Diplomarbeit

A Split TCP/IP Stack Implementation for GNU/Linux

Martin Unzner
munzner@os.inf.tu-dresden.de

29. April 2014

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer:	Prof. Dr. rer. nat. Hermann Härtig
Betreuender Mitarbeiter:	Dipl.-Inf. Julian Stecklina

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den 29. April 2014

Martin Unzner

Danksagung

Mein Dank geht an Professor Hermann Härtig und Julian Stecklina, die es mir ermöglicht haben, diese gleichermaßen herausfordernde und dankbare Aufgabe anzunehmen. Mit seiner geduldigen und gründlichen Art und richtungsweisenden Ratschlägen hat Julian diese Arbeit entscheidend geprägt. Björn Döbel gebührt Dank für die Weiterbetreuung der Arbeit in ihrer Endphase und viele hilfreiche Anmerkungen zu Stil und Inhalt. Meinen Kommilitonen im Studentenlabor möchte ich für die ausgezeichnete Atmosphäre und die teils unbequemen, aber stets tiefschürfenden und ehrlichen Hinweise danken. Meine Eltern haben mich während der Entstehungsphase dieser Arbeit auf vielerlei Arten unterstützt, ein großes Dankeschön dafür. Zuletzt danke ich meiner Schwester Grit fürs Korrekturlesen.

Abstract

The TCP/IP protocol suite is the foundation of the worldwide internet, which is the world's most widespread computer network. Today, in most systems, TCP/IP protocols are still handled in the kernel. Faulty packets and attacks like SYN flooding can affect the whole operating system, even if they are just targeted towards one single network process. A userspace TCP/IP stack, on the other hand, exposes only networking applications to those threats.

Most userspace TCP/IP stacks are designed as independent application libraries that send and receive fully-fledged Ethernet frames. To communicate among each other, they each require separate addresses and need to be connected in a separate virtual network on the host computer, which is complicated to set up and unnecessarily increases the use of spare addresses.

In this work, I propose Swarm, a userspace switch that reunites arbitrary independent userspace TCP/IP stacks into one logical host by administrating the system's port address space. In contrast to existing solutions, Swarm runs at user level and is thus fully isolated from non-network applications as well as critical kernel code. I am going to implement and evaluate a corresponding prototype, keeping the trusted part as slim as possible, using a custom-tailored routing component and the Rump TCP/IP stack [Kan12b].

Contents

List of Figures

1 Introduction

The TCP/IP technology is the foundation for the worldwide internet, which, with over 2.4 billion users [Gro12], is one of the world's most widespread computer networks. A software suite implementing all protocols necessary to communicate over the internet is known as a TCP/IP stack or an internet Protocol stack.

In monolithic operating systems like GNU/Linux, the BSD family, or Microsoft Windows, networking has always been a task of the kernel. However, when all applications share one network stack at the foundation of the operating system, one faulty packet, sent accidentally or on purpose, destined towards one single application, can cause vital parts of the system to crash. The widely deployed Linux operating system, which I am using in this work, provides a good example for those vulnerabilities. In 2012, a flaw in the Linux kernel was detected where users could make the system crash simply by querying statistics on TCP [Rap12].

The deployment of security fixes generally is disappointingly slow [Res03], but whereas many administrators still agree to deploy security fixes to their kernels eventually, the situation is even worse for new features. Recently, Honda et al. analysed TCP network connections for the capabilities they had enabled, and discovered that functionality which has been on by default for eight years now is still not supported by up to 70 per cent of network communication flows [Hon+14]. Part of the problem is that TCP/IP resides in the kernel, which is at the bottom of the operating system. If no always-on update functionality like Ksplice[AK09] is installed on the system, which is the case especially with older systems, a system reboot is required to apply the patch to the kernel. Most administrators try to minimise the reboot frequency and thus the downtime of their systems, and will shy away from kernel patches that they do not deem necessary. A single application, on the other hand, can be restarted quicker than a whole system, so new features in a userspace program are likely to be delivered earlier than new kernel features.

Hence there is a need for userspace TCP/IP stacks. There exist implementations such as lwIP [Dun01], LXIP [Gmb], or Rump TCP/IP [Kan12b], but each of those stacks is independent within its application. If there are multiple network processes running on the same host, each of them normally requires a singular hardware and IP address. MultiStack [Hon+14] offers a system-wide protocol switching solution that enables one host to appear under one address, and thus ease network configuration and save addresses as well as remain compatible to legacy applications. However, MultiStack uses VALE to switch packets, which is located at kernel level. Hence all the considerations concerning reliability and security made earlier for the network stack now repeat for the switch. There are also hardware solutions that can connect independent network stacks, such as Arsenic [PF01], but those require custom hardware and are not widely used.

In this work I introduce Swarm, which is a dedicated userspace port switch for interconnecting independent TCP/IP applications on the same host.

Swarm is kept so small that a code review is easily possible, and it further eases the use of userspace TCP/IP which enables a smaller kernel without networking functionality. Non-networking applications do not need to trust network code any more, and even though all network processes need to trust Swarm, they are still all isolated against each other, so that errors in one process do not directly affect another.

Swarm makes interconnecting userspace TCP/IP processes easier because all stacks are using the same address, which helps porting legacy applications to and integrating new programs with userspace stacks.

Swarm was designed to support arbitrary TCP/IP stacks, which helps to deploy new TCP/IP features and security fixes more quickly [Hon+14], and also facilitates maintenance.

However, Swarm does not perform as well as I expected, and imposes a decrease in performance between 30 and 50 per cent. I found out that the main reason for that lies within its implementation (see Chapter 4 for details).

I give an overview of Swarm's technical background and related work in Chapter 2 before I go on to describe the design and implementation of Swarm as well as the development of a prototype system in Chapter 3. I evaluate my results in Chapter 4 and present possible solutions to still existing problems in Chapter 5. Finally, I conclude my work in Chapter 6.

2 Technical background

This chapter serves two purposes: I provide the technical background that is necessary to understand this work, and in Section 2.5, I compare my work to related publications in the field. I begin with the most important technology, the internet.

2.1 The internet protocols

The entity of all protocols used in the internet is referred to as the internet protocol suite or TCP/IP stack. The latter term is derived from the separate stacked layers of the suite that comprise the individual protocols. The TCP/IP stack has four layers, which are depicted in Figure 2.1: At the bottom, there is the link layer, which transmits the raw data. Above it is the internet layer, which is reponsible for delivering packets to the appropriate host. The internet layer passes processed packets down to the link layer for transmission on the wire. The transport layer on top of both communicates data among the applications on different hosts using the internet layer. On the application layer, application-specific protocols make use of the general-purpose communication primitives that the lower layers provide them with to communicate web pages, speech, video, or even a BitTorrent stream. All communication that an application sends on the network passes through all layers before being transmitted onto the physical link, where it eventually reaches its destination and again traverses all layers, now in the reverse direction, to be delivered to the receiving application.

The highest layer that I treat in this work is the transport layer. There, the internet protocol suite offers the choice, among others, between a reliable transport mechanism, the Transmission Control Protocol (TCP), and the User Datagram Protocol (UDP), which is simple but unreliable.

TCP provides a logical stream once a connection between two fixed endpoints on the network has been established [Pos81c]. The protocol implementation numbers the individual messages to be transmitted over the network, and reassembles them in the right order at the receiving end. If a datagram is lost or corrupted on the network, TCP needs a means to retransmit the missing information. Therefore, each datagram is acknowledged by the receiving end if the content is correct. There is a checksum included in the transmission to check the correctness of the transmitted data [Pos81c].

UDP, on the other hand, does not provide guarantees for correct or even in-order packet delivery. Apart from an optional checksum, each message is sent to its destination without further precautions [Pos80].

As soon as the data has been prepared for transport, it is passed to the actual Internet Protocol (IP). Each network host has an IP address, so that IP datagrams can be routed using a source and a destination address, both of which are included in the IP header.

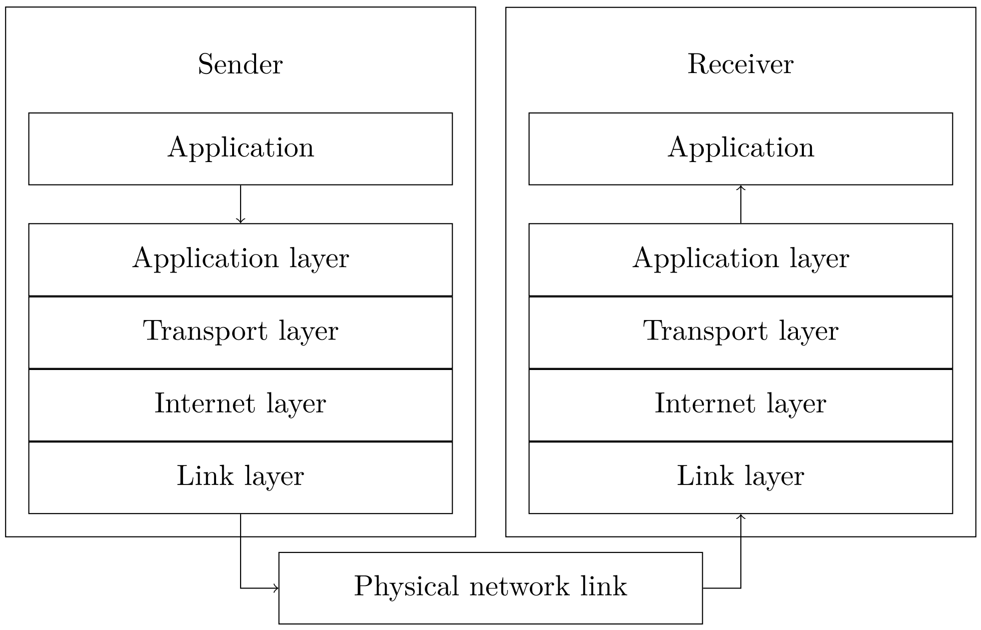


Figure 2.1: The working principle of the TCP/IP stack

In contrast to the transport protocols, IP secures only its header against corruption, not the payload data, and it provides no ordering of datagrams [Pos81b].

Before the data can be converted into electrical or optical signals that are transmitted on the physical network connection, further processing is necessary on the link layer. The only link protocol that I treat in this work is Ethernet. RFC 894 describes how to encapsulate IP into Ethernet. Essentially, the correct type field and source and destination hardware addresses need to be added to the IP datagram before handing it over to the device [Hor84].

For each network, the host system needs a network interface card (NIC), which translates the data flow from the host to the network protocol. If all networks the host is connected to are TCP/IP networks, the host has a different IP address for each network. There is also a technique called IP aliasing, where a host assigns multiple IP addresses to a single NIC.

Consequently, each NIC has one or more IP addresses exclusively assigned. The Address Resolution Protocol (ARP) [Plu82] retrieves the interface address for an IP address. In the beginning, a remote application that wants to send data to a specific IP address sends an ARP request to all NICs on the Ethernet bus, asking who owns the destination IP address. If a host has the address, it responds with an ARP reply. If two hosts reply with the same IP address, there is an IP address conflict in the network, and the user (more generally: the 'configuring agent') needs to be notified of the misconfiguration [Che08].

Although there are far more protocols in the internet protocol suite, I have used the protocols that I just explained, which are TCP, UDP, IP, ARP, and Ethernet. TCP, UDP, IP, and ARP are the essential protocols required for a working TCP/IP stack, and

Ethernet, a widely used link level technology tailored for local area networks (LANs), was chosen to comply with the test hardware used for the experiments presented in Chapter 4.

The hardware addresses of Ethernet NICs are also referred to as Medium Access Control (MAC) addresses. Because I only use Ethernet, I use the terms MAC address and NIC hardware address interchangeably within the scope of this work.

Ethernet MAC addresses, which are currently 48 bits long, are managed by the Institute of Electrical and Electronics Engineers (IEEE) and the network hardware manufacturers, and a sufficient amount is still available. IP addresses, on the other hand, are only 32 bits long in IP version 4 (IPv4), and more IP addresses than MAC addresses are required to operate a TCP/IP network. Thus, Stevens already predicted twenty years ago that they might soon run out [Ste94]. Eventually, thanks to measures like Classless Inter-Domain Routing (CIDR) [FL06] and Network Address Port Translation (NAPT), the last IPv4 address was sold eleven years later than he extrapolated [Law11].

To increase the number of available addresses, in 1998, the IETF introduced IP Version 6 (IPv6) [DH98] as the successor of IPv4. IPv6 has an address length of 128 bits and a completely restructured header, which means that IPv6 is not backwards-compatible to IPv4. Although it would provide a good solution to the internet addressing problem, the deployment of IPv6 is still going rather slowly. IPv6 connectivity among Google users, for example, is merely at 2.48 per cent as of this writing [Inc].

On the transport layer, both TCP and UDP use numeric endpoint identifiers, so-called ports. The theoretical port range, extrapolated from the field size in the headers of TCP and UDP, is 2^{16} (65536) for each protocol [Pos81c; Pos80]. However, part of the port space is reserved: The Internet Assigned Numbers Authority (IANA) provides a database of all assigned port numbers [IAN14].

Each message has a source and a destination port. Every application, client and server alike, needs at least one port to listen for incoming messages, which needs to be registered with the TCP/IP implementation. This process is called *binding*. Ports are either bound explicitly using a dedicated function call, or implicitly. For instance, when a TCP client establishes a connection, the TCP/IP stack implicitly allocates a temporary local port so that the server can properly address messages that it sends to the client. In contrast, a TCP server reserves a port when it starts and then receives connection requests on that port until it quits.

Network Address Port Translation (NAPT) [SE01] can relocate a portion of the IP address space into the port space to increase the pool of available addresses. NAPT maps multiple local IP addresses to one global IP address using the transport layer. In most use cases, a router is used to connect a local area network (LAN) to a comprising wide area network (WAN), such as the internet. In that case, whenever an application in the local network connects to a remote server and binds a dynamic port, the router binds a different dynamic port in its global port space directed towards the internet, which it uses for all future datagrams on this connection. An example is given in Figure 2.2. In each outgoing datagram, the router replaces the source IP address with the global IP address, and the source port with the global port (A). In turn, each incoming datagram is modified to contain the local client address and port (B). For each of those translated datagrams, new IP and TCP checksums need to be calculated [SE01]. Using NAPT,

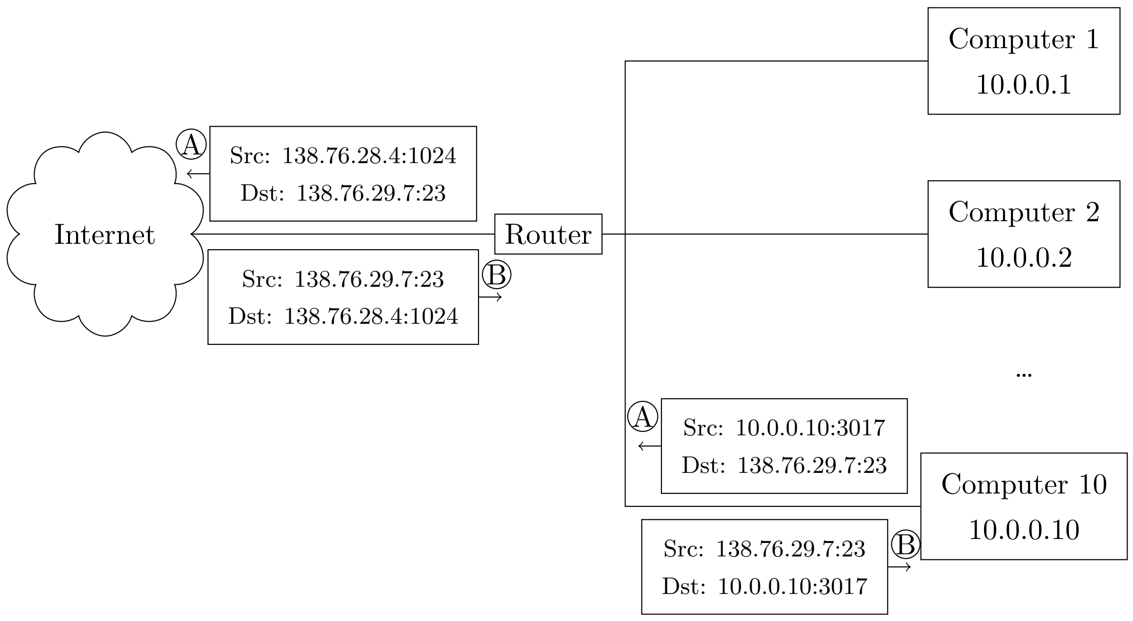


Figure 2.2: An example of Network Address Port Translation (NAPT) from [SE01]. The separate flows (A, B) are explained in the text.

millions of households worldwide can easily connect a dozen network devices to the internet using only one global IP address. However, the TCP and UDP port spaces only comprise 65535 ports each, which might not suffice when a large number of services are active in the local network. This does not have to be a problem in a private LAN, but it will be in a company with a few hundred employees.

