same memory loads recently. User-level networking places the network stack into the same context as the rest of the application. The result is that an application which consumes the data immediately after the protocol processing is complete should almost always find the data in the cache hierarchy. This will provide a performance benefit. Of course the operating system can preempt an application process at any time to execute another process so this will not always be the case.

Another performance related benefit of user-level networking comes from the ability to avoid the operating system kernel completely. This has the advantage of removing the overhead of context switches and the associated cost of cache flushes. The typical approach is to give the application some form of direct access to the network interface which does not require interaction with the operating system kernel. These designs often involve custom network interfaces which contain more intelligence than a standard Ethernet network interface.

As discussed in section 1.3 the in-kernel network stack can be a performance and scalability problem on SMP and multicore systems because it is a point of centralization. User-level network stacks on the other hand do not share any state information between instances so the lock contention problems created when using the in-kernel network stack will not exist. Notice that is this actually an end-to-end argument. By pushing the processing out to the application little state information needs to be maintained in the kernel.

2.2 User-level Networking for Customizability

One of the major downsides to the in-kernel network stack model is that it is a one size fits all solution. All applications choose from a very limited selection of transport protocols and are essentially forced to use one interface, the sockets API, for all network communication. In contrast user-level implementations can allow for almost any aspect of the network stack to be customized. The only constant is the interface between the application and the network. Depending on the user-level implementation this interface could be as simple as passing complete IP packets making it is no more limiting than a physical network interface.

Perhaps the most obvious point for network stack customization is the transport protocol. This could include deploying a more advanced standardized protocol such as SCTP or slightly modifying TCP for better performance in a particular environment. For example TCP reacts to packet loss by slowing the rate at which it sends new data. This model assumes that packet loss is the result of network congestion. However, some network technologies, wireless links in particular, are inherently lossy. When operating on networks

like this TCP will not be successful in reducing the number of lost packets by slowing the transmission rate. Instead throughput is reduced without reason. User-level networking makes it possible for the application developer or administrator to easily switch to a TCP variant with very different congestion behavior when the situation demands. Perhaps the most interesting possibility for user-level network is the ability to deploy domain specific transport protocols. Such a protocol could be tuned for a specific environment or designed to have exactly the features required for a demanding application.

Moving network protocols into user-space also gives the application the ultimate decision about which packets should be processed. An application could choose to process packets and do any associated work from a high priority connection before a lower priority one. One possible application of this technique would be to prioritize the processing of media packets for one of multiple senders in a multimedia conference application. Using the in-kernel model the application could choose to ignore data from a low priority sender but the CPU time required for layer three and four protocol processing will have already been consumed.

A less obvious place to customize the network stack is with the use of application specific information [53]. TCP's byte-stream abstraction forces all data to be acknowledged. However, there may be situations where the application messages alone indicate the receipt of data making the transport protocol's acknowledgments unnecessary [14]. Another common use of this technique may be for the application to send a single packet instead of two smaller packets. This would be possible if the application was expecting information to arrive soon which would change or add to the outbound data. By delaying the transmission for a short time the application can amortize packet header overhead which can by very high for small packets. In-kernel implementations of the TCP protocol use a heuristic called the Nagle algorithm [50] to accomplish this.

The sockets API provides a convenient high-level interface to the in-kernel network stack. There is no reason this API could not be duplicated by a user-level network stack in order to achieve compatibility. However, some applications may benefit from a different API. One possible option is the use of share semantics. The sockets API uses copy semantics

which means that when data is written to or read from the socket buffers a copy is made. The advantage of copy semantics is simplicity. After writing the data to the socket buffer the application is free to reuse the buffer and not concern itself with the kernel's operations. Similarly the kernel is free to reuse the socket buffer once the data has been copied into user-space. The downside of copy semantics is the performance cost. Copying data is a relatively high cost operation. This is especially true in current systems where the gap between CPU performance and memory bandwidth is quite large [30]. Data copying coupled with the very high data rates found in one gigabit and ten gigabit commodity networks leads to a situation where the network throughput bottleneck is memory bandwidth not CPU resources. According to [12] data copies and the checksum operation account for 41% of TCP's resource consumption. Share semantics essentially passes references to data buffers instead of copies of the data buffers. This allows the copy operations to be avoided but the cost is more complicated accounting of buffer usage. It is also interesting to note that the ability to use the virtual connections and the records based capabilities of the SCTP protocol also require changes to the standard sockets API [52]. It is quite possible that any transport protocol which differs significantly from TCP or UDP may require some amount of API change.

2.3 User-level Networking for Ease of Development

As discussed in section 1.3 protocol development using the in-kernel model is difficult. The protocol developer does not have access to many of the tools commonly used with user-space development or to the many libraries that make development faster through code reuse. Also, any faults can result in the corruption of the operating system kernel and thus bring down the entire system. In contrast user-level protocol development is much easier because all of the tools normally available to application developers can be used and any faults are isolated to the application process.

Another interesting possibility opened up by user-level networking is the ability to implement the network protocol stack in a high level language. Operating system kernels are

typically implemented in C/C++. As part of the kernel the network protocol implementation must also use these languages. However, a user-level protocol implementation can use whatever language or runtime environment is deemed appropriate. Imagine being able to prototype a new transport layer protocol and application in Python or Java. This could vastly reduce the time required for implementation even over a C/C++ user-level implementation.

2.4 User-level Networking for Ease of Maintenance

The maintenance and reliability aspects of user-level networking are very much related to the ease of development. When using an in-kernel network stack applying bug fixes to the protocol implementation requires booting the new kernel. This necessitates a complete reboot of the system. A user-level networking implementation on the other hand could simply install the new protocol library and restart the affected applications. The new protocol implementation will be linked against the application when it is restarted. This should result in a much shorter downtime and also has the benefit of not affecting the other applications running on the system.

2.5 Related User-level Networking Projects

There are a large number of user-level networking projects in the networking literature. This section briefly introduces the work that is in some way similar or relevant to the network stack design presented in this thesis. The goals for each of these projects are typically one or more of the common user-level networking goals discussed in sections 2.1, 2.2, 2.3 and 2.4.

Thekkath, Nguyen, Moy and Laxowska (1993) [53] provide the earliest description of user-level networking. This paper introduces most of the potential benefits that drive user-level networking including: performance, ease of prototyping, debugging, maintenance, co-existence of many different protocols and the ability to exploit application specific in-

formation. A trend towards a larger number of transport protocols is noted by the authors. It is interesting that this trend has not come to fruition as TCP and UDP are still by far the most commonly used protocols.

Druschel, Peterson and Davie (1994) [5] describe a hardware based user-level networking design. The hardware is the OSIRIS ATM network interface. Given the age of the design the hardware details are not interesting in the context of this thesis. However, the work also introduces the concept of application device channels (ADCs). ADCs provide applications with restricted but direct access to the network interface hardware allowing the common data path to bypass the operating system kernel. ADCs are implemented by mapping some portion of the OSIRIS's on-board memory into the virtual memory region of the process. This mechanism allows the application to implement all protocols in user-space. On the receive side the adapter is able to demultiplex directly to the correct receiving process by using the VCI field of the ATM header. This design requires each connection to be assigned a unique VCI value.

The U-Net system by Eicken, Basu, Buch and Vogels (1995) [8] is another example of a hardware based user-level networking design. The main arguments used in favor of user-level networking in this paper are performance and flexibility. Like OSIRIS, the first version of U-Net uses fields in the ATM header to identify application instances. A later version of U-Net [59] uses a modified version of Ethernet. The U-Net architecture gives each application a virtual network interface. Protection of the network and other applications is obtained by restricting control of connection setup and tear down to the kernel. This design also has the benefit of bypassing the kernel during normal network communication.

The Arsenic Gigabit network interface developed by Pratt and Fraser (2001) [34] provides a more modern hardware based platform for user-level protocol development. Arsenic provides applications with their own virtual network interface but unlike the OSIRIS and U-Net systems Arsenic works with normal Ethernet and IP networks. This is accomplished by adding features not normally found in Ethernet network interfaces. One of these features is the ability demultiplex the received frames directly to the correct application without involving the kernel. The operating system is responsible for providing the Arsenic network

interface with packet filters which it executes to demultiplex packets. Arsenic also implements filtering on the send path to ensure applications do not send malformed or malicious network traffic. Rate control is also implemented to stop applications from flooding the network.

Alpine [9] is a user-level networking implementation which focuses on reducing the amount of time required to develop network stack modifications. This goal is accomplished by virtualizing all of the services the FreeBSD network stack normally expects from the rest of the kernel infrastructure such as interrupts and timing information. The result is that the same network stack implementation can be executed in either kernel-space or user-space. This reduced the amount of time required to implement new protocols or add new features to the FreeBSD network stack. Once the features have been tested in user-space the changes are easily moved back into kernel if desired. In order to transmit complete packets Alpine makes use of raw sockets. On the receive side Alpine installs Libpcap [28] packet filters so that the user-space stack does not receive packets destined for the kernel network stack.

Braun, Diot, Hoglander and Roca (1995) [3] introduce a user-level networking implementation which separates the traditional in-kernel network stack into two parts and adds a new socket type. This design splits the in-kernel TCP implementation into an in-kernel component called TCPK which provides demultiplexing for incoming packets and TCPU which is the user-level library. A new socket type is added to the kernel which allows for the transfer of TCP segments to user-space. Since this design only allows for user-space implementation of TCP demultiplexing to the appropriate user-level stack is easily accomplished through adding a small amount of logic to also look at the TCP port information.

Paris, Gulias and Valderruten (2005) introduce a user-level TCP/IP stack implemented Erlang [32]. Erlang is a functional programming language designed for building fault tolerant distributed systems. An interesting aspect of this particular user-level stack is that the features of Erlang are used to build a fault tolerant network stack. This network stack replicates TCP state information on other nodes of the cluster in order to allow a connection to be recovered even in the presence of a node failure. The Erlang system uses a virtual machine similar to the ones used by Java and the .Net platform. This alone makes the use of Erlang

within a traditional operating system kernel impossible. In order to provide the user-level network stack access to the network a raw socket is used. Packets are demultiplexed to the correct network stack instance through the use of an unique Ethernet MAC address for each network stack. This arrangement is fine for an experimental system but fails in a production network because the use of raw sockets requires administrative privileges and each application instance in effect changes the network topology. Nevertheless, this paper does show a very innovative use of user-level networking.

In a paper that touches on many aspects of user-level networking Ganger, Engler, Kaashoek, Briceno, Hunt and Pinckney (2002) introduce a user-level networking implementation based on the Exokernel [14]. The Exokernel is a minimal operating system kernel whose only task is to time share hardware resources. Each application process is provided low level access to the hardware. The goal is to reach a high level of performance and flexibility by removing the intermediate layer (kernel) from common operations. In order to demultiplex packets to the correct application this design makes use of a packet filter engine called Dynamic Packet Filter (DPF) [10]. DPF uses dynamic code generation to construct efficient packet filters. Applications request access to particular set of packets (eg TCP port 80) by constructing the DPF filter and passing it to the kernel. It is up to the kernel to verify that this new filter does not conflict with other installed filters. The design presented in this paper is very interesting but it does require a specialized OS that is not used in production networks. This work is also noteworthy because it is the first work to make the observation that the application is the end of the network.

A recent example of user-level networking can be found in a presentation by Jacobson and Felderman (2006) [22]. In this presentation the authors introduce a user-level network stack with good performance characteristics. The design presented by the authors uses asynchronous queues between the operating system kernel and the application to transmit entire packets allowing all transport protocol processing to occur at user-level. Testing of this prototype shows a 75% reduction in protocol processing overhead over the TCP implementation found in the Linux kernel. This is especially impressive given the fact that the Linux kernel is considered the fastest network stack available [22]. Communication latency also shows improvement. This design introduces the idea of a transport signature to

direct packets to the correct application but the workings of this transport signature are not discussed. While the authors of this work explicitly state that this design is based on the idea that the application is the end of the network the use of an in-kernel transport signature limits some aspects of end-to-end communication. Since the time of the presentation (January 2006) no more details about this prototype have been made available.

The ability to provide network quality of service (QoS) features is one aspect of user-level network stacks that is not mentioned often but is the goal of a few papers [60][16]. One of the benefits of user-level networking for QoS is the reduction of hidden scheduling effects. Operating systems have many high priority tasks which may interrupt processing at any time. Also, the in-kernel network stack processes network data for all applications in the system. Both of these aspects allow the traffic from one application to affect the latency of network processing for other applications. By moving all protocol processing into a single context the opportunities for scheduling is reduced potentially resulting in better QoS performance. The ability to provide better QoS characteristics than the in-kernel network stack is not a goal of this thesis but the possibility is mentioned for completeness.

2.6 User-level Networking and Memory Swapping

The in-kernel network stack model is by far the most common network stack design. According to [22] the fact that most operating systems use this design is an architectural problem created by historical accident. The first operating system to contain a network stack was Multics in 1970. Multics ran on a GE-640 super-computer which was capable of 0.4 million instructions per second (MIPS). For comparison purposes a current Intel Core 2 Extreme X6850 CPU is capable of 54,960 MIPS [27]. ARPAnet (the precursor to the Internet) performance depended entirely on how fast Multics could empty its six IMP buffers. This combined with the fact that it took up to two minutes to swap in a new user meant that the network stack had to be placed in the kernel where it would not be swapped out of RAM. Inspired by Multics the in-kernel network stack design was then used in Berkeley Unix and became the standard network stack model.

Given that the in-kernel network stack design was motivated by the requirement of avoiding memory swapping one has to wonder what effects swapping will have on modern user-level implementations. This is a topic which none of the user-level network work found in the computing literature addresses. It is unclear if research in this area is simply being avoided or if there is a consensus that modern computers have enough RAM and are fast enough that this is no longer a problem. Interestingly, Jacobson describes a new user-level network stack in the same presentation where the original motivation for the in-kernel network stack is explained. Intuitively one might expect memory swapping on modern hardware to not have a large impact on user-level networking for several reasons. Firstly, the entire process will not have to be brought back into RAM to perform some small amount of protocol processing. Only the memory pages relating to the network protocol implementation will be required. Secondly, an application using the in-kernel network stack also has the potential to stop or slow network communication when it is swapped out of RAM because the application may not execute fast enough to process the data in the receive socket buffers. The operating system network stack can accept no more data when the application's socket buffers become full. The work in this thesis does not investigate the effects of memory swapping on user-level protocol implementations.

2.7 Summary

User-level networking is a powerful network stack design technique which places network protocol implementations within the application instead of the operating system kernel. This design has many benefits in terms of performance, customizability, ease of development and ease of maintenance. The concept of user-level networking and related works have been introduced in this chapter because user-level networking is a very important part of the IP per process network stack model presented in the next chapter.

This chapter also introduced user-level networking works found in the computing literature which are in some way similar to the work presented in this thesis. These works range from designs which are primarily hardware based to a complete network stack written in the

Erlang functional programming language. Two works of special interest are [14] and [22] both of which explicitly argue that applications are the ends of the network not operating systems.

Chapter 3

The IP Per Process Model

This thesis is not the first work to make the observation that applications are the ends of the network. While other works have accepted this observation the network stack architectures they present stop short of truly treating the application as the end of the network. This thesis presents a network stack architecture which brings end-to-end connectivity and all of its advantages to applications.

In IP networks, the ends of the network are identified by globally unique IP addresses. If applications are indeed the ends of the network and not the operating system kernel, then each application should be identified by an unique IP address. As long as intermediate nodes, including the operating system kernel, forward packets sent to and from the application in the same manor as any other IP router this model easily achieves all of the benefits of end-to-end functionality for the application. This network stack model which transfers complete IP packets to and from the application process and identifies each application by an IP address is referred to as the IP per process model for the remainder of this thesis.

This chapter introduces the benefits that the IP per process model has over other other user-level networking designs and the traditional in-kernel network stack design. In general these benefits come from the architectural simplification created by treating applications as the end of the network. Later sections discuss some consequences of the IP per process model as well as possible designs for its implementation.

