Imperial College London

Final year project

June 14, 2010

Improving Networking

by moving the network stack to userspace

Author:
Matthew Whitworth

Supervisor:
Dr. Naranker Dulay

Abstract

In our modern, networked world the software, protocols and algorithms involved in communication are among some of the most critical parts of an operating system. The core communication software in most modern systems is the network stack, but its basic monolithic design and functioning has remained unchanged for decades.

Here we present an adaptable user-space network stack, as an addition to my operating system Whitix. The ideas and concepts presented in this report, however, are applicable to any mainstream operating system. We show how re-imagining the whole architecture of networking in a modern operating system offers numerous benefits for *stack-application interactivity*, *protocol extensibility*, and improvements in *network throughput* and *latency*.

Acknowledgements

I would like to thank Naranker Dulay for supervising me during the course of this project. His time spent offering constructive feedback about the progress of the project is very much appreciated.

I would also like to thank my family and friends for their support, and also anybody who has contributed to Whitix in the past or offered encouragement with the project.

Contents

Chapter 1

Introduction

In this report, I present an adaptable interactive TCP/IP user-space network stack as an addition to my operating system *Whitix*.[1] As well as decoupling the policy of producing and processing packets from the mechanism of sending and receiving them, it scales and adapts better than existing solutions as well as adhering strongly to the *end-to-end principle* of the Internet. First of all, I will list the current problems and limitations in the field of network stacks, before proposing my solution in detail and the original contributions that I will make, along with a list of objectives that need to be achieved to satisfy the project goals.

1.1 Motivation

In our modern, networked world the software, protocols and algorithms involved in communication are among some of the most critical parts of an operating system. The core communication software in most modern systems is the network stack, but its basic monolithic design and functionality appears to have remained unchanged since the 1970s. With the progress in processor technology aiming towards *multi-core* and *many-core* systems,[38] there is a need for a new network stack design to replace the current monolithic stacks of today with (what I believe to be) their restricted legacy designs. First of all, I shall summarize the problems inherent in current implementations.

1.1.1 Adaptability and interactivity

The first major problem is the lack of adaptability in and feedback from the network stack in general. Although certain transport protocols like TCP can adapt their data transmission algorithms (through mechanisms such as *window scaling* and *congestion control*) to suit the circumstances, this useful information, which

[1] *Whitix* is a 32-bit Unix-like operating system that I wrote from scratch. It is detailed in Section 1.2.

includes estimates about the *round-trip time* and other knowledge about the network, is lost once the socket is closed or the application exits – most estimates would be very useful in future connections.[5] Applications receive no feedback on the state of the network from the lower layers, and can unknowingly flood a congested network with packets. Applications also vary in their usage patterns; a SSH session consisting mainly of single keystrokes will need a different strategy for data transmission compared to a large file transferred over a HTTP session – the exact use of TCP and network bandwidth varies by protocol.[9]

Wouldn't it be useful for the network stack to offer feedback to the application, so that in times of network congestion, the application could adapt accordingly? This would allow the application to be more efficient with bandwidth and offer a better user experience in all network conditions. Such a symbiotic relationship between the application and the network could minimize delays, network congestion and reduce temporal disturbance (i.e. "buffering") to the user. If we move the network stack into userspace, we can let the application know when to throttle TCP traffic. This would really help with network congestion; in current implementations, there is no way of notifying interested stakeholders about network congestion involving a particular host.

Another problem with current network stacks is that they treat their incoming data as a stream of bytes, leaving error checking, protocol verification and other networking-related issues to the application. Just as some applications have predictable network usage patterns, some applications have predictable network protocols. For instance, every HTTP session over a TCP connection comprises messages with a defined request and response format. If we moved the network stack down to userspace, we could place high-level network functions like validation and verification at that level as a common userspace library. By letting the network stack handle verification of packets in a generic way, this could lead to much more secure, smaller and safer code.

1.1.2 Multiprocessor systems and locking

The second problem is the ever-more multi-core design of today's systems. The typical design of the network stack is running into a number of problems as systems move towards a *many-core* design; a design where the number of cores is large enough that traditional multiprocessor paradigms are no longer efficient.[64] For example, passing a packet through different layers of the network stack often involves switching to different CPUs in the process, although network stack developers have improved CPU locality recently. To illustrate this, a typical path in the Linux kernel from receiving a Ethernet packet to distributing it to the application is:[29]

1. **Interrupt**. The Ethernet card signals an interrupt, which is handled by an arbitrary CPU. The interrupt routine is run, which typically schedules a *soft interrupt* (softirq) to handle the packet.

2. **Device processing (optional).** The remainder of the device driver work usually takes place here. The routine fetches the network packet from the

card via *direct memory access* (DMA) or *programmed I/O* (PIO) into RAM. A socket buffer is then allocated for the data, added to a work-queue list, and then the receive soft-interrupt is marked for execution.

3. **Packet processing.** Most of the network stack processing occurs here. The packet itself is checked for integrity (such as a valid IP checksum), then, if possible, it is processed as part of a UDP session or TCP connection. Typically the packet data is copied to a per-socket list, ready for the application to process it.

4. **Application.** At this point the application which owns the packet's socket has made a `read` or `recv` system call to retrieve the new data to process it. The data is copied from the per-socket list into the user-space buffer, and the application returns back to user-space to process the data.

Although Linux's packet processing has in fact been optimized for today's processors (typically, the first three stages run on the same CPU as a result of assigning a work-queue per processor), there are still several issues present in most monolithic network stacks (which are similar in design to that of Linux):

- **Lack of CPU locality.** There will still be a chance that the application will be running on a different CPU to the one that processed the packet. This will involve a big cache penalty as every memory access on the new processor will involve a trip to lower-level caches or main memory.

- **CPU usage.** Since the processor that handles the initial interrupt is chosen arbitrarily, the processing of the network packet could be assigned to a CPU that is currently running other tasks. For example, in the worst case, the window manager could be bound to a certain CPU, and if the processing of incoming packets occurs on that CPU, there will be issues with data throughput and latency from the network, especially if the window manager is busy rendering graphics at the time.

The way to address these issues to optimize the workflow as much as possible, by processing the packet in the hardware interrupt and saving as much work as possible for the application's context, while preferably eliminating the middle two steps in Linux's path. This saves multiple context changes and reduces CPU usage; the lack of CPU locality is unavoidable if we allow a thread to run on a range of processors.

In traditional network stacks, and in some where a kernel-space network spinlock[2] exists, the tendency to frequently take locks moves a cache-line between all processors and requires a cross-system atomic operation.[2] Both are expensive, and certainly do not scale well for increasing number of processors. Therefore, locking should be avoided as much as possible, especially in the network stack.[3]

[2] A *spinlock* is a synchronization primitive that provides mutual exclusion to a *critical section*, including code running in the network stack (for example, exclusive access to a particular network interface).

[3] Note that the Whitix kernel does not yet support SMP, but it does not prevent us from designing a multiprocessor-friendly network stack

1.1.3 Cache performance

The third and final problem, and also linked to today's multi-core systems and locking, is the cache performance of code. In today's systems, loading a cache-line from memory often takes 50 nanoseconds or more to complete.[25] As a result, when considering the performance of kernel code, and therefore of today's network stack, **cache performance** is often the dominant factor.[29] Linked to this, particularly in the Linux kernel, is the extensive use of linked lists to queue packets, which generally have poor cache performance and often lead to cache thrashing, especially in multi-core systems.

Although modern kernels have attempted to tackle the above problems, mostly by employing synchronization algorithms that avoid locking, such as *read-copy-update*,[2] I would argue that the way to making the network stack scalable, flexible and multi-core-friendly is to **perform as much work as possible in user-space**. This avoids many of the situations where data may need to be shared, and so it is unlikely that a cross-processor lock will have to be taken.

Another aspect of cache performance that relates to the network stack is the **copying of packets**. Large memory-to-memory copies are notoriously cache unfriendly, as the same data is duplicated in the L1 data cache, evicting useful data that may be used following the copy. After a copy operation, typically only one of the copies is used again before it is evicted from the cache.[15] Therefore, to increase cache performance and thereby speed up code performance, copying should be avoided where possible. A zero-copy send operation, for example, would be ideal for performance and memory usage.

1.2 Whitix

My userspace network stack has been developed for my operating system Whitix. Whitix is a 32-bit Unix-inspired operating system based around a monolithic but modular kernel. I have developed it from scratch over a number of years, and it is not forked from any other operating system. Unless noted otherwise, the Whitix design can be assumed to be similar to that of Linux. The only networking that was present at the start of the project was a small *local socket* implementation, developed for the operating system's windowing environment. This is likely to be made obsolete following the project, and replaced with a form of local network channels. The key architectural points of difference are:

- **System configuration**. Whitix kernel and the drivers do not use `ioctl`. Instead, the *Information and Configuration Filesystem* (IcFs) is a hierarchical tree for system configuration, broader in scope than `sysfs` in Linux and intended as a complete replacement for `ioctl`. Devices can expose a set of name-value pairs to allow userspace applications to read and write configuration values via the filesystem or by using special system calls such as `SysConfRead` and `SysConfWrite`.

- **Processes and threads**. In Whitix, new processes are created via `SysCreateProcess`

– there is no `fork` equivalent. Instead, threads, using the kernel threads implementation, can be created by calling `SysCreateThread`, suspended and restarted using `SysSuspendThread` and `SysResumeThread`, and destroyed by calling `SysExitThread`.

1.3 Outline

To solve the above problems and to add increased performance and flexibility to networked applications, I have developed a novel networking subsystem (applicable to almost any operating system) that places most of the functionality into user-space, with a small zero-copy array of buffers linking user-space and kernel-space. The components of this new network stack are:

1. **Network drivers.** We present the Linux driver layer, a framework for porting Linux network drivers and providing a compatible runtime interface, and a set of components for managing packet I/O and network interfaces. *(Chapter 2)*

2. **Network channels.** Building on top of the Linux Driver Layer, we present *network channels*, high-performance packet arrays designed for zero-copy send and receive. We detail an innovative fully-featured layer that is much more complete than competing research implementations.

   We chronicle the development behind network channels, and explain their superior cache performance and scalability as well as their memory layout. We describe the small protocol-specific kernel components for IPv4, and the framework for classifying incoming packets. The process for sending and receiving packets is outlined – we show that even adding a packet filter to the kernel is small, stable and fast. We detail the *usercode library*, a flexible mechanism that abstracts the internal structures of a channel for flexibility and stability. *(Chapter 3)*

3. **Userspace network stack.** We demonstrate that a modern userspace TCP/IP stack can be built as a shared library on top of network channels for increased security and performance, with a special focus on the *Transport Control Protocol* (TCP) as the most complex part of the stack.

   We investigate the userspace interface for network channels, and how the TCP/IP stack is built upon this. We detail the fully functional implementations of transport protocols such as UDP, ICMP, and the most complex, TCP, and their accompanying applications and utilities – many of which use the innovative adaptability and interactivity features of the network stack. We detail the two APIs available for the stack, the *native* and *POSIX* APIs, and how the former exposes the new functionality available in the stack. *(Chapter 4)*

4. **Network stack interactivity** We will then explore the concept of *events* in a TCP/IP connection, focusing especially on TCP events. We then detail a innovative method of exposing these events to an application through the use

of *callback functions*, a finer-grained primitive compared to other implementations, which use whole processes to handle events. We then detail a framework for user applications to monitor, filter and respond to these events.

The ability to establish a flow of events and hints, through the use of callbacks and helper functions, between the application and the network stack breaks new ground compared to other monolithic kernel-based stacks. We describe the myriad use of events in providing applications with performance and diagnostic information about network conditions, as well as the ability to profile network usage and feed the data into future runs to optimize performance. *(Chapter 5)*

Chapter 2

Hardware and the LDL

The basis of any networking software is the exchange of data over physical or virtual network interfaces. If we are to transmit or receive data and pass it to the network stack, we will require a layer for network drivers, and a framework to manage the raw packets involved.

In this chapter, we demonstrate how we have transferred support for a range of Ethernet cards from Linux, by developing the *Linux Driver Layer*. We explain the caveats of supporting Linux drivers through a binary interface, and how the similarity of the Linux and Whitix driver model allow us to use Linux drivers to transmit and receive packets on almost all Ethernet cards.

The low-level networking layer provides one main abstraction: the network interface. The network interface provides an extra layer around a network card device for sending and receiving data packets, assigning network and hardware addresses via management functions, and for referencing in routing tables (by a human-readable name). There are *two* classes of network interfaces found on Whitix. One is the loopback interface, a *virtual* interface implemented in software only, which is always available as `Loopback0`. Any packets sent to the interface are immediately received on the interface. Second is the hardware Ethernet interfaces, where one *physical* interface corresponds to an Ethernet card. The Ethernet interfaces are named `Ethernet0`, `Ethernet1` and so on, although most desktop machines only have one such card.

2.1 Architectural overview

There are four main components of the low-level networking layer, as detailed in Figure 2.1. All are placed in the kernel for reasons of speed and direct hardware access (and the idea of placing network drivers in user-space is evaluated and discarded in Section 3.2.1). The **network drivers** are the main abstraction of the layer; they allow the other components to interact with the device via management functions, as well as providing a single interface for sending and receiving packets

on the device. One other key driver component is the Linux driver layer, which is explained in detail in Section 2.3 and allows standard Linux network drivers to run as part of the Whitix kernel.

The **network device manager** manages the different physical and virtual network cards, assigning them a human readable name and exposing them to the *Information and Configuration filesystem* (IcFs). The manager registers a standard set of values for configuration purposes: each device has a *network address*, used by higher-level routing code, and, for physical interfaces, a *hardware address*. (For example, the MAC address of the first Ethernet card in the system is available at `/System/Devices/Network/Ethernet0/macAddress` as a file of six bytes). These values are used by network utilities to configure the system appropriately.

For physical network interfaces, which actually transmit packets to other immediate hosts based on their hardware address, a global **hardware address cache** is present in the kernel. This is mainly used by the routing code, which queries the cache for a translation from network to hardware address (in that respect, it functions as a *translation look-aside buffer*). To satisfy these requests, this particular implementation of the hardware address cache searches the generic cache, and if a suitable entry is not found, issues Address Resolution Protocol (ARP) requests to translate IP addresses to Ethernet MAC addresses. However, the caching code is written to function with any combination of request packet type, network address format and hardware address class.

The last component of the layer is the **packet I/O** layer, which interfaces with the high-level layers to provide an abstraction for sending a single network packet, as well as asynchronously notifying upper layers of a packet receive event. These packets are treated by the layer as a stream of bytes; hardware addressing and packet construction takes place at the network channel layer, with the help of device-class-specific functions for retrieving hardware addresses. A per-card queue for outgoing packets is provided, and used if the network card is busy transmitting a previously queued packet.

2.2 Network drivers

The driver architecture of Whitix is inspired by that of the Linux kernel, although with a stricter attitude to modularization; drivers are distributed as modules, and there are very few device drivers built into the kernel itself. The Whitix kernel is a monolithic kernel, with a range of subsystems available to drivers, such as timers, scheduling and bus subsystems. Device drivers use functions provided by the kernel and associated modules. The different subsystems of the Whitix kernel, such as the different bus components (such as PCI and USB), make functions used by drivers available via the `SYMBOL_EXPORT` macro (e.g. `SYMBOL_EXPORT(TimerAdd)`, from the part of the kernel that provides single-shot timers for use by other components).

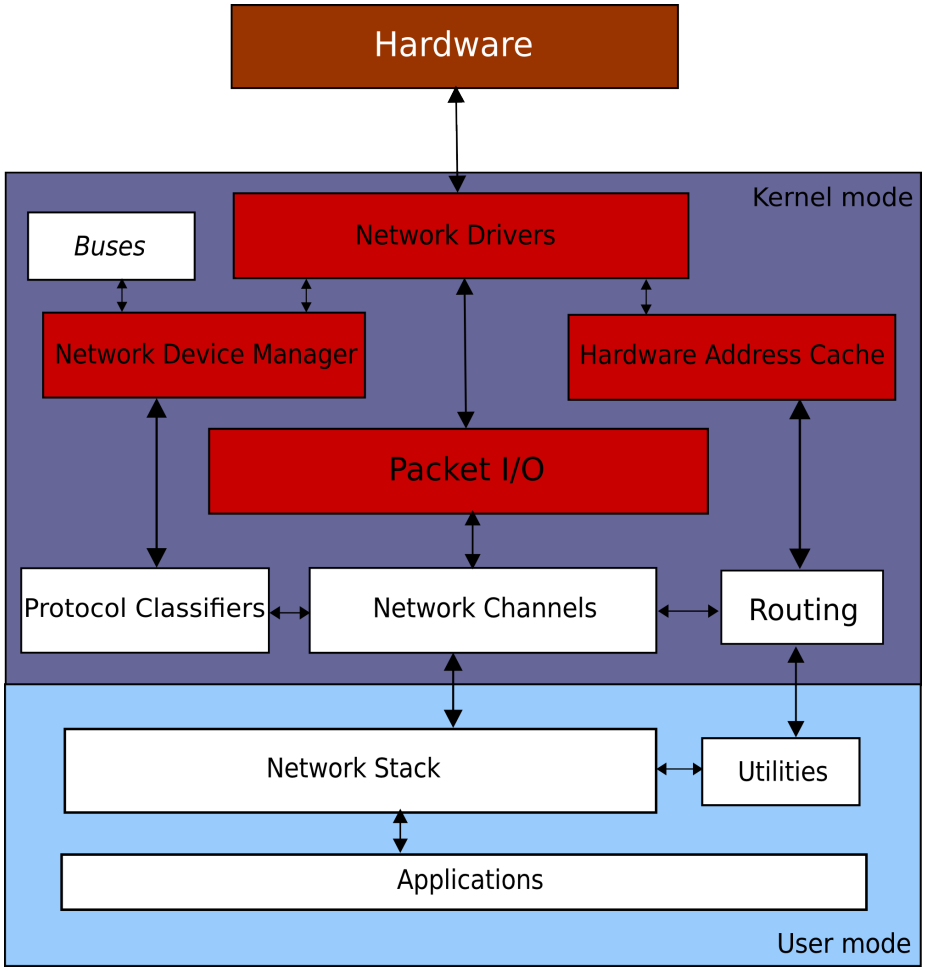

Figure 2.1: The low-level networking layer comprises the network drivers, the network device manager, the hardware address cache and the packet I/O layer. It interfaces between the networking hardware and the network channel layer to abstract the class-specific and hardware-specific functions of each device to allow packets to be transmitted and received in a device-transparent manner.

```
struct PciDeviceId pcnetIdTable[]=
{
    {0x1022, 0x2000, PCI_ID_ANY, PCI_ID_ANY, 0, 0, NULL},
    {0x1022, 0x2001, PCI_ID_ANY, PCI_ID_ANY, 0, 0, NULL},
    PciTableEnd(),
};
```

Figure 2.2: An example of the list of supported devices supported by the Whitix `pcnet32` driver (not used in this project). The first two fields are the vendor ID and device ID, two defining fields describing a device.

2.2.1 Driver and device setup

Whitix drivers, denoted by the `.sys` suffix, are loaded via the `SysModuleAdd` system call, and the module is then linked into the kernel by resolving these symbols. The module's list of symbols is then in turn made available to other modules. Network drivers mainly interact with the bus manager and network device manager layers. Ethernet devices are typically found on the Peripheral Component Interconnect (PCI) bus, and drivers wishing to find devices register their list of supported devices (provided as a filter on the different fields of the device's PCI configuration space), as in Figure 2.2, and a function to be called when a suitable PCI device is connected. Drivers (and modules) remain in memory until explicitly unloaded via `SysModuleRemove`.

On being assigned a device, the driver generally performs the following setup procedure (this example is taken from the `Ne2kInitOne` function in the Whitix `ne2k-pci` driver, available at `devices/net/ne2k-pci.c` in the source tree):

1. **PCI device and resource setup**. First of all, the device is enabled, with any relevant PCI settings (such as DMA bus mastering) performed at this point. The driver then locates the device resources, such as interrupt lines (a function is assigned to handle the interrupt), I/O ports and memory regions. The NE2000 driver locates the base address for its physical registers in the x86 I/O space, which it then accesses via the `inb` and `outb` machine instructions.

2. **Hardware setup**. The driver then performs a device-specific setup procedure. Note that the device does not have to be ready for packet sending and receiving immediately after setup; this can be delayed until the interface is brought up. This may also involve retrieving the hardware address; in the NE2000 driver, the Ethernet MAC address is read from the programmable ROM.

3. **Network device registration**. This is the part of the setup procedure that is of interest to the network device manager. The exact registration process, along with the information the driver supplies about the device, is detailed in Section 2.4. The driver also registers a structure containing device operations, essentially its interface to the higher layers, with operations specific to

```
struct NetDevOps
{
        ...
        int (*buildHeader)(struct NetDevice* device, struct NetBuffer*
            sockBuff, void* address);
        int (*send)(struct NetDevice* device, struct NetBuffer* sockBuff);
        ...
};
```

Figure 2.3: A subset of the operations available in the `NetDevOps` structure, which describes the device-specific operations for a device. `send` is the packet transmission function and assigned by the driver, and `buildHeader` is used by the higher-level code to build a class-specific hardware header. For example, `EthDevRegister` assigns `EthBuildHeader` if no device-specific function is supplied.

the device (`Send` for different devices) and its class (`GetHeaderLen`, for Ethernet cards), with the help of class-specific functions like `EthDevRegister`.

After the device is registered by the driver, the device stays idle until the network interface is *brought up* by `dhcp`, essentially allowing it to transmit and receive and function as a "live" interface. Bringing up an interface, in terms of the network driver, involves awakening the device and preparing it for transmitting and receiving packets. *Bringing down* an interface is the opposite process.

2.2.2 Sending packets

Packet transmission is a synchronous operation from the viewpoint of higher layers. It is an operation described in more depth in later sections and carried out through multiple layers of the networking code. From the driver's viewpoint, the only send-related operation that needs to be implemented is the `send` operation (as described in Figure 2.2.1), which, along with the device, takes a single packet (available as a `NetSendBuffer`) and transmit it in a device-specific manner. The network device may also signal to the higher layers that it is busy transmitting the packet as a result; the packet I/O code can respond by queuing packets, and once the network device is not busy, can resume packet transmission.

2.2.3 Receiving packets

In constrast, receiving a packet is an asynchronous operation. In most cases, the network card will raise a receive interrupt, and the driver's interrupt code will be called as soon as possible to handle it. The driver typically allocates a temporary network buffer (the `NetBuffer` structure) to transfer the packet from the card, fills in key fields in the buffer structure, and passes it up to the packet I/O code by calling the `NetRecv` function, which is part of the channel layer. Ignoring device-specific code, the receive code is simple and does not inspect the content of the packet.

Due to the nature of receiving packets, there may be a queue of received packets on the network card itself. This is invisible to the driver and the rest of the networking subsystem, and is usually handled by the device raising a series of interrupts to handle each packet, or noting to the driver via a device register that multiple packets need to be received in one interrupt.

2.3 Linux Driver Layer

Developing a driver for network hardware that works effectively and efficiently on all models is a time-consuming and difficult task. To write network drivers for Whitix, a laborious task that replicates almost exactly previously written drivers, for even only the set of machines that I will test the project code on would take away many hours of development time from the more innovative aspects of the project. I decided therefore to look to porting network drivers for another operating system, such as the Linux or one of the BSDs. I chose to port Linux drivers, found at `drivers/net` in the recent Linux source trees, instead of the BSD network drivers due to licensing compatibility concerns and the extent to which the kernel would be a "derived work" of the drivers.[1]

It became clear I would have to write a compatibility layer, named the **Linux Driver Layer** (LDL), to emulate the functions provided by the different subsystems of the Linux kernel. However, the similarity in the driver architecture of the two operating systems, as noted in Section 2.2, was a bonus; in particular, the network device and bus functions of the two kernels are very similar. This is fortunate: performance could have been degraded if major translation of structures and data was needed between the Linux layer and the corresponding native Whitix interfaces. Although the concept of Linux driver compatibility is extendible to virtually all types of device driver, I focused only on the functions and subsystems needed to run certain drivers.

Device setup, interrupt handling and transmission works in a very similar fashion to the native network drivers; the only difference between the two drivers is that the Linux driver believes it is running on a Linux kernel; the emulation of the Linux subsystems, including the translation of structures in the layer, is sufficient for this to hold. The LDL code itself is located in `devices/linux/` in the source tree, and is organized approximately by subsystem type.

2.3.1 Caveats

However, one major problem I foresaw involved the stability of the Linux kernel driver interface. Unlike the (binary) kernel to user-space interface of Linux, which is very stable, the internal driver interface at source-level changes frequently from version to version, and at binary-level, because of the wide range of configuration

[1] The Whitix kernel is licensed under the GNU General Public License (GPL), whereas the network drivers from the BSD kernels would be licensed under the BSD license.

options for the Linux kernel,[2] changes from machine to machine. This is suitable for the Linux kernel developers, who can update kernel drivers in the development tree, but awkward for closed-source driver developers who develop outside of the kernel tree; they must adapt for each version of the software. [34]

As a result, and to simplify the implementation, I have selected the kernel modules supplied with my Ubuntu Linux distribution. These are built for the same kernel version (2.6.28.11) and with the same configuration options. Therefore, the Linux Driver Layer only supports modules built against that kernel version. Any new devices and drivers that are released will mean the LDL will need to be updated to support a new kernel version. This is the disadvantage of an unstable kernel binary interface and driver API, but the time saved on my part outweighs any flexibility cost involved.

2.4 Network device manager

To enable efficient routing of packets and interface management, the different network devices are managed by the **network device manager**. The manager represents devices as interfaces to higher level code; the extra abstraction allows the storing of statistics and the status of the interface, as well as group devices by hardware class (Ethernet and virtual devices are just two examples of device classes) a common set of IcFs configuration name-value pairs and a human readable name to describe the interface.

The network drivers register the device by calling `NetDevRegister` (or a wrapper function), as described in 2.2, using a structure that contains the attributes used to describe the new interface. An Ethernet card typically calls the `EthDevRegister` function, which fills in the fields specific to the hardware class, such as the device name prefix, the hardware addressing callbacks and the packet sizes (a similar process would occur for IEEE 802.11[3]) when support for them is implemented).

To take the packet size as an example, 10 megabit Ethernet has a minimum packet size of 64 bytes, and a maximum size of 1500 bytes. This information is important when allocating buffers for network channels; the maximum size of a buffer corresponds to the *maximum transmission unit* (MTU) of the underlying device. For loopback interfaces, the maximum packet size is arbitrary (there are no limitations like in hardware), but the maximum packet size is generally limited to under the page size of the machine, mainly to ease memory management in the network channel layer.

The class-specific registration function calls `NetDevRegister`, which attaches the device to the internal device tree and registers generic configuration variables

[2] For example, kernel data structures will contain different field alignments, depending on the C compiler and a particular version. A SMP or non-SMP kernel can be built, which means some fields may not be present in memory at all for a non-SMP build

[3] Also known as WiFi devices, they use MAC addresses like Ethernet devices, but transmit and receive over a radio-based transmission medium and have different packet header requirements from Ethernet Devices.

```
/System/Config/Devices/Network/>ls
Loopback0/
Ethernet0/
/System/Config/Devices/Network/>ls Ethernet0
running
macAddress
netAddress
/System/Config/Devices/Network/>ls Loopback0
running
```

Figure 2.4: A list of basic information and configuration variables that be read and written by applications and utilities. An implementation of device management would expose more statistics and internal hardware information and configuration options, similar to class-specific programs such as `ethtool` or `iwconfig` on Linux.

with IcFs: `running` is a single byte value that, when written to, actually calls `NetDeviceSetRunning`. The written value, either 0 or 1, brings the device down or up respectively. The class-specific registration function also adds suitable IcFs name-value pairs; `EthDevRegister` adds two entries for addresses: `macAddress` and `netAddress` (for IPv4). These configuration values are used by network utilities such as `dhcp` to construct raw packets and register any new network addresses. The configuration directory structure for a test machine is shown in Figure 2.4.

As the network device manager has a list of all the interfaces in the system, it also provides some functions to search the list or aggregate information over the set of interfaces. `NetDeviceFind` finds a `NetDevice`, given a human-readable name; this is useful for applications using network channels that want to bind to a particular interface, `dhcp` again a good example.[4]

Functions that aggregate information over the set include `NetDevFindMinMtu` and `NetDevFindMaxHeader`, used for determining the maximum packet size and header size for channels (and their list of buffers) whose packets could be sent out over any interface; one example is a UDP channel, whose destination address could be the local host (then the packet is sent over the `Loopback` interface) or any other host on the Internet (in which case the `Ethernet` interface is used). Both have different minimum and maximum packet sizes.

2.5 Hardware address cache

The low-level networking layer deals with two classes of addresses: the network address (e.g. IPv4, addresses like 143.23.2.43) and the corresponding hardware address that the packet should be routed to. For example, when building the Ethernet header, we need to resolve the destination network address to the destination hardware address (which may or may not correspond to the actual destination's

[4] `dhcp` is a rather unique system utility; it assigns a IPv4 address to a device, and so, transmitting and receiving over a network interface with no IPv4 address, requires special routing and configuration needs that no other program requires

```
int ArpGetLinkAddress(struct NetDevice* device, ulong sourceAddr, ulong
    destAddr, char* address);
```

Figure 2.5: Definition of the ArpGetLinkAddress function. It is called by higher layers to translate `destAddr` to a hardware address, that is then stored in `address`.

network address – this depends on the routing table). The **hardware address cache** helps hardware address resolving by providing a framework for translating between the two classes of addresses, formatting address request packets using the Address Resolution Protocol (ARP) and parsing replies. The cache also contains a mechanism for discarding old or out-of-date entries, and is optimized for random read accesses.

Currently, the hardware address cache only supports resolving IPv4 network addresses to Ethernet MAC hardware addresses. (There is no support for a reverse lookup, as very few network-related operations need such a lookup). The central function that provides to the higher layers (and most calls come from the routing layer) is `ArpGetLinkAddress`, which takes the parameters described in Figure 2.5.

The `ArpGetLinkAddress` function first looks up the `destAddr` in the cache. If it is not found, or is out of date, the function then constructs an ARP request packet, which broadcasts a message over `device` asking "*Who has* `destAddr`*? Tell* `sourceAddr`."[5] The calling thread then waits for a reply for the packet; ARP replies and requests from other machines are handled separately in a dedicated kernel thread, with waiting threads being notified asynchronously of success or failure. The returned hardware address is then copied to the buffer at `address`, and the function returns successfully.

If there is no hardware address corresponding to the network address, because the host does not exist on the network or there is no ARP reply, the cache repeatedly sends an ARP request message before returning an error to the caller, who usually returns an `-ENOROUTE` value to userspace to indicate this.

Since, in a rapidly changing network, the mappings between network addresses and hardware addresses can change frequently, the ARP cache always checks if the stored entry is out of date. Incoming ARP packets also update the cache; *ARP announcements*, optionally sent by hosts when their IP or MAC address changes, always update the cache, and we can store the hardware address of another machine that sends a ARP request packet to us.

2.6 Packet I/O

The main abstraction that the low-level code provides is a means for sending and receive packets over a selected network interface. The **packet I/O** layer includes the generic `NetDeviceSend` function, per-device queues for outgoing packets, and

[5] This is how the Wireshark packet sniffer describes ARP requests in its short packet summary.

the `NetBuffer` and `NetSendBuffer` structures, which abstracts incoming or outgoing buffers respectively. These buffers can originate from either user-space or kernel-space (e.g. `ECHO REPLY` messages) and the structures store context (depending on whether the packet is incoming or outgoing respectively) such as the length, the start of the data and the current read pointer.

The `NetBuffer` and `NetSendBuffer` structure is the most important abstraction of the packet I/O layer; it may at first appear, to take incoming packets represented by `NetBuffer` as an example, that only the memory address of the start of the packet and its length need to be passed through the different networking layers of the kernel. However, storing the device that the network packet was received becomes important when selecting a channel type to distribute the packet to in `NetRecv` in the channel layer.

The `NetSendBuffer` structures are statically allocated by the channel. One `NetSendBuffer` maps to one send buffer in the channel memory, and stores context information about the channel buffer while it is passing through the kernel.

2.7 Summary

In this chapter, we covered the extent of network device support in the kernel. Because of the comprehensive **Linux Driver Layer**, which provides a translation layer between Ethernet drivers for Linux and the native network device API, we can support virtually every Ethernet card available on the market. We detail the paths taken to **transmit and receive packets** in a lightweight fashion in the low levels of the stack, and how we queue packets and represent them in the system.