Summary

The internet protocol suite offers a set of flexible and powerful protocols that can help to implement almost all networking scenarios. A subset of those protocols has been presented in this section.

The layer model of the TCP/IP stack provides a concise separation of concerns, which increases flexibility. Nevertheless, each layer needs to maintain its own separate state, including a proper address space. If one of the address spaces in the network fill up, it becomes difficult to add new clients.

2.2 TCP/IP stack implementation

Traditionally, TCP/IP has resided in the operating system kernel in Unix operating systems like Linux (see Figure 2.3(a)). In such a setup, the in-kernel network device driver controls the network hardware, and the TCP/IP stack filters packets, forwards them to the respective processes, manages the protocols' port space, and, for the connection-oriented protocols, assures that sessions are properly shut down even if the corresponding network process has quit.

The kernel networking code is designed to support as many use cases as possible, because it is supposed to be the single reference point for any network activity in the system. However, this means the kernel code cannot be optimised for performance: Intel reduced the networking overhead to 200 clock cycles per packet with a custom userspace network stack, which is about two orders of magnitude below that of the Linux kernel [Cor; Gra]. Also, design decisions that are taken for the network implementation always apply system-wide. An example is latency bounding. The earliest TCP implementations sent user data onto the network immediately when they were ready. With this strategy, a single remote shell session could congest a network. The telnet protocol, which was popular at the time, transmitted keystrokes letter by letter. A user typing in a command caused a sequence of packets with a payload of one byte, each baring a 40 byte protocol header. A TCP stream filled with such packets carries 40 times as much header information as user data, which is an overhead of 4000 per cent [Nag84]

In the next step, fixed timers were introduced in the TCP code. The TCP/IP stack accumulated payload data for a specific interval, and then wrapped the data into one or more packets and sent it. Naturally, latency-critical applications required different timer values than throughput-intensive ones, and setting low timer values could lead to congestion if the network was too slow to cope [Nag84].

The algorithm introduced by John Nagle in 1984 in response to those problems [Nag84] accumulates data until all previously sent data has been acknowledged by the receiving end, and then sends them, so that no fixed timer values are required. This algorithm works well for most interactive TCP applications, but Nagle himself already admitted that heavily throughput-intensive applications suffer from the additional initial delay of the algorithm, because the TCP implementation waits for one round-trip time until it sends the rest of the data. In addition, Nagle's strategy does not avoid small datagrams, which are still sent immediately if the line is clear. Therefore, modern Unix operating systems like Linux or the BSD family offer a socket option that forces the stack to send only TCP segments with maximum length [Bau06]. Such an option makes the kernel code more complicated than it has to be: If the networking code resides within the applications, every implementation can be adapted to a specific purpose independent from all others, and the separate code bases each become more straightforward and consequently easier to maintain.

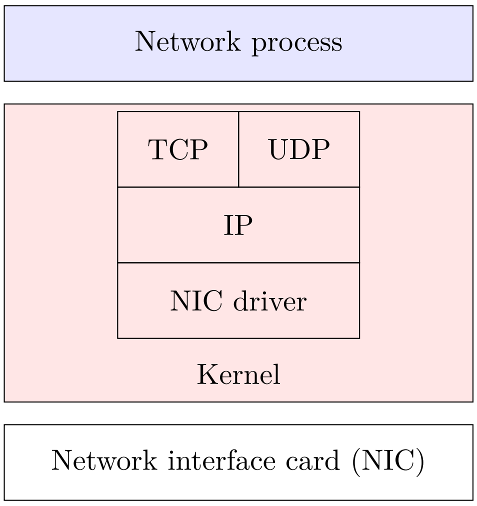
Therefore, the need for flexible TCP implementations was the main motivation behind a number of custom research implementations of the internet protocol suite in the beginning of the 1990s. Most authors initially shied away from complete userspace network stacks, stating performance problems, and concentrated on moving the TCP protocol handling to userspace, leaving UDP and performance-critical TCP functionality in the kernel (see Figure 2.3(b)). An exception is the earliest paper on the topic [MB93], where the authors already implemented everything except the NIC driver in userspace, in a setup that resembles that in Figure 2.3(c).

Today we see complete TCP/IP stacks in application libraries, which are capable of producing ready-to-send network frames. I highlight a few of those custom TCP/IP implementations in Section 2.5.3. Among the available userspace stacks, I chose the Rump TCP/IP implementation as the base for my prototype. I explain Rump kernels in Section 2.2.3, and Rump TCP/IP will be covered in more detail in Section 3.5.1.

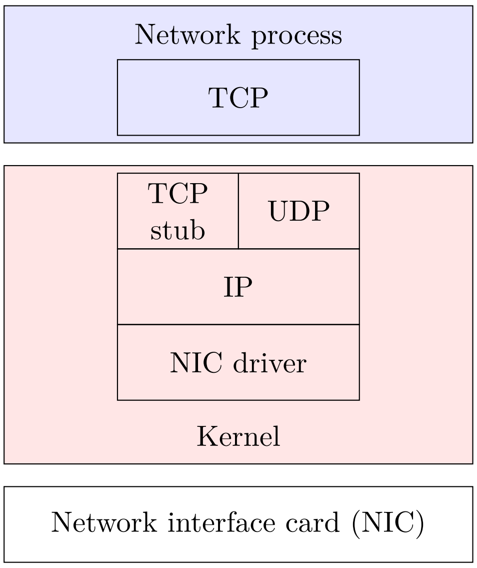
The development of full-featured protocol implementations was also quickened by better interfaces for forwarding network frames from userspace to the hardware. The best-known example is tap networking. This approach enables an authorised process to check and modify the network packets before the kernel sends them, and to send custom packets itself, on a virtual Ethernet interface provided through a standard Unix file descriptor. Similar mechanisms include Linux packet sockets [Pro] and the BSD packet filter (BPF) [Har02]. Because tap networking is not optimised for efficiency, direct access to the network card is more attractive in many cases. A device driver is still required for link-level access, though, and may be located either in userspace (see Figure 2.3(d)) or in the kernel (see Figure 2.3(c)).

If the NIC driver resides in userspace, applications can send packets to the driver via message passing or shared memory, or, if the driver resides in a library, via simple function calls (the latter is shown in Figure 2.3(d)). The driver has the network card's memory mapped into its address space and can transfer the data directly to the hardware, so the kernel is hardly involved in networking at all. Unfortunately, this model is not widely used yet, although the sv3 software switch has provided promising numbers with its attached userspace device driver [Ste14b].

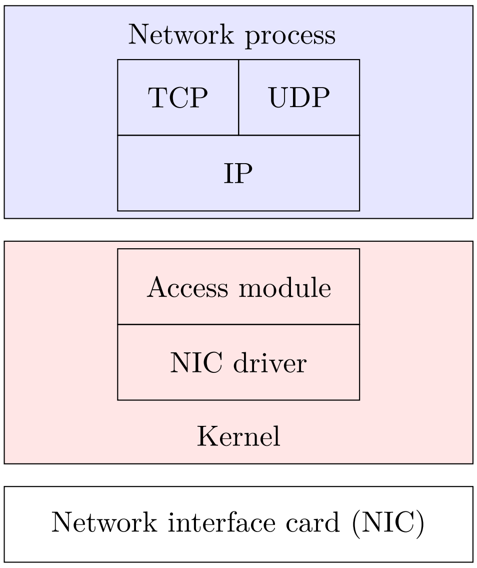
In case of an in-kernel driver, there are a number of implementations for direct network access from userspace. With netmap [Riz12], for instance, the user can access a virtual network card interface through a library that mimics the hardware layout. The netmap kernel module forwards the changes from userspace to the physical NIC as soon as a system call arrives, and thus reduces the system call frequency by batching multiple packets in one transmission, which is the main reason for its performance gain: context switches from kernel to usermode and handling interrupts consume more time than the mere data transfer itself [Riz12], and with tap, basically every sent packet results in a system call, and every received packet causes an interrupt. Netmap can easily transmit 10 gigabits per second [Riz12], which is also true for the alternative from Intel, the Data Plane Development Kit (DPDK) [Cor13]. In contrast to netmap, which reuses kernel drivers and offers a standard POSIX file descriptor to userland applications, DPDK is a software framework that redefines network access entirely. DPDK is optimised for throughput instead of usability. It processes the network data in parallel on multiple logical cores, which exchange data through common ring buffers. Interrupts are disabled because interrupt processing would be too costly, so the programmer has to poll the network interface for incoming packets. As the name says, DPDK provides merely the data plane, so all data paths and all required resources need to be preallocated before the actual network processing starts (*run to completion*). An Environment Abstraction



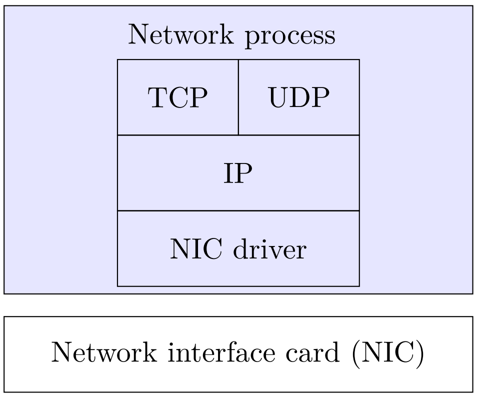
(a) kernel networking



(b) TCP protocol library



(c) direct NIC access (kernel driver)



(d) direct NIC access (userspace driver)

Figure 2.3: The four different networking software architectures mentioned in the text

Layer (EAL) serves as a kind of device driver, so that network applications can be programmed independent of a specific hardware architecture. DPDK's simple pipelined layout enables high throughput rates: As I stated in the beginning of this section, Intel was able to outperform the default in-kernel network stack by two orders of magnitude using DPDK.

Apart from flexibility, the isolation of userspace TCP/IP stacks also improves the security of the whole system. Attackers gaining hold of the stack only have user privileges, and can only disclose information from one network process, not from all networking activity on the system. Techniques like sandboxing can help enforcing the boundaries between the applications. In contrast, the in-kernel networking code is executed with all available rights, and intruders have the whole system at their hands. Also, isolation of the networking code also benefits fault tolerance, because bugs in one application need not necessarily compromise another process in a separate address space. Faulty kernel code, however, is capable of bringing the whole system down, including entirely unrelated components.

In addition to reducing the implications of an attack, the attack surface can be decreased if the applications in question rely on a codebase that is as small as possible. I explain the so-called *trusted computing base* in Section 2.2.2.

Summary

We have learned that fully isolating the TCP/IP stack is desirable, for reasons of performance, flexibility, security and fault tolerance. However, I have not yet discussed the integration of multiple independent TCP/IP applications on the same system. Tasks that are traditionally centralised in the kernel, like port space administration and packet multiplexing, are now scattered across multiple processes.

2.2.1 Integrating independent network applications

With userspace TCP/IP stacks, as soon as there is more than one application, there is always the problem of how to connect them to each other without conflicts. Each application hosting a userspace TCP/IP stack produces ready-to-send network frames, which means we need a way to distribute those frames in an orderly fashion.

Currently, each application requires its own MAC and IP address to be distinguishable from the others, which increases the demand for addresses of both of those address families. Given that we are running short of IPv4 addresses, an increased demand is problematic there. The techniques to date are the same that I mentioned before: CIDR and NAPT; only now they are not only applied on the network, but on every single physical host, too. That means that an additional switching layer is required. As explained by Honda et al., the separate stacks can either be connected to the hardware directly, or they can be switched in software [Hon+14].

If the hardware does the switching, each frame will have to be copied out to the NIC, where it may have to be sent back again if the destination is on the same physical machine. Basically every network card with dedicated application memory, such as Arsenic [PF01], can perform basic switching tasks. I detail on improved network hardware in Section 2.5.

In software, the most notable solutions are VALE [RL12] in the kernel and sv3 [Ste14b] in userspace. Both can achieve 10 gigabits per second line rate, and are especially fast when transmitting on the same machine: in that case, ideally, the packet can be transmitted using a pointer in an address space shared between the two communicating processes, requiring no copies at all.

Summary

Existing techniques for interconnecting independent network applications supply sufficient performance, but tend to drain the MAC and IP address space.

2.2.2 Trusted computing base

Most programs import functionality from independent libraries that need to be correct and secure. If the software depends on an operating system and its application binary interface (ABI), that operating system needs to be trusted. There are also applications that rely on the functionality of other processes running on the same system.

When the development of secure operating systems started at the end of the 1970s, computer scientists and security experts examined complex operating systems for their security properties, and stumbled upon the distinction between those system components that were security-critical and those that were not.

In the course of that debate, J.M. Rushby published a paper where he introduced the concept of a logically distributed operating system as a solution [Rus81]: All system components should be thought of as isolated machines that were only connected to the others via dedicated lines of communication with strictly limited protocol capabilities. A secure kernel would now merely have to enforce the security policies on those lines, and as a consequence the whole system would adhere to the security guidelines.

Alongside his considerations, Rushby coined the term *trusted computing base*, 'the combination of kernel and trusted processes' [Rus81]. 'In order to guarantee security [...] we must verify the whole of the "trusted computing base"' [Rus81]. Hence a smaller TCB decreases the validation cost, and has the additional advantage of a smaller attack surface.

The Nizza secure-system architecture [Här+05] tries to minimise the TCB of a system implementing the original principles of Rushby's work: on top of a small kernel, each operating system functionality is provided in a separate server process. Secure applications need only rely on the kernel and the server facilities they require.

This architecture has two main advantages: First, the trusted computing base can be reduced to what the secure applications on the system require. Second, in the

Nizza architecture, the trust chain of the individual processes varies, too: If there is a security flaw in one operating system server, that flaw does not necessarily affect another application running on the same system if it does not make use of the server.

The principles of the Nizza architecture can be applied to Linux, too. Figure 2.4(a) shows a simplified Linux setup where the applications run directly on top of the Linux kernel. If all high-level operating system features remain in the kernel, networking applications need to trust unrelated kernel subsystems like the storage unit and its underlying drivers. Vice versa, programs that do not require kernel networking still need to rely on a correct network stack.