3.1 Advantages of the IP per Process Model

The single overwhelming advantage that the IP per process model has over the in-kernel network stack and other user-level networking designs is that this model acknowledges the idea that the application is the end of the network and extends the benefits of the end-to-end principle to applications. All of the other advantages discussed in this section are the result of this.

Chapter 2 introduced user-level networking and its many advantages. Since the IP per process model relies to a large extent on user-level networking the advantages of user-level networking also extend to the IP per process model. Where the IP per process model really shines is in simplifying many of the problems that make other user-level networking implementations overly complex.

3.1.1 Transport Layer Port Usage

Perhaps the largest source of extra complexity introduced by user-level networking is the distribution and maintenance of state information among the individual stack instances. In many cases the distribution of state information to the application is a very good thing. For instance per connection TCP state is not in any way shared with the TCP flows from other applications so following the end-to-end principle this data is best placed in the application's network stack. Unfortunately, there is some state information which must be consistent between all user-level network stack instances which share a single IP address.

TCP and UDP identify the application which is sending and receiving packets by a source and destination port number. The selection of the port number that the application uses can be made by the application or a free port can provided by the network stack to the application. If the application is initiating a connection it will usually allow the network stack to choose the port number because there is no benefit to using a specific port. On the other hand if the application is offering a service to other hosts the port used by the application is very important. Offering SMTP service requires listening on port twenty-five, HTTP requires listening on port eighty. These port assignments must be unique across all

network stack instances which share an IP address. If two applications were both allowed to listen on port twenty-five the kernel would have no way of deciding which application should receive the packets.

The fact that transport layer port assignments must be unique poses a problem for most user-level stack implementations. An individual user-level stack cannot simply choose a port that it is not currently using because another network stack instance may be using that particular port. Working around this problem requires the individual network stack instances to negotiate port usage or the introduction of a intermediary process to arbitrate port usage.

The Alpine project [9] virtualizes the FreeBSD network stack in order to allow protocol development in user-space. To avoid conflicting transport layer port usage Alpine employs a central server process. When an application using the Alpine network stack wants to use a port it asks this central server. If the port is not in use then the server creates a dummy socket with the in-kernel network stack to ensure that the in-kernel stack does not use this port for an application which is not using the user-level stack.

The approach taken by [53] also uses a central server process called a registry server to manage port usage but this server is given even more responsibilities. Instead of just arbitrating port usage and leaving all protocol operations to the user-level network stack the central server in this design is also responsible for connection establishment and tear-down. For example to establish a TCP session the user-level stack asks the registry server to do the TCP connection establishment. The registry server then provides the user-level stack with a transmission channel.

A more decentralized port arbitration scheme is used by [14]. The scheme used by this work is to have the application select the port number it requires or if no specific port is required to randomly pick one. This particular user-level design requires the operating system kernel to maintain packet filters which are used to direct packets to the appropriate destination application. The kernel is also able to detect conflicts in these filters. If the port requested by the application causes a conflict the kernel will refuse to install the packet filter thus not allowing the reception of the potentially conflicting packets.

The user-level network stack described in [32] side steps the port allocation problem by requiring that every user-level stack is assigned its own unique MAC address. In effect each user-level stack instance functions as a separate node on the attached Ethernet network. This design requires access to a raw socket in order to capture all packets on the local network.

Another low level approach to the transport port allocation problem can be found in U-Net [8]. U-Net relies on ATM VCIs to identify the packets destined for a particular application. While this approach does avoid conflicting transport layer ports it simply moves the problem to a lower level. Instead of a transport layer port server U-Net has a server process to establish these low level identifiers. The obvious deficiency of this approach is that it is not applicable to the public Internet.

Although the port usage problem can be solved with a trusted server or an in-kernel service as is evidenced by the preceding approaches this model does introduce some problems for user-level network implementations. Firstly, the entity performing the port arbitration must have some understanding of the transport layer network protocol in order to successfully arbitrate port usage. This requirement makes it impossible for applications to deploy transport layer protocols which the port arbitrator does not support. Secondly, all user-space network stacks on a given system will be forced to negotiate port allocation using the same method. While this is possible it may increase the difficulty associated with porting an user-space network stack from one operating system to another.

All of the complexity associated with transport layer port usage comes directly from the loss of end-to-end connectivity between applications. The fact that the transport layer port space is shared state is not apparent when using the in-kernel stack but becomes a major complicating factor once multiple network stacks are deployed.

The IP per process model of user-level networking neatly side steps all port allocation problems by bringing end-to-end functionality to the application. When each network stack in the system has its own IP address there is no shared port state and thus no state synchronization required between stack instances. This model allows the application's network stack to use any transport layer port it wishes and indeed any transport layer protocol it wishes because each user-level stack instance is completely independent.

An interesting side effect of the IP per process user-level networking model is the ability to offer multiple instances of the same service on a single operating system instance. Normally an operating system can only host one SMTP service because only one process can be listening on port twenty-five on a single IP address. However, when using the IP per process model every application instance has the ability to be a SMTP server without causing conflicts.

3.1.2 Demultiplexing

Closely related to the transport layer port allocation problem is packet demultiplexing. Packet demultiplexing is the task of deciding which application or user-level network stack instance should receive a particular packet. When using the traditional in-kernel network stack design this process is relatively simple. For example when a packet is received by the IP processing code it can look at the protocol identifier stored in the IP packet and decide which transport layer input function to call. The transport layer input function can then process the packet and decide which application socket should receive the data. When user-level networking stacks are introduced things become slightly more complex. It is no longer possible for the IP layer to identify the next step in the protocol processing from the transport layer protocol identifier alone. Instead, the data within the transport layer header must be inspected. There are two possible solutions for demultiplexing packets to the correct application: transport layer specific demultiplexing and general packet filters.

Transport layer specific demultiplexing is the simplest of the two options but comes at the cost of flexibility. For example the user-level network stack in [3] only allows for user-space implementation of TCP. This design is not a general user-level network architecture. As a result demultiplexing to the user-level stack simply consists of first comparing against a list of TCP ports used by user-level stack instances and if none are found the packet is passed to the in-kernel network stack for further processing. Transport layer specific demultiplexing has the disadvantage of limiting user-level stacks to using only those transport layer protocols for which the operating system kernel contains filtering features.

Another approach to packet demultiplexing is to use a more general packet filtering mech-

anism. General packet demultiplexing is a complex topic to which *Network Algorithmics* [56] dedicates an entire chapter. The discussed packet demultiplexing approaches include: the Berkley packet filter (BPF) [29], Pathfinder [2] and Dynamic Packet Filter (DPF) [10]. BPF was designed to be a high speed mechanism for selecting which packets to capture for use in network monitoring tools such as tcpdump. Pathfinder fills a role similar to BPF but it is designed to support a much higher number of filters and with greater performance. In order to accomplish this Pathfinder creates a trie where each node in the trie is a comparison to be made against the packet being processed. Finding a match using Pathfinder consists of finding the longest matching path in the trie. Finally, DPF is a compiler based approach to packet demultiplexing. Dynamic code generation is used to create what is basically an optimized version of the pathfinder trie for any given filter set. While DPF is much faster than Pathfinder, dynamic code generation adds complexity to the implementation.

The Alpine project [9] accomplishes packet demultiplexing through the combination of a packet capture library and firewall rules. The packet capture library Libpcap [28] is used in combination with a raw socket to obtain a copy of all packets on the network link. Since Libpcap captures packets but does not stop further processing firewall rules are also installed by the central port server to stop the in-kernel network stack from receiving packets destined for a user-level stack instance. Other approaches the packet demultiplexing include [53] which requires either hardware support for packet demultiplexing or the use of BPF in the kernel and [14] which makes use of DPF. Also interesting is the Arsenic network interface hardware presented in [34] which performs packet demultiplexing on the network interface itself.

Another possible method to demultiplex packets is to simply demultiplex at a lower layer and completely ignore the transport layer information. Examples of this method can be found in [8, 59]. These user-level network designs use information in the ATM header or a modified Ethernet header to choose the destination application. These approaches require that these identifiers be negotiated between the two hosts on a per connection basis and thus do not generalize to functioning on the public Internet.

The lowest layer which is common across the entire Internet is the network layer, specifi-

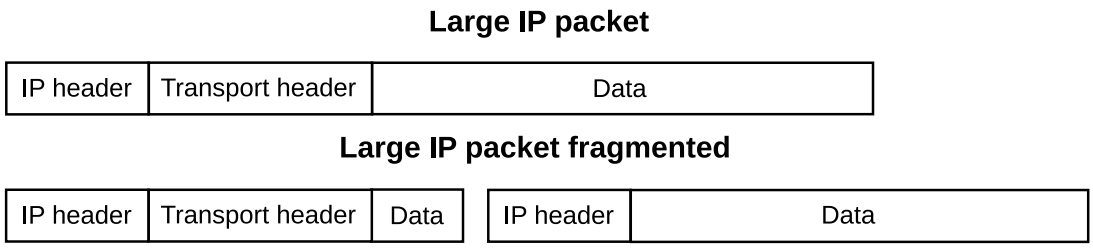

Figure 3.1: IP Fragmentation

cally the IP protocol. This makes the IP address the lowest layer on which demultiplexing can occur on the Internet. By demultiplexing at the IP layer the IP per process model reduces application demultiplexing to IP routing. IP routing is a simple and scalable operation that does not have the complexities of general purpose filters or the limitations of transport layer specific filtering.

Another factor which complicates packet demultiplexing is IP fragmentation [6]. IP packets moving through the network may cross many different layer one and layer two networking technologies. These technologies may have different limitations on the maximum size of a data packet. This maximum packet size is typically referred to as the maximum transmission unit (MTU). If an IP packet arrives at a router and it is larger than the outgoing interface's MTU the IP packet will be broken into two or more smaller IP packets called fragments. Each of these fragments contains some portion of the original packet's data payload. The fragments will be sized to allow their transmission on the outgoing interface. This allows the data to continue towards its destination.

Each individual IP fragment has the information necessary for IP routers to forward it towards its destination. Fragments that have reached the destination must be buffered until all other fragments have been received and the original IP packet can be reconstructed or some predefined time expires and the fragments can be discarded. This buffering is necessary because one or more of the IP fragments may be lost in the network. Although the IP protocol does not offer reliable packet delivery it does assure that the packets which do arrive are intact. If a network stack were to pass only partially reconstructed IP packets to the transport layer arbitrary portions of the packet could be missing. The application would have no way to know which portions were missing and any length information in the

transport layer header or application data would be meaningless.

The in-kernel network stack and most user-level network stack designs perform transport layer demultiplexing within the kernel. This forces IP fragment reassembly to also occur at this point because transport layer demultiplexing cannot be completed until the complete IP packet has been received. The reason for this is that only the first fragment contains the transport protocol header (see figure 3.1). All in-kernel network stacks implement IP fragmentation within the kernel so it is also possible for user-level networking designs to do the same. However, IP fragment reassembly is a relatively complex task with a history of bugs in multiple implementations [57, 58, 61]. The IP per process model has the advantage of moving IP fragment reassembly into the application along with the transport protocol implementations. This allows the kernel to forward IP packets to the application without any concern for whether or not each packet is a fragment just as every other IP router on the Internet does. This simplifies the kernel and thereby reduces the possibility for kernel implementation bugs. IP reassembly bugs found in user-level implementations will be much easier to fix for all the same reasons protocol development is easier in user-space. Another advantage of user-level IP fragment reassembly is that the fragments waiting to be reassembled are stored in the application's virtual memory space. An application's virtual memory space is very large and can be swapped to disk when the system requires free RAM. This avoids using kernel memory for fragment reassembly which some operating systems cannot swap to out of RAM.

3.1.3 Error Message Demultiplexing

Error message demultiplexing is a special case of demultiplexing with its own set of difficulties for user-level protocol implementations [14]. Consider an application attempting to initiate a connection with a remote host. Hopefully the connection will be successful and the destination will reply. Of course there are also many errors which are possible. For example the destination host could be behind a firewall or be offline. In both of these cases an intermediate node in the network path may send an error packet to the source node informing it of the problem.

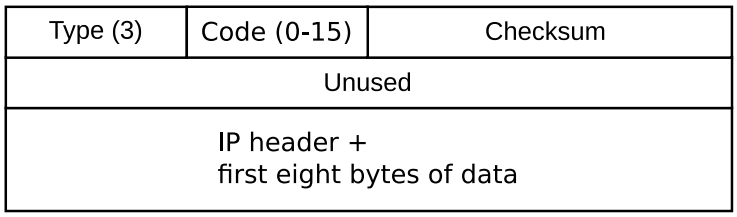

Figure 3.2: ICMP Unreachable Packet

IP networks signal errors using the Internet control message protocol (ICMP) [35]. In the case of the destination node being offline the message received by the source host could be an ICMP destination unreachable message. See figure 3.2 for the layout of this type of ICMP message. Notice that unlike the packet that elicited this response the transport layer header does not immediately follow the IP header in the response packet. This poses problems for demultiplexing the error back to the user-level network stack that originally sent the packet. A general purpose filter installed to match the packets destined for the user-level stack will not match these error packets. This forces either an extension of the transport layer specific filter rule to also match ICMP messages relating to the user-level packet flow or a second general purpose filter to match these errors. In either case the work of the intermediary, often the operating system kernel, increases. Here again the end-to-end nature of the IP per process model pays off. Handling these errors when using the IP per process model requires no special features in the kernel beyond simple IP routing which is already necessary for network communication.

3.1.4 Network and Application Protection

User-level networking gives applications new capabilities beyond what is available to them using the sockets API. These new capabilities come from the ability to have much more control over the packets which are transmitted into the network. However, with more power comes greater potential to harm applications on the system and the attached network.

When using the in-kernel network stack design applications receive data only after the protocol processing is complete. This ensures that an application only receives network data that was destined for it. User-level implementations which rely on raw sockets to get

packets to the application do not offer the same guarantee. Raw sockets give an application a copy of every packet received on a particular interface allowing a single process to snoop traffic from every application. User-level implementations which use raw sockets include [9, 32]. User-level network designs which perform demultiplexing within the kernel or on the network interface do not suffer from this problem. Using the IP per process model all demultiplexing occurs within the kernel and is based on the destination IP address so this model is also not vulnerable to inter-application snooping.

The ability to construct and transmit whole packets is what gives user-level networking flexibility but this capability can be used maliciously. The most obvious way to abuse this capability would be for one application to spoof another's traffic. For example a process could impersonate a local SMTP daemon by transmitting packets with a TCP source port of twenty-five. One possible solution to this problem is the use of outbound filters. This is basically the inverse of demultiplexing. Before transmitting a packet the kernel or some other trusted intermediary would verify that the packet to be transmitted matches a filter. If the packet does not match the filter it is dropped. The approach taken in [53] is to have a central port server be responsible for installing these filters. This adds further protocol specialization to the central port server making it even more difficult to deploy new transport protocols. A somewhat more extreme view can be found in [14] which argues that it is not necessary to stop processes from transmitting arbitrary packets because any security relying on a such a design is faulty anyway. Just as the IP per process model greatly simplifies demultiplexing it also greatly simplifies outbound packet filtering. It is very easy to filter outbound packets based solely on the source IP address. If this simple comparison fails the packet can be discarded preventing source spoofing.

Another potential problem with user-level networking is the rate at which an application may send packets into the network. When using the in-kernel TCP implementation the application is only able to transmit data at the rate TCP will accept. This rate will depend on the bandwidth available and packet loss. Obviously an application with its own network stack could transmit data at any rate it wishes potentially causing congestion on the network. However, an application could also flood the network with UDP packets even when using the in-kernel network stack so this problem is not limited to user-level networking. Also,

it is very difficult to limit a single application's network bandwidth when using the in-kernel network stack model. In order to accomplish such a limitation all TCP and UDP packets from the application must be classified into a single group to which the bandwidth limitation can be applied. This classification requires knowledge of exactly which ports are being used by the application and denying the use of other ports. Since the IP per process model associates each application with a single unique token in the network traffic, the source IP address, it is much easier to establish per process bandwidth limits using this model.