We then described the **network device manager**, and how it represents the network devices in the system as **interfaces**. We show how information about the interface and the class of interface, such as the MTU and size of the hardware header, are used by the higher layers to configure the memory layout for channels. The **hardware address cache** helps the network stack translate the network addresses of higher layers down to the hardware addresses each interface needs to transmit packets. Finally, we summarized the structures and functions comprising the **packet I/O** layer in the kernel.

Chapter 3

Network channels

In this chapter, we discuss the role of network channels in the user-space network stack architecture. The network channel, with its shared list of buffers, allows zero-copy sending and receiving of packets via a variety of network interfaces, with extremely fast buffer allocation and deallocation. This project's implementation of network channels adapts to different classes of connection paradigms seamlessly and with minimal configuration by user-space libraries and applications, and shared data structures are encapsulated using a special user-level library provided by the kernel.

Throughout this chapter, we discuss the design decisions involved, the tradeoffs inherent in any implementation, and a detailed survey of the project's implementation, noting its operation and memory layout. We conclude with an overview of the API provided by the network channel libraries, and several use cases by applications and libraries.

3.1 Background

Van Jacobson, in his 2006 talk,[29] after discussing the early history of network stacks and how they settled upon the Unix "standard model" of design, shows how locking, synchronization, multiple software interrupts and poor cache behavior conspire to reduce the scalability and performance of the current design of network stacks. He also notes that a network connection can be abstracted as a *servo-loop*,[1] whereas a kernel-based implementation converts this to two coupled loops. After noting that a "very general theorem" (Routh-Hurwitz) says that two coupled loops are less stable than one, he also says that the kernel loop hides the application dynamics from the sender, affecting socket time estimates and causing spurious retransmissions.

He states that the existence of two loops in most implementations has more than

[1] A *servo-loop* is a self-regulating feedback system or mechanism.

tripled the size of TCP, added the term "window" to distinguish between the arrival of data at the host and its consumption, and issues between implementations, such as the Silly Window Syndrome (SWS). SWS happens when poorly implemented TCP flow control algorithms request that the sender reduce their window until the window becomes so small that data transmission becomes extremely inefficient.

He also goes on to say that even for TCP, a "one size fits all" protocol implementation does not exist, implying that a network stack able to automatically adjust to different classes of connections (such as *transactions*, *bulk data* or *event streams*) would be preferred, rather than the static network stacks of today. He relates the problem to the increasing network speed of many connections, followed by the mismatch in the multiprocessing response to this development. He claims the end-to-end principle should be followed, by saying that in a multiprocessing system, the protocol work "should be done on the processor that's going to consume the data". To ensure this, he reworks the idea of packets being stored in a linked list to a modelling them as a lock-free FIFO structure called a **network channel**, noting that many network components have a producer/consumer relationship.

My project will not be the first implementation of **network channels** on a UNIX-like system. In 2006, Evgeniy Polyakov wrote an implementation of network channels for the Linux kernel, accompanying his proof-of-concept implementation of a userspace network stack.[2] However, it radically differs in design from my eventual design, as noted in later sections. Most importantly, it is not a zero-copy implementation; `netchannel_copy_from_user` and `netchannel_copy_to_user` copy network channel buffers to and from userspace, with the static array of buffers not shared with userspace.

3.2 Design

To make a user-space network stack work, there has to be an efficient method of linking the network stack and the drivers. There are a number of possibilities here. In line with the project's aims, in particular that of increasing cache performance, we can evaluate each design in turn with regards to the current architecture of Whitix:

3.2.1 Possibilities

User-space network drivers

One possibility is to place network stack drivers at the same privilege level as the network stack itself, to avoid expensive system calls into the kernel (and depending on the exact design, the overhead of data copying). However, the plan is unworkable, insecure and certainly not scalable, especially considering the current architecture of Whitix, for the following reasons.

[2]The source code for the implementation can be found at http://www.ioremap.net/cgi-bin/gitweb.cgi?p=netchannel.git;a=tree;f=net/core/netchannel;h=dfde00af264aa144a459d212bdeaa5f748e070db;hb=HEAD

First consider a single privileged process per hardware driver, known as a *server process*, as it handles requests from *client* or *user processes* via a message passing mechanism. This a design typically used in many microkernel operating systems, where each driver is given its own address space. This is mainly to address reliability concerns with driver stability, especially at a privileged level.

This first design was quickly discarded; the main disadvantage was the overheads associated with message passing. For example, copying data (in the form of sending a packet) from the client process to the server process might involve two memory copies and two context switches on a uniprocessor machine (from the client process to the server process, and the reverse in order to return the packet status the client process), even without including the overhead of system calls. For a bandwidth-intensive application, these costs would result in very limited performance.

The next design involves a compromise of less stability and much less security to gain higher performance. In this design, network drivers are linked into the user-space network stack library by dynamic linking, which is treated as a special type of user-space library. This design is unworkable as well; as well as the network stack library having different semantics compared to all other shared user libraries, each process wanting to use network capabilities must obtain or be given the appropriate privileges to manipulate hardware, which is undesirable for many potential insecure applications that contain security holes, such as web browsers!

Raw sockets and file I/O

The next design I considered was an adaptation of the raw socket for unprivileged applications. **Raw sockets**, a class of socket available in Unix-like operating systems such as Linux that can be used to generate any type of network packet. It functions as a normal socket, and allows users to bypass kernel network packet processing (usually for pure performance or special network setup, such as DHCP in IPv4).

Although this appears to be the ideal type of kernel object to use, there are number of disadvantages that means a more specialized design is required:

- **Security**. Allowing any user to construct their own packets without oversight by the kernel or a trusted process can lead to *network abuse*. For example, in TCP/IP, raw sockets can be used to launch *SYN flood attacks*, where a particular host is flooded with SYN packets (which function as connection requests in TCP) or perform *session hijacking*, where a malicious host sends a specially constructed FIN packet to spoof a connection close by one of the hosts.[30]

- **Copying and performance**. When sending a raw packet from user-space to the kernel,[3] the kernel must copy the packet to a private buffer or allocate

[3] In Linux, this is performed by calling `sendto`, or `write` and `send` if the application has specified the destination address to the kernel previously.

kernel-mapped pages to map the packet, to avoid the possibility of a malicious application unmapping the packet memory from another thread and causing the kernel to crash or fail to transmit the packet. As detailed in the project's introduction, this is not ideal for cache performance or the memory footprint of the application.[59]

- **Packet classification**. Owing to the very nature of raw sockets, received packets cannot be classified accurately by the kernel, as *no information* is given by the raw socket about the socket's connection context (if any) to the kernel. Depending on the layer at which the raw socket operates (either the data link layer or network layer), we must duplicate packets among all raw sockets in all processes (ignoring those that do not pass some optional criteria that the user-space can specify about the socket's packets). This is a major disadvantage, as it leads to very poor cache performance, massive data duplication and a range of security issues. With raw sockets, different applications run by different users can snoop each other's packets with ease, which is one reason why raw sockets are limited to privileged users.[59]

The above points, in particular the lack of efficient packet classification, means that the raw socket design will need to be adapted to be suitable for systems with many bandwidth-intensive network connections.

Network channels

The final design that I considered was an idea presented by Van Jacobson, known as **network channels**.[12] The key to more scalable network stacks is to make sure as much processing work is done on the CPU where the application that needs the data is running, extending the *end-to-end principle* as much as possible. Jacobson intends user-space applications to deal with one network channel per connection; a *network channel* is a circular buffer of constant size containing packet pointers. Circular buffers remove the need for *locking* and *writable cache lines* (which improves performance greatly, as both operations are very cache-unfriendly) between producer and consumer, with the kernel classifying packets to appropriate channels given certain protocol-specific information.

This appeared to be an ideal object for further investigation. Although Jacobson did not go into detail about a possible implementation, he gave the example of a user-space network stack (after some kernel-space examples concerning driver and socket code) that utilized zero-copy I/O. To send a complete TCP/IP packet in user-space, the application would place a pointer to the buffer in the network channel and signal, most likely via a `write` system call, that the packet should be transmitted. The kernel would then copy the packet to the network card without any intermediate copying involved.

When receiving a packet, the kernel utilizes channel context information given to it when the channel was created. Through a small amount of protocol-specific code, the kernel can then classify any incoming packet into the appropriate chan-

nels and copy the data into the channel.[4] The application, presumably polling the channel for incoming data via `poll` or blocking on incoming data in user-space, can then read the packet, process it as a TCP or UDP packet (for example) and return it to the user if applicable.

Conceptually, the network channel addresses the three major disadvantages of a user-space network stack that uses only raw sockets, mainly through the small protocol-specific areas of the in-kernel implementation:

- **Security**. With raw sockets, much of the security issues arise from unmonitored packet transmission. By using the protocol-specific information passed to the kernel when a channel is created, the kernel can verify, at any point during in-kernel packet transmission – right up until the packet is copied to the network card – that the packet source and destination match that of the supplied context. This verification can easily stop session hijacking, as the packets in a channel can no longer masquerade as packets from another host.

- **Copying and performance**. Because of the circular buffer design, packets can remain in user-space while the kernel transmits them. Each packet stays in the circular buffer until it is successfully transmitted (or in this implementation, the application has the option to manage transmission buffers); this allows the kernel to copy data from the underlying physical pages (via a temporary in-kernel mapping) to the network card without copying to a temporary buffer.

- **Packet classification**. As described above, protocol-specific context can be supplied upon channel creation to allow the kernel, with the help of protocol-specific code, to match the packet with the correct channel, with no duplication or unnecessary copying.

Possible disadvantages of Jacobson's network channels, and particularly my implementation, mainly stem from the circular buffer design; leaving the packets in user-space will allow applications to unmap the packet or change the contents before transmission, which, if not handled carefully by the kernel, may lead to a system crash or attack packets slipping past the kernel's packet verification procedures. *Jacobson does not specify exactly how userspace would share to-be-transmitted packets with the kernel*.

Without a last-minute packet verification check, one possible hole, which is effectively a *time-of-check-to-time-of-use* bug, can be exploited by a malicious application in the following manner:

1. **Create two threads**. One will send the packet using `SysWrite`, and the other will alter the packet's contents while the call is taking place.

[4] This memory copy is unavoidable. We must read the packet into a buffer in kernel space in order to inspect its contents for classification. One possible optimization, especially with large packets, is to read part of the header into the temporary buffer, classify the packet, and then copy and read the rest of the packet into the appropriate channel buffer - however, this split receive operation is not supported by many network cards.

2. **Create the network channel** with typical source and destination values, as well as specifying a protocol the channel will follow. In this example, we will consider TCP.

3. **Construct** a data packet in a network channel buffer that will pass verification, such as a TCP SYN packet, and pass the buffer address to the other thread.

4. **Signal** that the channel buffer is ready to be transmitted, via a `write` (or similar) system call, and (especially on a multiprocessor machine, where two threads in the same process can run concurrently) alter the buffer contents subtly to form an attack packet. One possible attack might involve changing the source address field in the IP header. If the attacker wins the race, a malicious packet has been transmitted.

However, due to the nature of packet transmission the attacker is unlikely to win the race, and even less likely if the packet is verified immediately (via a quick header checksum) before transmission or run on a uniprocessor machine. The chances of an attack packet being successful are diminishing with increasingly secure network stacks; for example, TCP session hijacking or reset attacks, without snooping, is now extremely unlikely with both hosts performing *initial sequence number randomization*.[4] However, it is possible to detect manipulated packets (perhaps not with a 100% success rate), and the kernel could possibly block the application from using network channel objects as a precaution to prevent further attacks.

The **goals of network channels** is to help build more cache-friendly, stable and scalable network stacks in the following fashion: Jacobson appears to design his network channels so no locks or shared writable lines are needed between the producer and consumer, so adding and removing data to the channel becomes a cache-friendly operation.[12] This appears to line up with most of the project goals and so Jacobson's network channels appear to be the best design to build upon. My changes to his proposed kernel object are detailed in the following sections, along with a more complete description of the implementation and operation of network channels.

3.2.2 Tradeoffs

The main tradeoff with moving network processing to user-space is that the sockets no longer map to file descriptors in the kernel.[5] This is especially important for operations shared with other file descriptors, such as typical files, which will still be represented as file descriptors. Operations like `poll` or the equivalent `SysPoll`, which take a list of file descriptors and returns, informing the application of events on those file descriptors, may be tricky to emulate with objects such as network channels or raw sockets.

[5] What type of file descriptor depends on the particular user-space interface; the class of file descriptor for a network stack based on raw sockets is not the same as an implementation based on network channels for instance.

Emulation is especially complicated because incoming data on these file descriptors does not always map to user-readable data (consider ACK messages on a TCP socket represented by a network channel). This means the application, if it is not carefully, will be woken up by `SysPoll` or `poll`, attempt to read the socket via `SocketRead` or `recv`, and block, perhaps for an undetermined amount of time, or not be woken up at all (if the network stack is simultaneously polling that channel. However, the documentation for the `select` function (an equivalent) notes the presence of spurious read notifications, even with kernel sockets. This issue is further discussed in Section 4.4.6 in the context of TCP sockets in the userspace network stack.

Other tradeoffs include being unable to aggregate or enumerate the network connections present in a machine (or, depending on the protocol, only at a very low-level). This is not normally important, but when we come to implement utilities such as `netstat`, which (among other things) prints a list of open sockets, or implement SNMP MIBs,[6] exactly how to retrieve or expose this information when most connection context is in separate userspace processes is difficult.

3.2.3 Architectural overview

Figure 3.1 depicts the network channel layer with regards to the overall networking architecture of Whitix. The user-space network stack interacts with the network channel layer via system calls such as `SysChannelCreate` and `SysChannelControl`, as well as general I/O calls such as `SysWrite` and `SysPoll`, where the network channel is treated as an ordinary file descriptor. The layer performs the following functions:

- **Channel management.** Each process in Whitix has a private array of file handles. An application may open channels using the `SysChannelCreate` call, specifying the source and destination addresses in a protocol-specific manner for the protocol classifier code, and receive a file handle for use with functions such as `SysWrite` and `SysPoll`. These handles are then closed with `SysClose`, and modified via `SysChannelControl` and `SysFileControl`.
- **Memory management.** The memory for incoming and outgoing packets needs to be managed correctly. The channel is represented as a static array of buffers shared between the kernel and userspace, partitioned by the buffer's purpose; the array of send and receive buffers are formatted and treated differently. Allocation and deallocation is performed through setting bits in one of these two bitmaps, with information about incoming and outgoing packets (such as the packet length) specified in a standard header format.
- **File emulation.** Since the network channel is represented as a file descriptor, it must fulfill a certain set of operations. Operations such as `SysWrite` and `SysPoll` should obey all existing semantics where possible, so that calling the functions on a channel file descriptor is indistinguishable from that of any other file descriptor.

[6] A future SNMP implementation is discussed in Section 8.2.4

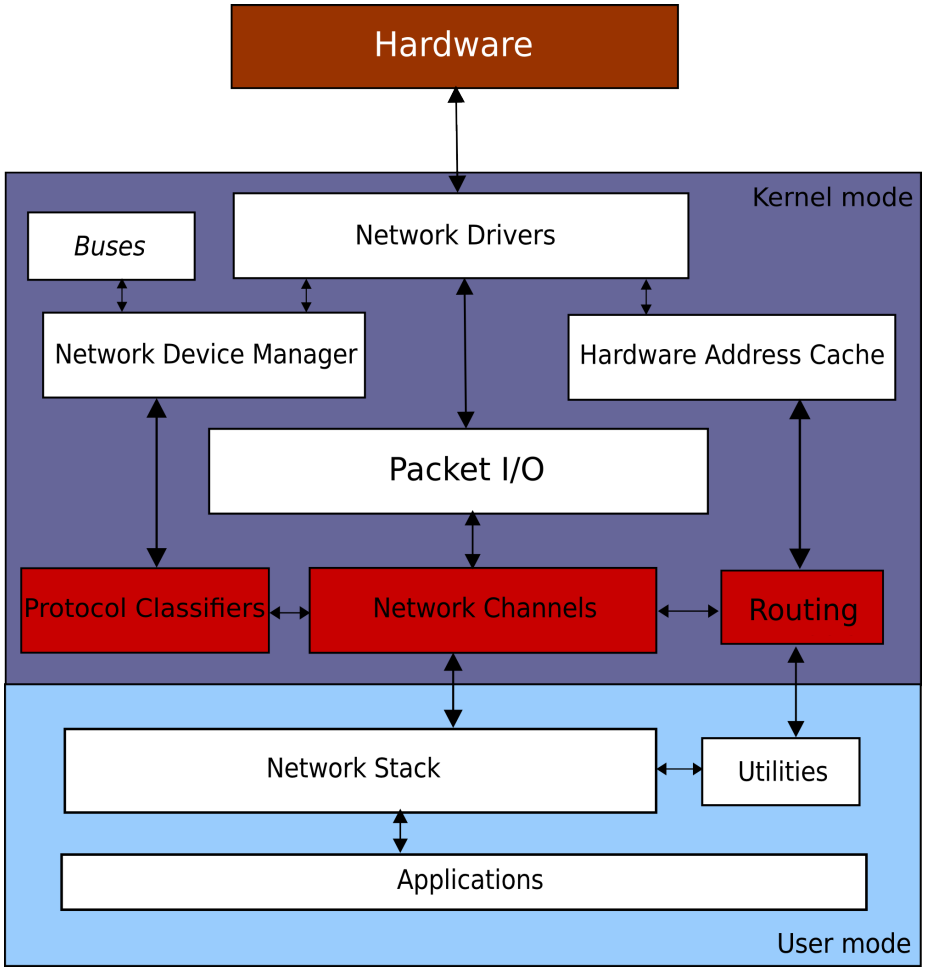

Figure 3.1: The network channel layer interfaces between the lower-level networking code (including the packet I/O code) and user-space code. The network channel layer structures packet I/O, routes outgoing packets to the appropriate network interface and classifies incoming packets.

- **Packet classification.** Incoming packets must be inspected, in a protocol-specific manner, to match them with a channel. For example, incoming UDP packets are matched with a channel that has the same source port; since only one IPv4 channel can map to a (IPv4) port, it must be destined for that channel. Because classification is performed in a interrupt context, the process is as efficient as possible ensure a responsive system.

- **Routing.** The userspace network stack has no knowledge of the routing table. Outgoing packets have their source address applied by the kernel after signalling a packet transmit via `SysWrite`; the protocol-specific code then updates any other dependent fields, such as the packet's checksum, and transmits the packet. The routing layer is updated by the `dhcp` utility as part of its initial configuration, and it calls the lower layers to map network addresses to hardware addresses.

- **Firewall.** A basic stateless firewall, or *packet filter*, has been implemented as part of the packet classification code. Support for filtering incoming and outgoing TCP and UDP packets by properties such as protocol, destination or source port, and packet direction (or a combination) has been added to a generic firewall layer that barely affect the throughput and latency of the networking subsystem.

- **Usercode library.**. Because network channels share knowledge of data structures between the kernel and the user, we supply a small usercode library that abstracts the lowest layer of structure manipulation (both for the channel in general and individual packets) into a user-space library.

  This kernel code is made available via a user-accessible shared page; the network stack on startup finds this code via an in-memory directory and links a set of function pointers (representing operations such as `ChanSendBuffAlloc`) to the export table, so that the channel data structures can be updated in a manner invisible to the application. This allows for greater maintainability and flexibility in the kernel implementation.

3.3 Channel management

Before we can send and receive packets on network interfaces, we need to specify to the kernel the *address family* (in this implementation, only IPv4 is supported), *protocol* (TCP, UDP or ICMP) and *source and destination addresses* we will be dealing with. This is to ensure security for outgoing packets and also to give the kernel enough context to classify incoming packets to that channel. As a result, it means the network channel truly is a bidirectional interface for data communication.

In channel creation, we can also specify several other options, such as retaining the ability to *manage deallocation of sent packets* (important for a connection-oriented protocol that retransmits lost or corrupt packets, such as TCP) or to later *bind to a certain interface* (useful for `dhcp`, which initializes a particular interface). The exact number of send and receive buffers needed can also be specified.

This information is used to set up the network channel generically, as well as any protocol-specific code that may be interested in the source and destination addresses.[7]. The exact manner depends on the sub-protocol (such as TCP or UDP) and what purpose the channel is intended to serve. For example, TCP master sockets, which create child sockets to accept connections, are allowed to accept TCP packets from any source on a certain port, but cannot then write to those sources without creating a channel with a more specific channel address.

After creating the channel, we can also manage channel-specific options through the `SysChannelControl`[8] system call interface. For instance, if we create a channel that supplies raw packets (functioning like a *raw socket*), like `dhcp`, which initializes a particular interface, we need to specify what interface we sending these packets out on. We can use the human-readable name of the interface to do this, and the channel will then be *bound* to this interface. (Appendix B.2)

After creation and channel-specific control calls, the rest of the channel management interface comprises the **standard filesystem calls**, bearing in mind the network channel behaves like a *character device* in UNIX.[9] In particular, closing the channel and releasing the associated resources is achieved by calling `SysClose`. The file descriptor referring to the channel can be passed to new processes via `SysCreateProcess`, duplicated (`SysFileDup`) or controlled using `SysFileControl`.

How exactly channels are internally organized by the kernel depends on the channel's address family and protocol; different protocols use their protocol-specific source and destination address to arrange channels for the most efficient access. However, the method of **channel organization** is generic; incoming packets are hashed and matched in a hash table; this is common to all address families and protocols. The exact mechanism is described in Section 3.3.3.

3.3.1 Setup

As part of socket creation or connection in the userspace network stack, we call `SysChannelCreate`, specifying the address family and protocol, source and destination addresses and optional features of the channel. (See Appendix B.1 for the exact parameters). After validating the parameters, the kernel then inspects the specified channel address family and protocol type. The address family is used to determine the protocol-specific channel operations for the channel, in the form of the `ChannelOps` structure (Figure 3.3).

Instances of the ChannelOps structure are registered by protocol-specific code via the `ChanRegisterFamily` call, which links a address family, identified by an integer, to a particular instance of the structure. This structure is then used to call

[7]For example, IPv4 assigns a temporary source port, known as an *ephemeral port*, if the source port of the channel is set to zero.

[8]The exact description of the two channel-specific system calls that comprise part of the channel management interface can be found in Appendix B.

[9]There are two types of device in UNIX-like operating systems: *block devices*, which are, able to read at any point in the file using `SysSeek` and represent a repository of bytes, and *character devices*, which represent a stream of bytes (in our case, packets!), and so, as a result, are cannot be read from at a random location; this affects other operations such as synchronization.

```
struct ChannelOps
{
    int (*create)(struct Channel* channel, struct ChannelHead** head,
        struct
            ChannelOptions* options);
    int (*control)(struct Channel* channel, int code, void* data);
    int (*write)(struct Channel* channel, struct NetSendBuffer* buffer);
    int (*recvBuffer)(struct NetBuffer* buffer);
    int (*close)(struct Channel* channel);
};
```

Figure 3.2: The `ChannelOps` structure is used by protocol-specific components, such as the IPv4 code, to present a generic interface to the general channel layer code. The operations are used in general channel management and packet transmission and receiving.

the protocol-specific `create`, which verifies the source and destination address (updating them if necessary), and if possible, assigns the channel to a particular network interface, updating the `ChannelInfo` structure described in the next section with values for the hardware header bytes needed and maximum packet size permitted to send packets out over the interface.[10] Finally, the channel is assigned to a list (represented by the `head` parameter in `create`) for internal organization, using information from the source and destination addresses.

Memory and the VFS

At this point, the channel has been assigned to a particular list of channels, the source and destination addresses have been verified and modified, and if possible, the channel has been assigned to an interface. We still need to initialize the channel's memory, which comprises an array of buffers, complete with a general channel header and packet headers. The exact setup is described later in Section 3.4, but in brief, the memory setup operation (`ChannelMemorySetup` in `net/channels/memory.c`) calculates the number of pages needed to store the number of send and receive buffers (the exact number may be specified in the `options` parameter of `SysChannelCreate`), which is a function of the maximum packet size (determined by the interface or interfaces that the packet can be sent and received on), the number of buffers and the hardware header size.

Once we have calculated the number of pages needed for the send and receive buffers, we can allocate a list of page pointers and set up the general channel header structure. At this point, applications can now interact with the network channel's memory correctly. After **memory initialization**, the last operation to perform is binding the channel to a file descriptor using the generic *virtual filesystem* (VFS) infrastructure; the channel layer supplies a list of file operations it can implement, and the VFS returns an integer corresponding to an index in the file descriptor table for use in later file operations. This file descriptor value is returned to the calling application in userspace – channel setup is now complete, and the

[10] See Section 3.6 for the exact rules involved in assigning IPv4 channels to network interfaces.

application can now send and receive packets on it.

3.3.2 Control

It may not be efficient or possible to supply all the needed optional features for a channel at channel creation. For instance, a suitably-privileged application may want the network channel to ignore addresses (for raw network access) so it can specify a binding to an interface, but then later use the routing functions in the kernel when raw network access is no longer needed. This can be achieved with the following series of calls to `SysChannelControl`, (which is further described in Appendix B.2):

```
...
SysChannelControl(channelFd, CHANNEL_GET_FLAGS, &flags);
flags |= CHANNEL_IGNORE_ADDRESSES;
SysChannelControl(channelFd, CHANNEL_SET_FLAGS, &flags);
SysChannelControl(channelFd, CHANNEL_SET_INTERFACE, "Ethernet0");
.. (send raw packets) ..
flags &= ~CHANNEL_IGNORE_ADDRESSES;
SysChannelControl(channelFd, CHANNEL_SET_FLAGS, &flags);
```

At the moment, there are only a few options that can be controlled by the user; this is a testament to the generality of network channels as well as the fact that most connection state is stored in userspace. In constrast, Linux provides three system calls for controlling kernel socket operation, `ioctl`, `getsockopt` and `setsockopt`; in Whitix, virtually all of the socket options are handled in by the userspace network stack without complex demultiplexing of `code` arguments in the kernel. Other options will be added when needed, but the options currently available (see Appendix B.2) seem sufficient for virtually all applications and utilities.

3.3.3 Organization

There are two goals to be achieved in channel organization:

- **Efficient organization**. The exact arrangement of channels is up to the address family, and is often dependent on the matching criteria and information available. This usually differs between protocols in a family. The IPv4 uses a hash table per protocol, with the protocol determining the exact inputs to the hash function. (Section 3.6)

- **Generalized lookup**. It is useful to make the lookup and organization code as generalized as possible, to avoid code duplication and subtle bugs. Therefore, the high level operations, such as *adding* and *removing* channels should be a general mechanism, with the protocol-specific code providing the *placement policy* with the most efficient organization possible as well as matching code, as stated above.

To achieve this, during channel setup, the protocol-specific `create` function returns a list, represented by a `ChannelHead` structure, to attach to. The protocol classifier code supplies the same `ChannelHead` pointer to the generic matching function `ChannelSearchList` (`include/net/channels.h`) when the `recvBuffer` operation is called (see Figure 3.3), along with a callback to match against any channels found. When the channel is closed, it is removed from the list without calling any protocol-specific code.

3.3.4 Destruction

When the channel is no longer used by the application, it may be freed to save memory and processing by calling `SysClose`, passing the file descriptor. Unlike closing a kernel-level socket, closing a channel does not mean any close messages are sent.[11]. In a connection-oriented protocol, userspace code should send the appropriate connection termination messages (along with the usual application-level protocol close handshaking) before closing the channel.

Since the channel's context is kept mostly in a `Channel` structure instance, a pointer to which is kept in the `File` structure, a channel close involves removing from a `ChannelHead` list and freeing the `Channel` structure, with a short in-memory protocol-dependent close procedure, which may include freeing any ports allocated to the channel.

3.4 Memory management

The exchange of packets in a network channel is achieved via a **shared memory-mapped area** between userspace and the kernel, divided into two main regions: *send* and *receive*, each with a different number of buffers, header structure and allocation map. The core of the network channel memory management involves a tight coupling between kernel-space and user-space, with both the kernel and the userspace application updating the packets and allocation maps as data is sent and received using the network channel.

Because of this interaction between kernel and user-space, the implementation *relies on correctly functioning user-space code to work correctly*. This is not an ideal situation; a good rule of thumb is to assume that any user-space code is incorrect and unreliable by nature. However, if the kernel can detect any errors made or inconsistencies created by user-space, then we recover as quickly as possible or let the application know in as detailed a fashion as possible.

It is also undesirable for user-space libraries to have knowledge of the internal channel data structures involved, and potentially encountering issues about keeping the kernel and user code in synchronization, especially with a less tightly-coupled development process than at the moment. To resolve this, most of the low-level accesses to shared data structures is provided by a **user-code library** that the kernel makes available via a dynamic mechanism (Section 3.4.2).

[11] One example of a connection termination protocol is the FIN handshake of TCP that is initiated when either host in a connection signals a socket close.

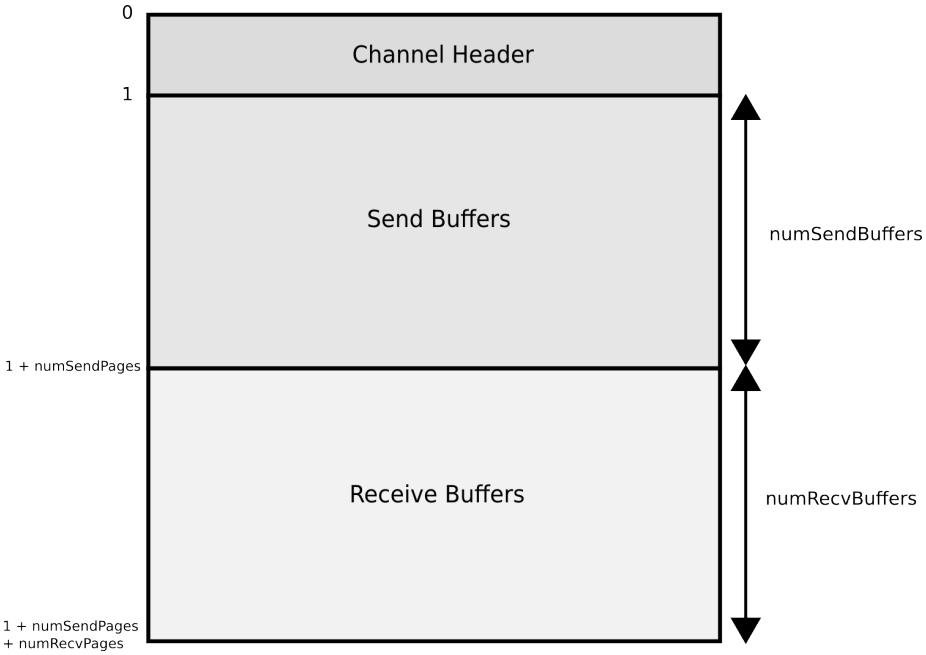

Figure 3.3: Layout of the network channel mapped contiguously in virtual memory. The left side of the diagram shows the number of pages used for each region, with the number of buffers in each region shown on the right-hand side. Note that send pages are mapped with an unspanned organization (to avoid sending a buffer on two discontiguous pages), but receive pages are mapped with a spanned organization.

The core motivation of the network channel design is the avoidance of costly packet copying where possible; we follow a **zero-copy philosophy**. The zero-copy philosophy in this case states "place the data where it is needed to begin with", as a copy essentially states the data was not in the desired place to begin with. With the correct design, placing the data in the correct place is perfectly possible to achieve.

3.4.1 Memory layout

Figure 3.4.1 shows the memory layout of a fully memory-mapped network channel (from the viewpoint of userspace). It can be divided into three regions:

- **Channel header** The channel header stores the channel information structure (Section 3.4.1), the identification number of the last received packet (updated by the usercode library and used when polling the channel), and two allocation maps, one each for send and receive.

- **Send packets** This is a static array of sent or soon-to-be-sent packets. Each packet comprises send headers and their accompanying buffers, which is the

```
struct ChanUserHeader
{
    int magic;
    WORD recvId;
    BYTE sendAllocMap[(CHAN_MAX_SEND_BUFFERS+7)/8];
    BYTE recvAllocMap[(CHAN_MAX_RECV_BUFFERS+7)/8];
    struct ChannelInfo info;
};
```

Figure 3.4: The general header of the channel, placed in the first page of the shared memory. It contains the information structure and the memory allocation bitmaps for the two memory regions of the channel. The `magic` field could be used by the usercode library to verify the pointer to the various functions points to a `ChanUserHeader` structure. The role of `recvId` when receiving packets is described in Section **??**

maximum length that allows the packet to be transmitted over a certain interface or all possible interfaces (depending on the semantics of the data protocol). Send buffers follow an *unspanned* organization, as network cards may be unable to transmit buffers that span two virtual pages, where the pages may correspond to non-contiguous physical pages.