If we move the drivers and the network and storage functionality into userspace, neither application needs to rely on functionality that it does not use. The result is displayed in Figure 2.4(b). The components that provide the required code are now encapsulated within the respective processes. The trusted computing base is now smaller and thus easier to verify, and none of the applications need to take code into account that they do not use.

As a rough estimate of what could be removed from the kernel, I have measured the size of the net subfolder of a current Linux 3.14.0 kernel, which alone comprises 586,916 SLOC[1]. My work provides a convenient way to move the TCP/IP network code from the kernel into the applications, thus reducing the system's TCB, while at the same time maintaining the well-known connection characteristics of in-kernel networking.

Summary

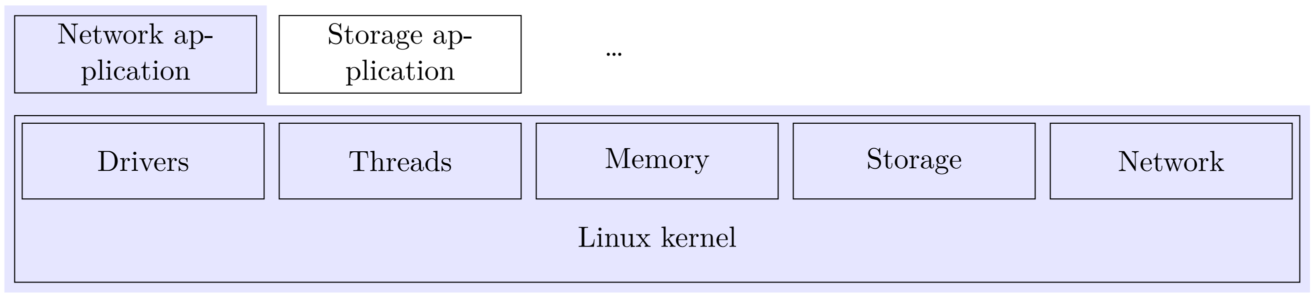
The trusted computing base (TCB) closely relates to a system's complexity as well as its attack surface. Decreasing the size of the TCB to a minimum is vital for a secure system. Many commonly used contemporary operating systems retain a monolithic design approach, which makes them less secure.

2.2.3 Reusing existing kernel code

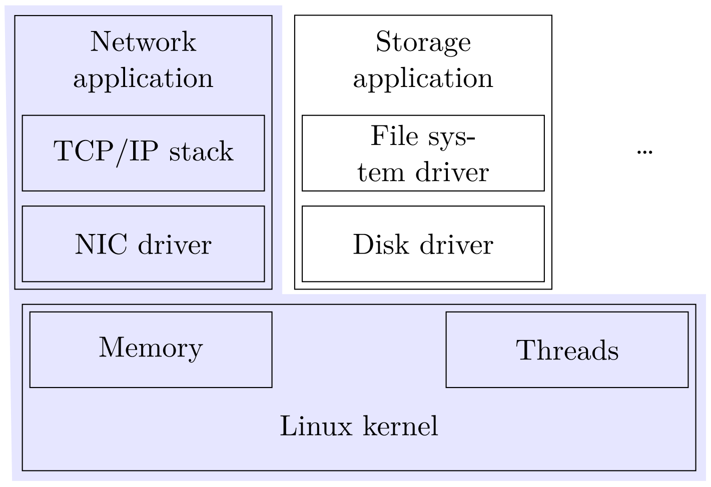
The entire Linux source code repository comprised 15 million lines of code in 2012 [Lee12]. More than half of that code base consists of drivers [Pal+11]. One of the main problems with driver development is testing the driver, because debugging the kernel is harder than debugging userspace applications. Hence isolating drivers is a good measure to improve their code quality, and in turn system stability and security.

However, reusing code that is tightly integrated into the OS kernel is difficult: on the one hand, the former kernel code cannot refer to other in-kernel units through simple function calls any more, and on the other hand, the direct view of the hardware that the kernel provides, such as access to physical memory addresses or device registers, is not available from userspace.

[1] Source Lines of Code, measured using SLOCCount 2.26 [Whe]



(a) in the standard setup



(b) using a stripped-down kernel

Figure 2.4: The TCB (light blue) of a network application under Linux

The device driver OS project [LeV+04] pragmatically decided to leave the driver code in its original environment, but to run it as an isolated process in a virtual machine (VM). The virtual machine executing the reused driver gets unlimited access to the device in question, hence the driver can fully utilise it. An RPC server, a so-called *translation module*, integrates the driver VM into the host system. For each device class, such as network or block devices, there is a translation module for clients on the host, so that the hardware access is independent of the particular device. The network performance of the device driver OS was comparable to that of a natively running Linux, with performance decline between 3 and 8 per cent. However, each device driver VM required several megabytes of main memory, and at times consumed more than twice as much computing time as the native driver.

Despite the high resource consumption, device driver VMs are still a popular concept for driver isolation: The Xen driver domain [MCZ06], for example, works in a similar fashion. The device driver is encapsulated in an ordinary Xen virtual machine that has elevated rights on the device for which it is responsible. The other virtual machines access the hardware through a generic virtual interface, which forwards their requests to the driver VM.

To decrease the overhead for running a device driver, it may be worth to take a look at the actual requirements of the driver. Recently, two major research projects examined partial virtualisation of kernel code: DDE [Groa] and Rump [Kan12b].

DDE, the Device Driver Environment, was created as a part of the L4/Fiasco operating system. In contrast to the device driver OS, which has a bottom-up approach to driver integration, DDE approaches the problem from the driver top-down: it provides the driver with the interfaces that it needs, such as memory management or hardware input–output access, and forwards requests on those interfaces to the individual L4 servers. Vice versa, the device driver acts as a server that processes requests from L4 clients that want to access a specific hardware component. DDE is split into the DDEKit, which provides general driver wrapper functionality, and the DDE backends for specific operating systems. At the moment, DDE supports drivers from Linux and BSD systems.

Rump, on the other hand, provides a way to run arbitrary kernel code outside the actual kernel. Like DDE, Rump provides that code with high-level abstractions for the required operating system functionality. Rump is specifically tailored to the NetBSD system, so it can only support parts of the NetBSD kernel. In return, those code fragments run in manifold environments: as part of a process on a POSIX-compatible system, as part of the Linux kernel [Kan13a], without an operating system in the Xen hypervisor [Kan13b], and even in a browser [Kan12a], among others.

To avoid having to trust all of the NetBSD kernel, the kernel code is separated into three so-called *factions* and several device drivers that depend on a selection of those factions. Figure 2.5 depicts a specific example for an application that uses the shmif bus explained in Section 3.2. Because no driver currently requires the whole NetBSD code, the compute time and memory footprints of drivers in Rump kernels are about one order of magnitude below those of the device driver OS [Kan12b].

Rump kernels on a POSIX system can either be embedded into an application as a library (*local* mode), or they can serve local or remote procedure calls (modes *microkernel* and *remote*). The microkernel mode resembles DDE's mode of operation.

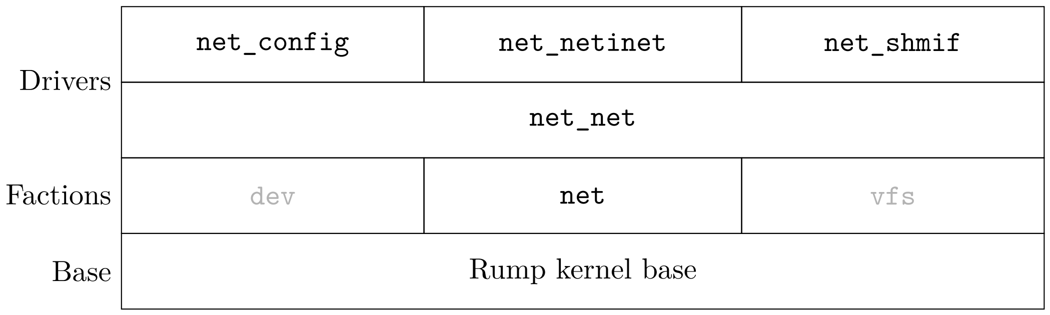


Figure 2.5: An example of Rump kernel partitioning for an application that uses the *shmif* bus (see Section 3.2). The factions that are labelled grey are disabled and thus not part of the application.

I decided to use Rump kernels because their flexible design enabled me to embed the well-tested and feature-rich TCP/IP implementation of the NetBSD kernel directly into my applications without any additional work to be done (see Section 3.5.1).

Summary

Device drivers are one of the major points of failure in an operating system. Consequently, the research community conceived several projects to isolate drivers from the critical parts of the operating system. Starting from there, Rump has evolved into a software that runs arbitrary parts of kernel code in manifold environments. Due to the efforts of the Rump developers, well-tested and feature-rich low-level software is available to higher-level applications as well. It is now important to use those capabilities to achieve a smaller TCB.

2.3 The problem

From the facts presented in the previous sections I conclude that the following open problems need to be solved:

1. Decreasing the size of the TCB to a minimum is vital for a secure system. Many commonly used contemporary operating systems retain a monolithic design approach, which makes them less secure.

2. Due to the efforts of the Rump developers, well-tested and feature-rich low-level software is available to higher-level applications as well. It is now important to use those capabilities to achieve a smaller TCB.

3. Userspace TCP/IP stacks are a promising technology, but they introduce a distributed network state that the system needs to handle.

4. Existing techniques for interconnecting independent network applications supply sufficient performance, but tend to drain the MAC and IP address space.

5. The TCP/IP standards define that IP addresses should specify hosts and TCP and UDP ports should specify applications [Pos81b; Pos81c]. Existing interconnects partially support the original addressing scheme, but need to be explicitly configured to adhere to it.

My solution to those issues is described in the next section.

2.4 A dedicated switch

In this work I introduce Swarm, which is a dedicated userspace port switch for interconnecting independent TCP/IP applications on the same host.

Swarm decreases the system TCB, because it is kept so small that a code review is easily possible, and because it eases the deployment of userspace TCP/IP which enables a smaller kernel without networking functionality. Non-networking applications do not need to trust network code any more, and even though all network processes need to trust Swarm, they are still all isolated against each other, so that errors in one process do not directly affect another.

A networking setup including Swarm and a userspace TCP/IP stack will depend on much less kernel code than conventional in-kernel networking. Even with a conventional kernel, you only have to call the device driver in the Linux kernel, and not the whole socket API.

At the same time, Swarm makes connecting userspace TCP/IP processes easier, because technologies like virtual Ethernet or NAPT become unnecessary when all stacks are using the same IP address and the MAC address of the network card. Swarm uses MAC addresses for NICs, IP addresses for hosts, and ports for applications, and thus helps porting legacy applications to userspace stacks as well as integrating new applications with custom network stacks into existing networks. Because Swarm assigns the same addresses to all applications, it needs to distinguish between them using the ports of the transport layer. This approach is conceptually similar to NAPT, except that Swarm has only one global port address space, so the switching is transparent for the attached applications.

Due to the higher significance of IPv4, the current Swarm implementation supports only the older version of the IP protocol. Swarm can, however, help increase the available IPv4 address pool by assigning the same IP address to all its connected applications. Consequently, there is no need for a virtual local area network on the host which would require multiple IP addresses.

As a reference implementation, Swarm uses Rump TCP/IP, which provides a well-tested and feature-rich base. Still, developers can choose any TCP/IP stack they see fit. Swarm supports every TCP/IP stack that adopts the system MAC and IP address and understands Swarm's inter-process communication (IPC) protocol (see Section 3.4.2) as well as the shmif data exchange protocol (see Section 3.2). Allowing arbitrary stack implementations in the individual programs instead of forcing them to use the one in the kernel helps to deploy new TCP/IP features more quickly [Hon+14], and also facilitates maintenance.

2.5 Related work

The aim of this work is to move networking away from the kernel. Apart from software solutions, there is also special hardware that allows applications to bypass the operating system and send their content directly to the NIC.

2.5.1 Arsenic

I look into the Arsenic project [PF01] first, because it shares several methods and goals with Swarm.

Arsenic exports virtual NICs to be used by the applications, and allows the operating system to define packet filters that influence the distribution of network frames. The applications can map a part of the physical NIC's memory into their address space, so that packets need not be passed through the kernel and isolation can be ensured by the hardware memory management unit (MMU).

Arsenic shapes traffic and queue length for each virtual NIC according to the requirements the application has postulated. All virtual NICs that still have bandwidth credit are served in a round-robin fashion to avoid packet bursts. The attached network processes are notified of new packets through interrupts which arrive according to their latency requirements, thus latency-critical programs do not experience poor performance due to data-heavy ones using the same interface, and vice versa. Because of those sophisticated mechanisms, Arsenic's flow control can also be used to enforce real-time constraints on latency and bandwidth.

The TCP/IP stack can be a part of the individual applications (as described in Section 2.2), but network processes can use the kernel network stack, too. Arsenic uses the hardware to switch between the different stacks. Consequently, custom stacks need to register each connection in a central IP registry. A custom library provides abstract methods for access to the NIC (see Section 2.2).

Arsenic's position in the hardware enables faster packet processing compared to software solutions, and also speeds up the implementation of policies like traffic shaping or latency bounding. On the other hand, Arsenic *does* require highly customised hardware, which is often infeasible, whereas Swarm as a general-purpose software solution supports a wider range of systems.

Pratt and Fraser developed a research prototype for GNU/Linux 2.3 using an Alteon ACEnic adaptor with DMA-accelerated queues. Although Arsenic has an abstract design so that it can run on other hardware architectures, too, it has not been developed further. Instead, similar principles are nowadays applied in TCP offload engines.

2.5.2 TCP offload engines

TCP offload engines (TOEs) allow the user to offload TCP/IP partially or completely to the network hardware. The application merely provides raw send and receive buffers, and the TOE in the NIC is responsible for the network processing. Custom-tailored hardware is often faster than general-purpose hardware and consumes less energy performing a certain task [Ham+10]. Also, network hardware has become increasingly powerful, so that it can accomplish other tasks in addition to mere packet processing. Examples for NICs that have a TOE include the FPGA hardware TCP/IP stack of the Fraunhofer Heinrich Hertz Institute (HHI) [Ins], which even allows offloading part of the application layer, the Broadcom NetXtreme II family [Cor08], and the Terminator 4 and 5 by Chelsio [Com].

The advantage of a TOE is that the processes that require network connection improve their performance and still maintain a single network address for a single host. However, although the software TCB is significantly reduced, users now need to trust the hardware to do network processing and isolation correctly.

2.5.3 Custom software networking stacks

In 1993, Maeda and Bershad published a paper describing a split TCP/IP stack for the Mach operating system [MB93]. They were able to compete with in-kernel solutions available at the time, like that of the UX operating system, and in some cases even outperformed them.

The authors reuse the existing TCP/IP code of BSD, and establish the performance-critical send and receive functionality as an application library. An operating system server contains the rest of the protocol implementation. For instance, the server controls connection establishment or address resolution, and once the connection is established, the program can send and receive data without additional copy operations or costly context switches. The server and the applications communicate using Remote Procedure Calls (RPCs). The performance-critical parts were moved into the network process because the authors had encountered a performance decline of 2–4 times in an earlier work of theirs [MB92] using a purely server-based solution.

The decomposed Mach network stack was highly influential to Edwards and Muir from HP Labs Bristol who implemented a part-userspace, part-kernel TCP implementation, based on HP-UX and an Asynchronous Transfer Mode (ATM) network [EM95]. Because their approach is similar to mine, I detail on their work in the following paragraphs.