3.1.5 Protocol Implementation Portability

Section 1.3 discusses the portability problems inherit in the in-kernel network stack model. The main difficulty is the fact that any protocol implementation developed for a particular operating system kernel will not be easily ported to another operating system because there are no standard interfaces or architectures that would make this task easier.

User-level networking can greatly increase the portability of network protocol implementations in the same way that GUI tool-kits are often portable across multiple windowing systems. This is usually accomplished through similar low level interfaces which the library can hide from the application developer. There are only two points of contact between the operating system kernel and an user-level network stack: the packet exchange interface and any interface needed to communicate transport layer port usage and demultiplexing information. Porting network protocol implementations across simple interfaces such as these is a much easier task than porting between operating system kernels. Since user-level network stacks using the IP per process model do not need to communicate layer four information to the kernel or to other user-level network stack instances the task of porting between operating system kernels will be even easier than in other user-level networking designs. In this case it is only the packet exchange interface that may differ between operating systems.

3.1.6 Kernel Simplification

Not only does the IP per process model simplify the application and the associated user-level network stack it simplifies the operating system kernel as well. In the extreme case where all applications running on a particular operating system instance make use of the IP per process model the kernel's networking features can be reduced to IP routing. That is, all transport layer protocols and associated code such as the socket implementation could be removed. This would greatly decrease the complexity of the networking code in the kernel. Of course it is not likely that any OS will be able to remove its in-kernel network stack any time soon given the large number of applications developed for this model. Even if the in-kernel stack is not removed the IP per process model offers advantages over other user-level networking designs. Any user-level networking design that makes use of the same network identifier (IP address) for both the in-kernel network stack and user-level stack instances will require modification and added complexity to the in-kernel stack. These problems which are associated with layer four state information and demultiplexing were discussed earlier in sections 3.1.1 and 3.1.2. As will be seen in chapter 4 where the prototype IP per process implementation is introduced in detail, the IP per process model can be implemented in such a way that no part of the existing in-kernel network stack requires modification.

3.1.7 Summary of Advantages

In general the benefits associated with the IP per process network stack model come from a reduction in complexity. This reduction in complexity is accomplished through following the end-to-end principle by moving as much work as possible to the end of the network, the application. There are many positive results that come from this change. Most obviously, the application receives end-to-end connectivity and all of the associated benefits. The operating system kernel becomes simpler because the only IP networking feature required is IP routing. The implementation of the user-level stack becomes easier because the IP per process model removes the need for shared state between user-level network stack instances. The interface between the application and the operating system also becomes

simpler because the only required operations are sending and receiving complete IP packets. This simple interface may also aid in making user-level networking implementations more portable between operating system kernels.

3.2 Discussion

The IP per process model has many advantages over both the in-kernel network stack architecture and other user-level network stack designs. Of course any architectural change of this magnitude is going to have consequences and downsides. This section investigates some possible criticisms and problems associated with the IP per process model.

3.2.1 IP Address Consumption

The most obvious criticism of the IP per process model is the increased consumption of IP addresses. Given the present situation where most Internet end users are only assigned a single IP address it is obvious that these users cannot deploy applications using the IP per process model without introducing NAT into the network. While introducing NAT will work if the transport protocol is supported by the NAT device it destroys the end-to-end benefits of the IP per process model. Although end users may not be able to deploy the IP per process model on the current Internet there are other situations where it can be deployed.

Unlike Internet end users, Internet Service Providers (ISPs) are allocated much larger blocks of IP addresses. Given a compelling application requiring the IP per process model or customer demand it is very unlikely that an ISP would not have IP address space available to allocate a small subnet for this purpose. Another interesting application of the IP per process model may come from private networks. Most of the high performance computing today is executed not on a single computer but a cluster of computers [54]. Given the lack of available global IP address space these clusters are typically allocated IP addresses from non-globally unique IP space [40] such as 192.168.0.0/16 or 10.0.0.0/8. When building a network with private, non-global IP addresses the available address space is effectively unlimited. The 10.0.0.0/8 block for instance contains 2^{24} addresses available for internal use.

Given the specialized nature of these clusters and the distributed computation performed there may be benefits to using custom network stacks and specialized transport protocols which could be easily deployed using the IP per process model.

While the present Internet architecture may not have enough addresses to allow every end user to make use of the IP per process model future architectures might. IPv6 was designed to be the successor to the present IP protocol which is now often referred to as IPv4. The primary advantage of IPv6 is a much larger address space. IPv4 offers users a thirty-two bit address space for a total of 2^{32} available addresses. IPv6 offers a one hundred and twenty eight bit address space for a total of 2^{128} addresses. This is 2^{96} times the number of addresses available in IPv4. The recommended deployment model for IPv6 places 2^{64} addresses on every local area network (LAN) [19]. This may seem like a waste of addresses but it allows nodes to automatically configure themselves with unique IP addresses removing the need for DHCP on IPv6 LANs. Even more interesting is the fact that the recommended IPv6 deployment model will give 2^{80} addresses to every end-site. Assuming ISPs follow this model when deploying IPv6, every user on the Internet will have ample address space to deploy the IP per process model.

3.2.2 Proliferation of Layer Four Protocols

The IP per process model brings end-to-end functionality to applications allowing for the deployment of any transport layer protocol or network stack design. Two potential problems that arise with these capabilities are protocol identifier exhaustion and protocols with congestion control behavior that is incompatible with TCP.

The IPv4 and IPv6 headers contain fields called the protocol identifier and next header respectively. These fields identify the protocol that is layered immediately on top of IP for the packet. For instance TCP is given the value of six and UDP packets are identified with a protocol number of seventeen. Without this identifier the receiving network stack has no way to decide which transport protocol should be used to continue processing. In both IPv4 and IPv6 these fields are defined to be a single byte wide. As a result only two hundred and fifty-six unique protocols can be identified. With present network stack architectures and

the associated difficulty of developing and deploying new transport protocols this relatively small protocol identifier field appears to be large enough as only one hundred and forty-one of the available protocol identifiers have been allocated [38]. However, the IP per process model makes it dramatically easier to develop and deploy new transport protocols. This could lead to exhaustion of the protocol identifier space. One potential solution to this problem may be to create a new protocol which simply adds a single field that is used to create a larger protocol identifier space. One of the remaining free identifiers could be used to indicate this new protocol. Another possible solution may be to make use of the IP protocol header options to create a larger protocol identifier field.

The second problem associated with an increase in the number of transport protocols being used on the Internet is the behavior of these protocols with respect to congestion. The congestion algorithm used in TCP attempts to be fair to other connections sharing the same network path. If congestion is detected, usually through a lost packet, TCP will back off its sending rate in an attempt to reduce the congestion. When every flow on the network implements similar behavior each connection gets a fair amount of bandwidth. With the ability to easily deploy new protocols the likelihood of a protocol without TCP compatible congestion control increases. This behavior could be malicious or simply an implementation bug. While this threat to the stability of the Internet should not be taken lightly any application with access to the UDP protocol already has a mechanism for generating traffic without TCP compatible congestion control so the threat is not unique to the IP per process model.

3.2.3 TCP Time Wait State and Quiet Time

A single TCP connection is uniquely identified by the combination of four pieces of information, a four tuple. These four pieces of information are the local IP address, the local port, the remote IP address and the remote port. TCP implementations use this information to demultiplex packets to the appropriate receiving process. Within a TCP byte stream individual bytes are associated with a thirty-two bit sequence number. This sequence number is used to reconstruct the order of the bytes in the case of lost or reordered packet delivery.

Two applications on the same system cannot use the same four-tuple at one time because there would be no way to uniquely demultiplex the packets. Also, there is a time period after one application closes a connection during which another application should not be allowed to use the same four-tuple. The reason for this is that IP networks do not guarantee delivery of packets in order or otherwise so it is possible for a single packet to be delayed significantly behind other packets in the stream if the packet takes a longer path to the destination. Fortunately, there is some bound on this delay because each router which forwards an IP packet is required to decrement the time to live (TTL) counter by one. When this counter reaches zero the packet is discarded. However, the possibility for a long delay does exist. A problem occurs when one application closes a TCP connection and another application uses a TCP connection based on the same four tuple shortly later. If a packet from the previous TCP connection is delayed but still arrives at the destination host there is no way for the receiving TCP to know that this packet contains data from a previous connection. If the TCP sequence numbers of the bytes in this delayed packet are close to the current position in the new connection then completely unrelated data may be passed to the receiving application.

The TCP protocol's solution to this problem is the time wait state and quiet time. The time wait state is the final state a TCP connection enters when the local application initiates the closing of the connection. During this state the connection has been shutdown but the state information for this connection remains active for a set length of time. This stops another TCP connection with the same four-tuple from being created. The wait time is defined to be two times the maximum segment lifetime (MSL). The MSL is an implementation defined value that ranges from thirty seconds to two minutes [48]. The time wait state covers the situation when an application cleanly closes a connection. By maintaining the state information for the 2*MSL period the TCP implementation can be sure that no stale packets matching a new connection will be received. However, the time time wait state does not help if the operating system crashes or if the system suffers a power loss. In these cases there is no state information indicating which connections may receive stale packets. Thus there is no way for the newly booted TCP implementation to avoid using these connections during the MSL period. In order to deal with this problem TCP has the idea of a quiet

time. The quiet time is defined to be MSL seconds long. During the quiet time the newly initialized TCP implementation should not create any new connections to ensure that all packets in the network have expired. However, many operating systems do not implement the TCP quiet time because most hosts take longer than MSL seconds to reboot [49].

The need to maintain state information about recently closed connections poses a problem for user-level TCP implementations. If all connection state is maintained in the application, the application cannot exit until the time wait state is complete. Three possible solutions to this problem are outlined in [14]. The first is to simply require the application process to not exit until the time wait state is complete. This solution poses problems for interactive applications such as command shells because the wait time destroys interactivity. The second solution is to extend the OS to allow the user process to signal that it is complete and have the OS remove the process from the process list even though the process continues to exist until the time wait state is complete. The third solution offered is to pass the ownership of the connection to a trusted process when the application exits. Another interesting solution to this problem can be found in [53]. This user-level implementation makes use of a special process referred to as a registry server to open and close all connections. The application is simply given a handle to the connection when it is opened and responsibility for the connection falls back to the registry server when the application exists.

Using the IP per process model, an application restart becomes very similar to a complete system restart from the standpoint of the network stack because a new, completely independent network stack instance is created. This makes the problems associated with packets from old connections more important because of the speed at which applications can be restarted. One way to deal with this problem would be to have the user-level network stack save state information when the application shuts down. This would allow the newly started user-level stack instance to avoid reusing the same connections. While this solution will work for a clean application shutdown it does not help when the application ends unexpectedly. In this situation the new user-level stack instance will either have to wait the quiet time before transmitting TCP packets or simply ignore the possibility of receiving segments from old connections like many operating systems do. An interesting way to sidestep the problem of receiving old packets which is made possible by the IP per process model is to

simply use another IP address. Of course this is not always possible because some services will be required to listen on a predefined address.

3.2.4 Connection Passing

The desire to share network connections across process boundaries also creates problems for user-level networking. A common use of this technique is to have a single process listening for incoming connections. When a connection has been established the listening process creates a child process to handle the new connection. This allows the original process to continue listening for more connections. On Unix based systems the child process is created by forking the listening process. This operation gives both the original process and the new process a copy of the file descriptor referencing the new connection. The original process closes this file descriptor and the child process uses it to communicate with the remote device. There are also other methods which enable socket file descriptors to be passed between processes which do not require the parent child process relationship.

Some user-level implementations go to great lengths to support connection passing. In [53] connection sharing is accomplished by making use of the Mach ports mechanism which is a feature of the underlying operating system. The user-level implementation in [14] accomplishes passing connections between processes by mapping the connection information into the new process and unmapping it from the original process.

The completely independent nature of network stacks using the IP per process model makes connection passing difficult. The problem is that all packets for a particular destination IP address are delivered to a single application process. One way to support connection passing would be for the receiving process to simply transfer the received packets to another process through some inter-process communication method. Given the context switches and data copies involved the performance of this option may be poor. Another possibility would be to give the kernel the ability to splice certain traffic off to another process. This solution adds significant complexity to the kernel's demultiplexing and breaks the end-to-end nature of the IP per process design.

It is important to note that the IP per process model does not preclude the use of multi-threaded applications. For example, instead of forking a new process to handle a client connection a new thread could be created instead.

3.3 Designing the IP Per Process Model

In order to achieve end-to-end connectivity for applications the IP per process model strives to push as much work as possible into the application process. This includes making use of user-level networking to place the protocol implementations within the application instead of the operating system kernel. This arrangement adds the requirement for an interface which allows the application to transmit and receive complete IP packets.

There are many possible ways that the IP per process model can be implemented within the host operating system kernel. Perhaps the simplest is to make use of raw sockets to capture all network traffic and pass it to the user-level network stack. Raw sockets allow applications to receive a copy of all packets sent to a particular network interface and to transmit packets of their own construction. The advantage of implementing the IP per process model with raw sockets is that no changes are required to the operating system kernel. However, there are several problems with this approach. One problem is that every process reading from a raw socket receives all packets on the network. This makes it trivially easy for one application to spy on another. Raw sockets also allow the application to transmit any network packet it wishes thereby allowing one user-level stack to spoof another user-level stack's address or even the in-kernel stack's address. These problems combined with the fact that the use of raw sockets usually requires special privileges for the application makes raw sockets a poor choice for the implementation of the IP per process model.

Another possible design for the implementation of the IP per process model involves adding an unique IP address for each application to one of the kernel's existing network interfaces. The advantage of this design over using raw sockets is that the kernel is in a position to limit which packets each application receives and also limit the packets that the application can send. This provides for the protection of the network and other applications from a ma-

licious application. One disadvantage of this design is that it requires modification of the kernel's packet demultiplexing to pass received packets directly to the application instead of passing them to the in-kernel protocol implementations. The management of application IP addresses on the existing network interface also poses problems for this design because adding and removing IP addresses from a network interface usually requires administrator privileges. Possible solutions to this problem include preassigning a set of IP addresses to the interface or introducing a trusted server process to manage interface addresses. The lack of globally unique IP addresses also creates difficulties for this design because each application uses an IP address from the local LAN subnet. Due to the shortage of IPv4 addresses most LANs are not assigned a subnet with a large number of addresses beyond the number of nodes on the LAN. The small number of free addresses limits the number of applications which can utilize the IP per process model. Of course the LAN could be assigned a larger subnet but it may be hard to predict exactly how many applications will require IP addresses and running out of IP addresses could result in a new physical node being unable to join the network. This will become far less of a problem on IPv6 networks. As mentioned in section 3.2.1 the recommended deployment model for IPv6 places 2^{64} addresses on every LAN. In this situation it is very reasonable to allow each application to utilize an IP address from the local LAN without any realistic fear of running out of addresses. The technology to dynamically create an unique IPv6 LAN address has been defined in [31]. This technique was developed to address the privacy concerns associated with using the 48-bit Ethernet MAC address to create a node's local IPv6 address. While assigning the application's IP address to a local interface does have advantages over using raw sockets the difficulties associated with interface permissions, modification of the existing network stack and the shortage of IPv4 addresses makes this design problematic.