- **Receive packets** This is a static array of received packets. It has a similar structure to the send packets: one packet comprises the receive header and its buffer. Each buffer is the maximum packet length that can be received on a certain interface or all possible interfaces. Receive buffers do follow a *spanned* organization, as the incoming packet is copied into the receive region in software by the kernel and so can span two pages.

Channel header

The channel header structure is used for memory management of the channel's entire shared memory area. In the usercode library, the `sendAllocMap` and `recvAllocMap` are updated using bit manipulation functions such as `BitTestAndSet` and `BitClear`, to allocate and free buffers respectively in the various usercode library functions. The structure itself is also updated by the kernel, and remains opaque to user applications. The application can retrieve a pointer to the `ChannelInfo` structure by calling `uChanGetInfo`.

Our approach to allocation differs from both Jacobson's and Polyakov's implementations. In those, the network channel is a cache-aware, cache-friendly queue, with a circular buffer approach. Incoming packets, if we take receive for example, are placed at the tail of the queue, with any waiters being notified if there is data between the head and the tail. They both presumably employ a copy between userspace and kernel (for user-facing network channels), since the network channel consists of a list of pointers to buffers, whereas the Whitix implementation is a list of buffers in the channel itself. The queue (Jacobson and Polyakov) and allocation bitmap (Whitix) have similar cache and locking properties, but lookup in the case where buffers may persist in memory is much faster with bitmaps; in

```
static inline void net_channel_queue(net_channel_t *chan, uint32_t item) {
        uint16_t tail = chan->p.tail;
        uint16_t nxt = (tail + 1) % NET_CHANNEL_Q_ENTRIES;
        if (nxt != chan->c.head) {
                chan->q[tail] = item;
                STORE_BARRIER;
                chan->p.tail = nxt;
                if (chan->p.wakecnt != chan->c.wakecnt) {
                        ++chan->p.wakecnt;
                        net_chan_wakeup(chan);
                }
        }
}
```

Figure 3.5: Van Jacobson's network channel, as a queue implementation. This works well for a producer-consumer relationship. However, if we would like some packets to persist (say, for retransmission), iterating through the queue to find the first free buffer becomes less effective. There is still a shared cache line if the channel is accessed by two threads – the `q`. Whitix's allocation bitmaps optimize lookup for allocation, deallocation and location of the earliest unread packet for receives.

many processor architectures, finding the first free bit involves no loops and very few processor instructions.

ChannelInfo

As well as sharing memory between the kernel and userspace, the network channel also shares a structure describing the layout and configuration of the channel in the `ChannelInfo` structure. This is used by the usercode library, as well as the network stack library, to construct packets. The structure contains the members shown in Figure 3.4.1.

This is the key mechanism for exposing information about the channel to the

```
struct ChannelInfo
{
    unsigned short numSendBuffers, numSendPages;
    unsigned short numRecvBuffers, numRecvPages;
    unsigned short sendHeaderBytes, recvHeaderBytes;
    unsigned short sendPacketsPerPage;
    unsigned short hwHeaderBytes;
    unsigned short maxPacketSize;
};
```

Figure 3.6: The `ChannelInfo` structure, placed in the first page of the shared channel memory. Unlike the other structures shared between the kernel and userspace, it follows a known standard format and the internals are available to applications (who use it to construct and parse packets).

```
struct ChanSendHeader
{
    unsigned int magic;
    unsigned short length;
    unsigned short indexFlags;
    void* priv;
    struct Time time; /* transmitted */
};
```

Figure 3.7: The channel buffer send header. This is placed before the data of each send buffer. A transmitted buffer has a different set of properties to a receive buffer; for example, the time field is used to note the system time at which the packet was transmitted (for round-trip time estimates). Unlike receive buffers, send buffers are automatically freed when sent, unless the `CHANNEL_KEEP_SEND_BUFFERS` flag is specified upon channel creation (Appendix B.1).

general application. For example, to calculate the local *maximum segment size*[12] in TCP, we must have an idea of the *maximum packet size* (`maxPacketSize`, which excludes `hwHeaderBytes`) that the channel's interface can support. Other fields of the channel structure are generally used by the usercode library; `uChanRecvBuffData` uses the `recvHeaderBytes` field to return a pointer to the start of the received packet's data, given a pointer to the start of the `ChanRecvBuffer`.

Transmission semantics

To send a packet, the application (usually via a channel layer in a userspace library) executes the following steps:

1. The application allocates a send buffer by calling `ChanSendBuffAlloc` in the usercode library.
   (a) If there are free send buffers, the function is successful and sets the appropriate bit in the send allocation map, returning a (opaque to the application) `ChanSendHeader` pointer to the calling code.
   (b) If there are no free packets, the function is unsuccessful and a NULL pointer is returned; the application must wait for send buffers to be freed by the kernel, or, if it has chosen to manage it its own send buffers, free the buffers in the allocation map itself.
2. The application constructs the packet at the network layer and above (e.g. IPv4 and either TCP, UDP or ICMP), with an offset of `sendHeaderBytes`. (reserved for the data link layer header and the buffer header itself) Checksum fields may be left to the kernel protocol-specific code to fill in, but this depends on the protocol code.

[12]The *maximum segment size* is the largest packet that can be received by a host in one unfragmented piece, excluding the size of the TCP or IP header.

```
struct ChanRecvHeader
{
    unsigned int magic;
    unsigned short read;
    unsigned short id;
    unsigned short length;
    unsigned short indexFlags;
    void* priv;
};
```

Figure 3.8: The channel buffer receive header. This is placed before the data of each receive buffer. A received buffer mainly stores application information about the amount of processing the buffer has gone through. `read` is updated by the application to note how many bytes of the buffer have been processed, and `id` is used to indicate the place of the buffer relative to others in the incoming stream of data.

3. The application sets the length of the buffer and calls the `SysWrite` system call, passing the address of the start of the buffer (which is the send buffer header) and the buffer length (including `sendHeaderBytes`).
4. The `SysWrite` system call returns, indicating how many bytes were successfully transmitted.
5. By default, the buffer is freed when transmitted by the kernel. If the application has chosen to manage the send buffers for that application (for example, to build a retransmission list such as in TCP), the buffer can be later freed by the application by calling `uChanSendBuffFree` in the usercode library.

Receive semantics

Receiving a packet on the network channel can be an asynchronous or synchronous process, as well as blocking or non-blocking. The implementation details are left to the application. The most common kind of receive, a synchronous blocking receive, is as follows:

1. (optional) The application calls `SysPoll`, passing the channel file descriptor and the required event (`POLL_IN`) as a `PollItem`, along with an optional timeout which can be used to implement the receive timeouts familiar in user-level sockets.
   (a) If a packet was received before `SysPoll` returns, the kernel will have copied the packet into a free receive buffer, incremented the `latestRecv` field of the in-kernel `Channel` structure, and signalled that data was available to read in the channel by setting `POLL_IN` in the `revents` field. `SysPoll` returns a positive number to signal the number of file descriptors with events, and the receive process continues.
   (b) If no packet was received, no `POLL_IN` events are signalled for the channel file descriptor, and the `SysPoll` system call returns zero to

indicate no events happened on the file descriptor. The application can continue polling the channel, or perform other work in the meantime.

2. The application then calls the usercode library function `uChanRecvBuffGet` to retrieve the (opaque pointer) `ChanRecvBuffer`. If the operation could be performed in the context of multiple threads, the appropriate synchronization is left to the application – the function may return NULL if another thread has already acknowledged receipt of the buffer. It may also return NULL if `SysPoll` was not called beforehand to ensure waiting data was present.

   (a) `uChanRecvBuffGet` acknowledges receipt of the packet. It is a simple operation in the usercode library with a simple general fast case (where only several buffers are allocated in the receive buffer). The allocation map is scanned for any allocated buffers, and those allocated buffers are then inspected to see if they have a newer ID than the current `recvId` value in `ChannelInfo`. If it is a new buffer, it is returned to the application.

3. The data can then be inspected by calling `uChanRecvBuffData`[13] – the receive buffer remains allocated until the application has processed all the data and called `uChanRecvBuffFree`.

3.4.2 Usercode library

Because the internal structure of all the headers mentioned in the section may change (in the future, the arrangement of fields in `ChanSendBuffer` and `ChanRecvBuffer` might depend on cache line size, word alignment, or the processor's word size), it is preferable that we abstract the lowest layer of structure manipulation so that we can update the structures without worrying about backwards compatibility.

The kernel provides a small **usercode library** used for manipulating the packet and channel headers, as well as allocating and deallocating packets. The code itself is in `net/channels/userlib.c`, and is copied to a nominal high userspace-accessible address into each application along with a directory of pointers to various function pointer tables; these tables comprise a NULL terminated list of function pointers, in a prearranged order (not unlike system calls) for applications to call. The `userlib` mechanism is available to all parts of the kernel in `lib/userlib.c`.

The function pointer tables are used in the user-space network library, where they are copied from kernel space when the shared library is first loaded (via the `init` function available to shared libraries). General user-space code can then call these functions via a small *inline wrapper* for each function, but generally the internal network stack code calls these functions through the channel layer, which further abstracts channel operation into *high-level functions* (such as reading a packet from a channel in `ChanRecvNb` and polling a channel, described in the next chapter). Many of these usercode functions are "getters" and "setters", and they provide

[13] This pointer generally points to the start of the buffer data – there is no link-layer header present in received packets as such information is useless without more context

an abstraction layer for channel structures to allow for maximum flexibility in the kernel implementation.

For a list of the functions available in the usercode library, see Appendix C.

3.5 File emulation

Even though a network channel does certainly not appear to have the properties of a traditional file, and since `SysChannelCreate` (Appendix B.1) returns a file descriptor for use in file operations, we must implement a number of common file operations to satisfy the semantics expected by user applications from any file descriptor. However, some of these operations are not implemented, because the alternative is faster and therefore should be used by libraries and applications. The file operations implemented in `net/channels/file.c` so far include:

- **Writing.** (special semantics) The `SysWrite` system call is a signal to the channel layer that the application wants the buffer to be transmitted. The application, perhaps via a userspace shared library, constructs the buffer as described in Section 3.4.1 and passes the address of the buffer (and header) and the total length of the packet (including the bytes of the send header). To avoid the corruption of packets, `ChanWrite` verifies the magic number and length before assigning a `NetSendBuffer` and passing the buffer to protocol-specific code (such as `Ipv4Write`) for transmission – as a result, only correctly formatted `ChanSendHeaders` can be transmitted.

- **Polling.** Polling is the main mechanism, instead of blocking reads, for waiting for data from a network channel. The channel can always send out data (so long as it has enough free buffers), and waiting data is signalled by a change in the in-kernel `Channel` structure of the value of the `latestRecv` ID. (which is incremented whenever a packet is received by a channel)

- **Handle duplication.** This is handled automatically by `SysFileDup`. However, it does raise interesting questions about duplicating channels across processes – by passing a file descriptor to `SysCreateProcess`, it is automatically duplicated and given to the new process. Given this, how can userspace network stacks transfer whatever wraps around the channel (such as the TCP socket state) with the duplicated file handle? This is an open question that needs to be addressed.

- **Closing.** When `SysClose` is called, the channel destruction process begins in `ChanClose`. The protocol-specific channel resources are then freed followed by the channel itself, as described in Section 3.3.4.

- **Memory mapping.** Since the channel's memory is faulted in when needed, we implement the `ChanMapNoPage` virtual memory operation. This maps in a zero page into an appropriate place in the channel's memory (updating the internal array of `pages`, a list of addresses of kernel accessible network channel pages). The page then formatted by `ChanMapNoPage` to become

part of either the general channel header or send region, or, if the accessed page is part of the receive region, the page is left zeroed by `ChanMapNoPage` – the `ChannelCopyPacket` formats receive buffer headers before data is copied in.

I have chosen not to implement directly **reading from the file** (by calling `SysRead`), because of the superior alternative of, if we wish to block, polling the channel for data and then reading from userspace as described in Section 3.4.1, or, if we wish to perform a non-blocking read, scanning the packets in the channel for a newer ID than the last packet read using `uChanRecvBuffGet`. Using the channel library involves at most one user-kernel transition, compared to a `SysRead`, which may involve multiple system calls if the specified buffer is smaller than the maximum packet size. It also simplifies the in-kernel channel implementation, as we do not have to consider specifying timeouts for reads as with in-kernel sockets.

3.6 Packet classification

When packets arrive in the network channel, the only context associated with the data is the length of the packet and the device it was received on. To efficiently match packets with channels, we need to solve the **protocol demultiplexing** problem associated with moving the userspace network stack out of the kernel. This is done in the network channel layer, in the kernel, in two stages: (1) matching the packet to an protocol family in `NetRecv`, and (2) in the protocol family-specific code, calling a small protocol-specific matching function using extra context (such as the *protocol* field in the IP header)[14] for transport-layer protocols such as TCP, UDP and ICMP, along with a generic IP layer.

The only address family that has been implemented so far is the IPv4 suite of protocols, which currently also includes other lower-level IP-related protocols such ARP. An flowchart of the various steps taken when receiving a IPv4 packet is illustrated in Figure 3.9. The process is described in depth below:

3.6.1 Family matching

When the device driver receives a packet, it calls `NetRecv`, often through `eth_recv` or similar in the device driver layer. At this stage, we have a packet buffer and a set of address family operations (an array of `ChannelOps` structure – see Figure 3.4.1 for the layout), and we must first match the incoming packet with the correct address family.

To simplify the process, we can work from the assumption that network interfaces will generally receive packets from the same address family[15] – it then follows

[14] The protocol field is an 8-bit integer ID describing the protocol the IP packet encapsulates; for instance, TCP is given value 6, and UDP 17. This is useful in matching packets to protocol-specific handlers via an function pointer table indexed by the ID.

[15] This is an assumption that does not always hold. On the local area network used for testing, packets from both IPv4 and IPv6, which can be considered to be different address families, were received. Obviously there were more IPv4 packets than IPv6, but this assumption may be under question in the

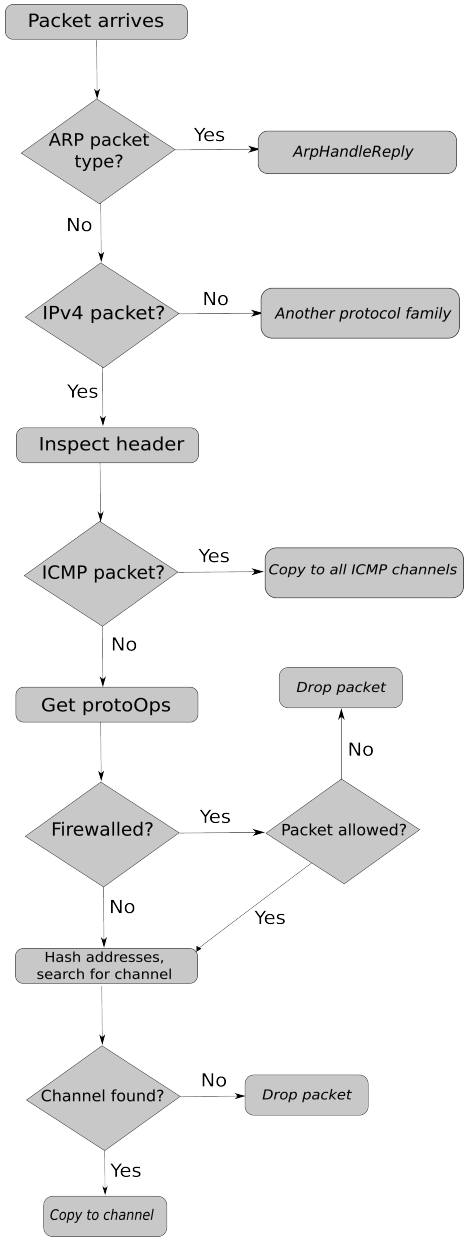

Figure 3.9: A flowchart of receiving a IP packet, as detailed in Section 3.6, and matching it with a channel. Processing of the packet takes place in an application-specific manner. Compare with the process for processing an IP packet in Linux, described in Section 1.1.2.

that the address family of the current received packet is likely to be the same as the last packet.

Working from this assumption, we store a pointer in the `NetDevice` structure to the `ChannelOps` structure of the address family that successfully handled the last packet. The `recvBuffer` method is used to handle the incoming packet (and hopefully match it to the channel), and it will return `CHAN_HANDLED` if matched, and `CHAN_NOT_HANDLED` if it becomes apparent the incoming packet does not match the address family.

In the case where the `recvBuffer` operation does not handle the packet, or where it is the first received packet on an interface, we also try the `recvBuffer` of *other address families*. If one of them matches, we (not always correctly) predict the next packet will belong to the same address family. However, as there is only one address family supported (IPv4), this infrastructure is only useful when more address families (like IPv6) are implemented. If no address family can successfully recognise the packet, it is silently dropped by returning from `NetRecv`.

3.6.2 Protocols: Matching packets to channels

Once we have matched the incoming packet to the address family in `recvBuffer` (if the packet does not belong to the address family, the function returns with a `CHAN_NOT_HANDLED` status), we can start matching the channel to a particular channel. To take the IPv4 address family as an example, the following steps in `Ipv4RecvBuffer` (depicted in Figure 3.9) are performed to match the incoming packet to a protocol and channel:

1. The packet type is translated from the data-link layer header – in our case it is the *type* field in the Ethernet header, and inspected by `Ipv4RecvBuffer`. If it is an ARP packet, the IPv4 code lets the *hardware address cache* (Section 2.5) handle the packet. `Ipv4RecvBuffer` then returns, signalling it has handled the packet.

2. Having handled any non-IP packets (i.e. ARP) in special cases, we then return if the packet is not marked as an IP packet.

3. We can now inspect the packet to verify it is an IPv4 packet, and retrieve the source and destination addresses (along with port numbers for TCP and UDP) from the IP header.

4. If it is an ICMP packet, this is handled as a special case and the packet is copied to all ICMP channels. ICMP channels essentially function like the raw sockets described earlier, because of a lack of port multiplexing in the protocol.

5. Using the protocol ID in the header, we find the protocol operations structure (`ProtoOps`) that corresponds to the packet's protocol (either TCP or UDP). If there is no operation structure for the protocol, then we cannot handle it.

future.

```
struct RouteEntry
{
    DWORD destination;
    DWORD gateway;
    DWORD mask;
    int flags;
    struct ListHead next;
    struct NetDevice* device;
};
```

Figure 3.10: The `RouteEntry` structure, similar in format to routing tables in other network stacks. The destination address of a packet is used to retrieve the corresponding `RouteEntry` in `RouteLookup`; if a gateway is present, the hardware address is set to that of the gateway's.

6. We use the protocol operation `fwInput` to verify that we are allowed to accept the packet. If we must drop it, we signal to `NetRecv` that we cannot handle the packet.

7. We then hash the source and destination address of the packet to locate the protocol-specific `ChannelHead` structure.

8. We then search the linked list represented by the returned `ChannelHead` structure using `ChannelSearchList`. There are two cases:

   (a) We have successfully matched the protocol to a channel (using the protocol-specific `compareFunc` passed to `ChannelSearchList`). If so, we copy the packet to the first free receive buffer in the channel and increment `latestRecv`. If there are no buffers, we drop the packet.

   (b) We could not find a channel to match the packet to. In this case, we call the `chanNotFound` function, which sends back a small RST packet (for TCP).

Although much of the implementation of the `recvBuffer` function is protocol-specific, the main steps apply to any protocol family: (1) inspection, (2) verification, (3) firewalling, (4) matching and (5) copying.

3.7 Routing

In the network channel layer, routing outgoing packets is the opposite problem of the classification of incoming packets. Given a channel and an outgoing packet, we must choose the network interface is used to transmit the packet (and to what hardware address). Contrast with classification, where we are given an incoming packet and its network interface, and must match it to a channel. Currently, the routing functionality is IPv4 specific and contained in `net/ipv4/route.c`, but it can be generalized to suit other protocol families.

Taking the IPv4 protocol family code as an example again, we inspect the *destination IP address* of the outgoing packet in `Ipv4Write` (unless we are in "raw channel" mode, such as the channels used by `dhcp`), and pass this to the core of the routing component: `RouteLookup`. This retrieves a `RouteEntry` (Figure 3.7), given an IP address. If the `gateway` field is non-zero, then the packet must be routed outside of the local network by sending it to the gateway. The implementation is similar to implementations in other networking subsystems, so it will not be covered in depth here.

The routing table is constructed with information from interface drivers (such as `Loopback0`, which knows packets sent to 127.0.0.0 to 127.255.255.255 should be routed to it) and the `dhcp` utility, which uses dynamic host configuration information from the local network to build the routing table. A routing utility may be developed in future that would also allow the addition, listing and removal of table entries, but only addition of entries via IcFs (`/Network/Routes/addRoute`) is supported at the moment.

3.7.1 Firewall

The Whitix stateless firewall, a simple *packet filtering* mechanism, also inspects the packet before it is routed to an interface (as well as after a packet has been received on the interface and sent to a protocol family). It was included to show that such filtering and security is still possible, even if most protocol processing takes place in userspace. Currently, firewalling is supported for the UDP and TCP transport protocols. The core functionality is the `FwRuleMatch` function, which, given a rule and packet context (currently the protocol, source and destination port), returns `FW_ACCEPT`, meaning protocol processing can continue, or `FW_DROP`, where the packet must be not be processed further (matched to a channel or sent on an interface). `FwRuleMatch` is generally called from protocol-specific code, either on input or output.

The abstraction of a firewall rule is the `IpFwRule` structure, which is translated from a `IpFwUserRule` structure passed to the kernel via the IcFs interface (at `/Network/Firewall/`). There is no support for ordering rules by generality, so a general rule to drop all input packets, if inserted first, will always take precedence. There is also no support for listing or removing rules, but this can be trivially added. See the description of the user application `firewall` (Section 4.5.1) for a further description of the firewall rule format and the available filtering mechanisms.

3.8 Testing

The main test framework used to test the network channel layer involved automated testing with the channel system calls using the `chan_test` test suite.The test suite could be described as *grey box testing* of the layer; we have knowledge of the internal data structures and algorithms when designing the test suite, but we test at the userspace level, just like typical users of the API.

```
Test set #1: Channel creation
=============================
Test #1.1: Invalid parameters to ChannelCreate ... [passed]
Test #1.2: UDP: Zero IP address as destination ... [passed]
Test #1.3: Invalid number of send and receive pages for channel ... [
    passed]

Test set #2: ChanSendBuffer: memory map
=======================================
Test #2.1: Create valid UDP network channel ... [passed]
        numSendBuffers > 0 ... [passed]
        numRecvBuffers > 0 ... [passed]
        numSendPages > 0 ... [passed]
        numRecvPages > 0 ... [passed]
Test #2.2: Allocate one send buffer ... [passed]
        correct address ... [passed]
Test #2.3: Freeing buffer then allocating again returns same buffer ... [
    passed]
Test #2.4: Allocating all (32) buffers ... [passed]
        Verifying buffer allocation fails afterwards ... [passed]
...
```

Figure 3.11: Part of the output of the `chan_test` program. The test suite tests a range of conditions, using the channel system calls and the usercode library to verify the channel implementation works correctly. Although it is not yet an exhaustive test, the test suite covers as much of the channel code as possible, with a focus on IPv4 channels.

Figure 3.11 depicts the part of the output of the test suite. The test suite is divided into sets, covering tests involving creation, send and receive buffer mapping, sending and receiving packets, and covering many of the channel library calls as well. The tests also cover the IPv4 protocol family code, but the tests can be easily reused across multiple protocol families.

`chan_test` was useful in locating bugs. One example involved the *send region allocation map*. The use of the bit operation `BitTestAndSet` function was present in `uChanSendBufferAlloc`, a usercode library function. Testing the creation of channels where the number of send buffers was larger than 32 caused the test to fail – `uChanSendBufferAlloc` seemingly failed to allocate all the send buffers in the channel. Using this test, I discovered that the processor instruction that the `BitTestAndSet` function was using, `bts`, only accepted a bit argument of 0 to 31, as the byte-sized argument was taken modulo 32. The automated test allowed me to identify the issue with ease (after all, the issue was located in the few lines of code in `uChanSendBufferAlloc`), which would have been hard to find later when sending packets using the userspace network stack.

Although the automated test suite was the *dynamic testing* performed on the layer, I utilized *static testing* methods, such reviews and walkthroughs. I reviewed the core kernel code often, often with a focus on justifying each operation and walking through rare cases in functions such as `Ipv4ChannelCreate` and `ChanWrite` to focus on cases where errors were not detected or pointers were not validated.

An ideal testing aid, especially for the automated test suite, would have been a mock *null protocol family*, so I could isolate the generic channel layer and test with a variety of invalid inputs – this would be to make sure any invalid inputs were detected by the generic layer, rather than just one protocol family.

One area of testing I should have explored more was *destructive testing*. One example of such a test would have been setting the fields of a `ChanSendBuffer` header to invalid values (having knowledge of the internal structure) – there are only a few such tests, and those are simple destructive tests, such as setting the entire buffer to zero and attempting to send it by calling `SysWrite`. Most of the memory corruption issues (and the lack of checking) were caught during testing in the various application programs of the network stack, and *not* by the channel test suite first.

3.9 Discussion

Network channels are a useful basis for handling packet I/O in a protocol-transparent manner and helping to transfer work out of the kernel. They would also be useful in a kernel-based network stack, avoiding the cache-unfriendly linked lists prevalent in many current implementations. Aside from some concerns about race conditions, they are also reasonably secure, and it is difficult to undermine the verification procedure for outgoing packets. Channels, especially during transmission, are multiprocessor-friendly – all the work of transmitting the packet is performed on the same CPU for one, and there is only typically one shared cache line (the allocation maps) that is atomically updated by multiple processors. Incoming packets are classified efficiently into protocol families and protocols in a generic manner – further research would most likely replace the simplistic hash table in the IPv4 code with a lookup-friendly data structure such as a *trie* for further performance gains.

Even though we have shown that it is possible to avoid *time-to-check-to-time-of-use* security holes by verifying the packet immediately before transmission, the fact that we can theoretically send **attack packets** is not ideal. We could deny access to network channels by programs whose packets pass all verification bar the final one before transmission; a checksum mismatch on the packet indicating that either they are corrupting their packets maliciously or they are overwriting channel memory (in any case, making the two causes of the bug as obvious as possible is useful). Even though it is becoming increasingly hard to construct useful attack packets (and regardless, attackers can still do so using their own computer and raw sockets), we should protect against future attacks by including other mechanisms.

Limiting the **bandwidth usage** per network channel will be important on public systems. If we do not enforce resource usage limits, or *bandwidth* quotas, then one process could perform a denial-of-service attack against the system, denying other processes and channels the bandwidth resources needed to carry to operate. A *quota system* could be implemented that enforced limits on the amount of data processes or users (such functionality already exists in the Linux firewall `iptables`) could send using network channels. With kernel-based sockets, this is

not a problem, as the kernel's network stack controls the data output through the use of send windows (depending on the protocol) and socket pages in the kernel.

3.10 Summary

This chapter detailed the **network channel** – a new kernel object for handling packet I/O for high performance. We first evaluated the current work and implementations of network channels; where they were lacking and how their design was changed in this implementation. Other possibilities, such as *userspace network drivers* and *raw sockets*, were evaluated and discarded for security and performance reasons along the way. We described how **channel management**, via the new system calls `SysChannelCreate` and `SysChannelControl`, worked by representing **channels as files**, and depicted the standard **memory layout** of the channel, how transmission and receive operations interacted with the shared memory, and the design's scalability and performance advantages.

Packet classification, which takes place in the kernel and involves matching incoming packets to protocol families, protocols and channels is then outlined for IPv4 – including the in-kernel **packet filter** that filters the packets of most transport protocols. We summarize the **routing** that takes place in the channel layer, matching outgoing packets to interfaces. Finally, we demonstrate the **automated test suite** `chan_test` and how it helped discover bugs and improve stability in the new network channel layer.

Chapter 4

Userspace networking

The network channel objects described in the previous chapter are of limited use to most applications. We would like present the abstraction of connections and packets to applications. To do this, the userspace network stack provides support for a range of protocols from the TCP/IP suite, including implementations of TCP, UDP and ICMP and a DNS resolver. The userspace network stack provides an implementation of the transport and session layers to the application.

We then list the various utilities and applications written to demonstrate and complement the networking subsystem developed for this project, and note any applications that use unique features of the network stack, including the extensibility and adaptability described in Chapter 5.

Distributed with the operating system and the networking subsystem is a simple HTTP server, `httpd`, FTP and Telnet clients, `ftp` and `telnet`, and a name resolver `dns`. Networking utilities include `nprof`, which displays network statistics and profiling data collected from applications, `firewall`, for new packet filtering rules, as well as key utilities such as `dhcp` and `ping`.

4.1 Background

This project is not the first implementation of a userspace network stack that uses the concept of **network channels**, albeit in a different form, to prove that alternative network stack designs are possible. In 2006, Evgeniy Polyakov, who also wrote a patch-set to implement network channels in the Linux kernel mentioned in Section 3.1, wrote a small network stack, `unetstack`, focusing mainly on the TCP implementation,[1] as a small proof-of-concept.

Although Polyakov's implementation is fairly complete in terms of basic TCP features, including TCP timestamps, fast retransmit support and support for UDP

[1] The code is available here at http://www.ioremap.net/cgi-bin/gitweb.cgi?p=unetstack.git;a=tree

and TCP, the implementation is lacking in several areas, such as a general socket API (the userspace network stack is distributed as part of a test program, not as a shared library) or the correct socket semantics for server sockets (no new networks channels are created to handle server connections using `accept`). Nevertheless, I have based the design of my TCP state machine and TCP option handling upon that of Polyakov's, but many of the semantics, especially those relating to server sockets, are completely different. There is also little similarity in how the network stack interfaces with the channels, as well as a different approach to allocation (Section 3.4.1). I have also added support for other features not found in Polyakov's stack: retransmission timeout and estimation, and of course the adaptability, profiling and interactivity APIs (Chapter 5).

4.1.1 Microkernel research

`unetstack` is not the only network stack that has been placed in userspace. Other researchers have utilized different kernel objects to transfer the network stack into userspace – it is a valid research objective, as the network stack logic is especially vulnerable to outside attack, and most of the code involves manipulating data structures and therefore does not require privileged execution in a monolithic kernel.

One example is the 1995 paper "Experiences implementing a high performance TCP in user-space", by Aled Edwards and Steve Muir,[20] which developed a userspace network stack running on the **JetStream** *token-ring* network[2] – even back then, the performance of the userspace TCP implementation, by reducing the number of data copies per packet from two to one, increased by 50%.

The research before then on userspace network stack mainly involved microkernels, a hot topic in operating system research at that time, and two different implementations for the **Mach** microkernel were developed. The first, described in "Implementing Network Protocols at User Level" by Chandramohan Thekkath et al. (1993) [62], argues that consideration of development efficiency and application-protocol mismatches mandate a more distributed network stack, with the network stack moving to userspace and divided protocol libraries.

Thekkath et al. develop three components: a set of *transport protocol libraries* (UDP and TCP), a in-kernel *network I/O module* to provide protected access to the network by the libraries, and a *registry server* that allocates and deallocates communication endpoints. They face the same problems with demultiplexing of incoming packets and they even mention *transport protocol adaptability* to applications as a benefit, but do not mention an implementation. "Protocol service decomposition for high-performance networking", by Chris Maeda and Brian Bershad (1994), mention a similar design to [62], but move most of the non-performance-critical operations and protocol management code, such as ARP requests and replies, to a single userspace network server. The performance-critical code interacts with the network stack directly via protocol libraries.

In summary, the two research projects involving the Mach kernel match well

[2] A *token-ring* network uses a special data-link messages, known as a *token*, to control access to the medium.

with the flexibility and adaptability-related goals of the project, but suffered poor performance due to excessive data copying in many cases (due to the more distributed architecture) – they also did not provide convincing reasons (through their implementation) concerning why the network stack should be placed in userspace. In constrast, Polyakov's implementation will be similar to mine, mainly due to the fact that we are both adapting the network channel as a shared static array of buffers, but the scope of his project and implementation is much less and the goals and mechanisms used different.