Edwards and Muir decided to extract the kernel TCP/IP code from their HP-UX system, leaving packet demultiplexing and buffering in the kernel for performance and space reasons. Because multiple processes now communicated using one kernel memory region, this region was separated into fixed-size memory pools that were statically assigned to the network clients to avoid resource conflicts.

To further accelerate packet demultiplexing, a system-wide *connection server* assigned an ATM channel ID[2] to each outgoing TCP connection, so that the device driver on the receiving side could forward incoming TCP packets to the correct location without consulting the kernel.

For kernel TCP to work in userspace, Edwards and Muir needed an instance that could perform the asynchronous TCP operations and make sure that connections could survive the processes that initiated them. In the end, they found a way to exploit the structure of their operating system core: HP-UX was derived from the original 4.3BSD, where the kernel consists of two parts: a *top half* geared towards the applications, which handles the syscalls, and a *bottom half* which handles interrupts and thus represents the kernel's hardware interface. Porting the kernel network code to userspace, they maintained this split structure using two processes: the application process with the network library, and the child process spawned by the connection server. The child process communicates with the network hardware and manages the low-level protocol features, such as triggering timers and sending remaining data after the application has quit.

With their solution, HP Labs were able to outperform the HP-UX kernel network stack, with a throughput of up to 170 megabits per second. However, the authors then ported their modified TCP code back to the kernel, and userlevel TCP could only achieve 80 per cent of the improved kernel implementation's throughput, at the cost of a 10 per cent increase in CPU utilisation.

In the same year, Braun et al. from the French INRIA published an experimental userspace TCP implementation that emphasized protocol flexibility rather than performance [Bra+95]. Like their colleagues from HP Labs, they left packet buffering and packet demultiplexing in the kernel, but while Edwards and Muir used their NIC driver and ATM channel IDs to avoid TCP multiplexing, Braun et al. decided to provide a hardware-agnostic TCP port demultiplexer, so that their solution would work on a broad range of systems. The remainder of the TCP code was maintained within an application library so that it could be exchanged easily. Braun et al. did not see the need to spawn a second process. The maximum throughput of their TCP implementation was 10 megabits per second, compared to 35 megabits per second that an ordinary implementation achieved on the same machine (DPX/20 at 42 MHz running AIX 3.2.5).

[2] ATM is a packet-oriented data transmission scheme that provides multiple logical connections on a single physical line. Those connections are specified using virtual channel identifiers and virtual path identifiers, abbreviated VCIs and VPIs, respectively.

2.5.3.1 Contemporary solutions

Solarflare's OpenOnload [PR11] is a recent example for a full featured part-kernel, part-userspace (hybrid) internet protocol stack. OpenOnload works with DMA queues in the NIC, which enable hardware multiplexing similar to TOEs and Arsenic. For access to the stack, there is a passive socket library for applications that stores the state of each socket in userspace and in the kernel.

The authors state several reasons to keep a part of the network stack in the kernel.

1. The kernel still maintains the network address spaces: each OpenOnload socket has a kernel socket as a counterpart, which reserves the IP address and port. The authors claim that UDP packets can be received from any interface, directly through the library for those that have OpenOnload support, and indirectly through the kernel stack and the kernel socket for those that have no OpenOnload support.

2. The kernel component makes sure that the state of a network connection persists after the process has exited, and that the stack responds timely whenever necessary, even when the corresponding application is not scheduled.

3. Highly threaded applications benefit from a resident kernel module because the switch between kernel and userspace does not involve any data copying.

4. The kernel part helps to maintain compatibility with the Unix programming model. Operations such as `fork` and `exec` duplicate or overwrite the application's userspace state, respectively, including the network stack. During such an operation, the kernel module preserves the socket state and the new application can map the appropriate kernel region back into its address space afterwards, if required.

As a consequence of the latest point, two processes can also share sockets, although the default behaviour of OpenOnload is not to share state between independent applications to avoid data leakage.

The problem with OpenOnload is that it extends the kernel and thus increases the TCB instead of decreasing it. Furthermore, the kernel module exports kernel memory into userspace. According to the creators, the pointer arithmetic for accesses from userspace is simple enough to be trusted [PR08], but it would be more reassuring if there was no kernel component to trust at all. Also, the access mode of the socket is migrated transparently, which is possible because every OpenOnload socket has a kernel counterpart. Consequently, network applications always need to check whether they are still operating in userspace or if their socket is now operated in kernel mode, especially after a `fork` or an `exec` operation.

Recently, Honda et al. presented a project called MultiStack [Hon+14]. Similar to the creators of OpenOnload, the authors argue that earlier userspace TCP/IP implementations like those presented in Section 2.5.3 have not seen widespread deployment because of missing backwards compatibility. Consequently, they allow arbitrary networking implementations in the applications, and connect them in the kernel using netmap to establish the data path and VALE as an efficient in-kernel switch. The kernel stack, the userspace stacks, and every NIC are each connected to one port of VALE. A separate so-called *namespace sharing module* administrates the port and address space. Each packet with a (protocol,address,port)-3-tuple that is not yet assigned is delivered to the kernel stack by default. With this central address administration, it is possible to reuse legacy applications without adapting them to the new model. Also, all applications on the same host can share an IP address, so long as they also share a NIC, although in contrast to Swarm a shared address space is not mandatory. Despite the good performance of 10 gigabits per second, MultiStack, like OpenOnload, has the weakness that it extends the kernel and thereby the TCB of the system.

On the contrary, Genode 13.11 [Gmb] offers a solution called LXIP that runs entirely in userspace. Genode Labs have developed a library embedding the complete GNU/Linux TCP/IP stack into a runtime library. A userspace virtual Ethernet switch routes packets among the separate application hosts. LXIP maintains a small system TCB; however, assigning each network process a separate IP address may be problematic regarding the sparse IPv4 address space.

3 Swarm — A userspace port switch

In this chapter, I introduce Swarm, the common component of the split TCP/IP stack.

Concluding from the observations made in Section 2.4, Swarm has to accomplish four goals to improve existing work:

1. Swarm needs to isolate the TCP/IP code from the kernel. The complex, untrusted portion of that code shall reside within the individual applications, so that they are also largely isolated against each other.
2. Swarm shall be kept so lightweight that a code review is easily possible to keep the TCB for network applications small.
3. Swarm shall enable all applications on the same host to use the same MAC address and IP address, to maintain compatibility with the original TCP/IP addressing scheme and increase the available address space without complicating the network setup.
4. Swarm shall allow the applications to use any TCP/IP implementation they see fit, so long as the implementation complies with Swarm.

Swarm implements an association between one host interface and one or more applications. Each application can use their own TCP/IP implementation (condition 4), so it is difficult to choose one stack to extract shared code from, because that contradicts the whole idea of having flexible stack implementations in the first place. Also, the different protocol layers of the internet protocol suite are closely linked and often crossed in the actual implementations. So instead of reusing parts of an existing TCP/IP implementation for Swarm, I decided to develop a separate component from scratch to accomplish packet switching.

Figure 3.1 shows Swarm's position in the software stack. On the system, there are multiple applications with complete TCP/IP stacks that can produce ready-to-send Ethernet frames. Swarm serves as a multiplex between all those stacks, controlling access to the network card and thus administrating the port space that is associated with the NIC's MAC address and the corresponding IP address. Hence, Swarm checks each datagram sent by the applications for its destination port on the transport layer, and forwards it accordingly, that is either to the network card using a dedicated kernel module or to another application (see Section 3.3).

An additional component in the software stack is required to be fast and lightweight. Hence, I decided to use a single thread and asynchronous communication on all channels. Recent research work has shown the advantage of asynchronous communication compared to synchronous, threaded models. IsoStack, for one, is the effort to run all

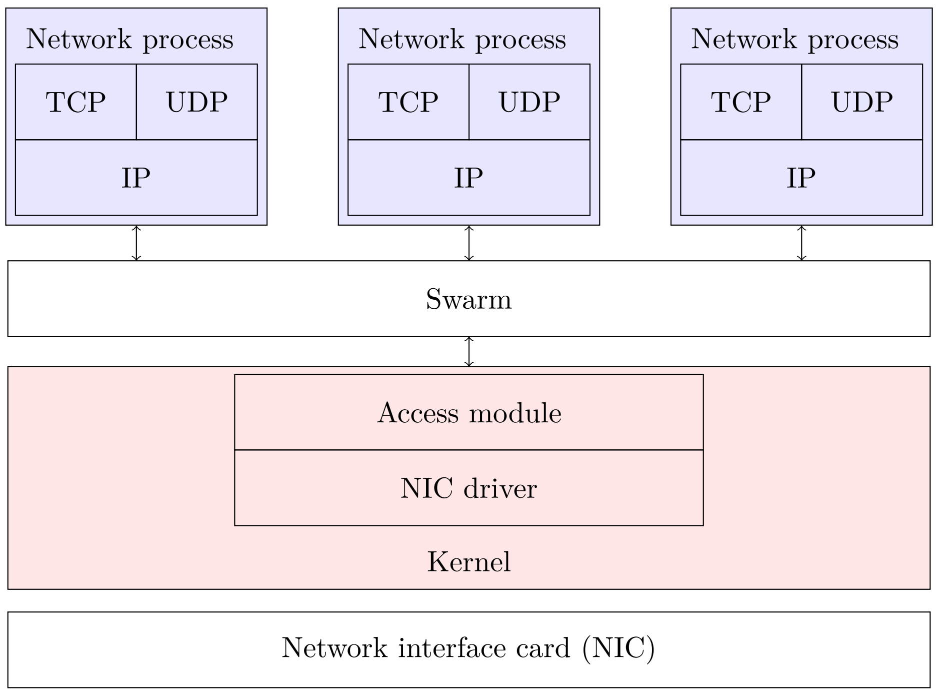


Figure 3.1: An overview of Swarm's architecture (explained in the text).

TCP/IP code on a single processor [Sha+10]. They make locks, interrupts, and synchronisation in the network code redundant, and introduce an efficient message passing scheme between separate processor cores, which results in a performance increase of up to an order of magnitude compared to a stack implementation that runs distributed across multiple cores. Another example is the Lighttpd webserver [Kne07], which, compared to its competitor Apache, handles more requests using fewer resources due to its asynchronous design: Lighttpd runs single-threaded with a single event loop, in which it accepts input and produces output on an arbitrary number of sockets. So using a central, asynchronous event loop, I could maintain a simple, lightweight design and avoid overheads for locking, synchronisation, and scheduling.

I conclude that Swarm consists of only one thread running an event loop. Now to find out what events Swarm needs to handle, we need to take a look at its communication ports. As Swarm is translating information from a virtual network of processes on the local system to a real network, it can also be thought of as a simple router. This analogy enables us to connect an interesting concept to Swarm: the separation of the router logic into a control plane and a data plane. The control plane is responsible for establishing the correct forwarding paths between the clients, and data plane enforces the packet forwarding policies of the control plane during data transfer.

Swarm's clients are the applications running on the system, and their addresses are the ports they are using. For Swarm's control plane, we need a way to register and remove ports, and for the data plane, we need a mechanism to transfer the application's payload data on its reserved ports to and from the physical network. To facilitate the following explanations, I refer to Swarm's control plane as Swarm core and to Swarm's data plane as Hive from now on.

First the applications need to know if they can reach Swarm, and where. Swarm is not a part of the applications themselves, so the applications interact with Swarm core via inter-process communication (IPC). Consequently, they need to agree on an IPC protocol and a well-known IPC address to enable a connection.

In a Unix system there are multiple ways to establish IPC connections. Most Unix systems ship with mechanisms for signals, sockets, pipelines, semaphores, and shared memory. I cannot use signals or semaphores because the IPC communication with Swarm core needs to transfer payload information like port numbers and protocol identifiers, and neither of the two mechanisms carry payload information, as they are mere wakeup signals. Also, control messages to Swarm core arrive asynchronously, so the used IPC mechanism should implement a signalling concept, too, which excludes shared memory. A pipe would only be an option for a one-to-one relation, because each end of a pipe cannot be held by more than one process at a time. What remains is a socket, so I decided to use a Unix domain socket as a control interface between Swarm core and the applications.

For their userspace TCP/IP stack, Edwards and Muir (see Section 2.5.3) also use a Unix socket [EM95]. In their solution, Unix sockets are used to pass messages for remote procedure calls (RPCs), but also to wake up the child process that they have associated with each network process. For simple message passing on a lossless data path on the same host, a datagram socket was the best solution. However, I intended to use one interface for all administrative tasks, also the data-intensive ones like setting up a connection to Swarm, so the messages could get arbitrarily long. To avoid maintaining overlong message buffers and potentially losing important information, I chose a connection-oriented Unix domain socket for Swarm core.

For the data plane, I decided to use a shared memory region backed by a file to conform with the Rump TCP/IP implementation, which offered the *shmif* interface for localhost communication that I implemented as a starting point. I explain the shmif interface in Section 3.2.

The next important question is how Swarm addresses its attached applications. Condition 3 states that all attached applications should use the same MAC and IP address, so the link layer and the internet layer cannot be used for addressing. Consequently, the packets are delivered to the applications according to the ports they reserved on the transport layer. Using TCP and UDP ports to specify applications is in accordance with the TCP/IP standards [Pos81b; Pos81c].

However, applications can reserve more than one port, so Swarm needs a way to find out what port belongs to which application. I decided to use a hash table that maps ports to their applications, because this data structure enables the quickest access. Address tables belong in the data plane, which is why the port-application table is a part of Hive.

With all data structures established, Swarm has but the following tasks in operation:

- Register and remove shared files for its applications

- Register (*bind*) and remove (*unbind*) ports for the connected applications
- Inspect the source and destination addresses and ports in incoming and outgoing packets and copy them to the according locations

With regards to my initial condition 2, I state that a reviewer could easily inspect a component that provides as few functions as Swarm. Also, Swarm is small enough so that it does not unnecessarily increase the TCB of its attached applications.

In the following, I explain the operating principles of Swarm (Section 3.1) and how bulk data transfer works (Section 3.3), before I get to explaining the details of the prototype (Section 3.4).

3.1 Swarm's operating principle

An application collaborates with Swarm in four stages which are depicted in Figure 3.2.

Firstly, the application needs to inform Swarm of its existence. To do so, it contacts Swarm at the well-known UNIX socket address and requests initial setup. In return, Swarm provides the application with the system's MAC and IP address and the name of the shared bus file (see Figure 3.2(a)). Each application has a separate bus file from all others, to ensure isolation of the applications.

Secondly, to communicate with the outside world, the application binds the ports that it needs to receive data on (sending data is allowed on all ports). If the port is already in use, Swarm returns failure, otherwise the port is successfully bound to the requesting process (see Figure 3.2(b)).

After binding the port, the application can start sending and receiving data through the established virtual shmif bus file, which is explained in Section 3.2, on the ports that it has reserved in the previous step (see Figure 3.2(c)). I detail on payload data transfer in Section 3.3.

Finally, the application may close the bus file, which will cause Swarm to remove all its port bindings (see Figure 3.2(d)). Swarm uses Linux' `inotify` [Lov05] mechanism to determine if a file has been closed. `Inotify` is a Linux library that notifies its user of certain file-related events like read, write or close operations performed on that file. The application may continue working, but to re-enable the networking functionality, it has to reconnect with Swarm.

3.2 The shmif interface

The shmif interface has been derived from the implementation of the shmif bus described by Kantee in his dissertation on Rump kernels [Kan12b].