A third design for an implementation of the IP per process model involves treating the operating system kernel as nothing but another IP router in the network. This version of the IP per process model consists of two parts: a kernel to user-space link layer and user-level protocol implementations. The purpose of the kernel to user-space link layer is to create a process area network (PAN) which is analogous to a local area network (LAN) but which consists of application processes instead of computers. A LAN which is connected to the

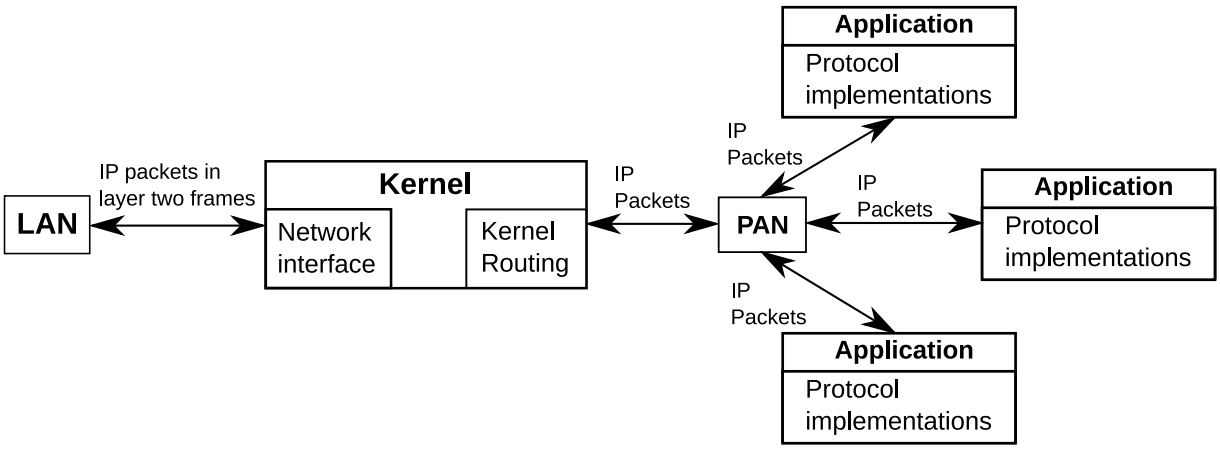

Figure 3.3: The PAN Based IP Per Process Model

Internet typically has a single device, the gateway, which connects it to the Internet. Often this gateway device will also be responsible for providing IP addresses to the individual computers on the LAN. In the IP per process model the kernel takes on both of these roles. The PAN is allocated a subnet just as a LAN is. Each application using the PAN IP per process model is assigned an IP address from this subnet and the kernel routes packets to and from the appropriate application process. The primary advantage of the PAN IP per process design is its conceptual simplicity. By extending the Internet into the host computer the PAN design follows the same principles as normal inter-computer IP networking. See figure 3.3 for a pictorial representation of the PAN IP per process model.

A PAN based IP per process implementation is conceptually very simple because it follows the same routing logic used throughout the rest of the network. However, there is one downside. Since this design relies on a separate subnet for the PAN it does require additional routing configuration in the network. Without a route entry the LAN gateway will not know how to reach the PAN. Fortunately routing protocols such as Open Shortest Path First (OSPF) and Routing Information Protocol (RIP) have been designed to dynamically manage route distribution but this is an added complexity not found in other network stack architectures.

It is worth emphasizing that the PAN based IP per process design is a routing based design. The IP packets contained in the layer two frames received by the system's network interfaces are removed from the frame and then passed to the application. The layer two headers

are not passed to the application. This behavior is identical to how an IP router moves packets between layer two networks. As a result, the PAN based IP per process design does not require each application to have its own layer two network address and it does not add any extra complexity to the attached layer two network.

As with any complex software system there are a multitude of designs that could be used to implement the IP per process model. The designs discussed here provide an overview of the possibilities and are by no means an exhaustive list.

The IP per process implementation presented in the next chapter follows the PAN design. This design was chosen because of its conceptual simplicity and the fact that it takes advantage of the routing infrastructure already found in the Linux kernel. In fact the prototype described in this thesis did not require any modification of the existing Linux network stack.

3.4 Summary

The IP per process model fully acknowledges the idea that the application is the end of the network by assigning each application an unique IP address. This allows all protocol processing from the network layer to the application layer to occur within the application process thereby establishing end-to-end connectivity for the application. A key component of the IP per process model is user-level networking. Other user-level networking designs are burdened by the complexities associated with sharing the network end point identifier, the IP address, between multiple user-level network stacks. The IP per process model greatly simplifies user-level networking by avoiding these problems.

Chapter 4

Prototype Implementation

The implementation of the IP per process model described in this chapter is based around the PAN model discussed in section 3.3. There are two components of this implementation. Pnet is a Linux kernel module which provides the PAN and associated interfaces to applications. The user-level networking library (UNL) is an user-space implementation of the IP, ICMP, UDP and TCP protocols for use with Pnet.

Figure 4.1 presents a high level and simplified picture of the Pnet/UNL design. The details of this prototype are discussed in subsequent sections. At a high level, Pnet provides UNL with ability to receive and transmit whole IP packets. Pnet accomplishes this by presenting itself as a network interface to the Linux kernel and offering a character device interface for interaction with applications. UNL makes use of the character device interface provided by Pnet in order to perform all network protocol processing at user-level.

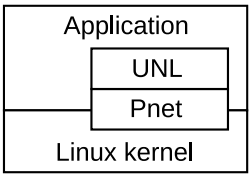

Figure 4.1: High Level Overview of the Pnet/UNL Prototype

4.1 Pnet

Pnet [46] is a Linux kernel module which provides the process area network (PAN) interface to applications. The Linux kernel uses a monolithic kernel architecture. Monolithic kernels execute all components of the kernel in the same address space without protection barriers between components. This does not require a single thread of execution. In fact, current Linux kernels employ many separate kernel threads. Traditionally monolithic kernels are contained in a single binary image. As a result, the size of the kernel image grows as new hardware devices are supported or new features are added. Linux avoids this problem by providing loadable kernel modules. Linux kernel modules are somewhat similar to user-space shared libraries in that they are loaded at runtime. This allows the main kernel image to remain small as new features and device support is added. For example, many Linux distributions ship most hardware drivers compiled as modules so that only the required modules are loaded for each particular computer system. The kernel module facility can also be taken advantage of to build and deploy kernel components which do not have to be compiled with the rest of the kernel. Pnet is designed as a kernel module specifically to make it easier for users to deploy the IP per process model by avoiding re-compiling the entire kernel and rebooting to start the new kernel.

4.1.1 Implementation

There are two main tasks required of Pnet in order to implement the PAN IP per process model. On one side Pnet must interact with the kernel network subsystem to pass IP packets to and from the attached network. On the other side Pnet must provide an interface to applications using the IP per process model which enables these applications to send and receive complete IP packets.

Within the Linux kernel all network interfaces are represented by a common abstraction, the *net_device* structure. This large structure contains many fields including device identification, network packet queues and function pointers for each operation a network device may wish to support. Device driver authors simply implement these functions and provide

the kernel with a *net_device* instance with the appropriate function pointers set. This allows the network subsystem within the kernel to be completely device independent. Pnet takes advantage of the device independent nature of the kernel by encapsulating the networking portion of its functionality within a *net_device* abstraction. As far as the kernel is concerned Pnet is another network device attached to the system just as a second or third Ethernet interface would be. The use of the *net_device* abstraction allows all of the standard networking tools to be used without modification. For example, configuring the IP address and netmask assigned to Pnet can be accomplished using the traditional Unix *ifconfig* program. It is also possible to attach a network sniffer such as tcpdump or Wireshark to the Pnet interface to monitor all application packets because this functionality is common to all devices using the *net_device* abstraction. The most important advantage of implementing Pnet using the *net_device* abstraction is that the kernel is able to route packets to this interface without modification to the kernel's routing infrastructure. Other advantages of this design include the ability to utilize Linux's powerful firewall and network quality of service features. Possible uses for these features are discussed in later sections.

While the *net_device* abstraction is a convenient way to implement the network facing side of Pnet it is not useful for the application interface. Most non-network devices on a Unix based operating system are characterized as one of two types, character devices or block devices. Block devices typically include storage media such as hard disks and flash memory. These devices operate on multibyte units of data called blocks. Since the size of network packets and their arrival time is unpredictable the block device abstraction is not useful to Pnet. Character devices on the other hand operate on a character by character basis. Keyboards, serial ports and normal files use a character based interface. The basic requirement for the application interface to Pnet is the ability to transmit whole IP packets to the *net_device* portion of Pnet. Pnet implements this interface through the use of a character device and shared memory.

Character devices function much like a regular file or socket and many of the same operations apply. When the Pnet kernel module is loaded it creates a character device located at */dev/pnet*. The application making use of the IP per process model accesses Pnet by opening */dev/pnet* with the open function call just as it would any other file. Once opened,

ioctl operation	Description
PNET_IOMTUGET	Used by the application to get the MTU size.
PNET_IOBUFSIZEGET	Used by the application to get the size of each packet buffer. This must be greater than or equal to the size of the MTU.
PNET_IOBUFNUMGET	Used by the application to get the number of packet buffers.
PNET_IOIPSET	Used by the application to set the IP address this application instance is using.

Table 4.1: Pnet ioctl Commands

```
struct pnet_msg {
        uint16_t type;
        uint16_t num;
        uint16_t len;
        uint16_t pad;
};
```

Figure 4.2: struct pnet_msg

the application is able to read and write to the returned file descriptor with the normal read and write system calls. As with many other character devices the Pnet character device supports several special *ioctl* operations. The *ioctl* system call is a mechanism for allowing communication with the character device which does not rely on the primary read/write communication channel. For example serial ports use the ioctl system call to set the parity and data speed. The Pnet character device supports the four *ioctl* operations described in table 4.1.

The *ioctl* system call is used by the application process to get and set configuration parameters but it is not used for the packet data transfer. Instead an application receives and transmits network packets by simply reading or writing a small data structure to the Pnet character device. The definition of this structure in C can be found in figure 4.2.

The *pnet_msg* structure is small and simple. It consists of four sixteen bit fields for a total of sixty-four bits or eight bytes. Obviously the *pnet_msg* structure is not large enough to transmit an entire network packet. The reason for this is that the *pnet_msg* structure is used for communicating information between the application and Pnet not the actual packet transfer. The meaning of each field of the *pnet_msg* structure can be found in table 4.2.

The final portion of Pnet's interface to the application is the packet buffers. The packet buffers used by Pnet are implemented using a memory region which is shared between the

Field	Description
type	The type of this message. See table 4.3.
num	The buffer number this message refers to.
len	The length of the data in the buffer this message refers to. Not always required.
pad	Unused. Fills structure to 64 bits.

Table 4.2: struct pnet_msg fields

Type	Description
PNET_MSG_DATA_BUF	This message type indicates that the associated buffer contains a network packet.
PNET_MSG_FREE_BUF	This message type tells Pnet that the associated buffer is not being used by the application.
PNET_MSG_NEED_FREE_BUF	This message type is used by the application to inform Pnet that it requires buffers to transmit packets.

Table 4.3: Possible Values for the Type Field of pnet_msg

kernel and the application process. Shared memory allows both the kernel and the application to create a packet in place and pass a reference to that packet rather than copying the entire packet to a new buffer for each send and receive. This has performance advantages because data copying is a very large portion of the overhead found in most network protocol implementations [12]. The shared memory region is divided into packet buffers which are each 2048 bytes long. This size was chosen for two reasons. Firstly 2048 bytes is large enough to contain a full Ethernet frame which has a maximum size of 1518 bytes. The second reason is that there are constraints on how memory is allocated within the kernel. Allocating a memory region which is an even multiple of the hardware page size simplifies the internal design of Pnet. The target architecture for Pnet is i686 which uses a 4096 byte page size. This makes 2048 bytes a good choice for the buffer size because two buffers can fit evenly into a single memory page. As a result of this choice the number of packet buffers is two times the number of pages allocated for the shared memory region (see figure 4.3). The packet buffers are numbered sequentially from zero. It is this buffer number that allows both the application and Pnet to refer to the same packet data. Since network packets can be any size from a few tens of bytes to the full MTU the *len* field of the *pnet_msg* structure tells the receiving context how big the actual packet is. Figure 4.4 gives a pictorial

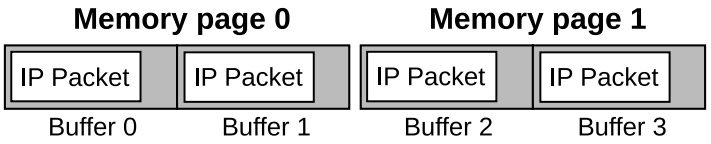

Figure 4.3: Pnet Buffers and Memory Pages

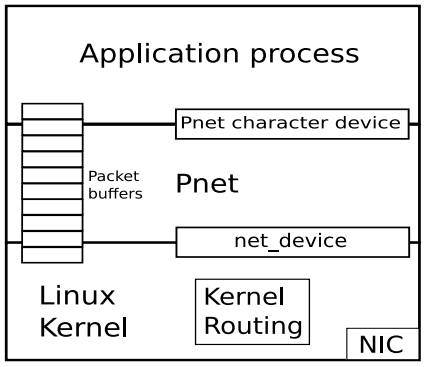

Figure 4.4: Pnet Overview

representation of the Pnet's architecture.

4.1.2 Interactions

With an understanding of the data structures involved it is now possible to walk through the use of Pnet from loading the kernel module through to communication between Pnet and the application process.

The Pnet kernel module takes two parameters at module load time. These two parameters are the number of packet buffers and the MTU size for the interface. Internally Pnet takes the passed MTU size and finds the next biggest power of two and uses that as the buffer size. Using a buffer size which is a power of two ensures that an even number of buffers fit into a single memory page. This buffer size is then multiplied by the number of buffers to get the size of the shared memory region for each process. The process of allocating the shared memory region occurs when the application opens the Pnet device. Each application shares a unique shared memory region with Pnet. Once loaded the Pnet network interface needs to be configured with an IP subnet just like a new Ethernet interface requires. This can be accomplished with any of the normal network configuration tools.

The first step for an application wishing to use Pnet is to open the */dev/pnet* character device. The file descriptor returned after a successful call to open is the only information required to communicate with Pnet. Once the character device has been opened the application asks Pnet for the current buffer configuration through the use of the *ioctl* operations described earlier. The configuration data read includes the MTU of the Pnet network interface, the number of buffers and the size of each buffer. With this information in hand the process is able to allocate the shared memory region. This is accomplished by using the *mmap* system call on the Pnet character device. Finally the prototype implementation of Pnet requires that the application inform Pnet which IP address it wishes to use. This is also accomplished with an *ioctl* system call.

Once the configuration parameters have been determined and the shared memory allocated, Pnet and the application are almost ready to transfer data packets. A key aspect of the Pnet design which has not been discussed up to this point is buffer ownership. Shared memory is a very convenient way to transfer large amounts of data between contexts because it removes the need to copy the data. However, using shared memory in this way requires communication between the sending and receiving contexts to indicate which one currently has permission to write to a particular portion of the shared memory. In the case of Pnet the portions of interest are the individual packet buffers. In order to arbitrate access to the buffers Pnet employs the concept of buffer ownership. Only one of either Pnet or the application process can own a given packet buffer at any time. The owner of a buffer has permission to write to the that buffer. For example, when Pnet receives a packet from the network it places that packet into a packet buffer it owns and then informs the application that a particular packet buffer contains packet data. It does this by sending a *pnet_msg* with the type field set to PNET_MSG_DATA_BUF. In so doing Pnet also transfers the ownership of the packet buffer to the application. The process works identically when the application wishes to transmit a packet. The prototype implementation of Pnet assumes that the application process owns all packet buffers when the shared memory region is allocated. Thus one of the first operations taken by the application is to pass ownership of some of the buffers to Pnet. Passing packet buffer ownership without indicating that the buffer contains a packet is accomplished by setting the type field of *pnet_msg* to PNET_MSG_FREE_BUF.

The current design of Pnet relies on the application process to ensure that Pnet has free buffers available. This is accomplished by passing free buffers to Pnet when the number of buffers owned by the application is over a threshold and by requested free buffers from Pnet when the number of buffers owned by the application drops below a threshold.