4.2 Overview

The only kernel object directly exposed to userspace through a set of system calls is the network channel. Although the kernel supplies a user-code library and supports classifying a variety of protocols, the channel is intended as a low-level object for zero-copy I/O between the userspace and kernel (with a focus on network transmission), unlike a kernel-level socket, which represents one endpoint of a bidirectional communication link. In short, the network channel has much more general applicability to data transfer than the socket, at the expense of only providing a low-level interface to userspace.

The **userspace network stack** provides the main abstraction for application developers wishing to use networking in their application. It is still possible for userspace applications to use the raw channel system calls, but operation can be prone to errors (as described in the previous chapter). The userspace network stack comprises wrappers and abstractions around all levels of the TCP/IP protocol suite, including a simple userspace abstraction of the network channel for raw network channel access (used by `dhcp` for example), and providing support for UDP, ICMP and TCP to applications through two APIs: the *Whitix native API*, designed for adaptability and interactivity and the *POSIX API*, supported for UNIX applications that use the *Berkeley Sockets API*.

Networking utilities interact with the userspace network stack and other parts of the networking subsystem through the Information and Configuration filesystem (IcFs). One example is `firewall`, the Whitix packet filter, which processes rules specified by the user and adds and removes firewall rules using the IcFs interface in `/Network/Firewall/`, without interacting with the network stack (unless functionality to test the new firewall rule is added). These rules are immediately processed by the kernel firewall and added to the appropriate chain of rules.

Other utilities interact exclusively with the network stack. `ping`, used to verify that a host is reachable, constructs ICMP Echo Request messages using the network stack. `dns` verifies that a DNS hostname resolves correctly. `dhcp` uses raw network channels to assign an IP address to a network interface. `nprof` does not directly interact with any network stack component, but reads and synthesizes the output of the network stack from a run of a networked application.

Applications constitute the majority of the users of the network stack. Applications (and utilities[3]) link to the `libnetwork.so` userspace shared library, and

[3]The differentiation between utility and application is traditionally a characteristic of the program's

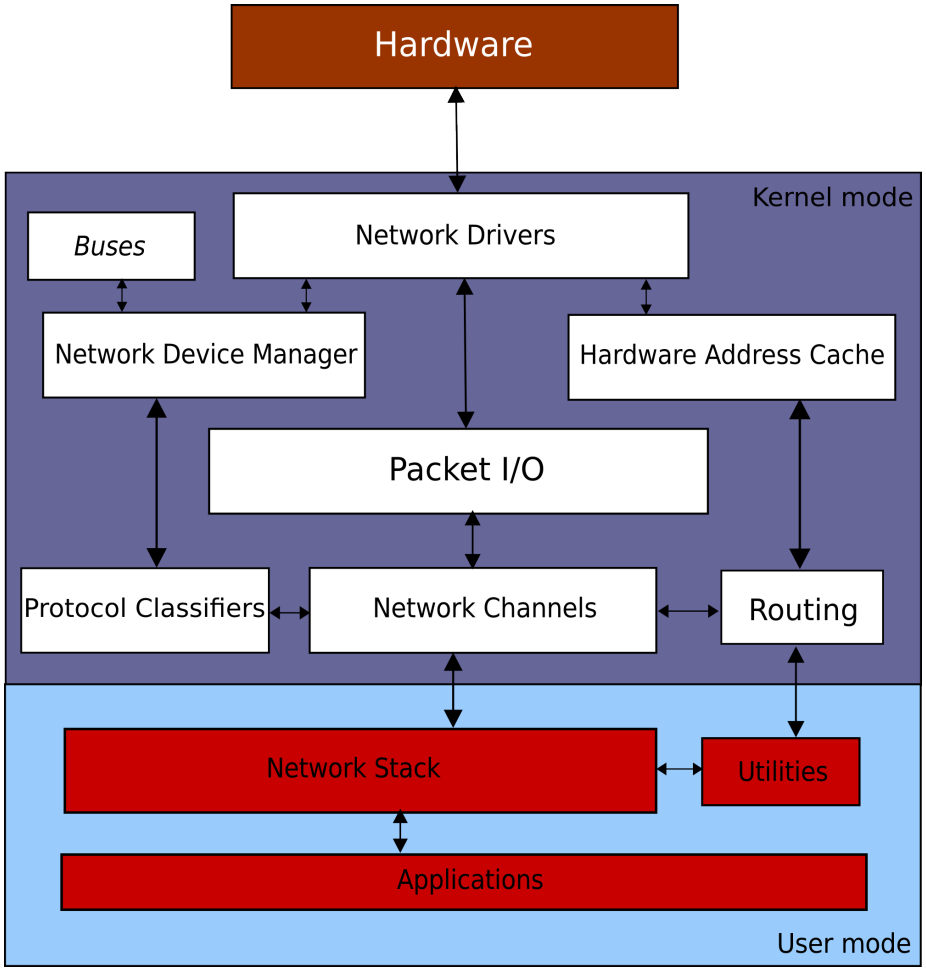

Figure 4.1: The network stack resides in userspace as a shared library and is the only component of the networking subsystem that applications interact with. Utilities interact with the network stack and kernel-level components such as the firewall and routing table (via the Information and Configuration filesystem).

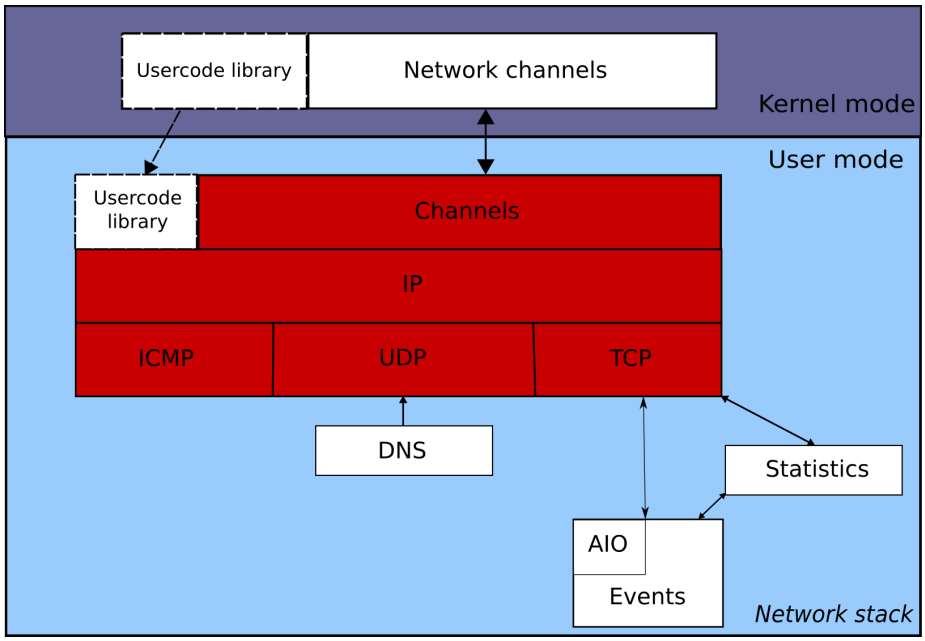

Figure 4.2: The userspace network stack architecture. The network stack is distributed as a shared library (`.so`). The diagram shows the usercode code library copied down to userspace – this is performed by the kernel and the code is in fact mapped into every process' address space. The statistics and events components are covered in Chapter 5.

then use call functions from of the two APIs to use networking functionality in their application. For POSIX applications, which Whitix supports for compatibility reasons, the POSIX API is seamlessly supported by the `libposix` shared library. The usage pattern of the network stack depends on the application and protocol: `ftp` and `telnet` use TCP client sockets, and `httpd` creates a TCP "server" socket to respond to HTTP requests.

4.3 Network stack

4.3.1 Architecture

The userspace network stack and library comprises a number of components that interface either with the kernel or applications and utilities. Figure 4.2 depicts the components and the interactions between them. The userspace network stack itself is distributed as a shared library; applications link to the `libnetwork.so` library that contains the socket API, protocol implementations and various libraries. As a result, much of the network stack code is shared amongst applications in

function, rather than its form.

memory; however, different processes contain different network stack state in the data section of the shared library.

To implement the userspace network stack and provide primitives for socket I/O and name resolution, the `libnetwork.so` library contains the following components:

- **Network channels** Functions to control network channels and transmit and receive data on them are provided. These mainly wrap around and directly link to the usercode library provided by the kernel, although higher level operations such as blocking receive are provided. The `Channel` structure, created when the kernel channel object is created using `ChannelCreate`, provides a wrapper for the lower-level channel file descriptor, as well as the memory resources associated with the channel.

- **IP layer**. Transport layer protocols that use IPv4 use the IP layer for constructing IP headers for later transmission. The IP layer is also used for receiving packets from the network channel layer, and verifies the packet's IP header before returning the packet as valid data to higher layers.

- **ICMP and UDP sockets**. These two datagram protocols are fully supported by the userspace network stack. Applications can create UDP servers and clients using the native API, and transmit and receive packets with the usual semantics. Because of the simplicity of the two protocols, we have little to add in original contributions to their implementation in the network stack.

- **DNS resolver library**. This component will probably be moved into a separate library in the future (`libdns.so`), but it is included in the main network library because traditionally name resolution functions are available along with sockets (c.f. `gethostbyname` in `libc.so` in Linux). As well as providing a high-level function that resolves a human-readable domain name to an address, the DNS resolver code also provides functions for the construction of DNS request packets and the parsing of DNS reply packets; these functions are used by command-line utilities such as `dns`.

- **TCP sockets**. The most complex part of the network stack is the TCP implementation, which supports all the major features of a typical TCP implementation and code for gathering statistics, profiling and using profiling for the adaptive algorithms. It abstracts TCP packets as a reliable ordered delivery of a stream of bytes, and handles slow start, congestion avoidance, window sizes, all TCP states and connection establishment and termination. Many of the advanced features unique to our implementation are covered in detail in the next chapter, however, we will summarize the architecture of the TCP implementation in this chapter.

- **Statistics and profiling**. Statistics from TCP connections are collected while the connection is active and written to disk after the application exits, to be later read by programs such as `nprof`. This component also reads in profiling data from a previous run of the application and uses it as additional input for

```
struct SocketOps
{
    int (*accept)(Socket* socket, Socket* child, struct SockAddr* addr);
    int (*bind)(Socket* socket, struct SockAddr* addr);
    int (*listen)(Socket* socket, int backlog);
    int (*create)(Socket* socket);
    int (*send)(Socket* socket, const void* buffer, unsigned long length,
        int flags);
    int (*sendTo)(Socket* socket, const void* buffer, unsigned long length
        , int flags, struct SockAddr* dest);
    int (*recvFrom)(Socket* socket, void* buffer, unsigned long length,
        int flags, struct SockAddr* dest);
    int (*recv)(Socket* socket, void* buffer, unsigned long length, int
        flags);
    int (*connect)(Socket* socket, struct SockAddr* sockAddr);
    int (*connectEx)(Socket* socket, struct SockAddr* src, struct SockAddr
        * dest, int flags);
    int (*shutdown)(Socket* socket);
};
```

Figure 4.3: The socket operations structure. All of these operations are available to user applications by calling functions such as `SocketAccept`, `SocketBind` etc, which are wrappers around this simple form of object-oriented design. Depending on whether the protocol is connection-less or connection-oriented, the exact set of functions implemented by a protocol differ.

fine-tuning of the TCP adaptive algorithms, which is described in the next chapter.

- **Asynchronous I/O and events**. The network stack provides a set of functions for handling events from and directly interacting with the network stack. Asynchronous I/O functions, such as `SocketAsyncSend` and `SocketAsyncRecv`, are special cases of network stack events and are handled accordingly. Applications such as `httpd`, `ftp` and `telnet` use these functions for performance reasons. The mechanisms involved in event registration and notification are detailed in the following chapter.

4.3.2 APIs

Two application programming interfaces (APIs) are provided for use in Whitix. There is the native API, which is part of the userspace network stack in `libnetwork.so`, and the POSIX API in `libposix.so`, which wraps around the native socket API and is intended to support POSIX applications ported from other operating systems. The two follow the same methodology in exposing networking functionality. Both abstract connection endpoints as sockets, and the function have similar names and the same order of parameters.

However, the native API includes sets of functions for statistics, events and asynchronous I/O, which either do not exist, are operating system-specific, or are

not widely implemented in the POSIX API. In short, the POSIX API supplies a subset of the functionality of the native API and the network stack.

Native

The native API is specific to Whitix. The main handle in the API is the `Socket` structure pointer, created using `SocketCreate` in a similar fashion to `socket` in `libposix.so`. Most of the basic socket functions (`SocketSend`, `SocketRecv` etc.) wrap around their corresponding entry in the `SocketOps` structure (Figure 4.3). As a result, if an operation is not implemented by a protocol, the native API will return an error.

The main addition, and the reason for creating a separate API for the userspace network stack, is that we can deal directly with socket objects (rather than indirectly through file descriptors or indices) and easily expose an interface to the new functionality of the userspace network stack. Functions such as `TcpSetTransferSize` take a `Socket` pointer (the function verifies it points to a TCP socket) and interact with the TCP implementation directly through the generic network events API detailed in the next chapter.

POSIX

The POSIX sockets API is part of the general POSIX compatibility layer available in `libposix.so`. It emulates the networking socket layer user interface of UNIX-like OSes; functions, instead of dealing directly with a `Socket` pointer, pass around a *file descriptor* to functions such as `connect`, `send`, `recv` and `shutdown`, which in turn call their native equivalents. Essentially, the POSIX API is a thin wrapper over the Native API, which is also a thin wrapper over the protocol-specific code and essentially the network stack itself. This file descriptor is actually indexes into a internal table of file pointers, with each file pointer having a different type (e.g. socket or disk file). The action for generic functions like `close` depends on the type of the file structure pointed to by the file descriptor; for normal files, the `SysClose` system call is issued, but for sockets, we call the `SocketClose` function in `libnetwork.so` to shut down the connection.

The "file descriptor as socket pointer" metaphor runs into trouble with the `poll` function, which, to work effectively, must poll both sockets and files efficiently. The workaround for TCP sockets, which has so far not been implemented, would involve creating a local pipe using the `SysPipe` call during socket creation, handing the write end to the network stack, and the read end to the application. The process is described below.

Whenever application data arrives at the socket, the network stack would write to the pipe to signal any waiters in `SysPoll` that there was new data available.[4] The only disadvantage of the workaround is that the `POLL_OUT` descriptor event would no longer signal, as writing to the read end of the pipe is never allowed. A

[4]Presumably UDP and ICMP sockets, which have no transport-level messages like TCP, could directly poll the channel using the channel file descriptor.

```
typedef struct tagChannel
{
    int fd;
    ChannelAddr src, dest;
    char* baseAddress;
    ChannelInfo* info;
}Channel;
```

Figure 4.4: In the network stack, the channel structure keeps track of key resources associated with the channel, including its file descriptor (`fd`), the address pair for connections (`src` and `dest`), the memory associated with the channel (`baseAddress`) and a pointer to the information structure in the channel (kept in the shared memory between userspace and kernel), the `ChannelInfo` structure.

workaround for that is to create pipes in both directions, and so the pipe with the write end given to the application would always signal `POLL_OUT`.

The overhead for this workaround has not been determined, but for servers that have a large amount of client connections and `poll` using the POSIX API, the overhead of three times the number of connections (one channel and two pipe file descriptors) may result in considerable overhead when processing translating `poll` to `SysPoll` calls in the kernel.

4.3.3 Internal layers: channels and IP

The channel layer of the network stack is not designed to be directly used by applications, expect in certain utilities such as `dhcp` (see Section 4.5.3). Instead, as depicted in Figure 4.2, the higher level protocol implementations use a small set of channel-centric network stack functions and the usercode library to interact with the kernel.

The main abstraction in the channel layer is the (small) `Channel` structure, shown in Figure 4.4. Each socket has a pointer to a private instance of the structure, with the functions in the channel layer, such as `ChanBufferSend` and `ChanBufferRecv` using it to access the resources associated with that channel when issuing a system call to the kernel. Other structures used include the opaque pointers `ChanSendBuffer` and `ChanRecvBuffer`, used for representing a send and receive network channel buffer respectively.

Other functions wrap directly around the usercode library provided by the kernel (Section 3.4.2), passing the `baseAddress` field, which points to the header page of the channel, to the main usercode library function. For example, `ChanSendBufferAlloc`, which allocates a `ChanSendBuffer` and the corresponding data pointer, calls `uChanSendBufferAlloc` (and then `uChanSendBufferData` to retrieve the data pointer) – both are function pointers that point to their implementations in the usercode library (see Appendix C) and directly manipulate the shared memory structures of the network channel on behalf of the channel layer.

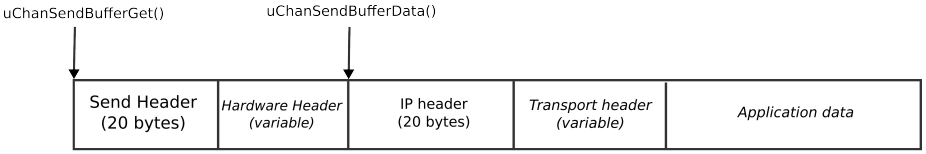

Figure 4.5: The layout of a IP packet in the channel send buffer represented by `ChanSendBuffer`. The send header is filled in by functions in the usercode library, and the hardware header and part of the IP header (the source address and checksum) are filled in by the kernel. The rest of the packet is constructed by the userspace network stack.

IP layer

The IP layer is not directly responsible for packet transmission, but IP layer functions are called to construct the packet. `IpBuildHeader`, given information about the packet, prepends an IP header describing the packet, following the format depicted in Figure 4.5.

All protocols in the userspace network stack use a common set of functions for receiving an IP packet on a channel. These functions, `IpChanRecv` and `IpChanRecvNb`, call the channel-layer equivalent to receive a `ChanRecvBuffer` pointer to the packet from the network channel, and perform basic error checking on the packet. For example, the received packet length is compared against the IP header's idea of the packet's length; if the received packet length is less, the packet has been truncated and may not contain valid data. Eventually, when IP fragmentation support is implemented, reassembly of the packet will take place at this layer.

4.3.4 UDP and ICMP sockets

These are two of the core protocols of the network stack, and are actually very similar in operation. UDP delivers datagrams to other hosts in a connection-less and unreliable manner, containing user-specified data. ICMP is used, mainly by the system and utilities, to send error and diagnostic messages. Since both are datagram-based, their mechanisms for sending and receiving data are very similar, but the data carried by them and their usage patterns are very different. They are also connectionless protocols, so little to no state is required for a socket. These characteristics make it an ideal place to start implementation of the higher level protocols.

As a result, both of the implementations are similar in operation, and involve simple packet handling and error checking. We shall now survey the different aspects of the protocols, noting any unique features of either protocol that affect operation. These include:

- **Socket creation.** When the SocketCreate function is called, both protocol implementations create a socket, noting their respective operation structures. The UDP socket then calls `UdpSocketConnect` to create a network channel for later calls; ICMP leaves this to the application or utility to call explicitly before allowing the application to send ICMP messages.

- **Socket "connection".** Although both are datagram protocols, we must allocate a source port (among other things) using the kernel's framework, and there we must notify the kernel of our resource usage by creating a network channel. Both create a network channel with a zero source address (so the IP-specific code in the kernel can assign a suitable source address and port, as in Section 3.6) and the broadcast address (255.255.255.255) is supplied to the `ChannelCreate` function to indicate that this socket can send to all hosts.

  The destination address provided to the function, if not `NULL`, is used as the default destination address when the application calls `SocketSend` instead of `SocketSendTo`. It is also the only address, if specified, from which `SocketRecv` datagrams can be received, although this is only enforced at a userspace level.

- **Sending and receiving data.** Both ICMP and UDP transmit data using `SocketSendTo`, or `SocketSend` if the default destination address has been supplied by calling `SocketConnect`. The difference between the two protocols involves the input buffer in transmission, since ICMP is a diagnostic protocol and UDP a transport one. Packets sent out over a UDP socket have their header automatically added, whereas applications using ICMP sockets must construct the header before transmitting the packet.

  The reasoning for these differing semantics is that use of ICMP sockets and channels in the future will be restricted to diagnostic applications run by privileged users (where, in a security context, constructing raw packets is permissible), and diagnostic applications typically need more control over the transmitted packet. For example, the `ping` application uses the ID field of the ICMP header in `ECHO REQUEST` packets to identify corresponding `ECHO REPLY` responses. (Section 4.5.5).

  Because UDP and ICMP are packet-based protocols, the send and receive calls operate with packet semantics. Transmissions larger than the maximum packet size of the channel (which corresponds to the minimum MTU of any interface) are not allowed, and will return an error. This simplifies the implementation, and only a couple of small packets are sent over UDP in a typical session anyway.

  In a similar fashion to sending a packet, UDP and ICMP sockets can receive packets by calling `SocketRecvFrom`, or `SocketRecv` if the default destination address has been supplied using `SocketConnect`. The same header and packet semantics also apply when receiving packets; only one packet is received in a call, and if the socket uses the ICMP protocol, the ICMP header is included in the received data. If the caller supplies a buffer that is too small for the entire packet, the packet is partially copied into the buffer, truncated and the rest dropped.

- **Socket close and destruction** Since UDP and ICMP are datagram protocols, closing the socket does not cause any messages to be sent; no connection

termination handshaking is involved. The socket is freed (there is no UDP or ICMP-specific state stored) and the channel silently closed.

4.4 TCP

Since it estimated more than 80% of the traffic on wide-area networks such as the Internet involves TCP,[9] it was key that there was a functional TCP implementation that would implement all the major features described in the relevant RFCs,[48] so we could build an expressive interactive and adaptable event layer on top of TCP. The major features supported are *low start and congestion control with retransmission* (including the **TCP Vegas** congestion control algorithm), *TCP client and server sockets*, *Protect Against Wrapped Sequence numbers (PAWS)*, *Initial Sequence Number Randomization*, *dynamic receive and send windows*, and other options, such as *maximum segment size* and *window scaling*.

4.4.1 Design choices

Since TCP involves a complex implementation (compared to the UDP and ICMP datagram protocols), the architecture of the TCP code involved two major design decisions, as well as deciding what optional features should be included, based on how it would advance the eventual goals of adaptability and interactivity or improve security.

The first decision involved how to **handle incoming packets** in TCP. Since datagram protocols do not have stream management, there are no *transport-layer-specific* packets that must be handled in UDP or ICMP. Since we have to send out ACKs and handle connection initiation and termination without the knowledge of the application (to obey general socket semantics), should we handle any incoming packets *in-band* in a *single-threaded* approach whenever the application receives data [5] or should we have a *multithreaded* approach, where there are separate thread or set of threads that wait and poll for incoming data?

In the end I chose the latter multithreaded option. It appears as if the single threaded route, in an initial observation, results in spurious retransmission of packets from the remote host; this is because if a socket is rarely used, acknowledgement packets are not sent when remote data arrives, but when the socket is next used (either to send or receive data). The remote host might also deduce that the network connection is congested and reduce bandwidth, reducing performance further.

Instead, newly created sockets (both client and server) are polled by a single thread, and packets are handled and processed as soon as they arrive. This results in low latency for ACK packets, and as a result the network stack may be busy even if the application is not; this may be a more efficient use of processing time, instead of paying the processing cost when the application chooses to receive or send data, which could be latency-critical. As a downside, this may result in less processing

[5] We may have to check any received packets when we send data as well, so we can send back ACKs for received but unread data to avoid unnecessary retransmissions.

time allocated to each socket if `SysPoll` indicates there are many incoming TCP packets waiting.

Another feature that differentiates established network stacks is their choice of measurement-based **congestion control** algorithm. Using the *round-trip time* , the amount of time from a data packet being sent to its corresponding acknowledgement being received, estimate as a guide, different algorithms adjust the *congestion window* , the network stack's idea of how many bytes can be sent without overloading the network with too much traffic,[6] differently: **TCP Vegas**, the algorithm that I chose to focus my efforts on, measures the round-trip time from each packet and linearly increases or decreases the congestion window to compensate.

Features that were not included (due to constraints on the length of the project) included **selective acknowledgement**, which would have further reduced wasted bandwidth by avoiding unnecessary retransmission, and a framework for an application to select from a variety of congestion control algorithms for a particular socket's needs.

4.4.2 Possible TCP changes

While developing the TCP implementation, I realized several of the features of TCP were focused on network stacks with a different design – especially those with in-kernel network stacks where the received data regions were allocated in terms of bytes, rather than buffers in the network channel.

For example, the receive window in **flow control** becomes irrelevant if we have enough bandwidth to process incoming data, but rarely receive data from the socket. Because of the way received packets are stored in the network channel list of buffers (to avoid unnecessary copies, but there other possibilities), the fact is that we could handle much more data, but are limited by the advertised receive window.[29]

My belief is that the receive window partly originates from environments where machines of diverse network speeds communicate, and also by the realities of memory management in the kernel; we can only allocate X bytes per socket because, especially in monolithic kernels, the pages allocated cannot be swapped out[7] and so the memory is limited.

Enforcing a limit on the bytes for each receive window also avoids especially high memory usage for one socket – however, if the received buffers are stored in user memory (so per-process, with more allocated pages per network channel meaning less memory available for the rest of the application) and can be swapped out if needed, there may be one less reason for the concept of a receive window in the protocol.[8]

[6] Note that this may be differ from the advertised receive window at the opposite host.

[7] In Linux some classes of pages can, but an interesting situation arises with swapping out network pages if the swap file is located on a networked filesystem!

[8] *Window scaling*, a TCP option that allows the scaling of windows beyond 64KB, allows us to treat receive windows as less important in the protocol by setting the value to an artificially high number if needed.

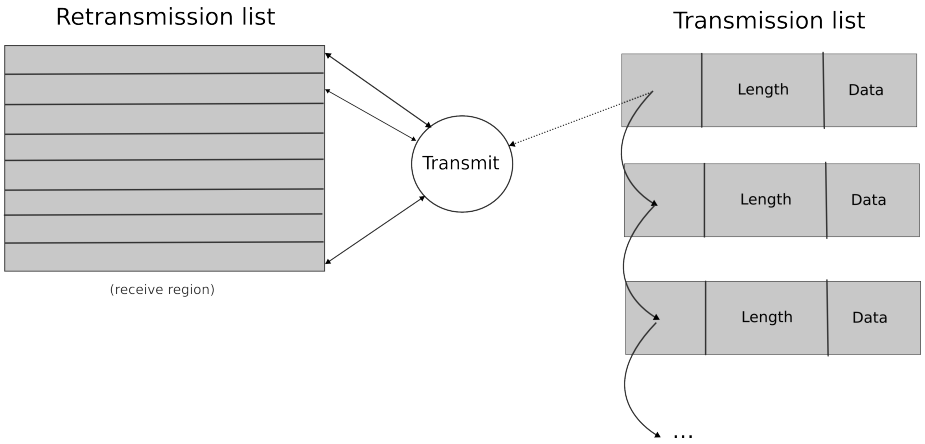

Figure 4.6: The *retransmission* list comprises the `ChanSendBuffer` pointers that point to memory in the network channel send region (hence the array layout in the diagram). The oldest transmitted unacknowledged packet is at the head of the list. The *transmission list* is a linked list of variable size buffers in ordinary memory that are waiting to be transmitted, but cannot be yet due to the receiver's window being full or no buffers being available in the channel's send region.

If the received buffers are kept in channel memory until the data in the packets are read by the application, (ACK packets containing no data are processed and discarded) then, in terms of memory, we are limited by the number of packets we can hold in the buffer at one time. The number can be changed by supplying a value for `numRecvBuffers` during channel creation, but the important thing is that currently the number of buffers per channel stays constant.

The particular channel memory configuration means, in the our implementation, that the receive window, defined in bytes, *may be larger* than the number of packets we can receive that fill the window.[9] This is a convincing argument for copying data out of the channel buffer – a sensible approach may be to perform copying in a situation where there are no free buffers in the receive array. However, since the receive region is meant as a temporary area for buffers, data tends to be copied out of it quickly.

4.4.3 Sending packets

Although at first sight sending packets may seem trivial – just construct a packet with the appropriate TCP header, update the relevant sequence numbers and send – the presence of `SocketSend` calls with a buffer larger than `maxPacketSize` or the *maximum segment size* (MSS) of the remote host and the potential for packet loss (and its cousin retransmission) complicates the implementation somewhat.

[9]For example, a channel with N buffers can receive N (non-fragmented) packets in the receive window. If the packets are one byte long and the application does not continually read to free the corresponding packets, then the receive window, in reality, will only be of N bytes.

The fact that the remote host may also have a small receive window (or that data is being sent quickly by the application) means we must also buffer data dynamically outside the network channels and transmit when there is free space in the remote host's window.

The typical state of the TCP socket with regards to transmission buffers is depicted in Figure 4.6. Applications wanting to send data through a TCP socket end up calling `TcpSocketSend`, which attempts to transmit any packets waiting on the *transmission queue*. If the transmission queue could not be completely emptied, the data to be transmitted is also added to the tail of transmission queue linked list. Otherwise, `TcpTryToSend` is called, which then calls `TcpDoSend` if enough space is available in the congestion and receive windows.

`TcpDoSend` sends one buffer, doing the work of splitting up the buffer into multiple TCP packets, updating sequence numbers, and adding packets to the transmission queue if there is no space in the channel send buffer array (which also functions as the retransmission list).

The two queues, the transmission list and the retransmission array, are updated at different times and have different usage patterns. The retransmission list is updated when an ACK is received – packets with a sequence number less than the acknowledged sequence number are regarded as successfully transmitted and discarded. The *return trip time* and dependent variables (such as the *retransmission timeout*) are recalculated, and the transmission list is updated in a similar fashion when packets are constructed, sent, and placed in the retransmission list – the two operations are usually sequential.

4.4.4 Retransmission

Following Karn's algorithm and the relevant RFCs,[43] each time a packet is transmitted, the *retransmission timer* is set up to expire in `rto` seconds, where `rto`, the *retransmission timeout*, is the amount of time the sender will wait for a given packet to be acknowledged. If the retransmission timer expires, the packet is quickly resent by calling `ChanBufferSend` with the oldest packet not yet acknowledged – the `ChanSendBuffer` at the head of the `retx` list. The retransmission timeout value is then updated and the *retransmission timeout* application-level event is fired.

4.4.5 Receiving packets and the state machine

Packets are received asynchronously through a separate worker thread, as described in Section 4.4.6. Once the packet arrives at the socket, it is processed by `TcpHandleData`, which first performs header processing (such as the processing of TCP options, described in [48]) and then runs the packet handler for the particular state (one of the eleven shown in Figure 4.7) – for example, sockets in the `ESTABLISHED` state have their packets processed by `TcpEstablished`.

Most of the state functions filter the packets with to find a single packet to move to the next state in the state machine. For example, the `CLOSING` state only moves to the `TIME_WAIT` state once a FIN+ACK packet is received. The most complex

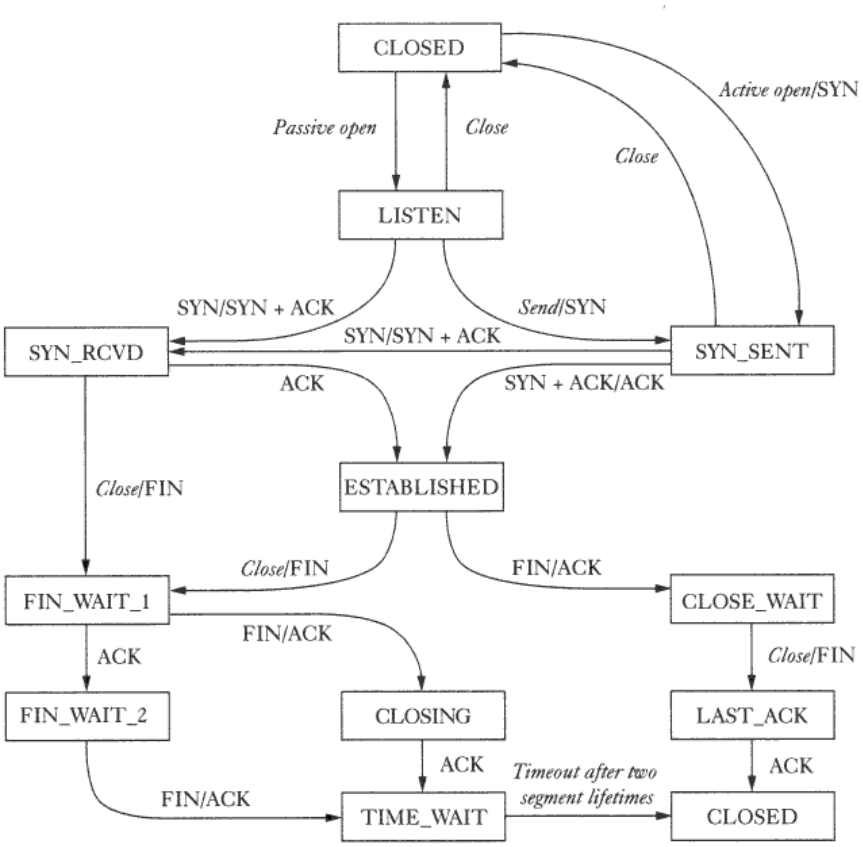

Figure 4.7: TCP state diagram. There are eleven standard states for a TCP socket; the exact path through the state machine depends on whether it acts as a client or a server socket. The Whitix userspace network stack implements all eleven, along with error handling for erroneously sent packets. Packets sent to closed ports are handled either by the kernel or by the channel corresponding to the now-closed TCP socket created by the application.

state function is the ESTABLISHED state (this is where the exchange of data takes place), which handles verifying sequence numbers, acknowledgement numbers, receiving data payloads, and handling any FIN packets received. Receiving a new ACK packet causes the retransmission and transmission queues to be updated as in the previous section. Successfully received ACKs also increase or decrease the congestion window, with the handling code in tcp_congestion.c.