Nowadays, Ethernet is often used for point-to-point connections, but originally, Ethernet was supposed to drive a bus that connected multiple computers. All hosts on a bus could see all Ethernet frames, and only copied those frames to the upper layers that were addressed to them. The idea behind the shmif bus is to connect multiple Rump kernels with each other with as little overhead as possible, using the original bus principle of Ethernet.

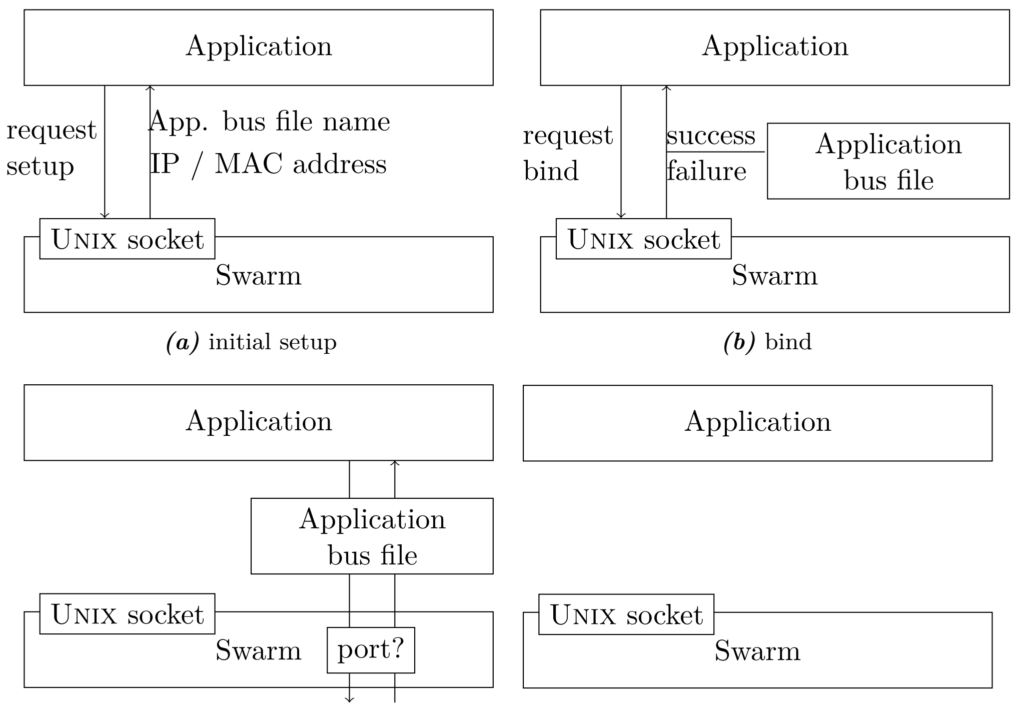


Figure 3.2: The operating phases of an application connected to Swarm. The separate stages are explained in the text.

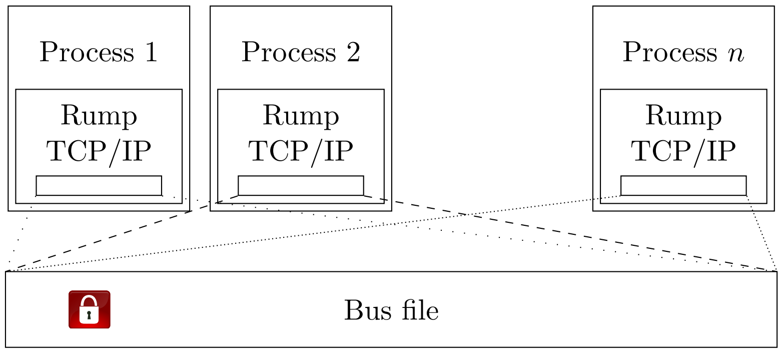


Figure 3.3: Diagram of the *shmif* bus with n connected processes. The processes have the bus file mapped into their address spaces to send and receive packets more easily. All clients agree to grab a lock before they perform a bus operation.

Figure 3.3 depicts the working principle of the shmif bus. All Rump instances share a file that constitutes the virtual Ethernet bus. Each Rump kernel maps the file into its virtual address space to facilitate bus access. To avoid runs on the bus, all participants agree to acquire a spinlock variable located in the header area of the bus file before performing a read or write operation on the bus. That means that access to the bus is serialised: one client can only either read or write at any given point.

To isolate applications from each other, only Swarm and one application are connected to one virtual bus, so that every process has its own data segment and possible misbehaviour of one program does not affect the others.

3.3 Sending and receiving payload

As I already mentioned, Swarm implements an association between one host interface and one or more applications. If there are multiple network interfaces on the same host, each requires a separate instance of Swarm. To access the physical NIC, Swarm uses the netmap interface [Riz12]. Netmap comprises a resident Linux kernel module and a userspace library for enhanced network access (see Section 2.2).

Data in Swarm flows in two directions: from the interface to one or more applications (*incoming*), or from the applications back to the interface (*outgoing*).

Incoming data from the NIC is first checked for its destination IP address. If that address equals the system's IP address or a local IP address (127.x.x.x in IPv4), and the transport layer protocol is either TCP or UDP, Swarm copies the IP datagram to the bus file of the application that has reserved the destination port. Section 3.4.3 discusses network protocols other than IP, TCP, and UDP in the context of Swarm.

Outgoing traffic handling works similarly. Swarm also checks outgoing datagrams for their source IP address. If the source IP address is either local or equals the system IP address, packet processing enters the next stage.

This check is necessary because data is transferred over a virtual Ethernet bus, and on an Ethernet bus, all clients receive all packets that are currently being transmitted. Hence Swarm also 'receives' packets that it sent to the network process itself. Packets that Swarm already sent must not be re-sent on the bus, or the system will quickly hang in an endless loop. It would not be necessary to check the source IP address if data transmission were realised using two separate ring buffers for sending and receiving data. I suggest an interface with that property in Section 5.1.

The next stage checks if the sending process has the source port reserved before sending the datagram to its destination. If the source port is not reserved, the packet is discarded. Otherwise, any process might try to established or destroy a connection on behalf of another, which allows malicious applications to impair the whole system's connectivity. Maeda and Bershad argue that such a source check is unnecessary and should be performed by a separate component like an application firewall [MB93]. However, I argue that it is vital for isolation that processes are not able impersonate each other either accidentally or voluntarily.

Data that is delivered locally is called *loopback communication*. Swarm also delivers loopback traffic according to the rules stated in this section, although I have not implemented this feature in the modified Rump TCP/IP stack that is currently used with Swarm.

3.4 The prototype

I have implemented a prototype that is logically divided into three subsystems:

1. **Swarm core**: the central component which also manages the data transfer
2. **Hive**: the part that administrates the connections and the local port databases
3. **Swarm IPC**: the remote procedure call (RPC) interface between Swarm, Hive and the applications

Figure 3.4 provides an overview of the packet send process in the Swarm prototype. The light blue boxes indicate separate userspace processes. After having registered with Swarm, an application can proceed to register a TCP or UDP port (step 1). Hive then stores an association of the port number to the ID of the bus file in its port registry (step 2). There is one hash table for each protocol port space, currently two for TCP and UDP. As soon as the port is registered, the application can start to send its Ethernet frames (step 3). Each frame is copied from the bus file (step 4) and passed to Hive for a sanity check (step 5). This pass is merely a function call, and is thus faster than the communication with external components (white boxes). If Hive returns a positive result (step 6), the frame is copied to the hardware interface or to a local recipient in case of loopback communication (step 7).

In the following subsections, I explain Swarm's three subcomponents, the Swarm core, Hive, and Swarm IPC.

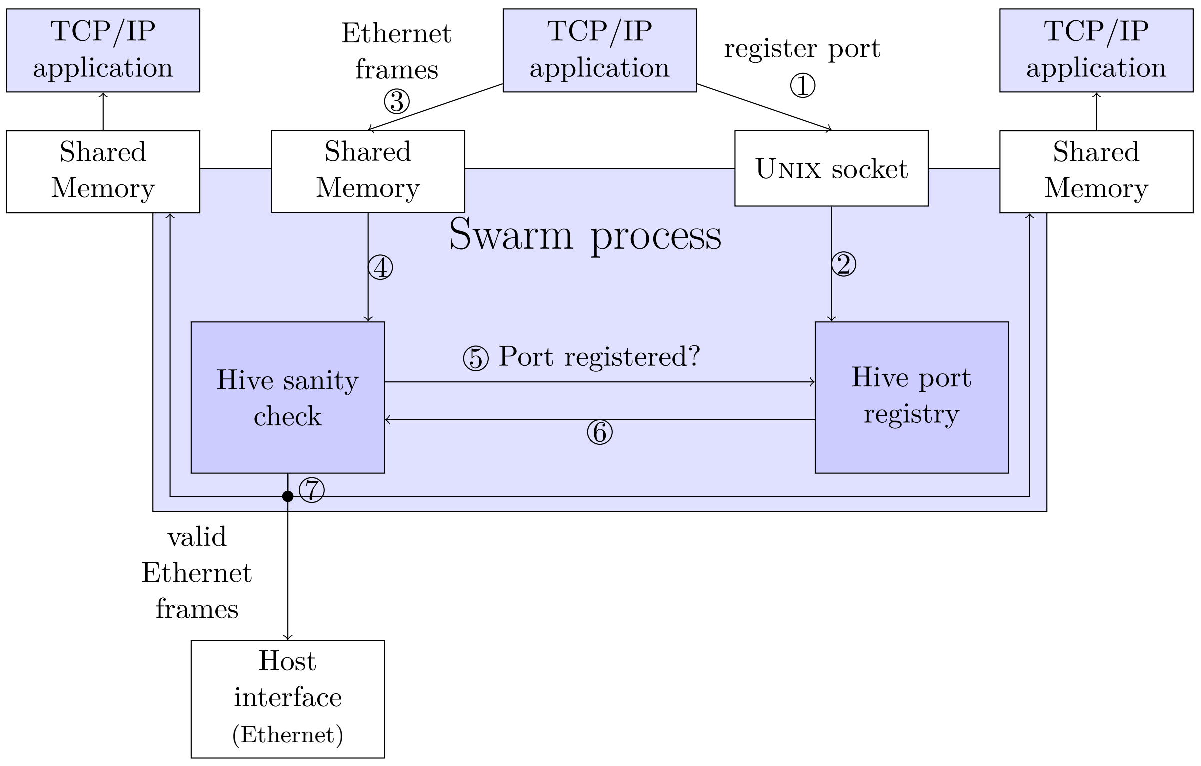


Figure 3.4: An overview of initialisation and frame send in Swarm. The separate steps (numbered) are explained in the text.

3.4.1 Swarm core

The Swarm core performs three main tasks.

Firstly, it binds a Unix socket to a well-known file system node, through which applications can register with Swarm to receive a bus file reference for sending and receiving on the physical network. The bus protocol is described in Section 3.4.2. Swarm core stores an association between an `inotify` event identifier and a bus description record in a hash table. It is possible for programs to allocate more than one bus, but there is no advantage in doing so because Swarm does not have a scheduling policy for packet forwarding yet.

Secondly, the core also administrates the connection to the physical NIC. The current mode of access is the netmap module in the kernel. I have not implemented support for userspace drivers.

Thirdly, Swarm core runs the single-threaded central event loop which asynchronously handles all events in Swarm. I have reasoned on asynchronous event processing in the beginning of this chapter.

3.4.2 Swarm IPC

The inter-process communication unit of Swarm is called Swarm IPC.

Swarm IPC transmits its messages through Swarm's well-known Unix socket. The server-side IPC library uses the asynchronous libevent API [MP], whereas the client library designed for TCP/IP stacks is based on synchronous socket calls. Swarm is single-threaded, and the Unix domain socket on which Swarm IPC operates is connection-oriented, so race-free execution of the remote procedure calls (RPCs) is guaranteed.

For the system to work, an application connecting to Swarm needs to support at least three messages:

- `SWARM_GETSHM`: The networking application sends this message to bootstrap its TCP/IP stack. Swarm allocates a virtual Ethernet bus and returns the file name of the bus file and the current IP address to the calling application.
- `HIVE_BIND(PROTOCOL, PORT)`: Hive attempts to allocate the specified port for the calling application and reports success if it is still available.
- `HIVE_UNBIND(PROTOCOL, PORT)`: Hive deallocates the specified port for the calling application.

As I mentioned above, Swarm destroys a virtual Ethernet bus automatically as soon as the respective file is closed by the application, so there is no need for a fourth IPC message type.

3.4.3 Hive

Hive associates each port with the process that occupies it. As there are currently two supported transport layer protocols, TCP and UDP, Hive requires two hash tables. Table 3.1 holds an exemplary set of port-process associations.

Port	SHM Key
21	none
22	1235
80	9878
⋮	⋮

Table 3.1: An example for a port assignment table in Hive

Based on those associations, Hive forwards every TCP or UDP packet to its registered destination: Packets destined for another than Swarm's IP address are sent to the physical network, whereas packets addressed to the local IP address are distributed according to their destination port number.

ARP, which I introduced in Section 2.1, is handled like this: every outgoing packet is sent to the physical network, every incoming packet is delivered to all buses. If ARP messages are only delivered along those two well-defined paths, the different TCP/IP stacks cannot find out about their coexistence, which avoids false alarms like duplicate IP errors. The stacks do not need to find out about each other because Swarm copies every frame that is supposed to be delivered locally to the network process that listens on the destination port automatically. The source and destination MAC addresses are inherently correct, too, because all stacks were set up to use the same.

My protocol implementation introduces a certain overhead due to duplicate ARP requests: Each stack implementation needs to perform address resolution independently from all others, so for six connected applications, Swarm's physical interface may request the same IP address six times.

You could argue now that Swarm might handle ARP directly, so that only one response will be generated for each request. I have decided against such a solution mainly for the sake of simplicity, but also because if the TCP/IP stacks generate ready-to-deploy Ethernet frames, they will need information on the destination MAC addresses of those frames in any case. Swarm would thus not only have to handle the ARP protocol, but also provide a proxy server for ARP requests and replies from the applications. Handling ARP in Swarm would not help gain much performance for all the overhead, because ARP requests are only sent when the application bootstraps and account for a small percentage of the average network traffic.

Every Ethernet frame that cannot be forwarded according to the above rules is discarded. One of the protocols that Swarm does not handle is the Internet Control Message Protocol (ICMP) [Pos81a]. ICMP is primarily used to check the availability of hosts and to trace routes through a network, and to deliver error messages when a network failure has occurred. ICMP support in Swarm may be useful for less reliable connections. I did not require ICMP in my setup, so I left it out.

3.5 Adapting network applications to Swarm

This section explains means to attach network applications to Swarm, and various issues that I experienced adapting the Rump TCP/IP stack to Swarm.

Before I began work on Swarm, I considered several alternatives for the network library that I was going to adapt. The code was supposed to be thoroughly tested and reliable, and had to support common TCP extensions, such as Selective ACKs [Mat+96] and Explicit Congestion Notification [RFB01]. Also, as this work is not about implementing a TCP/IP stack for userspace from scratch, the target implementation should already run as part of an application on GNU/Linux.

In the beginning, I intended to extract the TCP/IP stack of either FreeBSD or Linux. DDE had already been ported to Linux in an earlier work [WDL11], and I could have used it to run the relevant parts of the kernel source code in userspace. FreeBSD, however, had started a kernel virtualisation project under the codename *image* [Zec03], which also included a separate network stack called vnet. In that case, identifying the relevant code would have been easier, and I would have had more difficulties porting the code to userspace. Neither solution, though, provided a TCP/IP stack ready to be used in an application.

lwip [Dun01], on the other hand, would have provided an application library. It is designed to use as few resource as possible, focussing on a low memory footprint and code size. Unfortunately, due to its minimal design, lwip lacks the advanced TCP features that I requested, and there was no dedicated userspace library available that offered more capabilities than lwip when I started my work.