Packet transfer between Pnet and the application consists of reading and writing *pnet_msg* structures across the Pnet character device. Both sides of the communication receive packets to process via messages which have the type field set to PNET_MSG_DATA_BUF. In order to improve performance the Pnet character device reduces the number of required system calls by allowing the batching of *pnet_msg* structures when both reading and writing. Each time a system call is executed a context switch between application context and kernel context occurs. Minimizing context switches is important to achieving good performance. The application wishing to send more than one packet buffer to Pnet simply writes several *pnet_msg* structures in a single write call. Similarly, reading from the Pnet character device may return several *pnet_msg* structures to the application process. When this happens the amount of data read by the application process will be a multiple of the size of *pnet_msg*. The application then loops over each message and processes the associated packets while avoiding system call overhead. One difficulty that a character device interface introduces to application developers is that the read system call is usually a blocking operation. This poses a problem for applications using Pnet because there may be other work to be done while waiting for new packets to arrive. One solution to this problem is for the application to use threads but this is not strictly necessary because the Pnet character device also supports the *poll* and *select* system calls. These system calls, which are very commonly used in network applications, allow an application to be notified when a file descriptor becomes readable. The Pnet the character device becomes readable when a new packet arrives.

The overwhelming advantage of the Pnet design presented in this chapter is that the entire PAN is contained within a single kernel module. This allows Pnet to be built and distributed separately from the rest of the kernel. If Pnet required modification of the kernel's networking features or even if it could not be built as a module any user wishing to use the IP per process model would be forced to rebuild the entire kernel. Although not exceptionally difficult, this is a task beyond the skill of most users and many system administrators.

Another advantage of using the existing kernel networking infrastructure is the ability to make use of the Linux kernel's advanced firewall and network quality of service functionality. As discussed in section 3.1.4 it is important to be able to place some restrictions on applications using the IP per process model. Since Pnet is just another network interface from the perspective of the kernel it is trivial to add firewall rules which restrict applications using the IP per process model to certain transport protocols or to communicating with only a predefined set of IP addresses. This implementation of the IP process model also makes it very easy to apply bandwidth limits to each application; this is a task that is very difficult using the in-kernel network stack. The Linux kernel supports a large number of network packet queuing disciplines which can be used to prioritize packets or apply rate limitations. These queuing disciplines can be attached to any network interface making them also applicable to Pnet interfaces.

4.2 User-space Networking Library (UNL)

Once Pnet is in place to provide applications access to the PAN an user-level network stack is required. In this prototype the protocol implementations are collectively referred to as the user-space networking library (UNL) [47]. UNL provides an implementation of the IP, UDP, ICMP and TCP protocols for use with Pnet as well as an abstraction of the low level Pnet interfaces.

UNL has been implemented in C and makes extensive use of the GLib library [17]. GLib provides C programmers with many tools to speed development including data structures such as lists and queues, performance conscious memory allocators, debugging utilities as well as IO and main loop abstractions. Of particular interest is the main loop support. The GMainLoop is a high level wrapper on top of the select and poll system calls. This wrapper is used to create event based programs. As such, it is used extensively within Gtk+ based GUI programs. Using GMainLoop consists of adding event sources such as the Pnet character device and specifying callback functions to be executed when certain conditions occur. For example callback functions can be registered to be executed when the

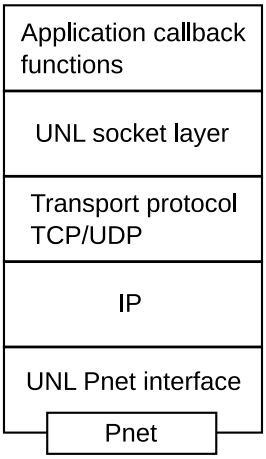

Figure 4.5: UNL Stack

Pnet character device becomes readable or writable. GMainLoop also allows for callback functions to be called at set intervals. By using GMainLoop UNL is able to achieve non-blocking behavior without requiring the complexity of threads. However, there is no reason that UNL's protocol processing could not occur in a separate thread if that is required by the application developer. Indeed, there may be performance benefits to performing protocol processing in a thread separate from the rest of the application.

As can been seen in figure 4.5 UNL is conceptually designed in layers like most other protocol implementations. In the case of UNL the lowest layer in the stack is specific to Pnet. This layer provides the higher layers with a more convenient interface to Pnet than reading and writing *pnet_msg* structures and it also registers the Pnet character device with the main loop and specifies which functions to call when new packets arrive. The input function at this layer has one job which is to determine what network layer protocol the received packet is carrying and then to call that protocol's input function. The prototype implementation of UNL only supports the IP protocol at present.

The next stage in processing an incoming packet is the IP protocol implementation's input function. This function's job is to verify the checksum field of the IP header is valid and to determine the next step in protocol processing. If the packet contains ICMP data the IP protocol implementation will process that data and respond accordingly. At present only a limited portion of the ICMP protocol is supported. Supported features include ICMP echo requests and echo replies (pings). If the packet is carrying UDP or TCP data then the

appropriate input function for these protocols is called.

Each transport layer input function performs the necessary protocol processing for its protocol. For UDP this processing is very simple. In contrast TCP has very complex processing to perform when a packet is received. At approximately 2500 lines of C UNL's implementation of TCP forms a significant portion of the entire code base and is definitely the most complicated part of UNL. The details of this implementation while complex and interesting, are not directly related to the IP per process model so they are not discussed here.

The final stage of protocol processing is passing the data to the application. It is important to note that the entirety of UNL's processing is occurring within the application. In this context passing data to the application refers to passing data from UNL to the application making use of the IP per process model. In both UNL's TCP and UDP implementations passing data to the application occurs through an interface which serves a function similar to the sockets API found in traditional network programming. After creating an instance of a transport protocol such as TCP or UDP the application creates a socket instance which represents a particular connection or packet flow. For each socket the application registers a callback function to be called when data is ready to be read from that particular socket. Parameters passed to the callback function allow the application to determine exactly which connection the data belongs to in situations when the developer uses a single callback function for many sockets. It is in this final callback function that the application performs its own work.

Transmitting data from the application through UNL and to the network functions much the same as data reception except that the function calls flow down the protocol stack instead of up. When an application writes to a socket the underlying transport protocol's transmit function is called. For UDP this will result in an immediate call to the IP protocol's transmit function after the UDP packet has been created. In the case of TCP the situation is much more complex because each data write to a TCP socket does not necessarily result in a network packet. When the TCP transmit function is called the TCP implementation queues the data and if all of the necessary conditions are met it generates a TCP segment and calls the IP transmit function. The IP layer's transmit function adds the IP header to the data it has received and then calls the UNL function that abstracts writing to the Pnet character

```
typedef struct {
        uint16_t buf_num; // Should never change.
        int16_t ref_cnt;

        void *head; // Start of buffer. Should never change.
        void *data; // Start of data.
        void *tail; // End of data.
        void *end; // End of buffer.  Should never change.

        void *h1;
        void *h2;
} UnlBufDesc;
```

Figure 4.6: UnlBufDesc Structure Definition

device. Once passed to Pnet the packet is out of the application's and hence UNL's control.

Since TCP is required to generate acknowledgment packets and perform other work at set intervals the TCP implementation also registers a timeout callback with the main loop instance. When this callback is executed TCP iterates over all TCP connections to see if any maintenance work is necessary or if any new segments must be sent.

An important task assigned to UNL beyond the protocol processing is the management and manipulation of the packet buffers shared between Pnet and the application. In order to manage packet buffers UNL represents each packet buffer with an *UnlBufDesc* instance. All of the required *UnlBufDesc* instances are created at start-up after UNL has determined the total number of packet buffers by querying Pnet. The definition of the *UnlBufDesc* structure can be found in figure 4.6 and the meaning of each field can be found in table 4.4.

As discussed earlier, an application using Pnet is responsible for ensuring that Pnet always has free packet buffers available to receive packets. The *UnlBufDesc* structure aids in this task by maintaining a reference count for each packet buffer. When a packet is received from Pnet, UNL sets the reference count in the associated *UnlBufDesc* structure to one. This indicates that UNL is holding a reference to this packet buffer so it cannot be given to Pnet as a free buffer. After a protocol layer has completed processing the packet there are two possibilities. The first is that the packet must be passed to a higher layer protocol for further processing. This is accomplished by passing a pointer to the *UnlBufDesc* structure to the higher layer protocol's input function. Since this all happens within UNL no reference count manipulation happens in this case. The second possibility is that the current protocol

Field	Description
buf_num	The number of the underlying packet buffer this UnlBufDesc represents. Used for communication with Pnet.
ref_cnt	The current reference count for this packet buffer. This value is greater than one if some part of the UNL or the application is using the buffer. Once the value reaches zero the buffer is placed on the free buffer list.
head	A pointer to the start of the packet buffer in the shared memory region.
data	A pointer to the beginning of the data to be processed in the packet buffer. This pointer changes as the packet moves through the protocol stack.
tail	A pointer to the end of the data to be processed in the packet buffer. Often used to determine the length of the packet.
end	A pointer to the end of the packet buffer in the shared memory region.
h1	A pointer to the link layer header in the packet buffer.
h2	A pointer to the network layer header in the packet buffer.

Table 4.4: Fields of struct UnlBufDesc

layer is the final destination of the packet and once processing is complete, the packet buffer should be marked as free. UNL accomplishes this by decrementing the reference count on the *UnlBufDesc* structure by one using the `unl_bufdesc_unref` function. Not only does this function decrement the reference count it also returns the packet buffer to the list of free buffers if the reference count reaches zero. By making use of reference counts it is possible for a part of UNL or even the application itself to continue to use the packet buffer after protocol processing is complete. A possible use of this technique is for the application to maintain a reference to the packet in order to later transmit the exact same data without having to inspect or copy that data.

As can be seen from the definition of the *UnlBufDesc* structure presented figure 4.6 and table 4.4 this structure is used for more than just maintaining a reference count on the underlying packet buffer. In fact, it is the pointer fields within the *UnlBufDesc* structure which each layer of the protocol processing uses to indicate where its data begins. For example, after a received packet has been processed by the Pnet specific portion of UNL the data field in the associated *UnlBufDesc* structure is set to point to the beginning of the IP header within the packet buffer. This tells the IP implementation where to begin processing. Once the IP layer has processed the IP header it moves the data pointer to point to the beginning of the IP payload and then calls the appropriate transport layer input

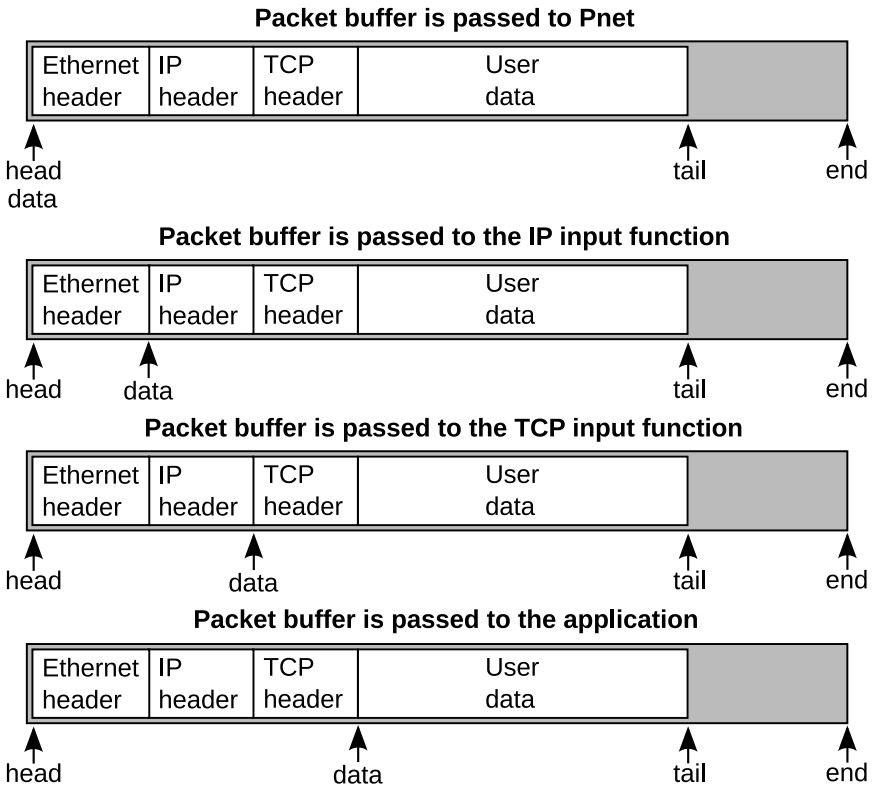

Figure 4.7: Processing a Packet with UnlBufDesc

function. This process continues until the packet eventually reaches the application's read callback with the data pointer pointing to the data payload (see figure 4.7). The *UnlBufDesc* structure is also used during packet construction. When TCP for example wishes to send a packet, it requests a free buffer which has the data pointer set such that there is room at the beginning of the buffer to prepend all of the necessary headers. The TCP implementation then creates the TCP header starting at the data pointer and copies the data to be sent into the packet buffer. Creating space for the IP header simply involves decrementing the size of the IP header from the data pointer and constructing the header at that point. The chief advantage of manipulating pointers to the packet buffer in this way is performance. This mechanism allows for the data to move through the stack without requiring data copying at each stage. The *UnlBufDesc* design is similar, but simpler, than the Skb and mbuf packet structures used by the Linux and BSD kernel protocol implementations respectively.

4.3 Limitations of the Prototype Implementation

A real world deployment of the IP per process model would require a method to assign individual applications an IP address and maintain a list of free and available address on the PAN. The prototype implementation presented in this thesis relies on the application to tell Pnet which IP address it wishes to use. There is, at present, nothing to stop two applications from specifying the same IP address. Adding the necessary duplicate address and subnet validity checks to Pnet should not be difficult. Address allocation and maintenance could be accomplished entirely within Pnet or through an user-level policy daemon which functions much like a DHCP server on a LAN subnet. This policy daemon could also be responsible for inserting the necessary firewall rules to ensure application and network protection as well as configuring any required QoS rules.

Although the protocol implementations in UNL serve as a good prototype of the IP per process model they are by no means complete or ready for production deployment. The IP implementation supports the features necessary to demultiplex packets to a higher layer protocol such as TCP or UDP but does not implement more advanced features such as IP defragmentation. There are no known reasons why IP fragmentation or any other part of the IP protocol could not be implemented in UNL. The UNL implementation of UDP is basically complete but since UDP is such a trivial protocol this is to be expected. The most significant amount of development time has been dedicated to UNL's implementation of TCP. The TCP protocol implementation supports most of the features found commodity operating system network stacks including: congestion control including slow start and congestion avoidance, MSS option, timestamp option, window scale option, triple duplicate acknowledgment detection and fast retransmission, protection against wrapped sequence numbers (PAWS) and RTT calculation. The only major TCP feature missing from UNL's TCP implementation is selective acknowledgment support (SACK).

4.4 High level Language Protocol Implementation

For the purposes of the testing and evaluation in the next chapter Pnet and UNL can be considered two parts of the same prototype. In reality Pnet and UNL are two very separate projects. In fact any application that can read from a file descriptor can make use of Pnet to send and receive whole IP packets. In order to demonstrate this flexibility a very small IP protocol implementation has been constructed using the Python programming language. Python is a very popular and powerful high level programming language. This small IP implementation contains only the features necessary to respond to ICMP echo requests. What makes this particular implementation interesting is that the in-kernel network stack model makes it impossible to develop network or transport layer protocols in a language such as Python. The combination of Pnet and a high level language has the potential to greatly reduce the work necessary to prototype a new transport layer protocol. The Python IP/ICMP echo implementation is only about two hundred lines long.

4.5 Summary

In order to demonstrate the IP per process model a prototype system has been created. This system is named Pnet/UNL in reference to its two components: Pnet and UNL. Pnet is a Linux kernel module which gives applications the ability to send and receive complete IP packets thereby bringing end-to-end connectivity to applications. The second component of the prototype is the Userspace Networking Library (UNL). UNL provides an implementation of the IP, ICMP, UDP and TCP protocols for use with Pnet. Both Pnet and UNL are written in C. The Pnet Linux kernel module consists of approximately 1400 lines of C and UNL is approximately 8400 lines.

Communication between Pnet and the application process occurs through a combination of a character device which is created when the Pnet kernel module is loaded and shared memory. Network packets are not written to or read from the character device. Instead packets are transferred via a shared memory region allocated when the application opens the Pnet character device. Both Pnet and the application create packets in this shared memory

region and signal when a packet is ready to be processed by writing to the character device. This design avoids copying the packet each time it crosses between Pnet and the application. Built on top of the low level communication mechanisms provided by Pnet, UNL provides applications with network protocol implementations and an interface similar to the standard sockets API.