Data received in the ChanRecvBuffers in the ESTABLISHED state is added to the data linked list and the number of packets waiting to be received by the application is increased. The application picks up the packets by calling TcpSocketRecv, which, in a producer-consumer relationship, copies out packets as they arrive until the buffer is full or there are no new packets arriving. The read field in the ChanRecvBuffer structure, described in Section 3.4.1, is updated when the packet is read by the application (and then freed in TcpSocketRecv if all the data has been read from the packet).

4.4.6 Socket polling

To efficiently poll for incoming data on all open TCP sockets, the network stack, through a worker thread that executes the TcpWatcherThread function, calls SysPoll on an array of file descriptors of the network channels corresponding to the list of TCP sockets. If there is no data arriving on any socket, the thread either suspends itself or sleeps in SysPoll, leaving as much processing time as possible to the application.

The array of file descriptors (and an array of the socket pointers corresponding to these file descriptors) is constructed via the TcpSocketAdd and TcpSocketRemove functions; when a TCP socket sends a connection request (TcpSendSyn) or reply (TcpSendSynAck) for example, and expects a reply, the channel's file descriptor is appended to the array. If it is the first socket opened, the worker thread is created and begins polling, so the cost of an extra thread is not paid by sockets that do not use TCP functionality. When the TCP connection enters into the CLOSED state and the socket stops sending and receiving data, the socket's channel file descriptor is removed from the array. The array dynamically expands to hold the number of open sockets, but does not shrink if these sockets are closed.

If any network channels have incoming data, revents in the poll structure is set to an appropriate value by the kernel. When SysPoll returns, along with a return value that indicates the number of ready channels, the data is read from the channel's list of receive buffers by calling ChanBufferRecvNb, as described in Section 3.4. This data is passed to TcpHandleData, which is the core of the state machine described above; the packet is appropriately handled – we then signal any events or the arrival of new application-level data – however, much of the traffic may be session messages such as data acknowledgements and keep-alive requests.

Application polling semantics

There is one caveat with this method. Due to the way TCP sockets are asynchronously polled in the shared library for new incoming data, using SysPoll on

```
input drop -p tcp -d 80 // Drop all incoming packets with destination port
     of 80 (HTTP)
input drop -p udp // Drop all incoming UDP packets.

// Let only outgoing TELNET packets through the firewall.
output accept -p tcp -d 23
output drop
```

Figure 4.8: Examples of the rules enabled by the Whitix packet filter. The `firewall` parses the command-line input and updates the kernel-level firewall, which is implemented in the protocol classifier code for the TCP and UDP protocols and described in Section 3.7.1.

the channel's file descriptor in the main application leads to unreliable notification of events on that file descriptor. This is because, while the application is polling for new data on that descriptor, the TCP worker thread is simultaneously polling on the same descriptor. This causes a race to handle new notifications, which the TCP worker thread, due to its frequency of polling, generally wins. The result of this is that the application typically never or only rarely receives an event on the file descriptor, which, if the application waits forever (specified by a negative value in the timeout parameter of `SysPoll`) on just that file descriptor, the application may appear to "hang", with negative consequences for interactivity and throughput.

The above is only true for TCP sockets that are polled using the direct system call `SysPoll`. For the users of POSIX sockets (which because of numerous ports of applications from UNIX platforms will be the main users of polling), a workaround can be implemented because one UNIX file descriptor functions as a wrapper around more than just one Whitix file descriptor. This workaround is described in Section 4.3.2.

4.5 Utilities

4.5.1 firewall

`firewall` is the command-line user interface to the stateless firewall implementation available at the channel layer (Section 3.7.1). The `firewall` interacts with the firewall via the `/Network/Firewall` directory in the Information and Configuration filesystem, allowing the user to add, edit and delete rules in the `INPUT` and `OUTPUT` sets of rules. `firewall` does not provide for ordering of rules, so a very general rule added first of all (such as `firewall -a input drop`) will take precedence over all others, including more specific rules that contradict the general one.

The program supports the same set of atoms for specifying firewall rules as the kernel implementation. Packets can be blocked based on protocol, source port and destination port (as well as a combination of the three, see Figure 4.8); the `firewall` program functions only as a simple wrapper around the firewall, sanity checking values and passing them directly to the kernel-level firewall.

4.5.2 nprof

`nprof` takes the (binary) statistical output (available as `.ns` files) from `libnetwork.so` and displays it in a human-readable format, also adding any derived figures (such as the average packet size and standard deviation of packet sizes) that are not directly included in the file. After verifying that the supplied file is an output file, it outputs information about the most accessed ports, the last accessed hosts and ports, and several aggregate functions over the data. There is no facility for editing the file. The internals are covered further in Section 5.3.

It is designed to be run in a similar fashion to `gprof`, which profiles the CPU time used in applications with a breakdown for functions; `gprof` itself only analyzes the automatically instrumented output from a program compiled with suitable command-line flags. `nprof` is designed to be used by developers and system administrators for diagnostic and analytical purposes; with the program, we can easily determine exactly how much bandwidth a particular connection is using, without any of the statistical inaccuracies of other profilers.

4.5.3 dhcp

`dhcp` is the Whitix implementation of a DHCP client, which contacts a DHCP server to retrieve its **IP address assignment** and other **configuration information**, like the IP address of the DNS server for the local network.[19] It is a key utility – it assigns an interface an IP address so that it can communicate as a host with the rest of the network.

`dhcp` is a unique case in applications and utilities; it is the only program that begins with no IP addresses assigned to the interface. After being supplied the name of a network interface by the user, it creates a network channel using `SysChannelCreate` with the `CHAN_IGNORE_ADDRESSES` flag, brings the network interface up and binds the channel to that interface using the "bind to interface" operation of `SysChannelControl`, before sending raw packets (including the broadcast Ethernet and IP addresses) and receiving packets through that interface.

`dhcp` uses the above mechanism to send out the `DHCPDISCOVER` and `DHCPREQUEST` messages, and receive the `DHCPOFFER` and `DHCPACK` messages. The program is also used to bring down the given interface, but does not yet send out the `DHCPRELEASE` message; the IP address is still assigned to the machine by the router but the host interface is no longer up.

4.5.4 dns

`dns` is the DNS lookup tool for querying DNS name servers. It performs DNS lookups and displays the replies generated from the queried name servers. It is similar to tools such as `dig` (the *Domain Information Groper*) on other systems. Currently only name lookups to other names (aliases) and IPv4 addresses are supported (the most common use of the tool); Figure 4.9 shows a typical run of the application.

```
/>dns www.google.com
www.google.com is an alias for www.l.google.com
www.l.google.com is an alias for www-tmmdi.l.google.com
www-tmmdi.l.google.com has address 66.102.9.147
www-tmmdi.l.google.com has address 66.102.9.105
...
```

Figure 4.9: `dns` output for www.google.com. The `dns` utility is useful for diagnostic purposes, and provides a simple test of UDP network channels and UDP client sockets.

```
/>ping www.whitix.org
PING www.whitix.org with 56 bytes of data
64 bytes from www.whitix.org: sequence=0
64 bytes from www.whitix.org: sequence=1
64 bytes from www.whitix.org: sequence=3
64 bytes from www.whitix.org: sequence=2
...
```

Figure 4.10: An example of the output of `ping`. Although Whitix's implementation does not have the configurability of the standard BSD implementation, it is intended as a diagnostic test of the ICMP protocol implementation, the packet classifier and as a small but comprehensive test of the low-level networking layers.

`dns` uses the DNS resolver library in `libnetwork.so` to construct a DNS packet and inspect the subsequent reply packet, which consists of a set of resource records. `dns` displays replies for records of type `CNAME` (aliases) and `A` (32-bit IPv4 addresses) in a human-readable format. Other resource records could be supported, but they are not as frequently used.

4.5.5 ping

`ping` uses ICMP sockets to test whether a particular host is reachable across through the local network or the Internet. It constructs ICMP Echo Request packets and sends them to a specified host, recording the type at which they were sent that we can use the reply (which duplicates the data section of the packet) to calculate the round-trip time of the packet in a stateless manner. It is the only test of ICMP channels and their corresponding userspace sockets devised so far, and is useful for comparing the estimate of the round-trip time to a host made by the TCP implementation with a more direct measurement.

4.6 Applications

Each of the applications developed for this project used a **special feature**, such as **profiling output and input** or **asynchronous I/O**, or combination of features unique to the project. This was mainly for testing and demonstration purposes, and

```
...
230-NFS and SMB/CIFS are no longer available.
230-
230-For comments on this site, please contact <ftpadmin@kernel.org>.
230-Please do not use this address for questions that are not related to
230-the operation of this site.  Please see our homepage at
230-http://www.kernel.org/ for links to Linux documentation resources.
230-
230 Login successful.
ftp> ls
227 Entering Passive Mode (204,152,191,37,75,39).
150 Here comes the directory listing.
drwxrwxrwx    3 0        0             109 May 27 04:06 bin
dr-xr-xr-x    2 0        0              28 Aug 29  1997 dev
d--x--x--x    2 0        0              49 May 20  1998 etc
drwxrwx---    2 536      528           124 May 21  2001 for_mirrors_only
drwxr-xr-x    2 0        0            4096 May 20  1998 lib
drwx------    2 0        0               6 Oct 02  2005 lost+found
drwxrwsr-x   11 536      536          4096 Feb 12  2009 pub
lrwxrwxrwx    1 0        0               1 Apr 21  2007 usr -> .
lrwxrwxrwx    1 0        0              10 Apr 21  2007 welcome.msg -> pub
    /README
226 Directory send OK.
ftp>
```

Figure 4.11: Output from the `ftp` program showing a connection to `ftp.kernel.org`. Statistics are collated for both the main *control connection* and the PASV *data connection*, but because of the variable destination port characteristic of PASV mode (e.g. the passive mode tuple above), only the statistics for the control connection are saved to disk and reused in future runs of the application.

their use is described below.

4.6.1 ftp

`ftp` is an implementation of a **File Transfer Protocol** client,[10] the protocol used for copying a file from one host to another.[51] `ftp` is an interesting test case for profiling data reuse, because it utilizes separate control and data TCP connections between the client and server applications; this two port structure is referred to as *out-of-band*. The control connection remains open for the duration of the session, with a number of data connections opened to the server to transfer directory listings and files.

`ftp` implements *passive mode* (`PASV`), where it connects as a client to an address and port tuple supplied by the control connection to the server; the port value in this destination tuple is variable and often an ephemeral port. As a result, for these *ephemeral data ports*, the port profiling output, which only stores persistent information for connections on well-known ports, generated by this program cannot be used at the moment as input to the next run of `ftp`; since the port com-

[10] `ftp` only supports read-only anonymous access currently.

binations are variable, it would be difficult to identify which ports were used in a previous PASV connection.

The host profiling output is still extremely useful, since the information in the host entry, which is especially useful for bulk transfers (as described in Section 5.4), is available for connections to all ports on that host. To this end, ftp was intended as a demonstration of the profiling and adaptive capabilities of the new network stack, even in the absence of much of the profiling information for many of ftp's connections.

4.6.2 telnet

telnet is an implementation of a **Telnet** client for Whitix, used to provide a bidirectional interactive text-oriented connection via a terminal.[50] It uses one TCP connection for the entire session. The unique feature of telnet (among well-known protocols), is that, after an optional initial negotiation (or *handshaking*) of terminal parameters, the protocol itself is entirely dependent on which host the client has connected to and what program is running at the remote endpoint.

Because of these bidirectional features, telnet is an ideal test of the asynchronous receive primitives available in the userspace network stack and described in Section 5.5. During normal operation in asynchronous mode, telnet creates the server connection, designates one asynchronous receive context (with a callback function TelnetSocketRecv) via SocketAsyncRecv and, during the main loop, polls the keyboard input for data to send to the server connection. When data arrives at the socket, TelnetSocketRecv is called in a separate thread context and handles the Telnet control and data (distributed *in band*) information, updating the console if necessary. This asynchronous feature of telnet was used for testing purposes and to avoid the issue of socket polling (as described in Section 4.4.6).

However, since asynchronous I/O is not really suitable for interactive programs, the asynchronous behavior can be switched on or off via a preprocessor #define statement. If the socket receive is synchronous, the socket itself is set to non-blocking, so that the keyboard can be continuously polled.

4.6.3 httpd

httpd is a HTTP 1.0 server used to test the application feedback capabilities of the userspace network stack.[6] httpd responds to GET requests by transferring files from the local filesystem. This typically involves distributing large files to clients, which is an ideal test of the userspace network stack's ability to reach peak performance quickly; it is useful as a benchmarking tool. httpd is the sole user of the POSIX API described in Section 4.3.2, with several Whitix-specific calls (when HTTP_TEST_HANDLERS is defined) added for testing features unique to the Whitix network stack. It is also a test of the TCP server socket implementation in the userspace network stack.

httpd tests the interactive and adaptive capabilities of the stack. When a client connects to the server, HTTP registers a number of callbacks with the socket for

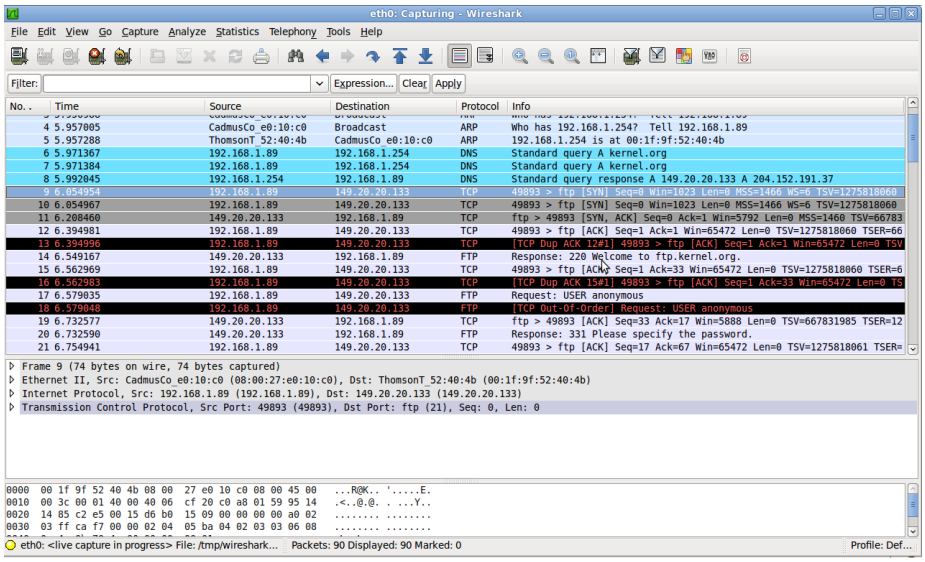

Figure 4.12: Wireshark used to inspect a TCP connection as the userspace network stack on Whitix runs on Virtualbox. Wireshark and Virtualbox were useful in debugging sequence number problems, checksum errors and packet formats for certain classes of application protocols, such as FTP. Shown here is the start of a FTP session using `ftp` to `ftp.kernel.org`.

performance, benchmarking, and diagnostic purposes (the exact combination of these can be expressed through a set of preprocessor `#define` statements). These include the *round-trip time measurement update* event and the *remote window size changed* event. In particular, the *remote window changed* event is used to adjust the buffer size used to read from disk in the application; they are probably more efficient ways of optimizing data flow with the right information and algorithms.

After the initial callback setup, the server handles the client request (and optionally printing out debug information). In HTTP, when the server responds to a GET request, the server spends a small packet containing the HTTP headers, followed by the data of the file. Before `sending` out the data, `httpd` lets the userspace network stack know much data will be transferred by calling `TcpSetTransferSize` with the size of the file and the size of the buffer used to transfer data from the file (using `SysRead`) to the network.

The information supplied by the application, combined with the previous TCP slow start values and the round-trip time estimate from previous connections from the client (automatically incorporated when the connection is created) and the transfer size could be used to implement a variation on the *network adaptive slow start* algorithm described in [7].

```
/>tcp_test --verbose
Test suite #1: Socket creation
==============================
Test #1.1: Create server socket with source port 3434 ... [passed]
Test #1.2: Create TCP channel connecting to 3434 on localhost ... [passed]

Test suite #2: Listen
=====================
Test #2.1: Send packet containing ACK ... [passed]
       (received packet with correct SEQ and RST set)
       (socket still in LISTEN state)
Test #2.2: Send SYN .. [passed]
       (received packet with correct SEQ, ACK and SYN, ACK set)
[accepting connection]
Test #2.3: Sending RST bit to socket in SYN-RECEIVED ... [passed]
       (child socket now closed)
Test #2.4: Repeat Test 2.2 ... [passed]
[accepting connection]
...
```

Figure 4.13: The output of `tcp_test`. Like *TIRTS* (Section 7.2.1), `tcp_test` constructs a series of raw packets to test the TCP stack conformance to RFC 793.[48] The technique is applicable to testing any host on the Internet, but `tcp_test` also creates a local raw socket and verifies its state after each raw packet has been sent, something *TIRTS* does not do.

4.7 Testing

During development of the userspace network stack, I used a variety of testing methods to ensure, as far as possible, that the stack, and especially the TCP implementation, worked under a variety of network conditions and network interfaces. The fact that there are few debugging tools available on Whitix, such as the lack of a traditional source debugger such as `gdb` or code coverage tool like `gcov`, hampered quick testing, but also meant that I had to be more inventive with testing methods, employing automated testing where possible. The fact that virtually all the software I wrote was network-facing aided testing greatly.

4.7.1 Specific methods

In the early stage of development, using the **packet sniffer** *Wireshark* (when the operating system was being tested in VirtualBox) was a useful diagnostic tool to ensure we generated correct output for ICMP, UDP and TCP packets. Figure 4.12 depicts the tool being used in development to snoop on a TCP connection; the fact it would display clearly any errors found in sniffed packets, at all layers of the stack (including various application protocols like DNS, FTP and HTTP) was useful for debugging packet construction and transmission at all levels.

Verifying that the packets followed the standard format using *Wireshark* was a useful method of finding obvious bugs (especially when network communication appeared not to work at all due to some low-level error), but it would be time-

consuming to exhaustively test each possible scenario in each protocol. To that end, I started to develop a unit test suite, `tcp_test`, with the objective of exhaustively testing the TCP implementation in an automated manner. Figure 4.13 shows the set of tests executing – although the test suite is not exhaustive yet, the goal is to achieve 100% code coverage and to test all states in the state machine. With the port of a code coverage tool such as `gcov`, I will soon be able to verify this.

Another possibility in testing was the **TCP/IP Regression Test Suite** (by Nanjun Li), because since it ran solely on FreeBSD systems (which I did not have easy access to), I could not test regularly with the software. (The software is further described in Section 7.2.1). The advantage of testing network-facing software is that it can be easily tested and verified remotely, which greatly increases the range of software available – however, since there appear to be no TCP unit testing suites that run on Linux, it was difficult to perform automated tests regularly.

One area of the network stack that is not well-tested is the `multithreaded aspect` of the TCP implementation. The locking that takes place between the main application using `TcpSocketSend` and `TcpSocketRecv` (where the `sendUpdate`, `txLock` and `retxLock` locks are taken to update the sequence numbers, transmission list and retransmission buffers respectively) can be taken through a number of different paths – although the common case (and many other tested cases) appears to be free of deadlocks, we have not been able to verify this. Errors in synchronization may lead to deadlock or gaps in synchronization protection, so testing efforts should be focused on verifying there are no multithreaded issues before continuing future development of new features.

4.7.2 Application testing and summary

When creating the range of user application distributed along with the network stack, I chose a range of application that would test the ability to handle general workloads of all the major socket types. Therefore, `ftp` and `telnet` are tests of TCP client sockets, `httpd` is a test of TCP server socket handling, `echoserv` is a very simple UDP server implementation, and `dns` is the test of the DNS resolver library and UDP client sockets.

In summary, the most effective type of testing was the automated testing suite `tcp_test` used for the TCP implementation; the use of unit tests, non-functional testing methods (such as sending incorrect packets that the TCP implementation should reject), and regression testing improved the quality of the TCP implementation – using the TCP RFC [48] as a guide, we could test for a variety of packets sent to the host and verify the response followed the RFC. The *Wireshark* packet sniffer was also useful initially as a first resort whenever we found errors; the ability to inspect a single packet and for *Wireshark* to describe errors in the packet was useful for quick verification of functionality during the main implementation stage.

4.8 Discussion

In the timescale of the project, we have managed to demonstrate that a modern TCP/IP stack can be implemented in userspace. Apart from issues with polling sockets, which is important enough to high performance servers to demand a permanent solution in future work, and more thorough testing of the multithreaded aspects of TCP sockets, there are relatively few issues with moving the stack down to userspace.

One major open problem that has yet to be addressed is how to **transfer socket state** between processes. Although the *fork and handle connection* metaphor is not used in Whitix, there are times when we would like to pass a socket to a child process for further handling. Possible solutions include a common area in shared memory for multiple applications to have access to the protocol state data,[20] which introduces the problem of locking state that we have tried to avoid by developing network channels. For Whitix applications however, passing sockets between applications, especially since they are only available as `Socket` pointers, will not be a very common operation and can probably be ignored as a result. However, this is an important issue for systems that do use `fork` regularly to create new processes, and future work should focus on this.

One area to be explored concerns the zero-copy I/O beneath the network stack. This could be utilized by user applications (avoiding memory copies while building packets) in a **high-performance network API**, such as the interface mentioned in [18], that exposes the network channel buffers to the application itself. Explicit management of these buffers by high performance applications, and so discarding the POSIX `send` and `recv`, would improve performance by enforcing the zero-copy philosophy through all the layers of the stack.

4.9 Summary

We described in this chapter the new **userspace network design**, and how a modern TCP/IP stack can be implemented in userspace with all major features of popular protocols, such TCP and UDP, included. We discovered that the **latest research** in the field, which was either at the proof-of-concept stage or intended for other kernel architectures, suffered from performance and security issues. After setting out the general architecture of the network stack, with the network channels at the base, we covered the **two APIs** available for Whitix, the *native* and *POSIX* APIs. Building from the basic *network channel* layer in userspace, we explored the simple **IP, UDP and ICMP** protocol support available.

In the rest of the chapter, we outlined the architecture of the **TCP** implementation, and how it used a multithreaded architecture to receive packets and post them to the application. The complete implementation of features like congestion control, retransmission, and polling sockets were also discussed. **Applications** such as `httpd`, `ftp` and `telnet`, as well as the numerous **utilities** (`ping`, `dhcp` and `dns` included) were described – many of the applications used the advanced **adaptability and interactivity** covered in Chapter 5. We then evaluated the **test-**

ing performed, including the automated test suite `tcp_test` suite and verifying network stack output using *Wireshark*, and then discussed, among other things, extending the APIs available with a **high-performance network API**.

Chapter 5

Dynamic protocols

In this chapter, we discuss the new next-generation features that have been developed, helped by the flexibility that a userspace network stack offers. We first survey the gathering and use of statistics by the network stack to generate persistent profiling data, its inspection by the `nprof` program to output and aggregate data in a human readable manner, and we then investigate how the network stack can use this performance data in future runs of the application.

The threshold for a interactive event in the network stack no longer involves a transition to and from the kernel. Instead, we have designed a simple *event delivery* for all sorts of TCP events using callback functions, where the application registers for events from the network stack that it is interested in and receives notifications in a timely fashion.

5.1 Background

5.1.1 Adaptable and interactive protocols

After I decided to write the network stack in user-space, I considered how it could produce a tighter coupling between applications and the network stack. After reading literature on event-driven systems, and how many APIs are partly event driven (for example, the Windows event messages that arrive at `WndProc`), I thought about how the network stack could notify the application of certain events, and decided to look for any research implementations.

The only implementation I found was a modification to the FreeBSD 4.3 network stack called the **Interactive Transmission Control Protocol** (iTCP).[32] The iTCP project at Kent State University investigated the possibility of making TCP extensible by introducing elements such as an event subscription, tracking and notification mechanism for certain TCP events that applications can subscribe to.

Their work was motivated mainly by the fact that congestion control for multimedia traffic (such as VOIP) has remained a difficult problem; many schemes

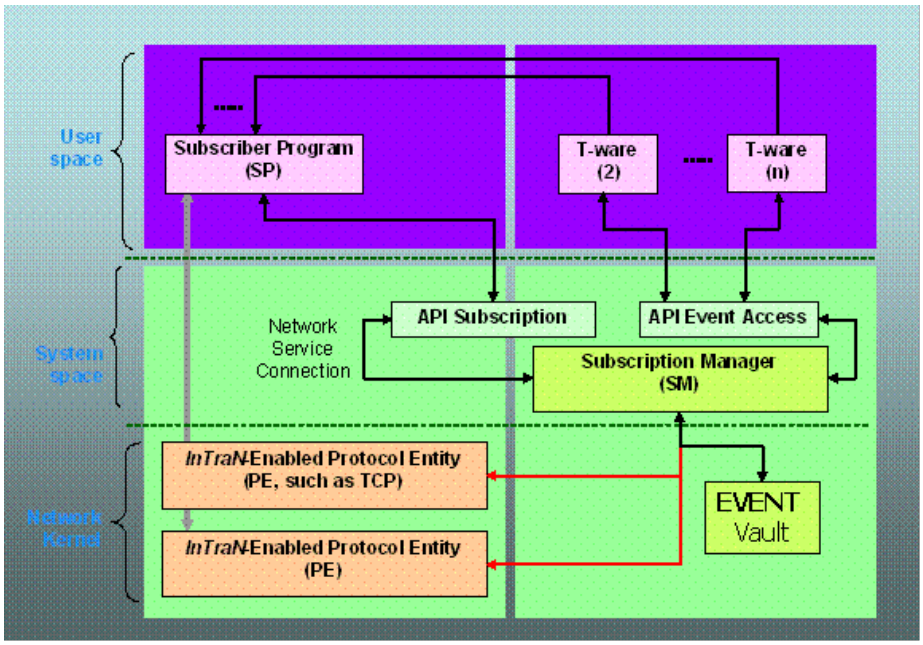

Figure 5.1: The architecture of **Project iTCP**, which adapts the FreeBSD 4.3 kernel-based protocol stack to generate TCP events. Applications (which are the subscriber programs in the model) subscribe to TCP events – these events are forwarded to the "*InTraN*-enabled protocol entity" (i.e. TCP) for processing. When an event occurs, such as a *retransmit timeout*, the in-kernel TCP implementation signals a separate user process, known as a *T-ware*, that probes the kernel for the relevant socket descriptor. The *T-ware* program then handles the original event.[32]

introduce *time distortion*, not ideal for time-sensitive data. As part of this application and network symbiosis, the application registers a subscription to a subset of TCP events, and to handle them, small *T-ware processes* are either supplied by the programmer or by a third party. These programs then alter the behavior of the subscriber program when the kernel network stack sends a signal with event data to the T-ware process.

Practical applications of the technology were detailed on their website and papers; they mostly involved a number of **symbiotic video and audio transcoders**. One demonstration of this was a *network-friendly adaptive MPEG-2 encoder*, which altered the stream bit rate based on the transport layer's perception of the level of network congestion. It was also used to implement a TCP extension, *FAST TCP*, without altering the core protocol code. It will be interesting to see in my project if this can be expanded to other parts of TCP.

The iTCP project may be the only implementation I found in my research, but other groups have had the idea of **interaction between the application and the network stack**. For example, Thekkath et al. in [62] advocate "exploiting application-specific knowledge to fine-tune a particular instance of a protocol". Other researchers, as early as 1991, (e.g. [63]) were noting that transport protocols should be specialized to the needs of a particular application.[1]

However, [62] proposes addressing this problem by **partially evaluating a general purpose protocol** such as TCP with respect to a particular set of requirements. Each application would use a slightly different variant of the general purpose protocol. They advocate the use of protocol compilers and language-based protocols to generate efficient specialized protocols – an approach that I did not consider in this implementation, but is perfectly possible with a userspace design. Developers using language-based protocols such as **Morpheus** [1] implement protocols by refining five base classes, deriving subclasses specific to the protocol.

In summary, event-based architectures for adapting protocols have rarely been implemented, and if so, have used heavyweight notification mechanisms, such as signals to processes located in separate address spaces. The specialization of protocols using language-based solutions transfers more of the burden onto the programmer for network usage optimization – however, such an approach is still perfectly possible in our userspace network stack. However, since event-based solutions are more dynamic, use only general-purpose protocols and offer more runtime feedback to an application, it seems a more fruitful avenue for generating adaptive and interactive network protocols.

5.1.2 Statistics, profiling and adaptation

In Bhumralkar et al. (2000) [7], they note that discovering bandwidth afresh each time for a host is extremely inefficient. The *bandwidth probing mechanism* used during slow start takes several round-trip time (RTT) estimates – if the transfer is small and the network has sufficient bandwidth, the transfer never moves into

[1] [62] also predicted that request/response protocols would co-exist with existing byte-stream protocols such as TCP, arguing that in systems that need to support both throughput-intensive *and* latency-critical applications, there is a good reason for both to exist.

the congestion avoidance phase, and so the duration of the transfer is only related to the RTT, rather than the actual amount of available bandwidth. Stating that studies show 85% of the packets transmitted by a typical high performance web server occur in the slow-start phase, (for example, [3]) which means the bulk of the transfer occurs while the connection is probing for bandwidth.

Their aim was provide an alternative to the TCP slow start algorithm, **network adaptive slow start**, so small files could be transferred much more efficiently. In their algorithm, they stores a history of the congestion window and the *smoothed round-trip times* (the average estimate over the course of an entire connection) at the end of previous connections. They also utilize the size of the file being transferred to choose the particular method of transfer (for large files, normal TCP is perfectly efficient), and use these values to estimate two values. First is the slow start time (*ss*), where $ss = R(log_2 W)$ (R is the round-trip time estimate, and W is the size of the congestion window in terms of the number of segments), then, using *ss*, we calculate the number of packets (and therefore the amount of data) we will be sending in slow-start, $maxpossible = (W \times ss)/R$

Bhumralkar et al. then use this *maxpossible* value to choose between using normal TCP for large file transfer (if $filesize > maxpossible$), and a modified slow-start algorithm for small transfers ($filesize <= maxpossible$). Combining that with the value of *ssthresh*, the window size value at which the connection moves into congestion avoidance mode, we can work out if we still have spare bandwidth and can increase the receive window size further.

However, there is no research on **persistent adaptive protocols**, because storing statistics and profiling data across connections is an option not feasible for kernel-based network stacks, which is where most of the adaptive TCP research takes place.

5.1.3 Asynchronous I/O

Asynchronous non-blocking I/O has always been a useful form of I/O that allows processing to continue while the I/O is being processed – I/O devices (especially hard drives) can be extremely slow (in a relative sense), and the application could always be performing other work while waiting for data. It makes sense to extend this metaphor to sockets, whose receive operation can block indefinitely if no new data arrives. As a result, socket AIO is already available in other operating systems, although how well is supported depends; Windows has supported asynchronous I/O for longer than Linux and its support is arguably more complete – in general, Unix-like operating systems do not support asynchronous I/O on sockets well.