Fortunately, shortly before I started development, I learned that Rump already supported userspace TCP/IP. It provided all the features of the NetBSD kernel, was well tested and published as part of a dissertation [Kan12b].

3.5.1 Rump TCP/IP

In the following, I provide details on the Rump TCP/IP implementation. I gave a general introduction on Rump kernels in Section 2.2.3.

Using a Rump kernel, the unmodified NetBSD TCP/IP code can be run in userspace. Still, being kernel code, Rump TCP/IP expects to have a device driver at the link layer that sends the IP datagrams to the network. Here, the Rump kernel provides the stack with three alternative interfaces: *virtif*, *sockin*, and *shmif*.

The virtif driver connects to an existing device driver on the host, such as a tap device, netmap, or Intel DPDK [Kan13c], and transmits its frames to the physical network. Virtif is the most straightforward solution to attach a Rump kernel to the network, but creating the backing device on the host requires root privileges.

If the designated Rump application does not require a full TCP/IP stack, the Rump kernel can intercept socket system calls using the sockin interface and forward them to the host. As this solution disables TCP/IP in the Rump kernel, it is not relevant in this work.

My goal was to establish an independent TCP/IP stack implementation in userspace. The virtif interface does not support connections in userspace, so I decided to use the shmif device, which I already described in Section 3.2. Therefore, the shmif interface had to use the system-wide MAC address, so I added another parameter to the corresponding Rump interface call. To set the IP address of the Rump network stack, I use another interface call that was already there when I started work on Swarm. Then, the TCP/IP stack needed to forward bind requests to Hive. I call the `HIVE_BIND` RPC twice in the modified NetBSD kernel code: in the actual `bind` system call, and in the random port selector used by the `connect` system call. `HIVE_UNBIND` is not yet supported in Rump because it was not required for my experiments.

Figure 3.5 delivers the general picture of a packet sent using Rump TCP/IP and Swarm. Suppose the client application passes a chunk of data to Rump. Then, the Rump kernel wraps the data in an Ethernet frame and writes that frame to the common bus file as soon as it has acquired the lock (step 1). Swarm reads the Ethernet frame from the virtual bus file (step 2), checks if it is valid, and sends it to netmap if it is (step 3). Netmap sends the frame out on the physical network, where it is routed to its destination (step 4). The receiving end can run any TCP/IP implementation, in the kernel or in userspace. In the example I depicted a kernel stack in the receiver. The receiving network stack copies the packet from the network, and removes the protocol headers, delivering only the payload data to the server process through the socket interface (step 5).

The data is copied from Rump to the bus file, from the bus file to Swarm, from Swarm to netmap, from netmap to the network, and from the network to the target stack. I can avoid one of those five copy operations using a shared memory interface that connects Swarm directly to the network process, such as the one suggested in Section 5.1. Section 4.2 shows that leaving out that one `memcpy` operation would result in a significant performance gain.

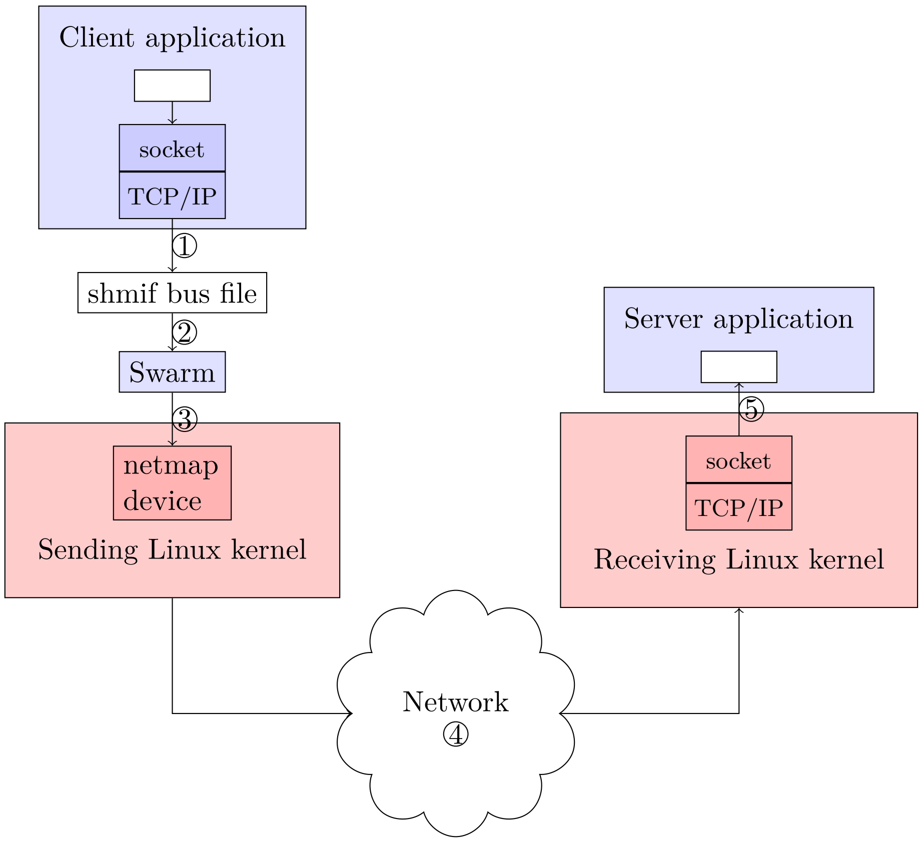


Figure 3.5: The way of a packet when using Swarm and Rump TCP/IP. The separate steps (numbered) are explained in the text.

4 Evaluation

In this chapter, I detail on the performance of the Swarm prototype, but also discuss the implications of my software model, and explain where there are still deficiencies and whether these deficiencies are systemic or not. Possible solutions to those deficiencies are discussed in Chapter 5.

4.1 Nonfunctional evaluation

I stated five major problems in Section 2.3:

1. Decreasing the size of the TCB to a minimum is vital for a secure system. Many commonly used contemporary operating systems retain a monolithic design approach, which makes them less secure.
2. Due to the efforts of the Rump developers, well-tested and feature-rich low-level software is available to higher-level applications as well. It is now important to use those capabilities to achieve a smaller TCB.
3. Userspace TCP/IP stacks are a promising technology, but they introduce a distributed network state that the system needs to handle.
4. Existing techniques for interconnecting independent network applications supply sufficient performance, but tend to drain the MAC and IP address space.
5. The TCP/IP standards define that IP addresses should specify hosts and TCP and UDP ports should specify applications [Pos81b; Pos81c]. Existing interconnects partially support the original addressing scheme, but need to be explicitly configured to adhere to it.

In this section, I am going to check to what degree I have solved them.

4.1.1 Problem 1: Decreasing the TCB

In the case of Swarm, the trusted computing base comprises Swarm, Hive, and Swarm IPC, as well as the backend used to communicate with the physical network and the operating system kernel. The whole Swarm subsystem comprises about 2,000 SLOC, whereas the currently used netmap backend contributes about 12,000 SLOC[1]. The biggest portion of the TCB, though, is the Linux kernel. A modern Linux system for a personal

[1] Source Lines of Code, excluding external libraries, measured using SLOCCount 2.26 [Whe]

computer requires about two million SLOC to be compiled into the kernel[2]. Neither of the last two elements is mandatory to run Swarm, though, because it could be set up to use a userspace device driver directly instead of using netmap in the kernel, and because it does not have to use Linux.

The userspace driver written for sv3 [Ste14b], for example, has a code footprint of 1,000 SLOC [Ste14a]. Also, I wrote Swarm for GNU/Linux, which is a UNIX-like operating system. Hence it should be feasible to port Swarm to a microkernel-based operating system that provides a POSIX-compatible API, like L4/Fiasco [Grob], Minix [Her+06], Escape [Asm], or the commercial QNX [Hil92]. Each of those systems has a kernel about an order of magnitude smaller than the Linux kernel [Fes06].

Swarm adds one component to the TCB that all network processes need to trust, and you could argue that this makes the system less secure. However, if you have more than one network process in a system, you will need to find a way to switch between those processes in any case. Currently, as I already explained in Section 2.5, switching is done in the hardware (Arsenic), in the kernel (VALE), or requires an additional software component just like Swarm (LXIP). Swarm is more secure than an in-kernel solution because it is isolated from all non-networking applications and all independent Swarm instances that serve other network interfaces. Compared to a virtual Ethernet switch such as the one used for LXIP, Swarm can aid configuration because it does not require a separate MAC and IP address pool for each host.

4.1.2 Problem 2: Using Rump kernels

Rump kernels already provide a wide range of functionality, such as PCI and SCSI drivers, file system handlers, the network file system (NFS), and the NetBSD network stack that I am using. They also support many possible hosts, for instance userspace libraries, Xen, or the Linux kernel (see Section 2.2.3). The range of both functionality and supported systems increases steadily as of this writing, but Rump is not widely used yet. Swarm can change that providing an easier way to migrate all of a system's network applications to Rump's TCP/IP implementation.

4.1.3 Problems 3 and 4: Integrating independent TCP/IP stacks

Swarm helps to connect userspace stacks more easily by tying them all to the same MAC and IP address and enabling localhost communication. So far, I have implemented a research prototype for Swarm. A production system that forms a fully operational bundle of independent TCP/IP stacks, however, will need support for more network protocols than TCP, UDP, IPv4, Ethernet and ARP.

[2] It is hard to estimate how much source code is actually used when users compile their kernel. Some of the kernel code is compiled into modules, which do not need to be active. Some functional units comprise a lot of code and are virtually obsolete, thus being never a part of the compiled kernel. To compensate for this, I left out the drivers and firmware entirely, supposing that really only a minimal amount of them is loaded. I also left out optional subsystems like sound and security. In the end, I ran the following command on a Linux 3.14.0 kernel:

```
$ sloccount arch/x86 block fs include init ipc kernel lib mm net
```

As a result, SLOCCount [Whe] reported 2,276,343 lines of code.

I already mentioned ICMP as a means to send diagnosis and control messages through the network. In addition, routing protocols like Open Shortest Path First (OSPF) [Moy98] as well as network management protocols like the Simple Network Management Protocol (SNMP) [HPW02] should also perceive all processes connected to Swarm as a single host, which means that Swarm would need to either include support for those protocols or rely on one or more separate processes to handle them.

Furthermore, Swarm does not support IP version 6 and the Stream Transmission Control Protocol (SCTP) [Ste07]. Both protocols introduce important and necessary improvements to the existing network stack: IPv6 extends the IP address range and includes advanced features like encryption or auto-configuration. SCTP offers message-oriented or stream-oriented transmission, and additionally supports multihoming, that means an SCTP stream can be sent using more than one IP address. A prominent use case are mobile clients which receive a new IP address whenever they change the network cell. With SCTP, existing streams can continue seamlessly during cell handover in such a setup.

An arbitrary TCP/IP stack that wants to communicate with Swarm needs to support the Swarm IPC protocol on the control plane, which is documented in the scope of this work (Section 3.4.2) and sufficiently easy to adopt. The shmif protocol on the data plane, however, is custom-tailored to the Rump kernel module of the same name, and is more complicated than for instance a two-buffer ring-based interface like the one described in Section 5.1. However, it has also been documented here in Section 3.2, and an implementation should still be feasible.

4.1.4 Problem 5: Preserving address spaces

Swarm helps saving MAC and IP addresses by assigning only one address each to all network stacks on the host. Swarm then addresses the stacks via the ports that they have reserved, as I already stated earlier. This approach is also in accordance with the network standards [Pos81b; Pos81c].

4.2 Functional evaluation

To measure the stream and request-response performance of Swarm, I chose the netmap framework, which provides a tool for each common use case: simulating a file transfer (`tcp_stream`), simulating web browsing, which basically consists of connection establishment, short request, short response, and connection reset (`tcp_crr`), as well as analysing latency without connection overhead (`tcp_rr`, `udp_rr`).

I conducted all measurements on two almost identical systems with an Intel Core 2 6700 at 2.66 gigahertz and about 3000 megabytes per second memory bandwidth[3]. Power management and frequency scaling were disabled in the BIOS. I used Debian Wheezy with a vanilla Linux kernel version 3.2.0 and netperf version 2.6.0 on an Intel 82566DM onboard Gigabit Ethernet adapter.

[3] Measured using `Memtest86+` 5.01 [Dem]

Figure 4.1 shows the performance of the stream benchmarks, Figure 4.2 displays the corresponding CPU utilisation of both processor cores. *Remote* marks the computer that ran the `netserver` process, which was adapted to Rump and ran on Swarm in the 'Swarm' case, and *local* is the machine that executed the corresponding `netperf` client. Swarm achieves about half the performance of the conventional Linux stack. We see a less grave performance drop with the request-response benchmarks (Figures 4.3 and 4.4), where performance drops by only about a third.

There are several reasons for the slowdown induced by Swarm. First, Swarm is an additional component in the data flow path compared to the ordinary Linux stack. Transmitting a packet requires two additional context switches: One from the network process two Swarm, and one from Swarm to the kernel.

As for the shmif interface, every new Ethernet frame wakes up all clients so that they can update their state of the file. Packet batching as seen in netmap is not intended. Furthermore, because the bus file has a constant size of 1 megabyte, but no constant slot sizes, frames may wrap around at the end, and so Swarm has to copy each frame before it can do any further processing. According to an exemplary Callgrind[4] measurement that I conducted, administrating the shmif bus introduces 24 per cent overhead, as well as a second `memcpy` operation which is responsible for 12 per cent of Swarm's computations. Concluding from that, a different bus might save up to 36 per cent computation power. 17 per cent of CPU time are required by the data plane (Hive), and another 15 per cent are used up by the `memcpy` that transforms the checked packet to netmap. Replacing that `memcpy` with a pointer operation would either require mapping netmap memory into the application or mapping application memory into netmap. The isolation of the separate applications among each other and from the kernel would be difficult to ensure if they directly shared memory with netmap (see OpenOnload in Section 2.5.3.1), which is why I would prefer to keep this `memcpy` operation for security reasons.

The implications of two `memcpy` operations instead of one are more significant when more data is transmitted, which is why the stream benchmarks and the connect-request-response benchmark perform worse than the simple TCP and UDP request-response benchmarks.

Because performance is Swarm's main weakness, Chapter 5 will primarily treat ways to improve it.