Chapter 5

Prototype Evaluation

5.1 Goals

The primary goal of the IP per process model is to bring end-to-end connectivity to the ends of the network, the applications. The Pnet/UNL prototype of the PAN based IP per process model has been designed to achieve this goal. The experiments described in this chapter show the application process in direct communication with a remote network entity and the operating system kernel functioning as a simple IP router. This demonstrates the desired end-to-end connectivity. A secondary goal of the Pnet/UNL implementation is to achieve a level of performance in terms of latency and throughput which is respectable in comparison to the Linux network stack. Since the Linux network stack is generally accepted to be the fastest network stack available [22] this sets a very high and perhaps over-optimistic performance target given the lack of protocol development experience and in-depth performance optimization knowledge of the author.

5.2 Experimental Environment

All experiments presented in this chapter were carried out on two Intel based servers. The specifications for these servers can be found in table 5.1. The two servers were connected directly by a cross-over Ethernet cable thereby avoiding the use of an Ethernet switch. The choice to avoid an intermediate Ethernet switch was made because of the lack of admin-

CPU	Cache	Front side bus	RAM
Intel 945 Pentium D @ 3.4 GHz	2 MB per processor	800 MHz	2 GB

Table 5.1: Experimental System Specifications

istrative access to a high-quality Ethernet switch and to simplify the testing environment as much as possible. The operating system installed on both servers is Fedora-Core 6 running the 2.6.20-1.2952 Linux kernel. All experiments were performed with one of the two servers running the UNL network stack and the second server using the Linux network stack through simple socket API based programs. Using the existing Linux network stack on one end of all experiments ensures a level of correctness in the UNL IP, ICMP and TCP implementations.

The experiments presented in this chapter evaluate Pnet/UNL based on latency and TCP throughput. These results are compared against the Linux network stack operating under the same conditions. Evaluating the latency of the Pnet/UNL prototype is important because protocol processing in this design uses two separate contexts, kernel and user-space. In contrast, the in-kernel network stack design performs all processing within a single context. Measuring latency aims to quantify the effects of queuing and scheduling between the two contexts in the Pnet/UNL design. TCP throughput is evaluated to provide a real world benchmark and because throughput is the most common measure of networking performance.

Each of the latency and throughput experiments were performed at two levels. The first level has the servers otherwise idle and second level places the servers under load. System load was generated with the *openssl speed* command. This command runs a series of encryption related benchmarks. Two copies of this benchmark were executed simultaneously for the duration of the experiments to ensure that both CPUs were under load.

5.2.1 ICMP Ping Latency Experiment

UNL contains a very simple implementation of the ICMP protocol which supports ICMP echo requests and echo responses (pings). In order to measure the latency introduced by

Pnet and UNL this experiment sends five hundred ICMP echo requests to the destination server spaced .2 ms apart. The average round trip time (RTT) of the five hundred responses are presented in the following graphs. There is nothing special about the choice to send five hundred echo requests at .2 ms intervals other than it provides a relatively large number of samples spaced over a reasonable time frame of one hundred seconds. The ICMP ping latency experiments were carried out at 100 Mbps and 1 Gbps.

5.2.2 TCP Data Throughput Experiment

The TCP data throughput experiments were performed with both UNL sending data and UNL receiving data as well as at network data rates of 100 Mbps and 1 Gbps. These experiments were also repeated with the system under load. In all cases the UNL stack was listening for new connections from a client program which uses the standard Linux TCP stack through the sockets API. Throughput was calculated as the time required to transfer a 1.5 gigabyte file. The graphs presented here use the average taken from five replications of each experiment. The graphs for the experiments where the system is unloaded also present system utilization percentages as reported by the *vmstat* utility.

In order to eliminate the effects of hard disk speed from the results all throughput tests were performed with the sender reading from a RAM drive and the receiver writing to a RAM drive. This is especially important for the 1 Gbps throughput tests. The highest throughout rate observed during these tests was 110 megabyte/sec. A simple disk speed benchmark performed using the *hdparm* program shows that the disks used in the experimental servers are only capable of approximately 56 megabyte/sec of sustained throughput. Therefore, without using RAM drives the hard disks could potentially become the throughput bottleneck.

The throughput experiments compare Pnet/UNL against two configurations of the Linux network stack. These two configurations are with network interface (NI) offload enabled and with it disabled. Many modern Ethernet network interfaces include the ability to perform some portion of the network stack's work. The goal of these techniques is to save CPU resources. The Intel e1000 network interface used in the experimental servers supports

transmission and receive checksumming, scatter-gather and TCP segmentation offload. The results presented as “Linux (no offload)” have all of these features disabled.

5.3 Results

5.3.1 ICMP Ping Latency

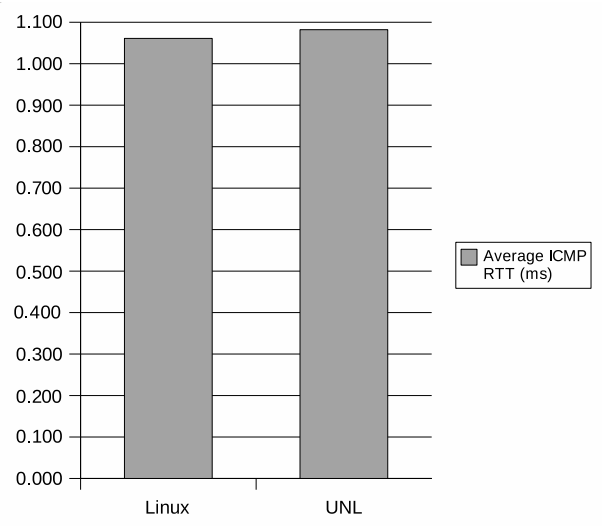

Figure 5.1: ICMP RTT at 100 Mbps

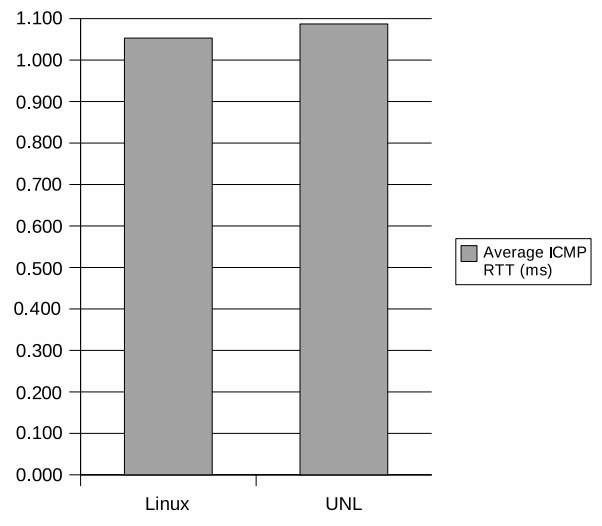

Figure 5.2: ICMP RTT at 100 Mbps Under Load

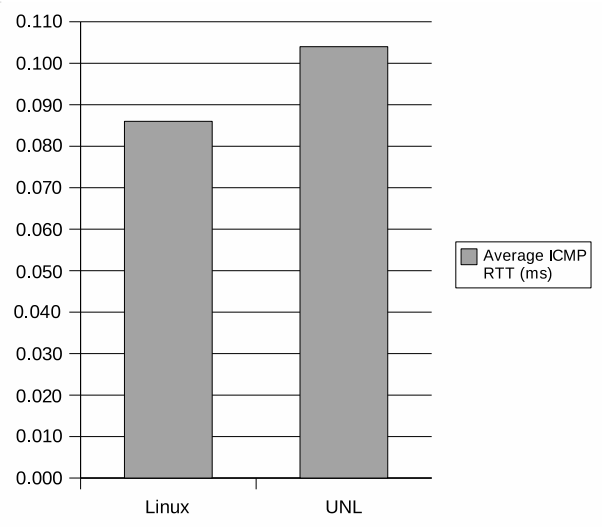

Figure 5.3: ICMP RTT at 1 Gbps

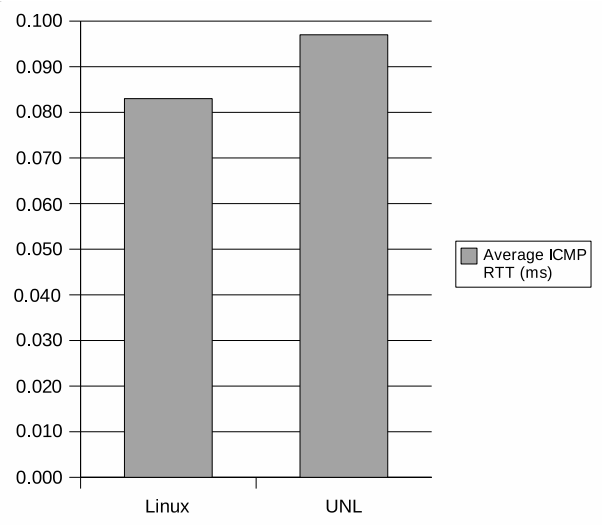

Figure 5.4: ICMP RTT at 1 Gbps Under Load

5.3.2 TCP Data Throughput

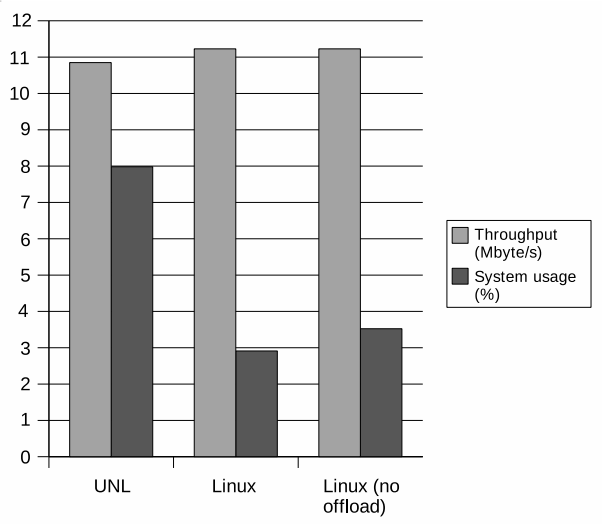

Figure 5.5: TCP Receive Throughput at 100 Mbit/sec

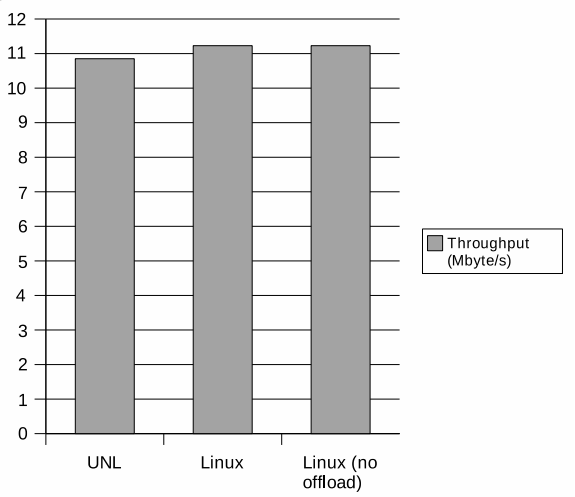

Figure 5.6: TCP Receive Throughput at 100 Mbit/sec Under Load

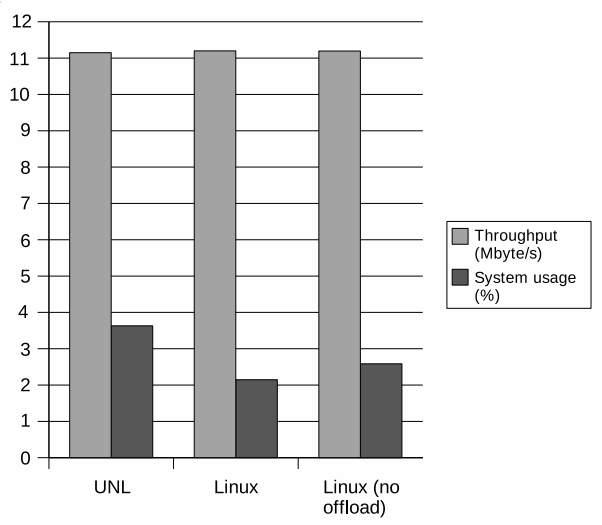

Figure 5.7: TCP Transmission Throughput at 100 Mbit/sec

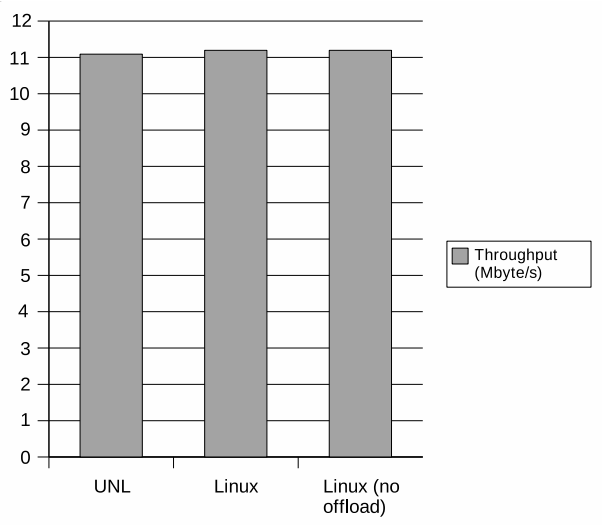

Figure 5.8: TCP Transmission Throughput at 100 Mbit/sec Under Load

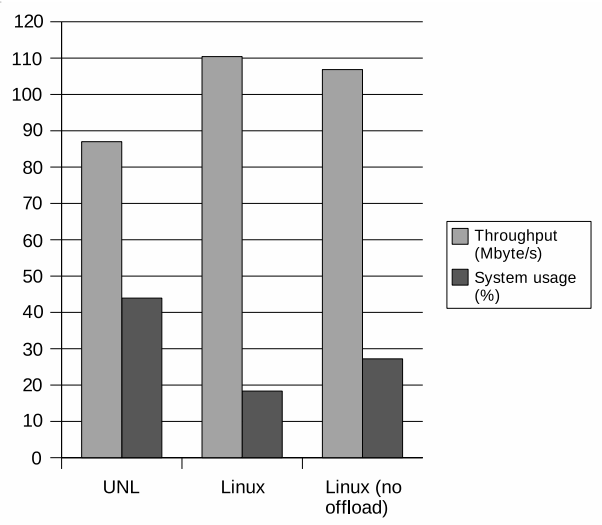

Figure 5.9: TCP Receive Throughput at 1 Gbps

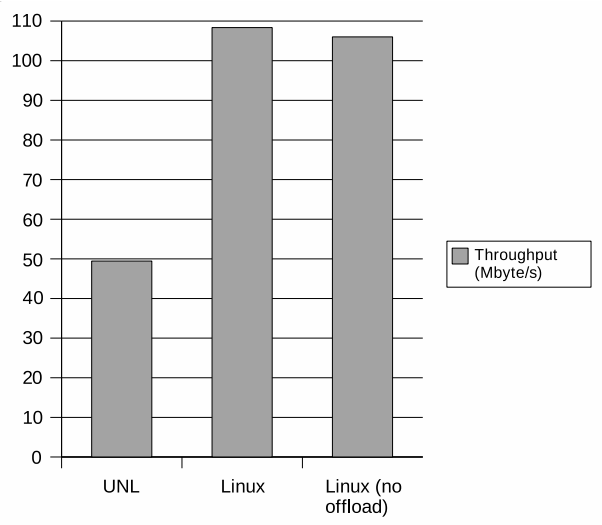

Figure 5.10: TCP Receive Throughput at 1 Gbps Under Load

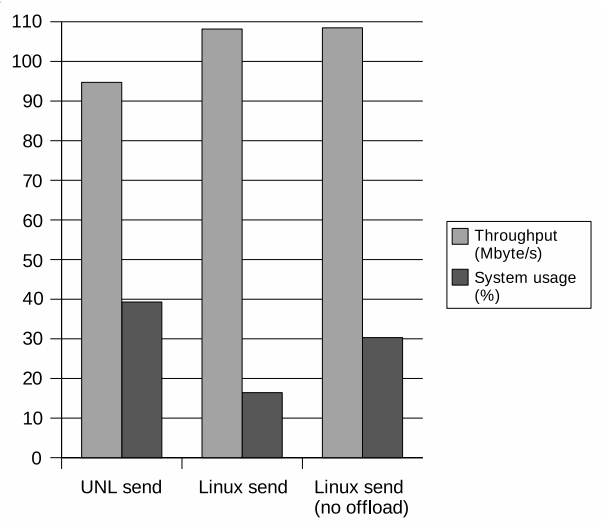

Figure 5.11: TCP Transmit Throughput at 1 Gbps

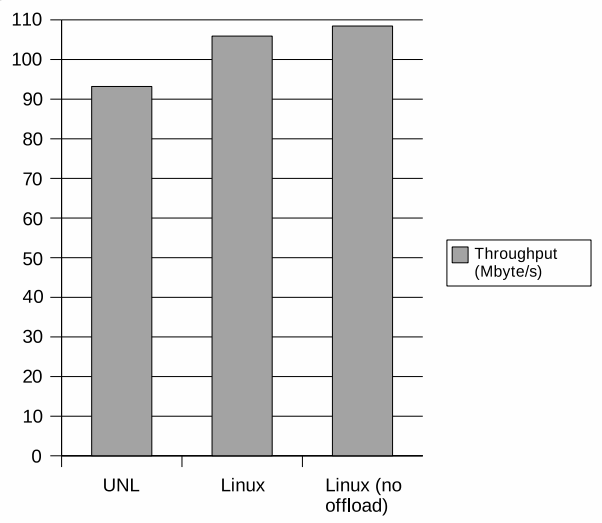

Figure 5.12: TCP Transmit Throughput at 1 Gbps Under Load

5.4 Discussion

The ICMP RTT experiments show that the Pnet/UNL prototype has slightly more latency than the Linux network stack. In the case of the 100 Mbit experiment the difference in latency is 0.021 ms when not under load and 0.034 ms when under load. The difference between the Linux network stack and UNL at 1 Gbit is even smaller. When not under load