My base for designing the asynchronous event interface was the Linux and POSIX AIO API. Incidentally, asynchronous I/O has been a problematic issue in the Linux kernel for many years, and is "difficult to use, incomplete in its coverage, and hard to support in the kernel".[14] However, most of the UNIX-like operating systems have begun to support the POSIX AIO specification. In FreeBSD, AIO support is not built into the default kernel and must be loaded as a module. In other Unix-like operating systems, such as Mac OS X, AIO support has only recently been added for file descriptor types other than disk files.[10]

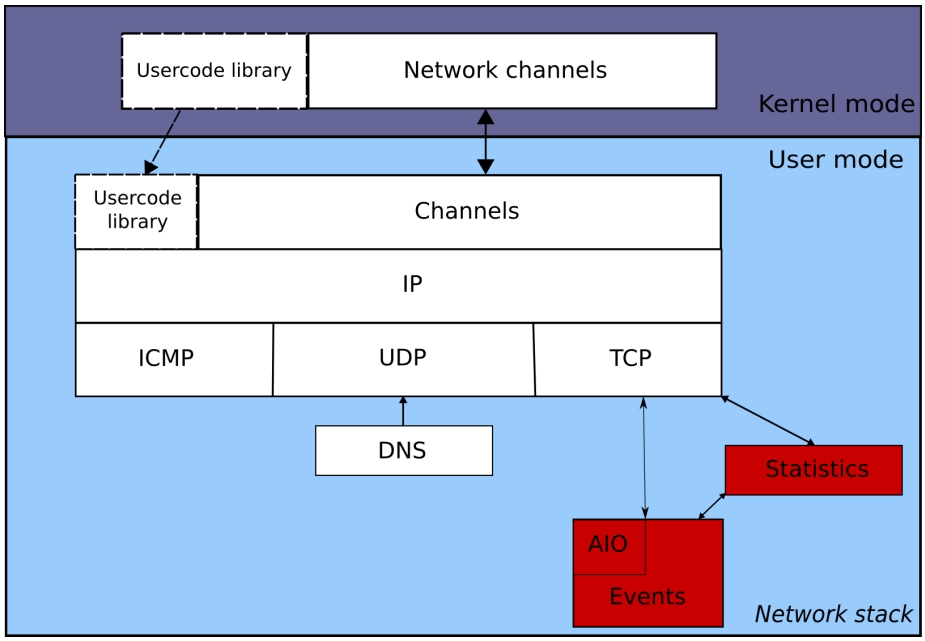

Figure 5.2: Architecture of the adaptable components of the network stack. Since TCP is a complex protocol, a number of TCP events have been created for applications to use, such as *retransmission timeout* and *dropped packets*, as well as an asynchronous I/O implementation.

In Windows, asynchronous I/O, referred to as *overlapped I/O*, has been supported for a number of years by the NT kernel. Developers use objects such as *I/O completion ports*, which provide a queue object and a set of threads to handle requests on the queue, to synchronize between threads.[40] Typically synchronous calls to `ReadFile` and `WriteFile` can be performed asynchronously by calling `CreateFile` initially with the `FILE_FLAG_OVERLAPPED` flag, and passing a pointer to an `OVERLAPPED` structure to subsequent I/O operations.

There is no mechanism for notifying applications of network stack events in either Windows, Linux or the BSDs. Since of all those operating systems employ kernel-based network stacks, the only method of signalling applications of events is through signals in the UNIX-like OSes and `WndProc` messages in Windows – there has not been any widespread adoption of the interactive protocols mentioned above.

5.2 Architecture

To enable the adaptability and interactivity features, there are three main components added to the network stack, as depicted in Figure 5.2. They are as follows:

- **Statistics and profiling.** The TCP implementation in the userspace network

stack now gathers a variety of statistics (such as the *total packets sent* to a particular port) and data (including the *round-trip time* estimate) about open connections; these statistics are stored in memory until the end of the program. If the application has specified an output file, we write the statistical data to a `.ns` file, which can be used for profiling the application's network usage. We then show how this network data could be used in the next run of the application in an efficient and useful manner.

- **Application feedback.** We can notify the TCP implementation of any upcoming data transfers with a known size (such as a large document being transferred over HTTP) to enable the various congestion control to adjust quickly in combination with previous profiling data about the host. We show how algorithms such as *network adaptive slow start* and algorithms involving other estimates can be optimized with the data, as well as how they extend the adaptability inherent in current byte-stream transport protocols such as TCP.

- **Event handling and asynchronous I/O.** Placing the network stack in userspace as a shared library allows to easily notify interested applications through the use of *event callbacks*. Combined with a mechanism for registering interest in events and submitting asynchronous I/O requests, we explore how applications could use this interactivity to optimize bandwidth usage and adjust their applications to suit network conditions.

5.3 Statistics and profiling

After consulting the adaptivity research, I realized that when a network socket is closed, lots of **useful context is lost** about the state of the network connecting the two hosts, including the information about the traffic profile and network conditions. This data has to be derived again and again, usually from estimates. Since IPv4 addresses (and many other protocol families and protocols that perform port multiplexing) consist of two elements, the address and the port, I came to the conclusion that the two represented different state spaces; there were certain estimates and statistics that held for hosts across all of their ports, and likewise for ports across all hosts. This meant, to make the statistics as reusable as possible, I would have to divide the collected statistics into two categories: *host* statistics and *port* statistics.

After storing this information throughout the lifetime of the program, I also considered that this data would be useful as **history** from previous runs. I could feed the collected data into sockets when they were created, and adjust their congestion control for maximum performance with algorithms such as *network adaptive slow start*. The information collected would also be useful for profiling the application's network usage via the `nprof` utility – tools such as `iptraf` exist, but it is not possible to feed the data collected in the program directly into the Linux network stack.

```
/>nprof /System/Temp/ftp.ns
There are 1 ports (20 bytes) and 3 hosts (52 bytes)
[Press any key to continue]

PORT 21
        flags = CLIENT
-------------------------------------------------
        total bytes sent = 32
        total packets sent = 3
        avg data bytes per packet sent = 10
-------------------------------------------------
        total bytes recv = 2265
        total packets recv = 10
        avg data bytes per packet recv = 226
-------------------------------------------------
[Press any key to continue]
```

Figure 5.3: Output from `nprof`, after running the `ftp` application several times. The port data could by the network stack to determine whether the the program focused on throughput (small amount of large packets) or latency (large amount of small packets) – however, it is unclear exactly how this information could be used to improve performance.

5.3.1 Categories

This section describes the mechanism by which we collect data about network usage and performance. The actual format for the network stack's persistent statistics file is described in Appendix A. In `libnetwork.so`, we **collect and store statistics** from TCP connections taking place that involve a well-known port (connections where the source or destination port number is below 1024). After we have collected statistics about a TCP connection and the connection is then closed (by calling `SocketClose`) by the application, the memory containing the actual TCP-specific state of the connection (`TcpSocketInfo`, which in turn contains the `TcpStatistics` structure) is not freed. Instead, it is written out, when the application exits, to a specified file in the above described format.

To enable efficient collection and use of collected data in subsequent runs of the network stack, we decided to divide the TCP statistics and the stored file into two main categories:

- **Port (protocol) statistics** These involve information about the packets sent and received using the connection. Statistics like `minSndPacketSize` and `maxSndPacketSize`, `totalBytesRecv` and `totalBytes` are useful for calculating the number of send and receive buffers needed on subsequent runs; application developers can inspect the statistics using `nprof` to find out the buffer size they should be using for optimum performance when calling `SocketRecv` (either as a result avoiding repeat calls to `TcpSocketRecv` or wasted memory for buffers that are much too large); information about the parameters passed to `TcpSocketSend` and `TcpSocketRecv` is stored and aggregated too.

- **Host statistics** These mainly focus on the actual network conditions during a connection, including the transport and retransmission functions and a measure of their success or failure. This can be used again if the application connects to that host again; however, to prevent very old data from being used, we include an expiry time in the host entry.

  Examples of host-based statistics include `droppedPackets` and `droppedBytes`, which count the number of packets dropped by the userspace network stack for invalid or out-of-range sequence numbers. `retxBytesSent` and `retxPacketsSent` is a similar statistics, summarizing the number of packets retransmitted due to timeout.

  Important host statistics that are immediately useful in adapting the TCP transmission algorithms include `rtt` and `rto` – an accurate estimate of the round-trip time and retransmission timeout value, which will not vary too much in the time between application runs. Using these values as an estimate will reduce the number of duplicate retransmissions and save network bandwidth. (Section 5.4.3)[43]

This organization of statistics is reflected in the format of the stored file written to disk at program exit, which aggregates information about all well-known ports used by the application and stores statistics about the *N* most recent port connections and hosts (as a *most-recently used cache* – useful if the application accesses the same host and port repeatedly) – *N* can be chosen by the user, who may want to balance the processing time cost at program and connection startup associated with many entries with increased adaptive performance coverage for TCP connections.

5.3.2 Use

The statistical data in the `.ns` file can be used by the developer in two ways. First of all, if the application specifies an output file for network data at runtime via `NetworkInitEx`, the data already present at that location will be **automatically incorporated into the adaptive components of the userspace network stack** – this is similar to how software tools such as `gcov`, the GNU code coverage tester, can use profiling information it collects to optimize the application by incorporating its findings when the application is next compiled using `gcc`. (However, our optimization and adaptability takes place solely at runtime).

A second use of the persistent statistical data is to manually **inspect the network usage using** `nprof`, the network profiler program I developed to complement the profiling mechanisms in the network stack. An example of its output is shown in Figure 5.3 – the per-port output provided especially illustrates how the program is using the network over a number of runs. One port may be sending very large packets, or packets whose variance is very large. This may be useful for developers wishing to break down and track their network usage by port and host to conserve bandwidth usage.

5.4 Adaptation

In this section, we describe how the statistics gathered above are used in the same and subsequent runs of the application to optimize TCP connections for maximal bandwidth usage. We also detail how the aggregate statistics are stored and quickly accessed to retrieve information about previous connections from a host and port tuple supplied to `TcpSocketConnect` or `TcpSocketBind`.

5.4.1 Current adaptive technologies

Current TCP implementations already employ algorithms for data transmission that adapt to the perceived congestion in the network. A number of interrelated algorithms are used for high performance and to avoid network congestion. They are:

- **Slow-start and congestion avoidance**. These are both transmission strategies; the aim of the pairing is to avoid *congestion collapse*, where network performance falls by a large amount due to an excessive number of packets being dropped. We try to avoid sending more data than the network is capable of transmitting.[2] The key object is a *congestion window* (see p. 67), and in *slow-start*, the protocol increases the congestion window exponentially in size before congestion is detected (or the *slow-start threshold* is passed) than afterwards, the *congestion avoidance* stage, where the *congestion control* algorithm, such as TCP Vegas, is employed.[57]

- **Fast retransmit and recovery**. This is a variation on the slow start algorithm – if we receive a number of duplicate ACKs (usually 3) for a single sequence number, we will assume that the next segment was lost in the network and quickly retransmit. We then reduce the congestion window size to the *slow-start threshold*, rather than the initial congestion window size.

- **Retransmission timeout**. One mechanism for retransmission is not based on duplicate ACKs, but on the estimated *round-trip time* (see p. 67) for the packets in a connection. The *round-trip time* essentially states the length of time we can expect to wait before we receive an acknowledgement; if we are waiting a great deal longer than this, we should retransmit. This *retransmission timeout* then retransmits the last packet not acknowledged by the receiver. These are all estimates, which is why previous data from persistent profiling data is very useful for more accurate estimation.

5.4.2 Application hints

Because the network stack operates at a lower level than the application (and unnecessary system calls tend to be avoided by the application, which could doing useful work instead of the system call overhead), the network stack and application do not exchange much information. This is undesirable, especially with large

[2] This *congestion control* is in comparison to *flow control*, where we avoid sending more data than the host is capable of handling.

file transfers, where, combined with the historical values of the TCP congestion window and the round trip time for packets, we know how much data we need to transfer and the size of buffers involved. Essentially, the application knows what the data transfer pattern will look in the immediate future, and the current implementations of network stacks are blind.

Take `httpd` for example. If the network stack knows in advance the file size that will be transferred, it can avoid the inefficiencies of the TCP *slow start* algorithm and help determine the degree of adaptability to network conditions that the transfer will need. A small file transfer over HTTP will experience a lot less variability in bandwidth compared to a large video or audio file being transferred (or streamed) over the same connection. The *network adaptive slow start* algorithm uses application hints, such as the file size, combined with historical data about the host to speed up the slow start mechanism. (The algorithm is currently not part of the TCP implementation, but is fairly trivial to implement given the supplied data)

5.4.3 Profiling data

Use of profiling data from previous runs is incorporated into various adaptable algorithms. Since many of the values TCP uses to adjust its algorithms are estimates, like the *retransmission* time, starting with a more accurate estimate will give the socket a more accurate idea. There are several ways in which the TCP stack uses (and potentially could use) data to improve performance:

- **Round-trip time estimates (host)**. All connections to the host could benefit from a more accurate estimate of the round-trip time – this means we have a much better idea of the state of the network between the two hosts (obviously the network and the network load could change between application runs, but the data is worth taking into account) and can use this to modify the retransmission timeout value, so we avoid unnecessary retransmissions in the slow-start process over a range of sockets in the same application.
- **Retransmission statistics and window sizes (host)**. We can gauge the congestion and reliability of the host from these measurements. These can be combined with the round-trip time estimates to gain an overall picture of the network congestion between the local and remote hosts in the past.
- **Packet sizes (protocol)**. It is not clear exactly how we can incorporate these statistics when the socket is created, and, as a result, they are ignored in socket startup. It may be useful in deducing whether the connection is throughput or latency-critical, based on the number of packets sent and their average size. One possible idea to automatically add the `PSH` flag for applications that sends many very small packets, as it is likely they are part of an interactive connection in a protocol such as `telnet` – currently the application must manually signal this.

When a TCP socket is created, the host and port linked lists (implemented in `stats_host.c` and `stats_port.c` in `user/sdk/network/stats/` respectively) are searched; if a suitable entry is found for either the host (in the list of

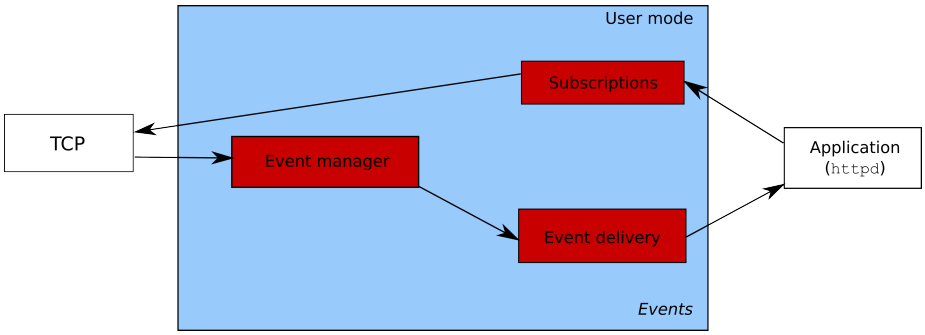

Figure 5.4: Diagram of the internal event component of the adaptable network stack. An ability to automatically log events would be useful for network stack and application debugging. There is no central storage of all events; each TCP socket has a list of asynchronous contexts per event which function as lists of *subscriptions*.

```
...
int SocketAsyncCreate(struct AsyncCtx* async, AsyncCallback func, int
    flags);
int SocketAsyncAdd(struct AsyncCtx** head, struct AsyncCtx* item);
int SocketAsyncCancel(struct AsyncCtx* sync);
int SocketAsyncReturn(struct AsyncCtx* async);
int _SocketEventFireList(Socket* socket, struct AsyncCtx* head, void* data
    , int length);
...
```

Figure 5.5: Some of the functions in the socket event API. Many applications call other APIs, such as AIO (Section 5.5.3) or the TCP event API (Section 5.5.2), that wrap around these event primitives and provide automatic storage or differing semantics.

recently accessed hosts) or the port, then the relevant group of statistics is found and incorporated into the `TcpSocket` structure – for example, the round-trip estimates are copied from the `HostEntry` structure to the `TcpEntry` structure, so we already have a reliable estimate of the value of *rtt* and *rto*.

5.5 Events

Events are asynchronous notifications generated by a protocol implementation in the stack. Interested applications register for a certain event or class events using a *asynchronous context*, a structure usually allocated on the stack or globally, and supply a callback called when the event fires. The idea behind events is to provide interactivity primitives to the application so that the application can base its network output on network conditions – they are a form of messaging coupling enabled by the fact that both the application and network stack are located in the same address space and privilege level.

5.5.1 Design

There were a number of design choices for the asynchronous API – the library code can be hard to write, and we would like to avoid potential resource conflicts and their associated failure conditions. For example, one potential problem with *callback functions* that we may have is the depth of the call stack growing unmanageably; if a event handler registers interest in another event and that event fires immediately, we could have problems with procedure call recursion if the stack context of the first callback is not unwound. This proved not to be a problem in the implementation.

We also wish to minimize the latency from a packet being received or sent successfully (or a TCP event firing) to the application handling it. This means we must find an efficient way of delivering the notification, performing as little work as possible without introducing synchronization issues and balancing the load so the events of one socket are not prioritized above another. Figure 5.4 shows the event subscription and notification architecture in the network stack.

I will summarize the major issues tackled in developing the asynchronous event code:

- **Repetitive or single-shot events**. I chose to leave this particular policy decision to each class of event. There should be no overall semantics dictated by the generic event handling code. Certain classes of events, such as an *asynchronous send*, should only be fired once; once the data has been sent, the event associated context is no longer needed as the event will not fire again.

- **Subscription**. How should interested stakeholders register for events? Most importantly, how should we handle storing event subscriptions and accessing them later? One possibility is a set of dynamic arrays in a central event subscriber component, but I eventually decided on a more distributed mechanism, with protocol code storing per-socket linked lists (updated by a common API for both the protocol) that stored a series of `AsyncCtxs` (a structure that served as an abstraction for an *asynchronous context*), which were allocated and registered by the application by calling event functions categorised by protocol. Removal and cancellation of subscriptions was then handled by the generic `Async*` API.

- **Event delivery**. Once notifications are generated by the network, how should they be delivered to subscribers? One design is to call the event functions as the events are generated by the network stack (in the same thread context in which we receive the packet) – however, this has stability and latency issues if an event callback hangs or the set of event handlers perform a large amount of processing. In the end, I opted for a separate global *event worker thread* to handle events, with an option for the application to create a *worker thread per socket* if needed.

- **Event data**. Should we supply a standard array for each event, or let events pass their own data to handlers? In the end, I chose the latter; since, in the

event delivery mechanism, I wanted to avoid unnecessary copying of general structures, I let each event pass its own data to handlers; in the main event-firing function, `_SockEventFireList`, I then copy the data (given the length) to each asynchronous context as part of the standard `AsyncCtx` structure, which was linked to the `Socket` the event fired on. If the event handler required more data about the socket, it could retrieve the information using the `Socket` pointer.

- **Time-based events**. Should we support timer events that update an application on the status of a socket periodically? These kinds of events are useful for diagnostics, but the overhead in managing many events using kernel timers means a clever global situation involving one kernel timer and the ability to register timed events from different classes was the preferred solution. However, I did not focus on these timer events, because they were not directly linked with the concept of a protocol event itself and therefore not the object of my research – these are *periodic updates* of socket state, rather than *notifications* of socket events.

5.5.2 TCP

I decided to focus on implementing a set of events for TCP and demonstrating their applicability applications such as `httpd`. As TCP is the most complex and widely-used protocol in the Internet Protocol Suite, the flow control and congestion control algorithms in TCP are vital in preventing data overwhelming the receiver[3] and congestion collapse – **feedback to the application** about when the host or the network is becoming congested or uncongested would allow the application to adjust its data transmission accordingly (e.g. adjusting the quality of streaming media). In a very congested network, it may make sense for the application to perform other work in the meantime, or display a warning message indicating the network conditions.

The events I imagined would be useful to applications, for diagnostic and performance reasons, are detailed below. This list is from the events available in `user/sdk/network/tcp/tcp_events.c`:

- **Retransmission timeout** This event is fired when the *retransmission timeout timer* expires. This is usually a reliable sign that the network is congested, and passing the current retransmission timeout value, the next one and the current estimate of the round-trip time should allow the application to judge whether it is more useful to perform other work (such as reading more bytes from a file on disk in a file transfer) and throttle its data flow accordingly.

- **Round-trip time change** This event is related to the *retransmission timeout*. Depending on the exact congestion control algorithm used, the round-trip time will be sampled regularly or rarely. This is the best possible estimate

[3] *Flow control* is not always a function of the host; the host may have a heavy traffic load from other networks, and so the receive window, the main abstraction in flow control, may be a function of congestion on other networks.

of network latency, which may be useful to latency-critical applications that use TCP connections – although many such applications use UDP combined with a custom application protocol, streaming video on web pages such as YouTube often utilizes a TCP connection to deliver data. If the round-trip time goes above a certain threshold, we can adjust the data rate accordingly to try and lower it again.

The only problem with this event is that it may fire rapidly (as certain congestion control algorithms such as *TCP Vegas* incorporate timings from every packet into the sample) – the ability to sample the round-trip time after every *N*th measurement would be useful for the general event load, but is currently not implemented.

- **Dropped packets** Obviously we cannot detect dropped packets if they are dropped during transmission through the network, but there are several cases in TCP where we may have to drop packets at the host. These include *out-of-order* packets, *duplicate* packets and packets with *incorrect checksums*. All of these events can be used to gauge network reliability (for instance, the error rate of transmitted packets in wireless networks is much larger than wired networks) and congestion. This is also a useful diagnostic tool, and can be adapted to other transport protocols that use checksums, such as UDP.

- **Local window change** This event is fired when the *window size* we advertise to the remote host changes. Although we should not allow the application to set the receive window directly, an idea of how much data the TCP socket can receive is useful. We would like to receive as much data as possible to process in one `SocketRecv` call, but avoid allocating buffers that are too large and will never be filled.

- **Remote window change** This event is fired when the *window size* advertised by the remote host changes in value. Using this, we can perform application-level flow control and deal with overwhelmed hosts at an application level – we can do other work while the size of the remote window is small, and restart transmission when the window size becomes larger, to help avoid problems such as *silly window syndrome* at an application level, where the receive window setting becomes so small that the data transmitted in each packet is smaller than the packet header. This is preferable to forcing the TCP socket to cope with a burst of data and the transmission backlog that follows.

Subscriptions

After a socket is created, we can register for an event using its corresponding registration function. This applies to TCP events and asynchronous I/O. For example, to **register** for the *remote window change* event, we execute the following steps:

```
int RemWinChanged(struct AsyncCtx* ctx)
{
        /* Handle event */
}
```

```
...
struct AsyncCtx ctx;
...
TcpEventCtxCreate(&ctx, RemWinChanged);
TcpRemWinRegister(child, &ctx);
...
```

1. We call the `TcpEventCtxCreate` function, passing a `TcpEventCtx`, which wraps around a `SockEventCtx` object and provides storage for a range of TCP events. The caller is responsible for allocating the `TcpEventCtx`. We also pass a callback function, `RemWinChanged` for example, whose only parameter is a `AsyncCtx` pointer – this is the *event handler*. `TcpEventCtxCreate` sets up the context structure, but does not register the context with any socket.

2. We attach the event context, represented by `TcpEventCtx`, to a socket by calling the relevant registration function for the *remote window change* event, `TcpRemWinRegister`. The TCP implementation adds this to the linked list corresponding to the event handlers for the structure. At this point, the event could now fire.

To **cancel** a TCP event subscription, we call `TcpEventCtxCancel`, which actually just calls `SocketAsyncCancel` on the internal `SockEventCtx` object. The mechanism of asynchronous event **delivery** is covered in Section 5.5.4.

Other protocols

It will be hard to asynchronously deliver events if the protocol implementation does not asynchronously receive packets (that is, does not handle receiving packets in a dedicated thread, like the TCP implementation) – event notifications would only occur once the application synchronously receives a packet. The datagram protocols, such as UDP and ICMP, are also rather simplistic; there would be few important performance-related events available. The only obvious class of events would a *checksum failure* event, although that would be more useful when considering diagnostics rather than performance.

One possibility is introducing the idea of a **UDP "connection"**. This would allow for the automatic timing of packets in the "connection"; an automatic estimation of the round-trip time taken for packets would be useful for applications, such as *online role-playing games*, that require a low latency connection. With an estimate of the round-trip time, we would then be able to introduce the *round-trip time change* event for UDP – typically applications perform their own *ping time* measurements. Adding socket operations for sockets using datagram protocols that are meant solely for the benefit of the local network stack, such as those described above, could be a fruitful avenue to pursue for multimedia applications.

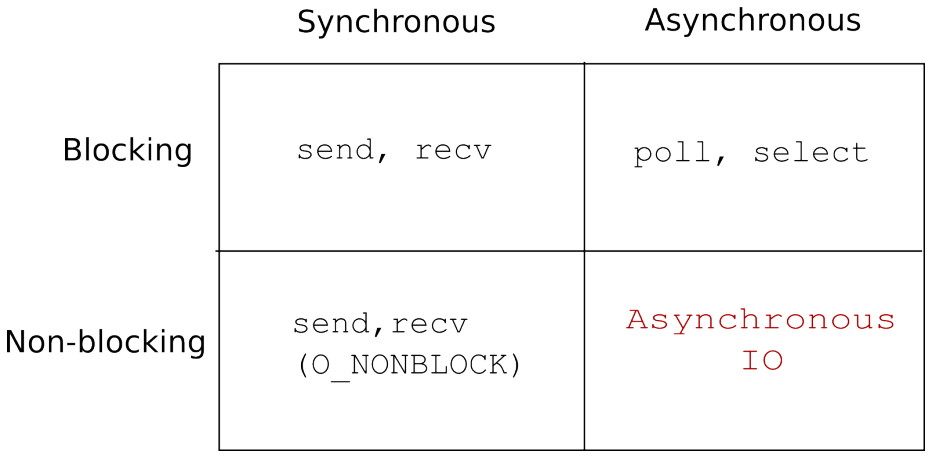

Figure 5.6: Asynchronous I/O in the context of sockets and how it fits into the I/O model of UNIX-like operating systems. Whitix's functions are similarly named.

5.5.3 Asynchronous I/O

Overview

Figure 5.6 depicts the four methods of performing I/O in operating systems today. In existing programs the most common model of socket interaction is the **synchronous blocking I/O** model, where the userspace application performs a system call such as `send` or `recv` on sockets and `write` and `read` on files; the file is represented on Unix-derived systems as a single file descriptor.[31] In a single-threaded application, no work can be performed while the system call waits for data to complete (or a timeout occurs). In a multithreaded application, other threads can still perform work, but this leads to synchronization issues if multiple threads access the same data. Cross-thread communication might also be difficult each time a blocking system call could occur.

Synchronous non-blocking I/O is also possible (see the `O_NONBLOCK` flag); however, this can be inefficient if the application has to busy-wait until the data is available anyway – the time between data becoming available to the file descriptor and the application calling `read` or `recv`, especially in the case where it decides to perform other work for example, will be non-zero, leading to increased latency and decreased throughput.

For networked applications with unpredictable I/O needs, especially ones where data arrives in an irregular fashion, asynchronous I/O is very useful. As Figure 5.6 shows, there are two types of asynchronous I/O, blocking and non-blocking. **Blocking asynchronous I/O**, `SysPoll` in Whitix, is useful if the application's work only involves reacting to events on multiple file descriptors; one class of examples is interactive applications like a command-line shell (`burn` in Whitix), where work is only performed upon user input. (There are several issues with `SysPoll` interacting with the userspace network stack and network channels - these are covered in

Section 4.4.6 - but in short we can assume that the semantics are similar).

The last I/O model is **non-blocking asynchronous I/O** (usually referred to as **AIO**, with blocking asynchronous I/O known as *multiplexing*). In this model, application post a request, either to send or receive data, to the network stack, supplying information about how to be notified upon I/O completion. The application is then free to continue other work while the background operation completes. A thread-based callback, usually performed by an *AIO worker thread*, can then be generated to complete the I/O transaction. This model is available with a wide range of configuration options, such the mapping of AIO worker threads for each socket to an I/O request. The use of multithreading, handled by the AIO layer, can really help an application take advantage of multiple processors.

AIO and the network stack

Only the *asynchronous receive* operation is available in this implementation of the userspace network stack. I chose not to implement *asynchronous send*, but to focus on other TCP events instead – the two operations are very similar to other events anyway. To register for an asynchronous receive event, the application performs the following steps:

1. The application allocates an instance of the `AsyncCtx` structure.
2. `SocketAsyncCreate` is then called, passing the asynchronous context and a callback function to handle the asynchronous receive event. This callback function is called every time a packet is received on the connection.
3. The application then calls `SocketAsyncRecv` to register the context to a socket, passing the data buffer (also allocated by the application) and its length. The asynchronous event could fire at any point on this socket as a result. Only *one* asynchronous receive context can be registered at one time to handle packets – this is because a *packet receive* event is fairly common and can involve a lot of data copying, limiting the asynchronous receive events to one context reduces the global event load greatly (and a well-designed application should only need to have one copy of the data, like an synchronous receive).

In the TCP implementation, the **asynchronous receive event fires** in `TcpEstablished`, the main data receive function, when we receive a packet containing the data. If the socket has registered an asynchronous context for the receive operation, then we call `_SockEventFireList`, which distributes the event to the subscriber as described in Section 5.5.4.

5.5.4 Event delivery

Mechanism

Once we have an event has occurred, we need to deliver a notification to the event as quickly as possible through the use of *event worker threads* to call the specified

callbacks; there is at least one global thread, and possibly a thread for each socket. The generic internal function for this is `_SockEventFireList`, which takes the following parameters:

```
int _SockEventFireList(Socket* socket, struct AsyncCtx* head, void* data,
    int length);
```

where `socket` is the socket where the event has occurred (we may want to use its private event worker thread), `head` is the linked list of contexts (with callback functions) that we wish to invoke, and `data` and `length` describe the event data, which is different for each type of event.

`_SockEventFireList` locates the appropriate event worker thread, and copies the data to each context's buffer, setting each context as *active* (i.e. in the process of being invoked). The function then signals to the worker thread that the notifications are ready to deliver by appending to the worker thread's list of waiting events and signalling the presence of work to the thread (resuming the thread if necessary). The *waiting event list* (the shared object) is appropriately synchronized between the two threads.

The worker thread (which runs the `SocketEventWorker` function) initially suspends itself. Once it resumes and discovers there is work, it copies the waiting event list by copying the head pointer (this is the critical section) – it then iterates through the list, calling each callback function (unless the context is marked as *cancelled*). Once there is no longer any work (the pointer to the list of waiting events is NULL), the thread sleeps until more work is ready.

Latency concerns

With multiple threads, especially on a uniprocessor, the delay, or *latency*, between posting an event or AIO response to an event delivery thread and the worker thread being scheduled by the processor may impact on the application's performance. This latency may be the result of extra event worker threads being active at once, along with a thread scheduler that is focused on throughput, rather than latency. The **load balance** between the main application's thread and the worker thread may not always be even as a result; CPU-intensive parts of the main thread, where it does not spend any time waiting for I/O and uses up its entire timeslice, may cause a backlog of events in an event worker thread if it is never scheduled (this includes the socket poller thread may also take or require a large portion of CPU time with high network traffic). High latency is of concern to responsive applications using asynchronous I/O.

To illustrate this, I ran a short experiment on an otherwise quiescent uniprocessor system, noting the number of cycles (using `rdtsc`, see Section 6.2.3) that passed between the event being fired and delivered in an asynchronous context and, in synchronous blocking I/O, the packet being made available on the list of socket data and the application copying the data to read. The results are available in Figure 5.7. This shows that currently, as the number of processors per system is fairly low, latency is still an important issue. If we do have enough processors

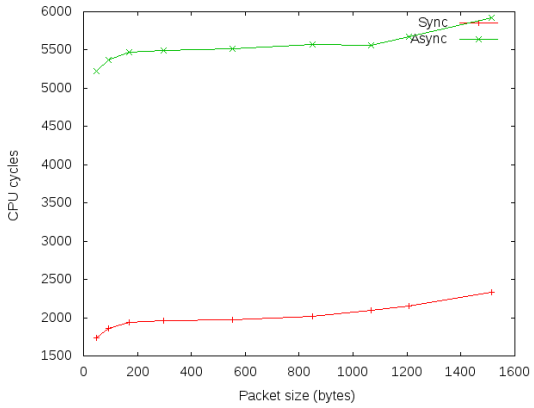

Figure 5.7: The cycles consumed in the system between the received data arriving at the socket and being delivered to `telnet`. `telnet` has the ability, due to the asynchronous properties of the protocol, to receive packets in a non-blocking synchronous or asynchronous manner. The synchronous I/O has less active threads running at once, which may be the issue in the asynchronous I/O's increased latency.

so the event worker thread of a process is always running, we would have a much lower average latency per event.