[4] Callgrind [Dev] is a profiling tool which is part of the Valgrind tool suite [NS07]

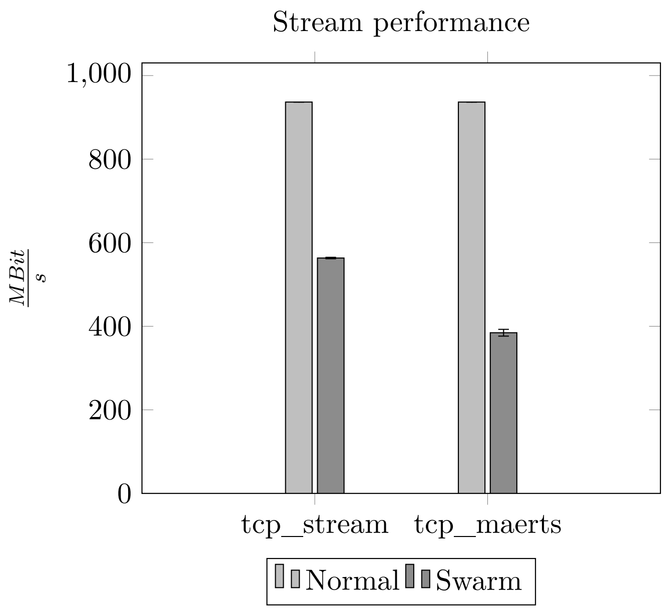


Figure 4.1: The stream performance of Swarm compared to standard Linux

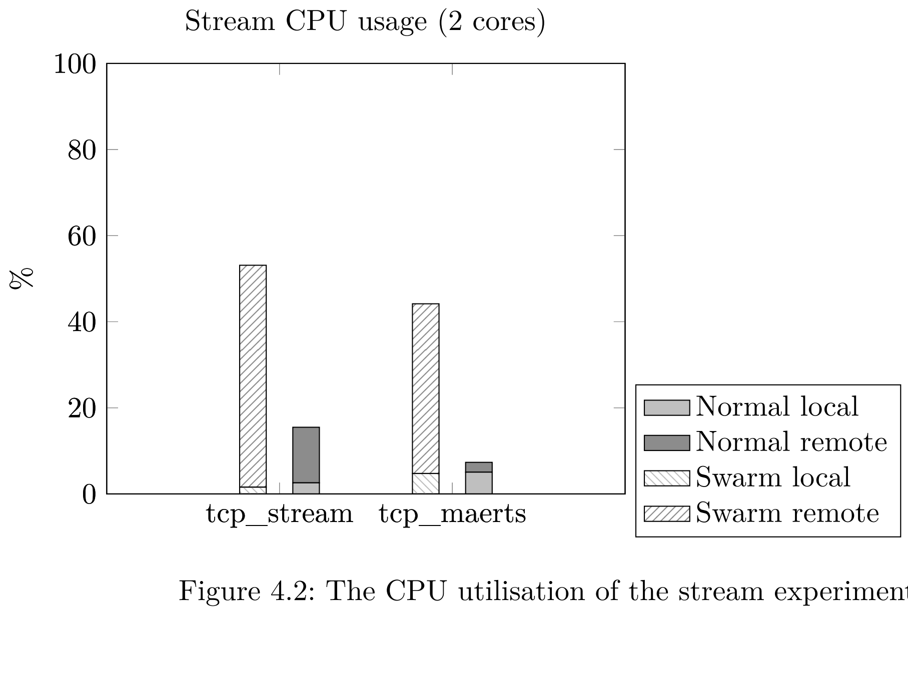


Figure 4.2: The CPU utilisation of the stream experiments

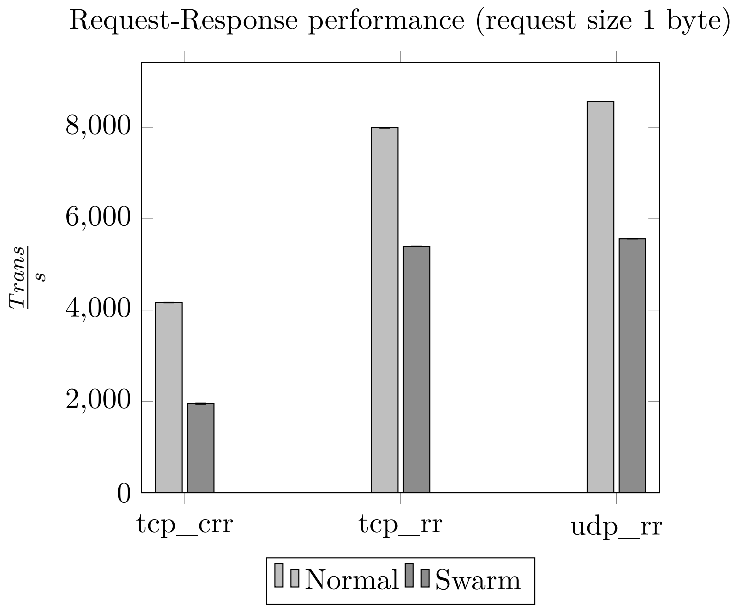


Figure 4.3: The request-response performance of Swarm compared to standard Linux

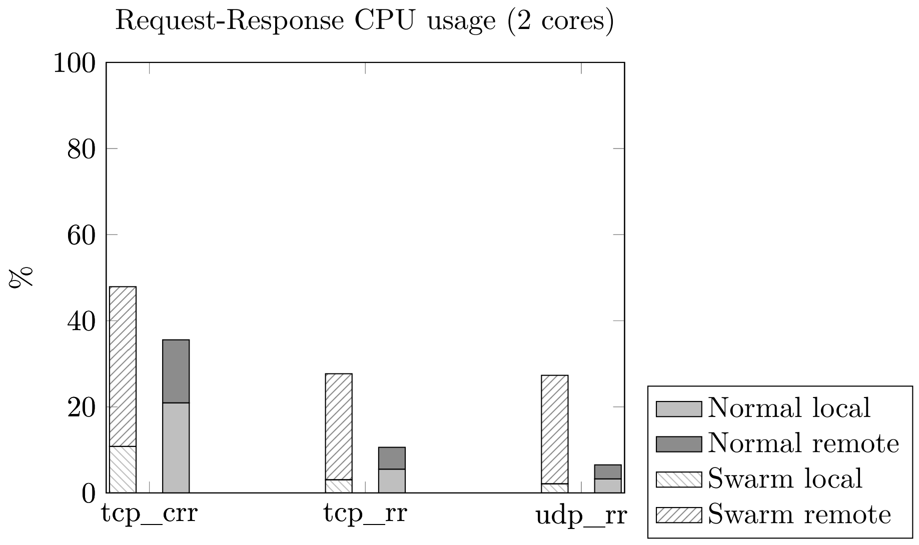


Figure 4.4: The CPU utilisation of the request-response experiments

5 Future work

The performance of Swarm is not yet satisfying. If we look for the causes, we first need to examine the surrounding components.

Rump TCP/IP can easily transmit 10 gigabits per second [Kan]. Netmap is capable of transmitting 15 million packets per second, which marked the hardware maximum of 10 gigabits per second in their measurement [Riz12]. Consequently, Swarm is the bottleneck.

Swarm uses a single-element buffer for packets, and two `memcpy` operations per packet. While this could be reduced to one copy, one gigabit per second is not yet in the same order of magnitude as my memory bandwidth (roughly 24 gigabits per second). The packet validation is rather simple and takes up only 17 per cent of the computing time. So the problem must lie in Swarm's connections to the outside.

The shmif interface used to connect to the Rump kernel, which I described in Section 3.2, is inefficient when used as a point-to-point connection, and is supposedly the main reason why Swarm cannot reach a bandwidth of one gigabit per second.

5.1 An alternative interface

In this section, I propose an alternative design for an application interface for Swarm.

The interface would use one receive ring buffer and one send ring buffer in shared memory. Such an interface would have no file descriptor, and would thus require packet scheduling. I propose a round-robin scheme which iterates over all applications and selects the send and receive ring of each application consecutively. If the selected ring has packets queued, Swarm processes up to a fixed number of packets before moving to the next one. In the event that all rings are empty, Swarm restarts as soon as there is data available in one of the rings. Such an event can be detected using `libevent` or a similar monitoring API. Alternatively, Swarm could retry after a fixed interval of time, in anticipation of more packets in the buffers after the timeout has expired.

The depicted scheduling scheme closely resembles that of sv3 [Ste14b]. I, too, would iterate over memory regions as close as possible and process multiple packets per ring to achieve better cache behaviour. Like sv3, I would need two threads: one that handles the packet queues, and one that runs the main event loop. Likewise, I would resort to userspace RCU to avoid data races that are still impossible now (see 3.4.2).

The size and count of the ring buffer slots should be the same as in the netmap device, so that buffers only need to be validated and copied to their destination, and not rearranged in any way.

Batching packet transmission would not only cause faster cache accesses, but there would also be fewer system calls involved in sending packets than there are now. If all buffers are identical in size and Swarm does not need to reassemble packets from the bus file in its storage memory, data transmission itself will be faster, too.

The current *shmif* data exchange protocol is quite complicated and does not scale as easily, nor is it widely used. Ring buffers however are a common concept in network development, and the new protocol would be easier to handle for network developers who want to adapt their implementation to Swarm.

5.2 TSO/LRO

If we look at hardware acceleration mechanisms for improved network connectivity, the technique most commonly used is a combination of TCP segmentation offload (TSO) and Large Receive Offload (LRO). Here, the operating system retains its packet processing method, but the size of the sent packets (TSO) and the received packets (LRO) are not limited by the Ethernet frame size limits. The network interface card (NIC) receives a big buffer from the operating system and splits it into several suitably sized frames to be passed on to the network. Using the specialised network processor to split the frame instead of the comparatively slow general purpose CPU increases the throughput of the system while retaining its latency characteristics. Most NICs produced in the last ten years support a measure of offloading, so introducing support for that in Swarm would probably increase its performance on most systems. Swarm's relative performance compared to other switching solutions, however, will not increase, because they can make use of the hardware offloading features, too.

5.3 Offloading Swarm

As I mentioned in Section 2.5, NICs have become powerful enough to perform general-purpose computation. Swarm is sufficiently lightweight to be offloaded to the NIC.

Such a modified NIC would be similar to the one developed for Arsenic. In contrast to their work, I am focusing exclusively on userspace networking, so there would be no need to collaborate with the operating system kernel in any way. The driver could reside in the Swarm IPC client library, so that RPCs would be sent directly to the hardware. Arsenic contains an IP registry, which is similar to the tables Hive operates on, and their packet scheduling algorithm is more complicated than what I presented in Section 5.1. Hence porting the algorithm to current hardware more than a decade later should not be problematic.

A hardware solution would eliminate the bottlenecks of context switches between the kernel, Swarm, and the applications, and reduce the significance of the kernel in networking even further, which, in case of Linux, might improve performance by two orders of magnitude [Gra]. But again, it would require custom hardware, which is often infeasible.

6 Conclusion and outlook

For the sakes of isolation and flexibility, it is desirable to move the internet protocol code into userspace. Userspace TCP/IP stack implementations exist, but they are entirely independent, which makes it difficult to use multiple stacks in different processes on the same system. Swarm establishes a transparent interconnect so that arbitrary independent stacks on the same system can collaborate without having to trust each other. Swarm is also sufficiently lightweight so that it can be reviewed and trusted by all other network applications, as it provides nothing more than the required switching functionality. Chapter 3 describes the principles of Swarm, the prototype that I developed, and the Rump TCP/IP stack which I ported to Swarm first.

However, although I was able to achieve my design goals (see Section 4.1), Swarm's weakest point is still its performance. Part of the slowdown may be systemic, such as the context switch overhead, but the sv3 project [Ste14b] demonstrated that userspace network switching is possible at high throughput rates. With the improvements that I outlined in Chapter 5, Swarm should be able to cope with higher line speeds, too.

The trend in TCP/IP computing goes towards userspace implementations. Private authors as well as researchers and companies have recently advocated userspace networking [Cor; Gra; Hon+14], not only because of flexibility, but mainly due to the increase in performance over the kernel implementations that those userspace stacks can bring. Honda et al. [Hon+14], however, already return to the kernel with the VALE network switch that is used in their MultiStack solution. By extending the kernel, they raise the security and maintainability concerns that I mentioned in Sections 2.2.2 and 2.2.3 again. In contrast, Swarm remains an isolated userspace application.

Generally speaking, switching solutions for userspace stacks like Swarm have the advantage to save MAC and IP addresses. With the number of sold network adapters around the world steadily increasing, the IEEE has seen it fit to issue an updated version of the MAC addressing scheme, where MAC addresses comprise 64 bits instead of 48 [IEE]. Addressing problems on the link layer have not been as pressing as those on the internet layer. Despite IPv4 addresses having long run out [Law11], the deployment of IPv6 is still ongoing, and widespread use around the world is still going to take years. Consequently, every measure to save IPv4 addresses is important at least in the medium term.

Appendix A

Source code

The source code of Swarm can be found on GitHub: `https://github.com/Logout22/buildrump.sh`. There is also a manual there for setting up Swarm and trying it out. The modified netserver of the netperf package that I used for the experiments in Chapter 4 can be found at `https://github.com/Logout22/netserver`.

Bibliography

[AK09] Jeff Arnold and M. Frans Kaashoek. 'Ksplice: Automatic Rebootless Kernel Updates'. In: *Proceedings of the 4th ACM European Conference on Computer Systems.* (Nuremberg, Germany). ACM. New York, NY, USA, 2009, pp. 187–198. ISBN: 978-1-60558-482-9. DOI: `10.1145/1519065.1519085`.

[Asm] Nils Asmussen. *Escape.* URL: `https://github.com/Nils-TUD/Escape`.

[Bau06] Chris Baus. *TCP_CORK: More than you ever wanted to know.* 6th Apr. 2006. URL: `http://baus.net/on-tcp_cork/` (visited on 9th Mar. 2014).

[Bra+95] Torsten Braun et al. *An Experimental User Level Implementation of TCP.* Research report 2650. Institut national de recherche en informatique et en automatique (INRIA), Sept. 1995. 20 pp.

[Che08] S. Cheshire. *IPv4 Address Conflict Detection.* Request for Comments (RFC). Internet Engineering Task Force (IETF), July 2008. URL: `http://www.ietf.org/rfc/rfc5227.txt` (visited on 6th Apr. 2014).

[Com] Chelsio Communications, ed. *TCP Offload Engine (TOE).* URL: `http://www.chelsio.com/nic/tcp-offload-engine/` (visited on 8th Jan. 2014).

[Cor] Intel Corp., ed. *Packet Processing on Intel Architecture.* URL: `http://www.intel.com/content/www/us/en/intelligent-systems/intel-technology/packet-processing-is-enhanced-with-software-from-intel-dpdk.html` (visited on 24th Mar. 2014).

[Cor08] Broadcom Corp., ed. *NetXtreme II Family Highly Integrated Media Access Controller — Programmer's Reference Guide.* 8th Oct. 2008. Chap. 2: Hardware Architecture / TCP-Offload Engine.

[Cor13] Intel Corp., ed. *Intel Data Plane Development Kit (Intel DPDK). Programmer's Guide.* Reference Number: 326003-005. Oct. 2013.

[Dem] Samuel Demeulemeester. *`Memtest86+` — An Advanced Memory Diagnostic Tool.* URL: `http://www.memtest.org/` (visited on 28th Apr. 2014).

[Dev] Valgrind Developers. 'Callgrind: a call-graph generating cache and branch prediction profiler'. In: *Valgrind User Manual.* Chap. 6. (Visited on 21st Apr. 2014).

[DH98] S. Deering and R. Hinden. *Internet Protocol, Version 6 (IPv6) Specification.* Request for Comments (RFC). Internet Engineering Task Force (IETF), Dec. 1998. URL: `http://www.ietf.org/rfc/rfc2460.txt` (visited on 20th Feb. 2014).

[Dun01] Adam Dunkels. *Design and Implementation of the lwIP TCP/IP Stack.* Swedish Institute of Computer Science, 20th Feb. 2001.

[EM95] Aled Edwards and Steve Muir. ‘Experiences Implementing a High Performance TCP in User-Space’. In: *Proceedings of the conference on applications, technologies, architectures and protocols for computer communication.* SIGCOMM ’95. (Cambridge, Massachusetts, USA). ACM. New York, NY, USA, Oct. 1995, pp. 196–205. ISBN: 0-89791-711-1. DOI: `10.1145/217382.318122`.