the difference at 1 Gbit is 0.018 ms and under load the difference is 0.014 ms. The fact that the Pnet/UNL prototype adds a small amount of time to the ICMP RTT is not surprising. The Linux network stack implements ICMP within the kernel. This allows the kernel to respond to the ICMP echo request without any kernel to user-space context switches. In the case of Pnet/UNL the received packet must be queued by Pnet for reception by the UNL application. Only after the kernel wakes the UNL application will the ICMP packet be processed and the response packet sent. Transmission of the packet also necessitates another user-space to kernel space context switch. Given the extra context switches that are required in the Pnet/UNL prototype the observed RTT difference appears quite reasonable.

The TCP experiments measure the TCP throughput and system load of the Pnet/UNL prototype. In all of the experiments performed at 100 Mbps the throughput achieved by Pnet/UNL is slightly lower than that of the Linux network stack. However, the difference is very small. When receiving data the Pnet/UNL prototype was only 0.379 megabyte/sec slower than the Linux kernel and 0.380 megabyte/sec slower than the Linux kernel with NI offload features disabled. When the system was placed under load Pnet/UNL was 0.377 megabyte/sec slower than the Linux kernel stack with NI offload both enabled and disabled. The system load percentage when receiving data at 100 Mbps was 7.98% for Pnet/UNL, 2.91% for Linux and 3.52% for Linux with NI offload features disabled. When transmitting data at 100 Mbps Pnet/UNL is 0.050 megabyte/sec slower than Linux and 0.044 megabyte/sec slower than Linux with NI offload features disabled. System utilization values found in this experiment show Pnet/UNL to use 3.63% of system resources while the Linux stack uses 2.15% and Linux without offload features uses 2.58%. When placed under load Pnet/UNL is 0.104 megabyte/sec slower than Linux and 0.105 megabyte/sec slower than Linux with NI offload features disabled. Given the results of these experiments performed with a network data rate of 100 Mbps Pnet/UNL appears to compare very well against the Linux kernel both in terms of data throughput and system utilization.

The second set of throughput experiments were performed with a network data rate of 1 Gbps. In these experiments the performance gap between the Linux network stack and the Pnet/UNL prototype widens. When receiving data at 1 Gbps Pnet/UNL achieved a data rate of 87.006 megabyte/sec which is 79% of the 110.425 megabyte/sec achieved by the Linux

kernel and 81% of 106.847 megabyte/sec achieved by the Linux kernel with NI offload disabled. When placed under load the throughput of both configurations of the Linux kernel drops slightly but the throughput of the Pnet/UNL prototype drops to 49.451 megabyte/sec. This experiment shows the largest difference between Pnet/UNL and the Linux kernel of any experiment performed. When transmitting data at 1 Gbps Pnet/UNL achieves 94.715 megabyte/sec. The Linux network stack achieves 108.167 megabyte/sec and 108.478 with NI offload disabled. Transmitting under load does not suffer the same performance consequences that the receive experiments experienced. In this experiment Pnet/UNL achieves 93.194 megabyte/sec while the Linux kernel achieves 105.917 megabyte/sec and 108.455 megabyte/sec with NI offload disabled.

Unfortunately, the current Pnet/UNL implementation contains a race condition which manifests itself as a stalled TCP data transfer. This race condition only happens when receiving data under load at 100 Mbps or during the 1 Gbps receive test. Four stalled transfers were observed when collecting the five samples used for the 100 Gbps under load experiment and two stalled transfers were observed when collecting the five samples used for the 1 Gbps receive experiment. During testing it is not uncommon for this bug to not appear for many trial runs only to reappear during a number of tests at a later time. The effect of this race condition is that an acknowledgment segment is transmitted in which some of the TCP header fields contain values which should be seen in segments from the other end of the connection. This happens after tens or hundreds of thousands of packets have been sent. Often the TCP checksum verification detects these corrupted packets and drops them. However, when only the TCP timestamp option fields are wrong the checksum value computed is still valid but the reversed timestamp values stall the connection. The reason that the connection stalls and never recovers is that the remote TCP peer uses the timestamp values to detect old packets in the network. Once the remote TCP receives an erroneous timestamp it ignores all of the TCP segments received with the proper timestamp value because these segments appear to be very old. An extensive amount of time has been invested isolating the effects of this bug but its cause has yet to be found.

There are many possible reasons why the performance of Pnet/UNL is lower than that of the Linux kernel. Certainly two major reasons are the lack of performance tuning of

Pnet/UNL and the extensive development work that has gone into the Linux network stack by a large number of talented people over many years. An important goal for the design of the Pnet/UNL prototype was simplicity. In order to achieve this goal simple data structures which are easy to understand and debug were used throughout the implementation. For example the data structure used to implement UNL's TCP segment reassembly buffer is a simple linked list. Also, the current design of Pnet uses very fine grained locking over its data structures. In retrospect this fine grained locking may be a performance problem not a benefit. Another design issue that may be affecting performance is the lack of a fast path also known as header prediction in UNL's TCP implementation. Most TCP implementations avoid fully processing a received packet if it matches a few criteria which identify it as the next segment that is expected. This feature can significantly reduce TCP overhead [4].

It is also worth noting that the TCP throughput results may be affected more by the quality of the TCP protocol implementation than the overhead inherit in Pnet/UNL. Creating a TCP implementation which performs well is a very difficult proposition. For example, if a required acknowledgment packet is not transmitted or processed in a timely manor the data transmission can stall thereby reducing the throughput of the connection.

In all of the TCP throughput experiments the difference in system utilization values between the Linux stack with and without NI offload features enabled are quite apparent. NI offload features clearly have performance benefits. At present the design of Pnet/UNL does not take advantage of any NI offload features. It is not clear how some offload features such as scatter gather memory operations would be useful to Pnet/UNL however other features such as checksum computation could be beneficial. An obvious optimization of Pnet/UNL would be to avoid recomputing the checksum on received packets when the Linux kernel passes packets to Pnet that have already had their checksum verified.

5.5 Summary

The goal of the Pnet/UNL prototype is to provide an implementation of the IP per process model which brings end-to-end network connectivity to applications and does so with a reasonable level of performance. The Pnet/UNL prototype implements the IP, ICMP and TCP protocols completely within the application. The ability of these user-level protocol implementations to communicate successfully with a remote Linux network stack shows that this model does indeed achieve its primary goal of bringing end-to-end connectivity to applications. The performance testing presented in this chapter shows that the Pnet/UNL prototype is very competitive with the Linux network stack when used on a 100 Mbps network. However, when the same testing is performed on a 1 Gbit network the performance gap between Pnet/UNL and the Linux network stack becomes wider. It is clear that Pnet/UNL will require further performance tuning to be competitive at this high data rate.

Chapter 6

Conclusions and Future Work

Over the last decade and a half the Internet has shown itself to be a powerful platform for innovation at almost all levels of its design. This great flexibility comes in large part from the application of the end-to-end principle in the design of the Internet. The end-to-end principle aims to place as much intelligence and processing as possible at the ends of the network rather than in the core as is found in networks such as the PSTN. This design allows for greater innovation and flexibility because new protocols and services can be introduced into the network by any single endpoint without requiring that intermediate network nodes be modified.

The usual definition of an end of the Internet is a single IP enabled device and by extension the operating system executing on that device. The operating system is included in this definition because it is the operating system kernel which usually contains the network protocol implementations that allow applications to communicate over the network. However, the actual end of any network communication is not the operating system kernel. The operating system kernel exists to manage hardware so there is little reason for a remote process to communicate directly with the kernel. Rather, remote processes are actually in communication with processes executing on the local system. The kernel is simply acting as an intermediary between the actual ends of the communication; the two application processes.

The sockets API is the interface most often used between applications and the in-kernel network protocol implementations. This interface allows the application to pass data to the kernel to be packaged and transmitted as well as allowing the reception of data from the ker-

nel after network protocol processing has been completed. The problem with this interface is that the application is isolated from the network and limited to using only the network protocols and features supported by the operating system kernel. When the application is considered the end of the network this model breaks end-to-end connectivity.

6.1 Summary of Contributions

This thesis offers a network stack model which makes use of user-level networking to achieve end-to-end connectivity for applications and which also has benefits over other user-level networking designs. This is accomplished by assigning each application an IP address and reducing the operating system kernel's involvement with the network packet flow to that of an IP router. The IP per process model results in a flexible, easy to modify network stack which allows the application to customize the network protocol implementations in almost any way desirable. The advantages of the IP per process model over other user-level networking designs come from a simplicity of design achieved through the application of the end-to-end principle. This allows problems such as packet demultiplexing and shared layer four port space to be simplified or sidestepped entirely.

In order to demonstrate the IP per process model a prototype implementation has been created called Pnet/UNL. This prototype consists of two components: a Linux kernel module named Pnet and an user-level implementation of the IP, ICMP, UDP, and TCP protocols named the Userspace Networking Library (UNL). Evaluation of this prototype shows that it achieves its primary goal of bringing end-to-end connectivity to applications. Performance testing shows Pnet/UNL to be very competitive with the Linux network stack at 100 Mbps. However, the performance gap widens when testing is performed on a 1 Gbps network.

6.2 Future Work

The Pnet/UNL prototype of the IP per process model serves as a good initial demonstration of the model but it is missing some features necessary for serious use. One obvious miss-

ing feature is the ability for Pnet to enforce which IP address the application process uses. The present design allows the application using Pnet to choose any IP address it wishes. One possible way to solve this problem is to introduce a user-level policy daemon which Pnet could consult when applications request IP addresses. This could also be implemented within Pnet itself but more complicated policies such as static IP assignments make a separate user-level daemon a better choice. Another useful task that could be assigned to the policy daemon is managing firewall and QoS policies to achieve network and application protection from malicious applications using Pnet.

Another area of UNL that warrants future work is the TCP implementation. While the current implementation is sufficient as a prototype the addition of TCP selective acknowledgments and the introduction of a TCP fast path would be very beneficial to production use. Further verification that the congestion control behavior is correct and compatible with other TCP implementations is also desirable. The ability to make use of NI offload features, especially checksum computation, may also be of benefit to UNL.

Section 2.6 discussed the historical relationship between user-level networking and memory swapping. It was also noted that none of the user-level networking works found in the computing literature have studied the effects of memory swapping on user-level networking with modern computer hardware. This could prove to be a very interesting research project.

It would be very interesting to prototype an implementation of the IP per process model which modifies the operating system network stack so that each application's IP address can be assigned to a local network interface. As discussed in section 3.3 this design would be especially interesting in an IPv6 network where the size of the subnet assigned to each LAN makes the random generation of IP addresses for each application possible. Implementation of this model likely involves more extensive modification of the existing network stack than was required when implementing the PAN model used in Pnet/UNL prototype.

As a longer term project the Pnet/UNL prototype could serve as a base for a much larger project investigating alternate transport layer protocols and network stack designs. There are many interesting areas worth investigating. One such area is a reliable transport protocol designed for lossy environments such as wireless links. This is an area where the perfor-

mance of TCP could be improved upon. Another interesting project would be to develop a reliable transport protocol which is simpler than TCP. According to [22] the complexity of TCP is tripled simply to deal with complications associated with in-kernel networking. Pnet/UNL will also be useful as a tool for experimenting with alternate interfaces between the network protocol implementations and the application. A simple example of an alternative interface may include making use of share semantics instead of the copy semantics found in the sockets API. More drastic changes could include a records based or virtual connection API. Transport layer protocols and the interface between the application and the network protocol implementation have remained relatively static for a long time. Pnet/UNL and the IP per process model in general make experimentation in these areas much easier.