One solution to this latency when there is a large backlog on the global event delivery thread is to **spawn a new event delivery thread** for a particular socket. This is a simple measure that I have incorporated into the event API: sockets with many events can call `SocketEventCreateThread` to spawn a separate worker thread at any time – however, only at most one event worker thread can be spawned per socket.

5.6 Discussion

Overall, one of the main advantages of a userspace network stack is being able to provide much closer coupling with the application to improve performance. By providing interactivity and adaptability through **events** and the **networking profiling** functionality, this implementation proves it is possible to use events in a constructive way to couple the application and network stack.

However, I believe this to be a proof-of-concept implementation, there is plenty more scope for other data to be incorporated into the TCP/IP stack, but it requires inventive uses of the stored statistics in order to realize performance gains. Most of the data stored in the network statistics, especially the port-related data, is either not used by the network stack when a related socket is next created, or it is not used to its full potential. I would investigate further the possibility of storing network data in a **separate service** that stores and analyzes socket data – this would provide information about other hosts across the system and avoid partitioning useful

information into per-process network statistic files.

More testing, especially **stress testing**, should also be performed on the event layer. The only tests we have performed with sockets have involved at most 4 or 5 sockets – it would be interesting to see if the event concept was still viable with many sockets (say at least hundreds) generating many events. A wider range of TCP events (along with several UDP events), a more specific language to specify subscriptions, and perhaps introducing composite events would increase the adaptability of the network stack. More research into what information the application can provide the TCP stack (apart from crude notifications such as `TcpSetTransferSize`) would also improve stack performance by incorporating more types of information.

5.7 Summary

This chapter covered how we used the userspace architecture of the network stack to add innovative new features, such as *protocol events*, *application hints*, *persistent profiling information* and *asynchronous I/O*, as part of achieving the goal of **feedback** between the application and network stack. After examining the **current research**, we noted that contemporary event-based network stacks signalled events using heavyweight processes, no other network stack kept persistent statistics, and asynchronous socket I/O was not well supported across all operating systems. We then covered how the network stack **collected statistics**, stored them to disk after program exit, and used the output data to *optimize future runs* or allow the user to inspect *networking data by host and port* using `nprof`.

Moving, we outlined the techniques used for **adaptation** in the network stack; after summarizing current adaptive technologies present in TCP, we explain how the application can influence the network stack's operation through **application hints** and **profiling data**. In this relationship, data also flows the other way; network stacks post **events** – several TCP events now available are described, along with the means of **subscription** to these events. Finally, we covered **asynchronous I/O** and the means to deliver all these events, before discussing storing these statistics in a centralized service to share with other applications, among other things.

Chapter 6

Performance

We now evaluate the completed network stack implementation. In this chapter, we begin by quantitively comparing the implementation against other similar operating system network stacks, using a series of benchmarks.

When inspecting the performance of the network stack implementation, it is important to note that there is no comprehensive performance measurement (and non-invasive) framework for the Whitix operating system to measure cache-related performance (such as cache misses) or support for high-performance timers. In spite of this, we used a number of existing timers to measure performance, mainly metrics involving CPU cycles (latency is a derived function of this) and throughput.

6.1 Background

Research into what constitutes high networking performance and the benchmarking of various network layers is widespread and involves all layers of the network subsystem[1] to optimize the operation of individual components, with much of the researching focusing on TCP and the various congestion control algorithms. There are a number of studies evaluating the performance of the **overall networking subsystem**, which, due to time constraints on the project, is the most feasible method of obtaining the most performance information within a general framework.

Perhaps the most comprehensive study is "Performance Characterization of the FreeBSD Network Stack" by Hyong-youb Kim and Scott Rixner (2005) [33]. They analyzed the behavior of high-performance web servers in the FreeBSD 4.7 operating system along three axes: packet rate, number of connections and communication latency. They used low overhead non-statistical profiling by employing the **performance counter events** available in all modern CPUs to count events such

[1] See http://www.csm.ornl.gov/~dunigan/netperf/netlinks.html for a directory of network performance related links.

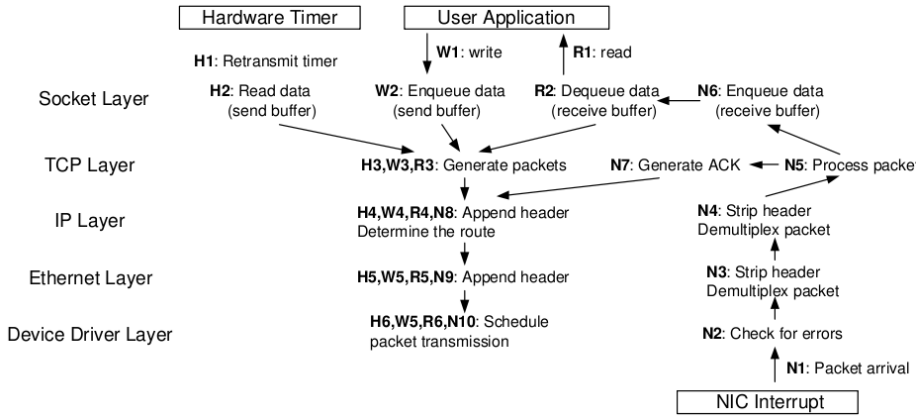

Figure 6.1: Figure of tasks generally performed by the network stack. There are three main actors in the system: the hardware timer, the user application and the network interface card (through its interrupt routine).

as L2 cache misses[2] and TLB misses. Such misses involve an access to main memory, and since memory latencies are now equivalent to several hundred processor cycles, the amount of time processors must now wait for main memory data affects throughput.

This project is not the first benchmark of **network channels**, though it is the first within a wider and more complete framework of a multithreaded userspace network stack. In 2008, Evgeniy Polyakov tested his implementation of network channels and a proof-of-concept userspace network stack. His bandwidth figures and CPU usage figures were impressive (see Figure 6.2). Using a third of the CPU time, it achieves three times the maximum bandwidth when sending small packets.[13] The difference is less clear when sending or receiving large packets – Polyakov has not implemented zero-copy I/O in his network channel patch.

6.2 Method

6.2.1 Caveats

Because there is no other implementations of a TCP/IP network stack on Whitix,[3] it will not be possible to perform a direct comparison of network stacks using the same operating system. This means only a comparison against the network stacks of other operating systems, such as Windows, Linux and the BSDs, is possible. There are caveats to this however.

[2] In the tested CPUs, an L2 cache miss would result in an access to main memory. The addition of an L3 cache, available in most modern multicore CPUs, would affect this particular measurement

[3] The closest equivalent is a simple implementation of local sockets. That probably can now be replaced with a local implementation of network channels, or *data channels*

Sending			
Object	**Packet size**	**Bandwidth**	**CPU Usage**
Netchannels	128	27-28 MB/sec	20-30%
Sockets	128	7-8 MB/sec	80-90%
Netchannels	4096	27-28 MB/sec	20-30%
Sockets	4096	30-31 MB/sec	30-40%
Receiving			
Netchannels	128	70-71 MB/sec	80-90%
Sockets	128	24-25 MB/sec	80-90%
Netchannels	4096	73-74 MB/sec	80-90%
Sockets	4096	79-80 MB/sec	80-90%

Figure 6.2: Evgeniy Polyakov's benchmark of network channels versus Linux kernel sockets. His implementation of the network channel I/O copies the data to and from userspace. It is likely the cost of copying the packets into and out of the kernel is the dominating factor for large packets.[13]

Much of the performance benchmarks concerning the network stack, especially a multithreaded one as in Whitix, also **indirectly measure the performance** of other operating system components. For example, performance benchmarks of a multithreaded web server would indirectly measure the throughput and latency of the filesystem, disk I/O and thread scheduler components. However, recent studies have shown that a modern, high-performance web server spends over 80% of its time within the network subsystem of an operating system,[33] and, so long as the rest of the system is quiescent and the served files are cached in main memory, the difference should be minimal. If in general this is not the case, we can perform relative comparison in terms of performance gains in the Whitix network stack against network stacks in other operating systems.

6.2.2 Procedure

A number of test were performed with the `echoserv` implementation available in Whitix, which echos back packets it is sent. Depending on a startup option, it creates a TCP or UDP server socket; for TCP sockets, it echos back any packets it is sent until the connection is closed. Unfortunately, unlike other operating systems, there is not a comprehensive framework available in Whitix for performance evaluation and profiling; I have not developed an API for retrieving the performance counters available in modern CPUs, unlike other operating systems.

To measure system performance, I measured the **cycle counts** for each operation. I measured the different layer counts separately and varied the packet size, (from the smallest possible to the largest possible packet size on a standard Ethernet local area network) keeping all other variables constant. The system was otherwise quiescent, with no other connections and no firewall rules added. I measured the best-case operation, even though there are many slower paths through the transmission and receive code.

For the *send* operation, the different stages of packet construction and transmission, as depicted in Figures 6.4, 6.5 and 6.8, are:

- **TCP/UDP.** The cycles spent in userspace constructing the packet to transmit. Timed from the `SocketSend` method for each socket type. For TCP, I made sure to only trigger the immediate send path, (i.e. not a path where the packet is added to the transmission list) where the data buffer is sent as a single packet using `TcpDoSend` during the call to `TcpSocketSend`.

- **IP.** I timed both the IP header construction performed in userspace (in `IpBuildHeader`) and the header modification (in `Ipv4Write`) and added both counts to obtain the total cycles spent building the IP header and calculating the packet checksum.

- **Ethernet.** I timed `EthBuildHeader`, which includes a call to `ArpGetLinkAddress` – I ignored any outlying figures that suggested a ARP request was sent by `ArpGetLinkAddress`, focusing instead on the common case where the hardware destination address would be found in the cache.

- **Driver.** I timed the `NetDeviceSend` function. Obviously the time spent in the driver will differ between Linux drivers, but a rough idea of how long transmission takes in the driver will indicate whether or not it is the dominant factor in transmission.

For the *receive* operation, depicted in Figures 6.6, 6.7 and 6.9, since normal header parsing did not take place in the kernel, I decided to divide the receive operation into a different set of layers:

- **Inspection** This is all of `NetRecv` the majority of `Ipv4RecvBuffer`, where the important contents of the IP header, such as the source and destination port for UDP and TCP, are copied to temporary variables for the matching process.

- **Matching** This is the cycles taken to match a packet with a channel, using the protocol hash function and `ChannelSearchList`. Since the system was quiescent, the only operation that is really timed is the hash function (since the only channel will be at the head of the list). To improve profiling, I should investigate under different system loads.

- **Copying** This is the cycles it takes to find a free receive buffer, set up the header and copy the data from the temporary buffer to the header. The majority of the cycles should be spent copying the data.

- **TCP/UDP** The overhead for the userspace network stack to receive the packet. Note for TCP this is the cycles it takes to return from `SysPoll`, read in the packet using `IpRecvNb`, and place any data in the packet on the `data` list for the TCP socket.

6.2.3 Timers

As Whitix lacks a profiling framework, this means that the only two means of measuring performance are the Intel 8253 **programmable interval timer** available in modern CPUs, and the **Time Stamp Counter** (TSC), a 64-bit register that counts the number of ticks since reset and is present on all Intel processors since the Pentium. There are disadvantages with both methods. With the PIT, the timing resolution on Whitix (since it is used for the scheduling quantum) is relatively coarse-grained, at 10ms – this means that events shorter in duration than 10ms will be not be timed. If we do repeat it many times, this also means the approximation error of the sample will be quite large.

As an alternative, the TSC initially appears to be a convenient method of retrieving CPU timing information. In the age of multicore and hyperthreaded power-saving CPUs and hibernating operating systems (lowered CPU clocks due to power saving is a concern with operating systems other than Whitix), the TSC can give varying results. However, since we are essentially running the benchmarks with the system at maximum performance, hibernation or low-clocked power-saving CPUs should not be an issue. The varying *cycle count* of the TSC (which due to out-of-order processors scheduling the execution of the `rdtsc` at different times) can be ignored by averaging the results.

6.3 Experimental testbed

I ran the benchmarks described in the previous section in two environments. Whitix was run off a CD-ROM drive and requests were run before benchmarking to ensure that both the programs and the pages to be benchmarked were available in memory – as a result, hard-drive and CD-ROM-drive speed were irrelevant when benchmarking. They were:

1. **Development laptop**. Intel Core Duo T2300 processor (1667 Mhz), 1GB of DDR2 SDRAM with a Broadcom BCM4401-B0 100Base-TX Ethernet interface, connected to the local area network by a 100BaseT Ethernet connection. The Core Duo processor has a unified L2 cache and separate L1 instruction and data caches. Each L1 cache in the processor is a two-way set associative 32KB cache with 64 byte lines, and the unified L2 cache is a shared 2MB cache with 64 byte lines.

2. **Virtualbox** virtualisation environment. This was the main machine used for testing the Whitix networking subsystem. On my development laptop (the above machine), the software emulates a 1600Mhz Intel processor with 256MB RAM and a AMD PCnet-PCI II (Am79C970A) Ethernet interface, sending packets over the Ethernet interface described in the previous setup.[4]

[4]Because of the way virtual machines run guest operating system code on the local computer, the virtual machine interacts with the host system's cache in a complex and involved way.

```
cycles = rdtsc();

for (i = 0; i < TIMES; i++)
        ret = device->ops->send(device, buffer);

cycles2 = rdtsc();
KePrint("%u, %u cycles\n", buffer->length, (DWORD)(cycles2-cycles)/TIMES);
```

Figure 6.3: `NetDeviceSend` annotated with cycle measurements. To average results, we sent `TIMES` packets to take the mean of the cycle timings, including outlying measurements in the final figures.

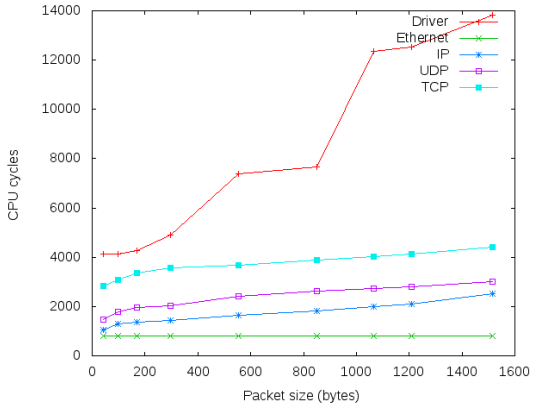

Figure 6.4: **Transmit cycles on VirtualBox**. The VirtualBox cycle counts (using `rdtsc`) for the various stages. Since the network device is emulated by VirtualBox, the driver cycle counts will are not an accurate representation of the real timings for the device.

Although VirtualBox may seem like a poor benchmarking environment, measuring performance with hardware virtualisation is important, because such environments are possibly how many will first evaluate the Whitix operating system, as well as being a popular way to run many operating systems.

6.4 Analysis

The results depicted in the supplied graphs and tables are useful for judging the CPU time taken between different layers of the networking subsystem. Although not directly useful as a comparison between operating systems or environments (or at least not as useful as instruction counts, cache misses or latency timings would be), we can judge what the **limiting factor of optimization** may be, and compare how long each network stack spends in each layer, comparing with studies such as [23] and [33].

From the results above, we can determine which layers are constant with re-

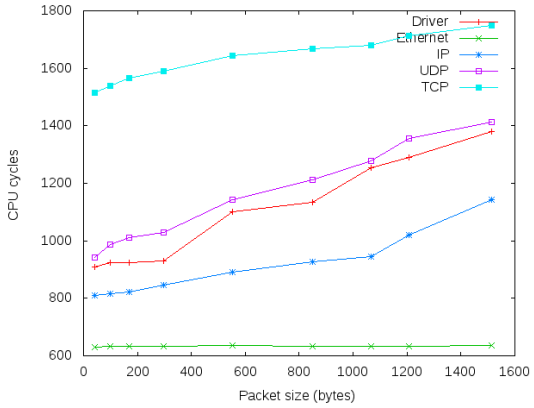

Figure 6.5: **Transmit cycles on laptop**. The laptop cycle counts (`rdtsc`) for the various stages.

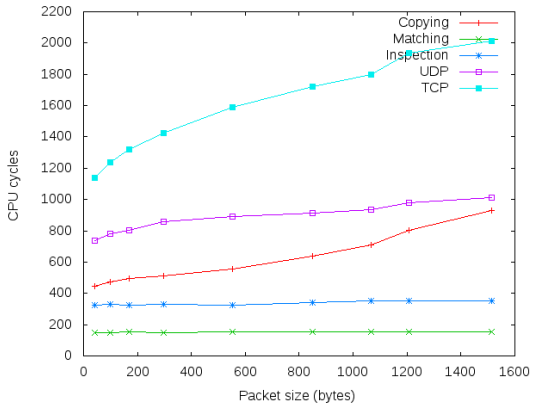

Figure 6.6: **Receive cycles on VirtualBox**. The VirtualBox cycle counts (`rdtsc`) for the various stages.

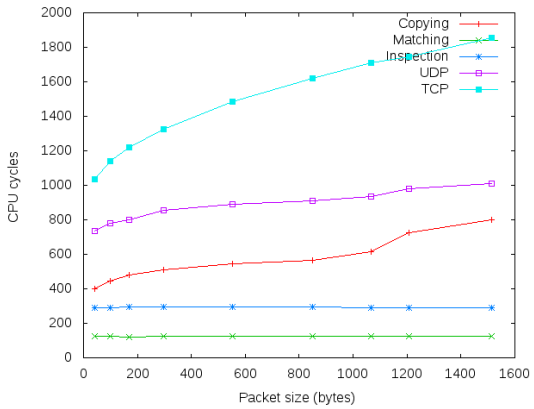

Figure 6.7: **Receive cycles on laptop**. The laptop cycle counts (`rdtsc`) for the various stages.

Environment	Packet size	Driver	Ethernet	IP	TCP	Total
VirtualBox	42	4117	804	1043	1454	7418
		55.5%	*10.8%*	*14.0%*	*19.6%*	
	850	7658	805	1810	2610	12883
		59.4%	*6.25%*	*14.0%*	*20.3%*	
	1514	13815	805	2532	3021	19543
		67.5%	*4.12%*	*12.3%*	*15.5%*	
Laptop	42	910	631	810	1522	3873
		12.3%	*16.3%*	*20.9%*	*39.2%*	
	850	1134	632	926	1669	4361
		26.0%	*14.5%*	*21.2%*	*38.3%*	
	1514	1381	635	1142	1750	4908
		28.1%	*12.9%*	*23.3%*	*35.7%*	

Figure 6.8: Summary of the **cycles consumed sending** small, medium and large TCP packets on Virtualbox and the development laptop. The two sets of figures are not comparable between environments generally.

Environment	Packet size	Inspection	Matching	Copying	TCP	Total
VirtualBox	42	361	183	443	1154	2141
		16.9%	*8.55%*	*20.7%*	*53.9%*	
	850	360	184	638	1757	2939
		12.2%	*6.26%*	*21.7%*	*59.8%*	
	1514	362	183	930	2004	3479
		10.4%	*5.26%*	*26.7%*	*57.6%*	
Laptop	42	290	123	398	1036	1847
		15.7%	*6.66%*	*21.5%*	*56.1%*	
	850	294	123	564	1621	2603
		11.3%	*9.00%*	*21.7%*	*62.3%*	
	1514	290	124	798	1855	3007
		9.64%	*4.12%*	*26.5%*	*61.7%*	

Figure 6.9: Summary of the **cycles consumed receiving** small, medium and large TCP packets on Virtualbox and the development laptop (in the best case). The driver cycles spent receiving the packet are not included.

gards to packet size, and which layers scale well or badly when the packet size is increased. These performance results will be useful for directing future optimization, and comparing how different layers scale to increased packet sizes compared to other network stacks.

6.4.1 Send

From the graphs, we can see that network channels scale well to larger packets, mainly because we perform no copying from userspace to the kernel to transmit. Figure 6.4 shows that most of the cycles are spent transmitting the packet – it also scales poorly compared to other layers involved in transmission. The cycles spent in the Ethernet layer, which comprises resolve network addresses and constructing a small hardware header (neither vary with the packet length), is constant for all packet sizes. Figure 6.5 shows a similar set of cycles count by layer, except that a lot less processor time is spent in the driver layer compared to the VirtualBox environment – the TCP implementation takes the most cycles on the laptop.

Inspecting the table in Figure 6.8, we observe that for larger packets, the cycles spent in the driver and in the userspace TCP implementation are the dominating factor, at least in the VirtualBox environment. On the laptop, the picture is much less clear; the TCP stack performs much better (the caches do not have to be shared with other processes as in the VirtualBox environment) – the focus for optimization here would focus on the IP layer, which takes up 10% more of the processing time than on the VirtualBox environment.

Because there is no dynamic allocation when sending a packet, the cycles spent transmitting a packet compares favorably with the network stacks in other operating systems. More work is needed to compare performance, especially CPU usage and bandwidth, between the Whitix implementation and others.

6.4.2 Receive

Receiving a packet has a similar workload – the inspection and matching stages are constant, since they only depend on the packet type and the number of channels in the system. Although we could not take driver measurements (as the current cycle counts were obtained by creating a special field in the NetBuffer structure, which is not present before packet receive), the receive results indicate most of the cycles spent receiving a packet involves the driver and TCP (and UDP) code.

Figure 6.6 shows that most of the time spent handling a TCP packet is in the TCP implementation in the userspace network stack. The UDP receive, which only involves reading the buffer from the channel and performing a checksum over the entire packet, takes much less cycles, although the two transport protocols dominate the processing time in a typical packet receive. The graph depicting the receive cycles on the laptop, Figure 6.7, shows similar results – in both graphs, inspection and matching take constant time, regardless of the size of the packet (matching is dependent on the number of channels registered to that protocol – in our example, there were no others).

What the *cycles consumed receiving* table shows, in Figure 6.9, is that 20-25% of the time receiving a TCP packet is spent copying the packet to the channel buffers. This copy, with the correct design, may be unnecessary – if allocate pages to represent receive buffers and copy the data from the network card into them, we can use the virtual memory of the operating system to remap the network channel's first free buffer (page) to point to that received page. If we can achieve that, we will be able to receive a packet in three quarters of the time, as well as improving cache performance greatly. Otherwise, in general, in both environments, most of the processor time is spent processing the TCP packet.

6.5 Summary

In this chapter, we first summarized the current techniques for general **network stack benchmarking**, focusing on the FreeBSD 4.7 study in particular. After a survey of previous network channel benchmarks, which delivered impressive throughput and CPU usage figures, we outlined a **method** for measuring this userspace network stack that involved a TCP and UDP echo server receiving and sending packets.

However, there were **caveats** in the experimental method, including the lack of a comprehensive profiling framework available in Whitix. After detailing the **experimental testbed**, we ran the experiments and produced an **analysis** of the relative performance and scalability of the userspace network stack. We offered comparisons to other network channel implementations and potential areas of optimization.

Chapter 7

Evaluation

In this chapter, we evaluate and compare the project's implementation against other stacks in a qualitative manner, based on metrics such as *flexibility and adaptability*, *correctness and stability*, *functionality and usability* and *scalability*. We also consider each level of this userspace network stack, and compare against the counterparts in other network stacks. After discussing each area, we propose ways to improve in the various areas, and conclude by summarizing the strong and weak areas of this project's implementation, as well as its successes.

7.1 Flexibility and adaptability

The flexibility and adaptability provided by the **asynchronous event and interactivity** layer (Chapter 5) allows for a much wider range of diagnostic and performance feedback algorithms to be incorporated into both client and server applications. Aside from academic proof-of-concept implementations of events for the TCP protocol using listener processes (see Section 5.1), there are no equivalent set of features in competing network stacks, mainly due to their kernel-based design. Event distribution to many sockets using an asynchronous notification mechanisms such as *signals* would not scale very well.

Although the set of events provided by the TCP implementation is fairly small, and their use by applications may fall short of a truly interactive relationship, the framework for specifying, registering and delivering events to sockets is **easily extensible** to more events and protocols. We have demonstrated this through a number of implementations in client applications such as `ftp`, `httpd` and `telnet`, and although some I/O events may not be suitable for interactive applications (see Section 4.6.2), generally the profiling, adaptability and feedback features, with more work, would improve application performance to an even greater degree with imaginative uses of the events.

Perhaps the only major thing missing from the implementation is a framework for filtering events by value when registering, in order to decrease the event work-

load. For example, if an application wants to register an interest in the *TCP retransmission timeout* event, it can do so by calling `TcpRetxTimerOutRegister`. If however, it only is interested in situations where the retransmission timeout is excessively high (to display indicating poor network connectivity as a message to the user), it must receive all the *retransmission timeout* events and performs its own filtering. In that respect, the event mechanism is not as flexible as desired.

7.2 Correctness and stability

In the absence of a framework for formal testing (such as *verification and validation*) and testing to ensure complete code coverage, and due to time constraints on the project, it was not possible to devise a comprehensive automated test suite for every one the four main sets of components of the networking subsystem. However, different methods were used for different layers; for example, unit testing suited testing the system call interface and memory management of the channel layer, but was ineffective in comprehensively testing the Linux Driver Layer.

The testing of **low-level networking** involved a number of different techniques, but the focus was mainly on automating testing at the higher levels and indirectly testing the packet transmission and receiving functionality. Since the Linux drivers used in the Linux Driver Layer were already well tested (since many of the drivers have been in the kernel tree for a number of years and their core functionality can be assumed to be well-tested) and there was a strict API to conform to when coding the layer, most of the testing of the layer was informal and comprised sending packets of different sizes and at different rates as part of general use of the layer.

The other components of the low-level networking layer, apart from packet send and receive, was the network device manager and hardware address cache. Again, since the input to the functions in this layer was assumed to stay relatively constant (for example, the `NetDevRegister` function would always be called from `register_netdev` in the LDL and would always register an Ethernet driver from the `b44` or `pcnet32` Linux drivers), I did not focus on testing this layer much under different inputs. To improve stability and correctness, more verification of the passed `net_device` structure should take place with a wider range of Linux drivers.

The unit testing (the `chan_test` program) performed for the **network channel layer** was sufficient to improve the code quality of the memory management and channel management components greatly, and as a result was a success in terms of improving system stability and correctness. An initial suite of tests written after the first iteration of the channel object was created uncovered a number of bugs that would have laid dormant had I only tested with the default flags. Writing automated tests for such a widely-used layer saved a lot of time later on when I began to develop the userspace network stack on top of the network channel layer.

The **userspace network stack** was well-tested by the range of applications that I developed, many of which functioned as basic tests of a class of sockets and transmission and receiving using the socket (c.f. `ping`). This also tested the protocol classifier code indirectly. Sending and receiving UDP and ICMP packets

was tested with a variety of packet sizes, but once I verified (using Wireshark) that the output of the implementation was correct, I considered the UDP and ICMP functionality complete (in terms of the project's scope) and did not see a benefit in writing further automated unit or regression tests.

Overall, my main concerns about correctness lie with the **TCP implementation** in the userspace network stack. It is relatively complex compared to the rest of the userspace network – in fact, the size of the TCP implementation in lines of code is larger than the rest of the network stack combined. The core functionality involves multithreading and multiple thread synchronization adds to the complexity and so makes it much harder to automatically test. Include the fact it was developed and built upon much later in the project than the other components, and it becomes clear that the implementation may have stability and correctness issues that the TCP automated test suite (`tcp_test` is described in Section 4.7) and the mock unreliable connection (which randomly drops packets) in the `TcpHandleData` function may not uncover.

The verification of the state machine performed by running `tcp_test` for automated testing (not complete) and, during general use, "sniffing" the TCP packet exchanges between the Whitix network stack and other hosts using *Wireshark*, was satisfactory for general use. Considering other aspects of the implementation, testing for deadlock conditions and multithreading bugs in the TCP polling thread will be more difficult due to their unreliable nature and the random delays of network packets.

In many ways, the **event handling and profiling** subsystem of the userspace network stack were well-tested by general application use, even though no automated testing took place. I tested each of the asynchronous events implemented, including all of the TCP events (Section 5.5.2) and the `AioReceive` event in different programs. This was one area where using the protocol sniffer *Wireshark* was useful for certain classes of events; we can verify that the *remote window changed* events fires when and only when the remote window size in a received packet changes easily. (It is harder to verify the *retransmission timeout* event firing pattern, but retransmission timeouts can be simulated in software anyway).

The most effective testing method overall for this project was the **automated unit and regression testing** performed for the TCP and network channel components, followed by in particular, verification of the network stack output by protocol sniffers such as Wireshark. Although it is relatively expensive in terms of time and effort, the benefit gained from ensuring, under most conditions, the correctness of possibly the two most complex and widely-used layers was key to the overall stability of the project.

With a larger timescale for the project, I would extended the automated testing to other areas of the networking subsystem. My focus would still remain on the complex parts of the system; the advantage is that testing networking functionality can easily be done using by running software on other hosts. Tackling the multithreading bugs that may be present in the various components would require a lock debugging suite and dependency modelling (illustrating why correct multithreaded software is so difficult to achieve), but the stability of such a core component may be worth the price in terms of time and effort.

7.2.1 Comparisons

The automated test suites present in Whitix compare favorably with those in Linux and the BSDs.[1] The **Linux Test Project**, which is a body of regression tests designed to evaluate the behavior of the Linux kernel, includes a large number of tests for application programs such as `ssh`, `ping` and `rcp` (included as shell scripts), internal networking functionality such as `iptables` (the standard firewall) and the Network Filesystem (NFS).[2] No equivalent exists in the Whitix test framework yet.

Crucially, there appears to be no test of the socket layer at a system call level in the Linux Test Project, and no automated testing of UDP, TCP or ICMP. In this respect, the small TCP test suite `tcp_test` is superior, and its high-level design could be adopted for automated testing of the network stack in Linux. The only similar project, and one that tests at a lower-level than my test suite, is the **TCP/IP Regression Test Suite** (TIRTS), a set of a programs that tests the state machine correctness (1) and the *transmission reliability*[3] (2) of a host by constructing a series of raw packets to see if the host meets the requirements of the TCP RFC 793.[4] However, unlike `tcp_test`, it does not verify the other local socket's state.

7.3 Functionality and usability

Is the current networking implementation adequate for both client and server applications? For **client applications**, such as `ftp` and `telnet`, the APIs provided are adequate to for a simple equivalent implementation, and since most of the functions called in these programs are the basic socket I/O functions such as `send`, `recv`, `close` (and their Whitix equivalents), we can consider the APIs provided to be sufficient for these class of programs. Other functionality provided in the BSD Socket API, such as the functionality to retrieve the local and remote endpoint address via `getsockname` and `getpeername`, will be trivial to implement anyway.

In terms of **server applications**, such as high-performance web servers, the polling quirks described in Section 4.4.6 and the lack of a high performance and change notification API (such as `epoll` in Linux), as well as only one thread used to poll all sockets for new TCP data, may negatively impact upon performance or make it harder to implement a server to handle many clients simultaneously. However, the ability to easily provide a new direct memory access API (as described in Section 4.8) will easily decrease latency and avoid unnecessary copies. To summarize, if we wish to make this networking subsystem usable by very high-performance servers, we must develop a high-performance socket buffer API and I/O event notification event facility. Our solution, as described earlier, has the po-

[1] Since Windows has a closed development process, the only diagnostic tools that are known about are the *Network Diagnostic Framework*, but this is mainly focused on helping users to diagnose network problems at runtime. It is likely however that they run a large range of automated test suites internally.

[2] The body of test cases can be found at http://ltp.cvs.sourceforge.net/viewvc/ltp/ltp/testcases/network

[3] *Transmission reliability* is the host's ability to handle segments with out-of-order sequence numbers.

[4] The code for the TCP/IP Regression Test Suite can be found at http://code.google.com/p/tirts/, and short technical documentation at http://wiki.freebsd.org/NanjunLi

tential to surpass other network stacks in performance if these two features are implemented.

In terms of the **end-user functionality** provided, they are no significant differences between the interface to networking applications, utilities and internal system configuration presented to the user in Whitix. All of the standard tools can be implemented using the new networking subsystem, and essentially their new functionality, while affecting performance, does not result in any functional changes to core features; this means that adoption of the system by end-users will be aided by a wealth of transferable knowledge from other Unix-like operating systems.