[Fes06] Norman Feske. *A Nitpicker’s Guide to a Minimal-Complexity Secure GUI.* 13th Sept. 2006. URL: `http://demo.tudos.org/nitpicker_tutorial.html` (visited on 24th Mar. 2014).

[FL06] V. Fuller and T. Li. *Classless Inter-Domain Routing (CIDR): The Internet Address Assignment and Aggregation Plan.* Request for Comments (RFC). Internet Engineering Task Force (IETF), Aug. 2006. URL: `http://www.ietf.org/rfc/rfc4632.txt` (visited on 20th Feb. 2014).

[Gmb] Genode Labs GmbH, ed. *Release Notes for the Genode OS Framework 13.11.* URL: `http://genode.org/documentation/release-notes/13.11` (visited on 12th Jan. 2014).

[Gra] Robert Graham. *Custom Stack: it goes to 11.* URL: `http://blog.erratasec.com/2013/02/custom-stack-it-goes-to-11.html` (visited on 24th Mar. 2014).

[Groa] TUDOS Group, ed. *DDE/DDEKit.* URL: `http://wiki.tudos.org/DDE/DDEKit` (visited on 9th Apr. 2014).

[Grob] TUDOS Group, ed. *Fiasco.* URL: `http://os.inf.tu-dresden.de/fiasco/` (visited on 24th Mar. 2014).

[Gro12] Miniwatts Marketing Group, ed. *World Internet Users and Population Stats.* 30th June 2012. URL: `http://www.internetworldstats.com/stats.htm` (visited on 1st Nov. 2013).

[Ham+10] Rehan Hameed et al. ‘Understanding Sources of Inefficiency in General-Purpose Chips’. In: *Proceedings of the 37th Annual International Symposium on Computer Architecture.* (Saint-Malo, France). ACM. New York, NY, USA, 2010, pp. 37–47. ISBN: 978-1-4503-0053-7. DOI: `10.1145/1815961.1815968`.

[Här+05] Hermann Härtig et al. ‘The Nizza Secure-System Architecture’. In: *In IEEE CollaborateCom 2005.* 2005.

[Har02] Daniel Hartmeier. ‘Design and Performance of the OpenBSD Stateful Packet Filter (pf)’. In: *USENIX 2002 Annual Technical Conference, Freenix Track.* USENIX Association. Berkeley, CA, USA, 2002.

[Her+06] Jorrit N. Herder et al. ‘MINIX 3: A Highly Reliable, Self-repairing Operating System’. In: *SIGOPS Operating Systems Review* 40.3 (July 2006), pp. 80–89. ISSN: 0163-5980. DOI: `10.1145/1151374.1151391`.

[Hil92] Dan Hildebrand. ‘An Architectural Overview of QNX’. In: *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures.* USENIX Association. Berkeley, CA, USA, 1992, pp. 113–126. ISBN: 1-880446-42-1.

[Hon+14] Michio Honda et al. 'Rekindling Network Protocol Innovation with User-level Stacks'. In: *SIGCOMM Computer Communication Review* 44.2 (Apr. 2014), pp. 52–58. ISSN: 0146-4833. DOI: `10.1145/2602204.2602212`.

[Hor84] Charles Hornig. *A Standard for the Transmission of IP Datagrams over Ethernet Networks*. Request for Comments (RFC). Internet Engineering Task Force (IETF), Apr. 1984. URL: `http://www.ietf.org/rfc/rfc894.txt` (visited on 20th Feb. 2014).

[HPW02] D. Harrington, R. Presuhn and B. Wijnen. *An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks*. Request for Comments (RFC). Internet Engineering Task Force (IETF), Dec. 2002. URL: `http://www.ietf.org/rfc/rfc3411.txt` (visited on 20th Apr. 2014).

[IAN14] Internet Assigned Numbers Authority (IANA), ed. *Service Name and Transport Protocol Port Number Registry*. 11th Feb. 2014. URL: `http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.txt` (visited on 13th Feb. 2014).

[IEE] IEEE Standards Association (IEEE-SA), ed. *EUI: Guidelines for Use of Organizationally Unique Identifiers (OUI) and Company ID (CID)*. URL: `http://standards.ieee.org/develop/regauth/tut/eui.pdf` (visited on 13th Feb. 2014).

[Inc] Google Inc., ed. *Statistics — IPv6 Adoption*. URL: `http://www.google.com/intl/en/ipv6/statistics.html` (visited on 13th Feb. 2014).

[Ins] Fraunhofer Heinrich Hertz Institute, ed. *High Speed Hardware Architectures — TCP/IP Stack*. URL: `http://www.hhi.fraunhofer.de/fields-of-competence/high-speed-hardware-architectures/applications/tcpip-stack.html` (visited on 17th Nov. 2013).

[Kan] Antti Kantee. *Mailing list communication*. URL: `http://sourceforge.net/p/rumpkernel/mailman/message/32126777/` (visited on 22nd Mar. 2014).

[Kan12a] Antti Kantee. *Kernel Drivers Compiled to Javascript and Run in Browser*. 7th Nov. 2012. URL: `http://blog.netbsd.org/tnf/entry/kernel_drivers_compiled_to_javascript` (visited on 9th Apr. 2014).

[Kan12b] Antti Kantee. 'The Design and Implementation of the Anykernel and Rump Kernels'. doctoral dissertation. Department of Computer Science and Engineering, Aalto University, Nov. 2012.

[Kan13a] Antti Kantee. *A Rump Kernel Hypervisor for the Linux Kernel*. 23rd Apr. 2013. URL: `http://blog.netbsd.org/tnf/entry/a_rump_kernel_hypervisor_for` (visited on 9th Apr. 2014).

[Kan13b] Antti Kantee. *Running applications on the Xen Hypervisor*. 17th Sept. 2013. URL: `http://blog.netbsd.org/tnf/entry/running_applications_on_the_xen` (visited on 9th Apr. 2014).

[Kan13c] Antti Kantee. *Survey of rump kernel network interfaces*. 17th Dec. 2013. URL: `http://blog.netbsd.org/tnf/entry/survey_of_rump_kernel_network` (visited on 9th Apr. 2014).

[Kne07] Jan Kneschke. 'lighttpd — Using lighttpd for faster WebApps'. In: *PHP Unconference 2007*. (Hamburg, Germany). 2007. URL: `http://www.lighttpd.net/download/php-unconf-lighttpd-2007.pdf` (visited on 25th Apr. 2014).

[Law11] Stephen Lawson. 'News Update: ICANN Assigns its Last IPv4 Addresses'. In: *Computerworld* (3rd Feb. 2011). URL: `http://www.computerworld.com/s/article/9207961` (visited on 19th Feb. 2014).

[Lee12] Thorsten Leemhuis. 'Kernel Log: 15,000,000 Lines, 3.0 Promoted to Long-Term Kernel'. In: *The H Open* (12th Jan. 2012). URL: `http://h-online.com/-1408062` (visited on 24th Mar. 2014).

[LeV+04] Joshua LeVasseur et al. 'Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines'. In: *USENIX 2004 Annual Technical Conference*. USENIX Association. Berkeley, CA, USA, 2004.

[Lov05] Robert Love. 'Kernel Korner: Intro to Inotify'. In: *Linux Journal* 139 (Nov. 2005). ISSN: 1075-3583.

[Mat+96] M. Mathis et al. *TCP Selective Acknowledgment Options*. Request for Comments (RFC). Internet Engineering Task Force (IETF), Oct. 1996. URL: `http://www.ietf.org/rfc/rfc2018.txt` (visited on 24th Mar. 2014).

[MB92] Chris Maeda and Brian N. Bershad. 'Networking Performance for Microkernels'. In: *In Proceedings of the Third Workshop on Workstation Operating Systems*. 1992, pp. 154–159.

[MB93] Chris Maeda and Brian N. Bershad. 'Protocol Service Decomposition for High-Performance Networking'. In: *In Proceedings of the 14th ACM Symposium on Operating Systems Principles*. 1993.

[MCZ06] Aravind Menon, Alan L. Cox and Willy Zwaenepoel. 'Optimizing Network Virtualization in Xen'. In: *USENIX 2006 Annual Technical Conference*. USENIX Association. Berkeley, CA, USA, 2006.

[Moy98] J. Moy. *OSPF Version 2*. Request for Comments (RFC). Internet Engineering Task Force (IETF), Apr. 1998. URL: `http://www.ietf.org/rfc/rfc2328.txt` (visited on 20th Apr. 2014).

[MP] Nick Mathewson and Niels Provos. *Libevent — An Event Notification Library*. URL: `http://libevent.org/` (visited on 25th Apr. 2014).

[Nag84] John Nagle. *Congestion Control in IP/TCP Internetworks*. Request for Comments (RFC). Internet Engineering Task Force (IETF), 6th Jan. 1984. URL: `http://tools.ietf.org/rfc/rfc896.txt` (visited on 6th Apr. 2014).

[NS07] Nicholas Nethercote and Julian Seward. 'Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation'. In: *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation.* (San Diego, California, USA). ACM. New York, NY, USA, 2007, pp. 89–100. ISBN: 978-1-59593-633-2. DOI: `10.1145/1250734.1250746`.

[Pal+11] Nicolas Palix et al. 'Faults in Linux: Ten Years Later'. In: *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems.* (Newport Beach, California, USA). ACM. New York, NY, USA, 2011, pp. 305–318. ISBN: 978-1-4503-0266-1. DOI: `10.1145/1950365.1950401`.

[PF01] Ian Pratt and Keir Fraser. 'Arsenic: A User-Accessible Gigabit Ethernet Interface'. In: *Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies INFOCOM 2001.* (Anchorage, AK). IEEE, Apr. 2001, pp. 67–76. ISBN: 0-7803-7016-3. DOI: `10.1109/INFCOM.2001.916688`.

[Plu82] David C. Plummer. *An Ethernet Address Resolution Protocol.* Request for Comments (RFC). Internet Engineering Task Force (IETF), Nov. 1982. URL: `http://tools.ietf.org/rfc/rfc826.txt` (visited on 6th Apr. 2014).

[Pos80] J. Postel. *User Datagram Protocol.* Request for Comments (RFC). Internet Engineering Task Force (IETF), 28th Aug. 1980. URL: `http://www.ietf.org/rfc/rfc768.txt` (visited on 13th Feb. 2014).

[Pos81a] J. Postel. *Internet Control Message Protocol.* Request for Comments (RFC). Internet Engineering Task Force (IETF), Sept. 1981. URL: `http://www.ietf.org/rfc/rfc792.txt` (visited on 20th Apr. 2014).

[Pos81b] J. Postel. *Internet Protocol.* Request for Comments (RFC). Internet Engineering Task Force (IETF), Sept. 1981. URL: `http://www.ietf.org/rfc/rfc791.txt` (visited on 15th Nov. 2013).

[Pos81c] J. Postel. *Transmission Control Protocol.* Request for Comments (RFC). Internet Engineering Task Force (IETF), Sept. 1981. URL: `http://www.ietf.org/rfc/rfc793.txt` (visited on 15th Nov. 2013).

[PR08] Steve Pope and David Riddoch. *The OpenOnload User-Level Network Stack.* 7th Feb. 2008. URL: `http://www.youtube.com/watch?v=1Y8hoznuuuM` (visited on 11th Apr. 2014).

[PR11] Steve Pope and David Riddoch. *Introduction to OpenOnload — Building Application Transparency and Protocol Conformance into Application Acceleration Middleware.* White Paper. Solarflare Communications, Inc., 2011.

[Pro] The Linux *man-pages* Project, ed. *Linux Programmer's Manual — PACKET.* URL: `http://man7.org/linux/man-pages/man7/packet.7.html` (visited on 9th Mar. 2014).

[Rap12] Rapid7, ed. *USN-1650-1: Linux kernel vulnerability.* 21st Dec. 2012. URL: `http://www.rapid7.com/db/vulnerabilities/ubuntu-usn-1650-1` (visited on 21st Apr. 2014).

[Res03] Eric Rescorla. 'Security Holes... Who Cares?' In: *12th USENIX Security Symposium.* USENIX Association. Berkeley, CA, USA, Aug. 2003.

[RFB01] K. Ramakrishnan, S. Floyd and D. Black. *The Addition of Explicit Congestion Notification (ECN) to IP.* Request for Comments (RFC). Internet Engineering Task Force (IETF), Sept. 2001. URL: `http://www.ietf.org/rfc/rfc3168.txt` (visited on 24th Mar. 2014).

[Riz12] Luigi Rizzo. 'Revisiting Network I/O APIs: The Netmap Framework'. In: *ACM Queue* 1 (Jan. 2012), pp. 30–39. ISSN: 1542-7730. DOI: `10.1145/2090147.2103536`.

[RL12] Luigi Rizzo and Giuseppe Lettieri. 'VALE, a Switched Ethernet for Virtual Machines'. In: *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies.* CoNEXT '12. (Nice, France). ACM. New York, NY, USA, 2012, pp. 61–72. ISBN: 978-1-4503-1775-7. DOI: `10.1145/2413176.2413185`. URL: `http://doi.acm.org/10.1145/2413176.2413185`.

[Rus81] J. M. Rushby. 'Design and Verification of Secure Systems'. In: *Proceedings of the 8th ACM Symposium on Operating Systems Principles.* (Pacific Grove, California, USA). SOSP '81. New York, NY, USA: ACM, 1981, pp. 12–21. ISBN: 0-89791-062-1. DOI: `10.1145/800216.806586`. URL: `http://doi.acm.org/10.1145/800216.806586` (visited on 15th Feb. 2014).

[SE01] P. Srisuresh and K. Egevang. *Traditional IP Network Address Translator (Traditional NAT).* Request for Comments (RFC). Internet Engineering Task Force (IETF), Jan. 2001. URL: `http://www.ietf.org/rfc/rfc3022.txt` (visited on 20th Feb. 2014).

[Sha+10] Leah Shalev et al. 'IsoStack — Highly Efficient Network Processing on Dedicated Cores'. In: *USENIX 2010 Annual Technical Conference.* USENIX Association. Berkeley, CA, USA, 2010.

[Ste07] *Stream Control Transmission Protocol.* Request for Comments (RFC). Internet Engineering Task Force (IETF), Sept. 2007. URL: `http://www.ietf.org/rfc/rfc4960.txt` (visited on 20th Apr. 2014).

[Ste14a] Julian Stecklina. Personal communication. 25th Mar. 2014.

[Ste14b] Julian Stecklina. 'Shrinking the Hypervisor One Subsystem at a Time: A Userspace Packet Switch for Virtual Machines'. In: *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments.* (Salt Lake City, Utah, USA). VEE '14. ACM. New York, NY, USA, Mar. 2014, pp. 189–200. ISBN: 978-1-4503-2764-0. DOI: `10.1145/2576195.2576202`.

[Ste94] W. Richard Stevens. *TCP/IP Illustrated — The Protocols.* Vol. 1. 1994. ISBN: 0201633469.

[WDL11] Hannes Weisbach, Björn Döbel and Adam Lackorzynski. ‘Generic User-Level PCI Drivers’. In: *Real-Time Linux Workshop 2011*. Real-Time Linux Workshop. (2011). 2011.

[Whe] David A. Wheeler. *SLOCCount*. URL: `http://www.dwheeler.com/sloccount` (visited on 22nd Mar. 2014).

[Zec03] Marko Zec. ‘Implementing a Clonable Network Stack in the FreeBSD Kernel’. In: *USENIX 2003 Annual Technical Conference, Freenix Track*. USENIX Association. Berkeley, CA, USA, 2003.