Appendix A

Results Data

A.1 Receive at 100 Mbps

Real (sec)	Idle (%)	User (%)	System (%)	Throughput (megabyte/sec)
134.626	91.963	2.889	5.352	10.849
134.674	91.970	2.897	5.404	10.845
134.706	92.116	2.868	5.335	10.843
134.593	92.014	2.933	5.389	10.852
134.547	92.022	2.911	5.397	10.855
134.629	**92.017**	**2.900**	**5.375**	**10.849**

Table A.1: Receive at 100 Mbps UNL

Real (sec)	**Idle (%)**	**User (%)**	**System (%)**	**Throughput (megabyte/sec)**
130.124	97.091	0.221	2.893	11.224
130.074	97.068	0.137	2.816	11.229
130.073	97.091	0.221	2.839	11.229
130.075	97.076	0.175	2.824	11.229
130.073	97.114	0.244	2.816	11.229
130.084	**97.088**	**0.200**	**2.838**	**11.228**

Table A.2: Receive at 100 Mbps Linux

Real (sec)	**Idle (%)**	**User (%)**	**System (%)**	**Throughput (megabyte/sec)**
130.080	96.530	0.204	3.477	11.228
180.076	96.518	0.150	3.383	11.229
130.072	96.442	0.137	3.511	11.229
130.074	96.435	0.198	3.503	11.229
130.075	96.462	0.166	3.477	11.229
130.075	**96.477**	**0.171**	**3.470**	**11.229**

Table A.3: Receive at 100 Mbps Linux (no offload)

A.2 Receive at 100 Mbps Under Load

Real (sec)	**Throughput (megabyte/sec)**
134.258	10.879
134.721	10.841
134.879	10.829
134.560	10.854
134.548	10.855
134.548	**10.852**

Table A.4: Receive at 100 Mbps Under Load UNL

Real (sec)	**Throughput (megabyte/sec)**
130.078	11.228
130.074	11.229
130.077	11.229
130.073	11.229
130.073	11.229
130.075	**11.229**

Table A.5: Receive at 100 Mbps Under Load Linux

Real (sec)	Throughput (megabyte/sec)
130.075	11.229
130.074	11.229
130.075	11.229
130.074	11.229
130.076	11.229
130.075	**11.229**

Table A.6: Receive at 100 Mbps Under Load Linux (no offload)

A.3 Transmit at 100 Mbps

Real (sec)	Idle (%)	User (%)	System (%)	Throughput (megabyte/sec)
130.997	96.323	0.992	2.789	11.150
130.985	96.436	0.947	2.812	11.151
130.978	96.424	0.962	2.795	11.151
130.980	96.325	1.000	2.795	11.151
131.002	96.333	0.954	2.810	11.149
130.988	**96.368**	**0.971**	**2.800**	**11.150**

Table A.7: Transmit at 100 Mbps UNL

Real (sec)	**Idle (%)**	**User (%)**	**System (%)**	**Throughput (megabyte/sec)**
130.466	97.902	0.015	2.330	11.195
130.472	97.856	0.015	2.393	11.195
130.152	97.894	0.015	2.383	11.222
130.474	97.716	0.014	2.380	11.194
130.478	97.893	0.015	2.378	11.194
130.408	**97.852**	**0.015**	**2.373**	**11.200**

Table A.8: Transmit at 100 Mbps Linux

Real (sec)	**Idle (%)**	**User (%)**	**System (%)**	**Throughput (megabyte/sec)**
130.476	97.424	0.015	2.810	11.194
130.475	97.393	0.015	2.787	11.194
130.479	97.393	0.015	2.787	11.194
130.475	97.439	0.022	2.833	11.194
130.476	97.431	0.015	2.750	11.194
130.476	**97.416**	**0.016**	**2.793**	**11.194**

Table A.9: Transmit at 100 Mbps Linux (no offload)

A.4 Transmit at 100 Mbps Under Load

Real (sec)	**Throughput (megabyte/sec)**
130.895	11.158
135.113	10.810
130.868	11.161
130.891	11.159
130.862	11.161
131.726	**11.090**

Table A.10: Transmit at 100 Mbps Under Load UNL

Real (sec)	**Throughput (megabyte/sec)**
130.477	11.194
130.477	11.194
130.476	11.194
130.477	11.194
130.479	11.194
130.477	**11.194**

Table A.11: Transmit at 100 Mbps Under Load Linux

Real (sec)	Throughput (megabyte/sec)
130.471	11.195
130.470	11.195
130.474	11.194
130.469	11.195
130.468	11.195
130.470	**11.195**

Table A.12: Transmit at 100 Mbps Under Load Linux (no offload)

A.5 Receive at 1 Gbps

Real (sec)	Idle (%)	User (%)	System (%)	Throughput (megabyte/sec)
16.659	56.842	14.263	29.105	87.675
17.016	55.500	14.500	30.111	85.835
16.767	54.500	14.333	31.166	87.110
16.741	56.736	13.736	29.684	87.245
16.756	56.736	13.894	29.368	87.167
16.788	**56.063**	**14.145**	**29.887**	**87.006**

Table A.13: Receive at 1 Gbps UNL

Real (sec)	**Idle (%)**	**User (%)**	**System (%)**	**Throughput (megabyte/sec)**
13.221	81.666	0.866	17.666	110.474
13.253	82.466	0.600	17.133	110.207
13.227	81.400	1.200	17.800	110.423
13.206	81.600	0.933	18.000	110.599
13.227	81.133	1.000	18.000	110.423
13.227	**81.653**	**0.920**	**17.720**	**110.425**

Table A.14: Receive at 1 Gbps Linux

Real (sec)	**Idle (%)**	**User (%)**	**System (%)**	**Throughput (megabyte/sec)**
13.847	72.266	0.933	27.333	105.479
13.462	73.400	1.000	25.933	108.496
13.851	73.312	0.875	26.250	105.449
13.336	73.333	0.933	25.933	109.521
13.872	71.600	1.000	27.867	105.289
13.674	**72.782**	**0.948**	**26.663**	**106.847**

Table A.15: Receive at 1 Gbps Linux (no offload)

A.6 Receive at 1 Gbps Under Load

Real (sec)	Throughput (megabyte/sec)
30.573	47.773
27.675	52.776
30.321	48.170
30.324	48.165
28.996	50.371
29.578	**49.451**

Table A.16: Receive at 1 Gbps Under Load UNL

Real (sec)	Throughput (megabyte/sec)
13.305	109.776
13.810	105.762
13.215	110.524
13.883	105.206
13.210	110.566
13.485	**108.367**

Table A.17: Receive at 1 Gbps Under Load Linux

Real (sec)	Throughput (megabyte/sec)
13.733	106.355
13.913	104.979
13.850	105.456
13.505	108.150
13.897	105.100
13.780	**105.100**

Table A.18: Receive at 1 Gbps Under Load Linux (no offload)

A.7 Transmit at 1 Gbps

Real (sec)	Idle (%)	User (%)	System (%)	Throughput (megabyte/sec)
15.167	60.294	14.352	25.352	96.299
15.431	61.777	13.888	24.333	94.652
15.544	60.411	14.411	25.176	93.694
15.418	61.166	14.333	24.611	94.732
15.550	59.941	14.764	25.117	93.927
15.422	**60.718**	**14.350**	**24.918**	**94.715**

Table A.19: Transmit at 1 Gbps UNL

Real (sec)	**Idle (%)**	**User (%)**	**System (%)**	**Throughput (megabyte/sec)**
13.312	83.633	1.133	15.200	109.718
13.558	83.266	1.133	15.733	107.728
13.543	83.800	1.133	15.066	107.847
13.555	83.533	1.133	15.266	107.751
13.550	83.666	1.066	15.266	107.791
13.504	**83.580**	**1.120**	**15.306**	**108.167**

Table A.20: Transmit at 1 Gbps Linux

Real (sec)	**Idle (%)**	**User (%)**	**System (%)**	**Throughput (megabyte/sec)**
13.474	69.666	1.333	28.933	108.399
13.477	69.733	1.333	28.633	108.375
13.474	69.866	1.333	28.733	108.399
13.428	69.312	1.250	29.437	108.771
13.468	69.866	1.333	28.800	108.447
13.464	**69.689**	**1.316**	**28.907**	**108.478**

Table A.21: Transmit at 1 Gbps Linux No Offload

A.8 Transmit at 1 Gbps Under Load

Real (sec)	Throughput (megabyte/sec)
15.291	95.518
15.394	94.879
16.539	88.311
15.768	92.629
15.434	94.633
15.685	**93.194**

Table A.22: Transmit at 1 Gbps Under Load UNL

Real (sec)	Throughput (megabyte/sec)
13.610	107.316
13.544	107.839
13.827	105.632
13.534	107.919
14.478	100.882
13.799	**105.917**

Table A.23: Transmit at 1 Gbps Under Load Linux

Real (sec)	**Throughput (megabyte/sec)**
13.438	108.690
13.421	108.827
13.454	108.560
13.604	107.363
13.420	108.835
13.467	**108.455**

Table A.24: Transmit at 1 Gbps Under Load Linux (no offload)

Bibliography

[1] G. M. Amdahl, Validity of the single-processor approach to achieving large scale computing capabilities. In *Proceedings of AFIPS Conference*, 1967, pp. 483-485.

[2] M. Bailey, B. Gopal, M. Pagels, L. Peterson, P. Sarkar. PATHFINDER: A pattern-based packet classifier. In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI)*, 1994, pp. 115-123.

[3] T. Braun, C. Diot, A. Hoglander, V. Roca. An Experimental User Level Implementation of TCP. *INRIA Rapport de recherche No 2650*, September 1995.

[4] D. Clark, V. Jacobson, J. Romkey, H. Salwen. An analysis of TCP processing overhead. *IEEE Communications*, pp 23-29, June 1989.

[5] P. Druschel, L. L. Peterson. Experiences with a High-Speed Network Adaptor: A Software Perspective. In *Proceedings of the SIGCOMM '94 Conference*, pp 2-13. August 1994.

[6] P. Druschel, G. Banga. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSD*I) *USENIX*, October 1996.

[7] K. Egevang, P. Francis. The IP Network Address Translator (NAT), RFC 1631, May 1994.

[8] T. von Eicken, A. Basu, V. Buch, W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pp 40-52, 1995.

[9] D. Ely, S. Savage, D. Wetherall. Alpine: A User-Level Infrastructure for Network Protocol Development. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS '01)*, 2001.

[10] D. Engler, M. F. Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *Proceedings of the SIGCOMM '96 Conference*, pp 53-59, August 1996.

[11] S. Floyd, M. Handley, E. Kohler. Problem Statement for the Datagram Congestion Control Protocol (DCCP). *RFC 4336*, March 2006.

[12] A. Foong, T. Huff, H. Hum, J. Patwardhan, G. Regnier. TCP Performance Re-Visited. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2003.

[13] V. Fuller, T. Li, J. Yu, K. Varadhan. Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy. RFC 1519, September 1993.

[14] G. Ganger, D. Engler, M. F. Kaashoek, H. Briceno, R. Hunt. Fast and Flexible Application-Level Networking on Exokernel Systems. *ACM Transactions on Computer Systems*, Vol. 20, No. 1, pp 49-83, February 2002.

[15] S. H. Goldberg, J. A. Mouton Jr. A base for portable communications software. *IBM Systems Journal*, Vol 30 No 3, 1991.

[16] R. Gopalakrishnan, Gurudatta M. Parulkar. Efficient User-Space Protocol Implementations with QoS Guarantees Using Real-Time Upcalls. *IEEE/ACM Transactions on Networking (TON)*, Volume 6, Issue 4, August 1998.

[17] Gtk+ The GIMP Toolkit. URL http://www.gtk.org.

[18] History of the PSTN. URL http://www.inetdaemon.com/tutorials/telecom/pstn/history.shtml.

[19] G. Huston. Just how big is IPv6? - or Where did all those addresses go?. URL http://www.potaroo.net/ispcol/2005-07/ipv6size.html.

[20] IEEE POSIX® Certification Authority. URL http://standards.ieee.org/regauth/posix/.

[21] V. Jacobson, R. Braden, D. Borman. TCP Extensions for High Performance. *RFC 1323*, May 1992.

[22] V. Jacobson, B. Felderman, Speeding up Networking, URL http://www.lemis.com/grog/Documentation/vj/lca06vj.pdf.

[23] E. Kohler, M. Handley, S. Floyd. Datagram Congestion Control Protocol (DCCP). *RFC 4340*, March 2006.

[24] Greg Kroah-Hartman. The Linux Kernel Driver Interface. URL http://www.kroah.com/log/linux/stable_api_nonsense.html.

[25] Bobby Krupczak, Kenneth L. Calvert, Mostafa H. Ammar. Increasing the Portability and Re-Usability of Protocol Code. *IEEE/ACM Transactions on Networking*, Vol 5 Issue 4, August 1997.

[26] J. F. Kurose, K. W. Ross. *Computer Networking: A Top-Down Approach Featuring the Internet*. Addison-Wesley, 2003. 55.

[27] A. Leibman. Intel Core 2 Extreme QX6850. *Computer Power User*, Vol 7, Issue 9, September 2007.

[28] Libpcap. URL http://www.tcpdump.org/.

[29] S. McCanne, V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Technical Conference Proceedings*, pp 259-269, Winter 1994.

[30] S. A. McKee. Reflections on the Memory Wall. In *Computing Frontiers (CF04)*, Ischita, IT, April 2004.

[31] T. Narten, R. Draves. Privacy Extensions for Stateless Address Autoconfiguration in IPv6, *RFC 3041*, January 2001.

[32] J. Paris, V. Gulias, A. Valderruten. A High Performance Erlang TCP/IP Stack. *Erlang '05*, 2005.

[33] R. Perlman. *Interconnections Second Edition: Bridges, Routers, Switches, and Internetworking Protocols*. Addison-Wesley, 2000. 5.

[34] I. Pratt, K. Fraser. Arsenic: A User-Accessible Gigabit Ethernet Interface. *IEEE Infoco*m *2001*, 2001.

[35] J. Postel. Internet Control Message Protocol. *RFC 792*, September 1981.

[36] J. Postel Ed. Transmission Control Protocol. *RFC 793*. September 1981.

[37] J. Postel. User Datagram Protocol. RFC 768, August 1980.

[38] Protocol numbers. URL http://www.iana.org/assignments/protocol-numbers.

[39] K. K. Ramakrishnan. Scheduling issues for interfacing to high speed networks. In *Proceedings of Globecom 1992 IEEE Global Telecommunications Conference*, pp 622-626, December 1992.

[40] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, E. Lear. Address Allocation for Private Internets. *RFC 1918*, February 1996.

[41] J. Rosenberg, J. Weinberger, C. Huitema, R. Mahy. STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs). *RFC 3489*, March 2003.

[42] J. Rosenberg, R. Mahy, C. Huitema. Traversal Using Relay NAT (TURN). *Internet Draft*, September 2005.

[43] J. Rosenberg. Interactive Connectivity Establishment (ICE): A Methodology for Network Address Translator (NAT) Traversal for Offer/Answer Protocols. *Internet Draft*, March 2007.

[44] J.H. Saltzer, D.P. Reed, and D.D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277-288, November 1984.

[45] D. Searls, D. Weinberger. World of Ends, URL http://www.worldofends.com/.

[46] D. Siemon. Pnet, URL http://projects.coverfire.com/pnet/.

[47] D. Siemon. Userspace Networking Library. URL http://projects.coverfire.com/unl/.

[48] W. Richard Stevens, TCP/IP Illustrated Volume 1 (Reading Massachusetts: Addison-Wesley, 1994), 243.

[49] W. Richard Stevens, TCP/IP Illustrated Volume 1 (Reading Massachusetts: Addison-Wesley, 1994), 246.

[50] W. Richard Stevens, TCP/IP Illustrated Volume 1 (Reading Massachusetts: Addison-Wesley, 1994), 267.

[51] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, V. Paxson. Stream Control Transmission Protocol. *RFC 2960*, October 2000.

[52] R. Stewart, Q. Xie, L. Yarroll, K. Poon, M. Tuexen. Sockets API Extensions for Stream Control Transmission Protocol (SCTP). *Internet Draft*, July 2007.

[53] C. Thekkath, T. Nguyen, E. Moy, E. Lazowska. Implementing Network Protocols at User Level. *IEEE/ACM Transactions on Networking*, Vol 1 No 5, pp 554-565, October 1993.

[54] TOP 500 Supercomputer Sites. URL http://www.top500.org.

[55] UPnP Standards. URL http://upnp.org/standardizeddcps/default.asp.

[56] G. Varghese. *Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices*. Morgan Kaufmann Publishers, Inc., 2004.

[57] *Vulnerability Summary CVE-2000-0305*. May 2000. URL http://nvd.nist.gov/nvd.cfm?cvename=CVE-2000-0305.

[58] *Vulnerability Summary CVE-2003-0364*. June 2003. URL http://nvd.nist.gov/nvd.cfm?cvename=CVE-2003-0364.

[59] M. Welsh, A. Basu, T. von Eicken. ATM and fast ethernet network interfaces for user-level communication. In *Proceedings of the Third International Symposium on High Performance Computer Architecture (HPCA)*, San Antonio, Texas, February 1997.

[60] D. K. Yau, S. S. Lam. Migrating Sockets - End System Support for Networking with Quality of Service Guarantees. *IEEE/ACM Transactions on Networking*, Vol 6 No 6, pp 700-716, December 1998.

[61] G. Ziemba, D. Reed, P. Triana. Security Considerations for IP Fragment Filtering. *RFC 1858*. October 1995.

VITA

NAME: Dan L. Siemon

PREVIOUS POSITIONS

University of Western Ontario — September 2005 to December 2006

Teaching Assistant in the Department of Computer Science

EDUCATION

University of Western Ontario — September 2005 to present

M.Sc. (Computer Science) expected August 2007,
Supervisor: Hanan Lutfiyya

University of Western Ontario — September 2001 to April 2005

Hons. B.Science. (Computer Science with Distinction) received June 2005

AWARDS

Natural Sciences and Engineering Research Council of Canada (NSERC) Postgraduate Scholarship (PGS-M) (2005 - present)

Western Graduate Tuition Scholarship (2005 - present)

University of Western Ontario Gold Medal for Honors Computer Science (2005)

NSERC Undergraduate Student Research Award (USRA) (2004)