Missing features in the **network channel layer** are few and far between. Several features, as summarized in Section 3.9, that may be needed by a more complete implementation of a network stack include the ability to receive multiple streams of packets. As one more example, some protocols supports a notion of *out-of-band* data – a mechanism to retrieve more than one stream of data, perhaps at the network channel level, will need to be added for completeness. Overall, in terms of widely-used functionality, the network channel layer is complete, and most of the work will involve adding mechanisms to support rarely-used features of TCP or diagnostic elements of ICMP.

The implementation is lacking in other areas in terms of **end-user functionality**, such as a comprehensive set of commands to manage Ethernet devices (configuring hardware MTUs and so forth) in a similar fashion to `ipconfig` in Linux or the BSDs, but I judged this to be outside the scope of building the network stack – many of the configuration options are purely diagnostic or of little direct use (such as configuring the medium type used by an Ethernet card) anyway. A more complete implementation would provide this, along with more options for diagnostic tools such as `ping`, `dhcp` and `dns`.

7.4 Scalability

Would the network stack scale to multiple processors and a much more intensive workload? In many ways this is the most important objective of the project; we began by initially arguing in Section 1.1.2, among other points, that the use of locking and poor cache performance in current network stacks would not scale well to systems with many more processors than today. Although we could not directly test this when evaluating performance (as Whitix does not support SMP currently), we can evaluate the implementation against the project objectives and the initial design to evaluate whether the network stack would indeed scale.

In the **low-level networking subsystem**, with regards to packet transmission, the only place where locking may still be an issue is inside the driver itself, and the queuing of packets in the Linux Driver Layer. Some locking must take place between different transmission contexts to avoid hardware corruption of packets. The implementation of the LDL has not addressed multiprocessor locking[5] – as a result, this may be the bottleneck on network transmission in future many-core

[5] `IrqSaveFlags` and `IrqRestoreFlags` are used to disable preemption and interrupts on the local processor, but obviously would not work on a multiprocessor system.

systems (as it is one of the few common locks continually accessed by every packet transmission context) and so may scale poorly as a result. More work is needed, by perhaps employing an alternative to spinlocks, such as a *readers-writer lock*.[2]

The **network channel** layer is, in my opinion, more scalable than competitors in distributing packets to the rest of the network stack. By distributing the process of channel matching into sets of channel lists, and in turn removing any linked lists from packet distribution, we have created a scalable layer, both improving cache performance (at least in theory) and removing widely-taken locks by partitioning channels into lists and using static arrays of buffers, structures that fit into one cache line and utilizing simple bit operations such as *test and set*. Overall, we have improved scalability and partitioned many of the locks in existing network stacks into individual network stack and process contexts.

The main issue in the **userspace network stack** with regards to scalability is the handling of incoming and outgoing socket data. There is little state shared between sockets (apart from the common socket array), so locking is not too big a concern, but the ability to dynamically create new threads for polling particular sockets based on workload would be much more scalable. Currently only one global thread polls all TCP sockets, so the ability to create new TCP data handler threads for a socket or group of sockets based on workload would address any scalability issues present.

Thinking along different lines from purely processor and memory scalability, the **events and asynchronous I/O** component of the network stack has *not* been thoroughly stress-tested with a large number of connections, asynchronous I/O events or protocol events. However, handling a large number of events quickly and responsively is partly a result of the event handling implementation, but mostly a result of the priorities of the scheduler (whether it is focused on throughput or interactivity). Although the likelihood of thousands of events occurring per second is slim, further work needs to be done on constructing workloads and optimizing both the event handling and the thread scheduler to deal with processing, or adding a lightweight userspace threads implementation customized for event handling.

Overall, I believe as the number of processors per system increases and further work is done on the Whitix network subsystem to address any locks, this implementation will prove to be **much more scalable** than others. However, lacking useful performance counters for cache misses and locks taken, we cannot assert this currently with performance data. Future work should however focus on the common locks in low-level networking and scaling events and asynchronous events to thousands of sockets, and verify improvements with performance benchmarks and scalability tests, perhaps on environments such as **PlanetLab** (a testbed for computing networking research); in general, more stress tests are needed at the higher levels in user-space, and a focus on reducing the number of cache misses and shared cache lines through careful design and locking is needed at the kernel level.

7.5 Summary

Overall, and taking into account that the Whitix networking subsystem is at the stage of a proof-of-concept compared to the much more established network stacks of Linux, Windows, and the BSDs, the entire networking subsystem compares favorably with other implementations. Although we lack some of the breadth in low-level configuration of network interfaces, the **low-level networking layer**, through the help of the Linux Driver Layer, supports packet transmission and receiving almost every Ethernet device available today, which is no mean feat, and its raw performance is respectable compared to others. Amid some concerns about stability and multiprocessor locking, the performance and functionality of the current low-level layer implementation is suitable for future work.

The **network channel** layer provides probably the most convincing set of reasons for adopting the new networking architecture. Its scalability, stability and high performance mean that, even if the network stack itself is kept in the kernel, it is worth porting to other operating systems as a comprehensive replacement for the linked lists of network packets in other architectures. Any gaps in functionality, such as those described in Section 3.9, can be easily addressed and are not limited by the initial design.

The **userspace network stack**, although not complete to the standard of competing implementations (especially with regards to TCP), functions as an acceptable host on the wider Internet and is stable and correct enough to function in general use. The UDP and ICMP datagram protocols can be regarded as fairly complete, with work needed solely on the TCP implementation, in areas such as retransmission, state machine correctness and reliability. We discovered that the functionality offered by the lower level easily satisfied the needs of the protocol implementations in the userspace network stack.

The implementation of **asynchronous I/O and events** in the stack worked well as a proof of concept; general use of the layer may be unsuitable unless we can show large performance gains and the ability to handle large number of events and sockets simultaneously. However, the functionality supplied by the AIO and event component is not found in such breadth in any other network stack, and this can be judged to be a validation of moving the network stack to userspace. Along with the **interactivity** API, the layer transforms the userspace network stack into an adaptable and flexible component of the userspace network stack.

In summary, and including the performance results detailed in the previous chapter, the ideal **applications** for this network stack would include both client and server applications, especially high-performance servers with varying network conditions. The latency and throughput gain between conventional sockets and the new network channel-based design would only become apparent with a large workload per application – however, the decreased CPU usage, illustrated by Polyakov's results (and if able to run those types of comparisons, we would possibly improve upon those results), would be useful for any application. Environments where our implementation may be questionable would be public servers where users are able to log in and run their own programs – the minimal possibility of fooling the kernel network channel layer into transmitting attack packets may be a risk for certain

high-security organizations.

Chapter 8

Conclusion

Throughout this report we have shown that our alternative design for the network stack, one of the key components in any operating system, increases performance and security, decreases latency and increases packet throughput for a whole range of applications. **Moving the network stack into userspace** brings with it a number of advantages, including more persistent and interactive networking that adapts to the needs of particular applications. We conclude the following:

- There is a faster means of moving packets of data through the networking subsystem. We adapted and implemented an alternative kernel object, the **network channel**, to move data through the network stack with low latency and minimal kernel-level code, while preserving most of the existing semantics of packet transport. (Chapter 3)
  - We saw how network channels are superior to other socket alternatives, and how the shared memory between userspace and the kernel allowed for **zero-copy I/O**, while avoiding many security concerns.
  - We explored the channel's innate high cache performance, and due to the static nature of the network channel's buffers and the small amount of work required to allocate and free those buffers, we hypothesized that the design would be much more scalable than the linked lists used to store socket data in current network stacks.
  - Replacing sockets with network channels allowed us to push packet processing out to userspace, and enabled us to create a fully-featured network stack in userspace. A minimal amount of privileged code was required to classify incoming packets and route outgoing packets in the kernel.
- It is possible to create a high performance **userspace network stack**, with the protocol processing and state in a shared library, with the benefit of increased stack flexibility and system-wide security. (Chapter 4)

  - Aside from certain issues involving file descriptors, such as polling, we noted how our network stack implementation included all the major features of complex protocols like TCP, with congestion control, retransmission and flow control. We outlined the implementation's support for datagram protocols such as UDP and ICMP.
  - We discussed the multithreaded design of the TCP implementation, and how it provided a base for flexible events and low-latency responses to transport-level messages in the protocol, while obeying the traditional socket semantics in Unix.
  - We described the two socket APIs available for Whitix: the *native API* and *POSIX API*, and explained how the native API offered a framework for interactive next-generation features such as event subscription and, in the future, zero-copy I/O to the application.

- **Network stack adaptability**, and the use of profiling data from previous runs of a networked program, is a viable technique for optimizing bandwidth usage and improving latency. Even a small amount of statistical output from a previous run can lead to a large relative increase in performance, for only a small time cost at socket creation and the program start. (Chapter 5)

  - The userspace architecture of our novel network stack was used to add features such as *protocol events*, *application hints*, *persistent profiling information* and *asynchronous I/O*, as part of achieving the goal of **feedback** between the application and network stack.
  - The automatic collection of statistics by the network stack was covered, along with the mechanisms for categorizing the data and storing it to disk at program exit. We described how this profiling output was used to inspect network usage and automatically incorporated into future runs of the application, along with application hints. These are innovative functions not part of contemporary kernel-based network stacks.
  - Events, asynchronous notifications about a socket, can now be posted to an application if requested. TCP events, such as *retransmission timeout* or *round-trip time change*, can be subscribed to through the native API. We also added the capability for asynchronous I/O using the new event framework.

8.1 Scale and schedule

The project's scale was relatively large for a six month development process (see Table 8.1 for breakdown of lines of code by component). By far the largest component in terms of lines of code was the *userspace network stack* (including the applications and utilities developed for it). However, reviewing my schedule, the layer that I spent the most amount of time researching was the *network channel*

Table 8.1: Lines of code per component

Layer	Component	Source tree	LoC
Low-level	Network driver layer	net/{eth,network}.c	112
	Linux Driver Layer	devices/linux/	1241
	Network device manager, packet I/O	net/device.c	130
	Hardware address cache	net/arp.c	223
	Total		**1706**
Network channels	Generic channels	net/channels/	418
	IPv4 channels	net/ipv4/	964
	Usercode library	net/channels/userlib.c	130
	Total		**1512**
Network stack	Native API	user/sdk/network/socket.c	98
	POSIX API	user/posix/socket/	173
	Channels	user/sdk/network/channels.c	243
	IP	user/sdk/network/ipv4.c	97
	UDP and ICMP	user/sdk/network/{udp,icmp}.c	214
	TCP	user/sdk/network/tcp/	1709
	Applications, utilities etc.	user/net/	1358
	Total		**3892**
Adaptability and interactivity	Statistics and profiling	user/sdk/network/stats/	257
	Events	user/sdk/network/	112
	TCP events	user/sdk/network/tcp_events.c	153
	Asynchronous I/O	user/sdk/network/aio.c	24
	Total		**546**
Total lines of code			**7656**

layer, to investigate competing implementations, and the *adaptability and interactivity* layer.

The *network channel layer*, by my estimates, took the longest to develop – planning the architecture of the layer, the various features, and constantly reviewing it for stability, security, performance and scalability involved a large design effort. The fact that the generic channel layer, at 418 lines, is relatively small and yet fully-featured is a testament to the succinct design of the layer.

With a longer development schedule, I would have been able extend the *adaptability and interactivity* layer and supplement the range of TCP events with events from other protocols – the total number of lines in that layer may double in a more complete implementation. The AIO layer may look tiny, but most of the asynchronous logic is placed in the TCP packet receive code – the generic layer merely manages the list of AIO contexts present in a socket.

8.2 Future work

The remainder of this chapter discusses improvements to the current implementation and extensions for particular parts of the networking subsystem. This project functions as a small overview of the eventual potential of network channels in Whitix. We also note any design considerations and potential compromises compared with the current design.

8.2.1 Merging shared memory and message queues

Currently, the only protocol suite that uses the new network channel objects is the TCP/IP suite of protocols. **Unix domain sockets** or **IPC sockets**, used by an application for local inter-process communication (without the overhead of a networking protocol). These sockets use the filesystem or a separate special namespace for named endpoints. Currently, the communication between two local endpoints uses buffers in kernel memory.

With network channels, it should be possible to arrange the internal channel memory management so that, for two local endpoints A and B, the send buffers of A correspond to the receive buffers of B. For this to work correctly, the send and receive headers for each buffer must exactly match (the total cost in memory would be very small).[1] If this is possible, then essentially we will have an object that corresponds to structured shared memory, which happens to map to both local and remote endpoints efficiently.

Following the above point, we could possibly look towards unification of local and remote endpoints and names; the application would specify what class of transport it needed (reliable, in-order delivery or packet-based transport), along with a name of an endpoint, and let an intelligent network stack handle the data trans-

[1] An alternative design might involve a special header structure for local IPC, that uses most of the network channel infrastructure, however, it would mostly likely lead to a lot of duplicated code and mechanisms.

port. In this respect, networked resources would then become location transparent to the application.

8.2.2 More protocol suites

Would our userspace network stack work with **other networking protocols**? We may have demonstrated that our userspace network stack provides (virtually) all of the functionality of TCP/IP, a now-ubiquitous protocol suite, but can the same mechanisms still work for others? It is hard to decide on another protocol suite that would be worth the investment of time, but protocols based on *Asynchronous Transfer Mode* (ATM),[41] a cell-based switching technique that uses time division multiplexing, may be worth considering.

To judge whether our design and other protocols, including ATM, would be compatible would require a deeper knowledge and research. Questions such as "*how would we represent a virtual circuit in userspace?*" are pertinent. Most protocol suites would probably be suitable, as they generally use a layered design based on the OSI model, a model with finer distinctions than the TCP/IP model. Such a layered design allows low-level packet code (the physical, data link and network layers in the OSI model) to be placed in the kernel, with higher level layers (the transport and session layers) to be handled by userspace code.

8.2.3 Stateful firewall

Although we have successfully implemented a basic packet filter (Section 3.7.1), which accepts or rejects single packets based on user-defined rules, for the new networking subsystem, the main challenge of integrating a firewall into the network channel layer lies in adding a knowledge of TCP (and, via heuristics, UDP connections) to the firewall and incorporating the part the packet plays in the connection into the user-defined rules to create a **stateful firewall**.

The code to keep track of the connection state should be fairly small; as a comparison, the userspace network stack's TCP state machine comprises about 20% of the total TCP codebase, and so stateful support for different connections would only involve a couple of hundred lines of new code, for a much more expressive firewall.

8.2.4 Remote network stack management

The **Simple Network Management Protocol (SNMP)** is a network protocol running over UDP that is used to monitor network-attached devices. It includes a set of data objects of a variety of basic types, packaged as a *management information base (MIB)* that describe the configuration of a system, including its network stack; in many ways, the SNMP standard implicitly assumes a central implementation of networking configuration. There are many different types of MIBs, but the ones that would prove the most challenging to implement would be the TCP [53], IP [56] and UDP [21] MIBs.

For example, to implement the `tcpCurrEstab` SNMP object [see 53, page 6], which is an integer containing the total number of established TCP connections in the system. We need to collate this information from all the processes with active network stacks in the system. We would either have to poll each network stack for information via a userspace SNMP server process or make sure that each network stack implementation exposed the relevant SNMP objects and statistics in a standard way to the userspace SNMP application. Both methods make it harder to implement an alternative network stack by requiring more information and mechanisms from an implementation.

8.3 Final comments

Overall, I believe the size of the project was an ambitious undertaking, but I have established proof of concept implementations in enough areas to dispel many of the preconceptions held about network channels and alternative network designs, as well event delivery and AIO. The fact that the network stack is easily extensible with user-level events and (relatively) low-latency asynchronous IO is ideal, and so as a result has features lacking in other network stacks.

If I had been allocated twice the amount of time to do the project, I am sure I would have delivered a much more fully-featured implementation that could have competed with existing established network stacks. In the six months I have worked on the project, I believe I have reached the point, however, where my implementation is a **viable base** for future development, and various elements, such as the userspace network, could easily be ported to other operating systems, thanks to the layered design. More time would have been spent on automated testing and performance evaluation – developing a simple performance framework would have proved useful in generating useful data, such as statistics on cache misses.

So in conclusion, even though tough goals were set at the beginning of the project, the implementation given here and my analysis is a success in terms of exploring and achieving the objectives.

Chapter 9

Bibliography

[1] M. Abbott and L. Peterson. A language-based approach to protocol implementation. *IEEE/ACM Transactions on Networking (TON)*, 1(1):19, 1993.

[2] T. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990.

[3] H. Balakrishnan, V. Padmanabhan, S. Seshan, M. Stemm, and R. Katz. TCP behavior of a busy Internet server: Analysis and improvements. In *IEEE INFOCOM*, volume 1, pages 252–262. Citeseer, 1998.

[4] S. Bellovin. Defending Against Sequence Number Attacks. RFC 1948 (Informational), May 1996.

[5] C. Benvenuti. *Understanding Linux Network Internals*. O'Reilly Media, 1st edition, 2005. ISBN 0-596-002-556.

[6] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer Protocol – HTTP/1.0. RFC 1945 (Informational), May 1996.

[7] Y. Bhumralkar, J. Lung, and P. Varaiya. Network Adaptive TCP Slow Start. 2000.

[8] D. Bovet and M. Cesati. *Understanding The Linux Kernel*. O'Reilly Media, 3rd edition, 2005. ISBN 0-596-005-652.

[9] R. Cáceres, P. Danzig, S. Jamin, and D. Mitzel. Characteristics of wide-area TCP/IP conversations. In *Proceedings of the conference on Communications architecture & protocols*, page 112. ACM, 1991.

[10] Chisnall, D. POSIX Asynchronous I/O. *InformIT*, 2006.

[11] J. Corbet. A proposal for a new networking api. 2006. Available online (last accessed 25-11-2009) at http://lwn.net/Articles/192410/.

[12] J. Corbet. Van Jacobson's network channels. 2006. Available online (last accessed 24-11-2009) at http://lwn.net/Articles/169961/.

[13] J. Corbet. The return of network channels. 2007. Available online (last accessed 24-11-2009) at http://lwn.net/Articles/260880/.

[14] J. Corbet. LCA: A new approach to asynchronous I/O. 2009. Available online (last accessed 03-04-2010) at https://lwn.net/Articles/316806/.

[15] P. de Langen and B. Juurlink. Memory copies in multi-level memory systems. In *Application-Specific Systems, Architectures and Processors, 2008. ASAP 2008. International Conference on*, pages 281–286, 2008.

[16] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460 (Draft Standard), Dec. 1998.

[17] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), Aug. 2008.

[18] U. Drepper. The Need for Asynchronous, Zero-Copy Network I/O. 2006. Available online (last accessed 13-02-2010) at http://people.redhat.com/drepper/newni.pdf.

[19] R. Droms. Dynamic Host Configuration Protocol. RFC 2131 (Draft Standard), Mar. 1997.

[20] A. Edwards and S. Muir. Experiences implementing a high performance TCP in user-space. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, page 205. ACM, 1995.

[21] B. Fenner and J. Flick. Management Information Base for the User Datagram Protocol (UDP). RFC 4113 (Proposed Standard), June 2005.

[22] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999.

[23] X. Fu, J. Demter, C. Dickmann, H. Peters, and N. Steinleitner. Performance analysis of the TCP/IP stack of Linux Kernel 2.6. 9. 2005.

[24] S. Gay, V. Vasconcelos, A. Ravara, et al. Session types for inter-process communication. *Preprint, Dept. of Computer Science, Univ. of Lisbon*, 2002.

[25] J. Hennessy, D. Patterson, D. Goldberg, and K. Asanovic. *Computer architecture: a quantitative approach*. Morgan Kaufmann, 2003.

[26] R. Hinden and S. Deering. Internet Protocol Version 6 (IPv6) Addressing Architecture. RFC 3513 (Proposed Standard), Apr. 2003.

[27] R. M. Hinden. IP next generation overview. *Commun. ACM*, 39:61–71, June 1996.

[28] V. Jacobson. Congestion avoidance and control. In *Symposium proceedings on Communications architectures and protocols*, SIGCOMM '88, pages 314–329, New York, NY, USA, 1988. ACM.

[29] V. Jacobson. Speeding up networking, 2006. Available online (last accessed 24-11-2009) at http://www.lemis.com/grog/Documentation/vj/lca06vj.pdf.

[30] F. Jennings. Linux socket programming - what lies beneath a socket? 1995. Available online (last accessed 03-03-2010) at http://linux.sys-con.com/node/34589.

[31] T. Jones. Boost application performance using asynchronous i/o. 2006. Available online (last accessed 13-05-2010) at http://www.ibm.com/developerworks/linux/library/l-async/.

[32] J. Khan and R. Zaghal. Symbiotic rate adaptation for time sensitive elastic traffic with interactive transport. *Computer Networks*, 51(1):239–257, 2007.

[33] H. Kim and S. Rixner. Performance characterization of the FreeBSD network stack. *Computer Science Department, Rice University*, 2005.

[34] G. Korah-Hartman. The Linux Kernel Driver Interface. 2008. Available online (last accessed 03-03-2010) at http://lxr.linux.no/#linux+v2.6.34/Documentation/stable_api_nonsense.txt.

[35] B. Loo, T. Condie, M. Garofalakis, D. Gay, J. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking: language, execution and optimization. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, page 108. ACM, 2006.

[36] A. Lounsbury. Gigabit Ethernet: The difference is in the details. *Data Communications*, 26(6):4, 1997.

[37] K. McCloghrie and M. Rose. Management Information Base for Network Management of TCP/IP-based internets:MIB-II. RFC 1213 (Standard), Mar. 1991.

[38] R. Merrit. CPU designers debate multi-core future. 2008. Available online (last accessed 03-03-2010) at http://www.eetimes.com/showArticle.jhtml?articleID=206105179.

[39] P. Mockapetris. Domain names - implementation and specification. RFC 1035 (Standard), Nov. 1987.

[40] MSDN. Synchronous and Asynchronous I/O. 2010. Available online (last accessed 23-05-2010) at http://msdn.microsoft.com/en-us/library/aa365683%28VS.85%29.aspx.

[41] P. Neelakanta. *A textbook on ATM telecommunications: principles and implementation*. Boca Raton, 1st edition. ISBN 0-849-318-05X.

[42] M. Neubauer and P. Thiemann. An implementation of session types. *Practical Aspects of Declarative Languages*, pages 56–70.

[43] V. Paxson and M. Allman. Computing TCP's Retransmission Timer. RFC 2988 (Proposed Standard), Nov. 2000.

[44] D. Plummer. Ethernet Address Resolution Protocol. RFC 826 (Standard), Nov. 1982.

[45] J. Postel. User Datagram Protocol. RFC 768 (Standard), Aug. 1980.

[46] J. Postel. Internet Control Message Protocol. RFC 792 (Standard), Sept. 1981.

[47] J. Postel. Internet Protocol. RFC 791 (Standard), Sept. 1981.

[48] J. Postel. Transmission Control Protocol. RFC 793 (Standard), Sept. 1981.

[49] J. Postel and J. Reynolds. Telnet Option Specifications. RFC 855 (Standard), May 1983.

[50] J. Postel and J. Reynolds. Telnet Protocol Specification. RFC 854 (Standard), May 1983.

[51] J. Postel and J. Reynolds. File Transfer Protocol. RFC 959 (Standard), Oct. 1985.

[52] N. Pryce and S. Crane. Component interaction in distributed systems. In *Configurable Distributed Systems, 1998. Proceedings. Fourth International Conference on*, pages 71–78, 1998.

[53] R. Raghunarayan. Management Information Base for the Transmission Control Protocol (TCP). RFC 4022 (Proposed Standard), Mar. 2005.

[54] T. Roscoe. Network architecture test-beds as platforms for ubiquitous computing. *Phil. Trans. R. Soc. A*, pages 3709–3716, 2008.

[55] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. RFC 3261 (Proposed Standard), June 2002.

[56] S. Routhier. Management Information Base for the Internet Protocol (IP). RFC 4293 (Proposed Standard), Apr. 2006.

[57] W. Shay. *Understanding data communications and networks*. PWS Publishing Co. Boston, MA, USA, 2 edition, 1999.

[58] W. Stallings. Ipv6: The new internet protocol. *IEEE Communications Magazine*, 34(7):96–108, 1996.

[59] R. W. Stevens. *Unix Network Programming, Volume 1: The Sockets Networking API*. Addison-Wesley Professional, 3rd edition, 2003. ISBN 0-131-411-555.

[60] A. Tanenbaum. *Computer Networks*. Prentice-Hall, 4th edition, 2002. ISBN 0-130-661-029.

[61] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2nd edition, 2001. ISBN 0-13-092641-8.

[62] C. Thekkath, T. Nguyen, E. Moy, and E. Lazowska. Implementing network protocols at user level. *IEEE/ACM Transactions on Networking (TON)*, 1(5):565, 1993.

[63] C. Tschudin. Flexible protocol stacks. In *Proceedings of the conference on Communications architecture & protocols*, pages 197–205. ACM, 1991.

[64] P. Tulloch. Discussing the Many-Core Future. *HPCWire*, 2007.

Appendix A

Network statistics file format

The **statistical output** from the network stack is stored on disk at the filesystem location specified in `NetworkInitEx` in the following format. As described in Section 5.3, the file is divided into three parts, the header describing the structure sizes and the number of structures in the file, the port entries, where protocol information is aggregated over all the runs of the application, and the host entries, which acts as a cache for connection information (such as the *retransmission timeout*) for programs that regularly connect to the same set of hosts.

A.1 Header

Item	Size (bytes)	Comments
Magic number	4	Always has the value 0xDE23FA11
Number of port entries	4	
Size of port entry structure	4	
Number of host entries	4	
Size of host entry structure	4	
Maximum number of host entries	4	Usually set by application or general system configuration value
Port entries	*variable*	
Host entries	*variable*	

A.2 Port entries

Item	Size (bytes)	Comments
Port	2	Value is always a well-known port i.e. <= 1024
Flags	2	Either PORT_SERVER (0) or PORT_CLIENT (1), depending on the port's role in the communication
Total bytes sent	4	All bytes sent in all runs of the application since profiling began
Total packets sent	4	All packets sent in all runs of the application since profiling began
Total bytes received	4	All bytes received from all hosts on the port
Total packets received	4	All packets received from all hosts on the port

A.3 Host entries

Item	Size (bytes)	Comments
Address	4	Of the remote host.
Flags	4	Either HOST_SERVER (0) or HOST_CLIENT (1), depending on the host's role in the communication
Retransmitted bytes sent	4	Total number of bytes that were retransmitted across all ports to the host
Retransmitted packets sent	4	Total number of packets that were retransmitted across all ports to the host
Retransmit timeouts	4	The total number of *retransmission timeout* events that occurred
rto, rtt, mdev	12	The three variables used in the estimation of *round-trip time* and the *retransmission timeout* value
minWin, maxWin, endWin	12	The minimum, maximum and last receive window size of the remote host

Appendix B

System calls

B.1 SysChannelCreate

```
int SysChannelCreate(int chnType, void* source, void* dest, struct
    ChannelOptions* options);
```

The parameters have the following meaning:

- **chnType**. (in) The protocol that will be used to inspect and classify incoming and outgoing packets. `chnType` is a 32-bit integer, with the high word specifying the address family, such as IPv4, and the low word specifying the exact protocol (TCP, UDP or ICMP). This is roughly equivalent to the `domain` and `type` parameters of the `socket` system call in UNIX, except with the `type` field depending on the protocol used.
- **source**, **dest**. (in/out) The exact format of these parameters depends on the address family. For IPv4 channels, the data pointed to by the two pointers has the following structure:

  ```
  struct Ipv4EndPoint
  {
      ulong address;
      ushort port;
  };
  ```

  The protocol family specified in `chnType` determines the exact semantics of the address and port fields. Generally, for all protocols that involve a source and destination port, specifying a zero port will cause a ephemeral source port to be allocated automatically by the `Ipv4PortAllocate` function in `net/ipv4/port.c`.
- **options**. (in/out, optional) This can be NULL if the caller does not want to specify any particular options. In this case, sensible defaults for the number

of send and receive buffers are supplied, and there are no special flags assigned to the channel. The exact structure of `ChannelOptions` is shown below:

```
struct ChannelOptions
{
    int flags;
    unsigned short sendBuffers, recvBuffers;
};

define CHANNEL_IGNORE_ADDRESSES     0x01
#define CHANNEL_KEEP_SEND_BUFFERS    0x02
```

If the call is successful, a valid file descriptor is returned to the user, which can be used in later file operations to transmit and receive data. On an error, one of the standard UNIX error codes is returned.

B.2 SysChannelControl

```
int SysChannelControl(int chnFd, int code, void* data);
```

The parameters have the following meaning:

- **chnFd**. The file descriptor returned by a previous call to SysChannelCreate.
- **code**. The operation to perform on the channel. There are currently three values available:

  ```
  #define CHANNEL_SET_FLAGS          0x01
  #define CHANNEL_GET_FLAGS          0x02
  #define CHANNEL_SET_INTERFACE      0x03
  ```

- **data** Dependent on the value of `code`. For the flag-related operations, `data` is the address of an integer to read from or write to respectively. For `CHANNEL_SET_INTERFACE`, it is the address of a string containing the human-readable name of the interface: "Ethernet0" for example.

The function generally returns zero if successful, or one of the standard UNIX error codes if unsuccessful.

Appendix C

Usercode library functions

The network channel usercode library contains a number of functions, as described in Section 3.4.2, that abstract away the low-level structure manipulation involved in the operation of any class of network channel. Each function acts on the general channel header (`ChanUserHeader`) or one of the two packet header types (`ChanSendHeader`, `ChanRecvHeader`); there is no extra context stored or invoked by the usercode library.

As a result, the channel memory appears opaque to the userspace application; the structure of any internal memory, like the `ChannelInfo` structure, is revealed on a need-to-know basis. The function prototypes, which can be divided into three categories, are shown below:

```
/* 1. Allocation map functions */
void* uChanSendBufferAlloc(struct ChanUserHeader* header);
struct ChanRecvHeader* ChanRecvBuffGet(struct ChanUserHeader* header);
void uChanSendBuffFree(struct ChanUserHeader* header, struct
        ChanSendHeader* send);
void uChanRecvBuffFree(struct ChanUserHeader* header, struct
        ChanRecvHeader* recv)

/* 2. Header functions */
int uChanRecvBuffLen(struct ChanRecvHeader* header);
void uChanRecvBuffSetPriv(struct ChanRecvHeader* header, void* data);
void* uChanRecvBuffGetPriv(struct ChanRecvHeader* header);
unsigned short uChanRecvBuffGetRead(struct ChanRecvHeader* recv);
void uChanRecvBuffAddRead(struct ChanUserHeader* header, struct
    ChanRecvHeader* recv, unsigned short add);
void uChanSendBuffSetPriv(struct ChanSendHeader* header, void* data);
void* uChanSendBuffGetPriv(struct ChanSendHeader* header);

/* 3. Information structure functions */
void* uChanGetInfo(struct ChanUserHeader* header);
```

Each function in each category performs a different task, and each category can be further categorized into whether the functions act on send or receive headers,

which have a different structure, semantics and memory layout. The functions can be summarized as follows:

1. **Allocation map functions**. These functions update the send and receive allocation bitmaps (there is nothing to stop another data structure being substituted); the semantics differ depending on the type of buffer. For send buffers, the order of allocation does not matter (although the order of transmission might), but for receive buffers, the application should `Get` the oldest unread buffer (following the semantics in Section 3.4). Once we are done with a receive buffer, we should free it, and, if the application manages the deallocation of send buffers itself, it should free the buffer once finished with it.

2. **Header functions**. The internal structure of the send and receive header are not exposed to userspace applications and libraries. The main reason for doing this is that it would make it impossible to update or optimize either header without recompiling every application (or, most of the time, the network stack library) that has ever used network channels; this would decrease the flexibility we would have if we wished to optimize the internal structure.

   These functions get and set the various fields of the structure. There are more fields in the receive header, because we store most of the context of a received packet in userspace anyway. It is the opposite for send headers. One use for the `Chan*Buff*Priv` functions is to build a retransmission list for sent packets or a backlog for received ones.

3. **Information structure functions**. A pointer to the information structure could easily be a parameter of the `SysChannelCreate` function. However, since the information contained in `ChannelInfo` is key to the channel's correct operation (including functions in the usercode library), and the fact that the channel can be passed between processes, means that storing the channel's context in the channel's memory itself is preferable to passing a pointer around different layers of code (such as between the usercode library and the application)