Design and Implementation of a Network Server in LibrettOS

Mincheol Sung

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Binoy Ravindran, Chair
Haibo Zeng
Ruslan Nikolaev

November 29, 2018
Blacksburg, Virginia

Keywords: multiserver OS, microkernel, hypervisor, Xen

Design and Implementation of a Network Server in LibrettOS

Mincheol Sung

(ABSTRACT)

Traditional network stacks in monolithic kernels have reliability and security concerns. Any fault in a network stack affects the entire system owing to lack of isolation in the monolithic kernel. Moreover, the large code size of the network stack enlarges the attack surface of the system. A multiserver OS design solves this problem. In contrast to the traditional network stack, a multiserver OS pushes the network stack into the network server as a user process, which performs three enhancements: (i) allows the network server to run in user mode while having its own address space and isolating any fault occurring in the network server; (ii) minimizes the attack surface of the system because the trusted computing base contracts; (iii) enables failure recovery, which is an important feature supported by a multiserver OS. This thesis proposes a network server for LibrettOS, an operating system based on rumprun unikernels and the Xen Hypervisor developed by Virginia Tech. The proposed network server is a service domain providing an L2 frame forwarding service for application domains and based on rumprun such that the existing device drivers of NetBSD can be leveraged with little modification. In this model, the TCP/IP stack runs directly in the address space of applications. This allows retaining the client state even if the network server crashes and makes it possible to recover from a network server failure. We leverage the Xen PCI passthrough to access a NIC (Network Interface Controller) from the network server. Our experimental evaluation demonstrates that the performance of the network server is good and comparable with Linux and NetBSD. We also demonstrate the successful recovery after a failure.

Design and Implementation of a Network Server in LibrettOS

Mincheol Sung

(GENERAL AUDIENCE ABSTRACT)

When it comes to reliability and security in networking systems, concerns have been shown in traditional operating systems (OSes) such as Windows, MacOS, NetBSD, and Linux. Any fault in a networking system can have impacts on the entire system owing to lack of isolation in the OSes. Moreover, the large code size of a networking system enlarges the attack surface of the system. A multiserver OS design solves this problem by running a networking system as a network server, which performs three enhancements: (i) isolates any fault occurring in the network server itself; (ii) minimizes the attack surface of the system; and (iii) enables failure recovery. This thesis proposes a network server for LibrettOS, an operating system developed by Virginia Tech. The proposed network has two-pronged merits: (i) provides a system server providing a network packet forwarding service for applications; (ii) enables the existing device drivers of NetBSD to be leveraged with low amount of modification. Our experimental evaluation demonstrates that the performance of the network server outperforms state-of-the-art and comparable with Linux and that a successful recovery is possible after a failure.

Dedication

This thesis is dedicated to my family.

Acknowledgements

There are many people without whom this thesis would not have been possible, and I would like to thank the following: My advisor, Dr. Binoy Ravindran, for providing me the opportunity to work with him and the amazing people at the Systems Software Research Group, and for his valuable guidance throughout. My committee members for taking the time to review and provide valuable feedback for my thesis. Dr. Ruslan Nikolaev, for sharing his immense knowledge and amazing ideas with me, for all the brainstorming and debugging sessions, and for editing and giving feedback for my thesis. It would not have been possible without him. All the members of the Systems Software Research Group, who were always ready to provide help and advice when needed. My parents and sister, for their love and support.

This research is based upon work supported by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA). The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the ODNI, IARPA, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

This research is also based upon work supported by the Office of Naval Research (ONR) under grants N00014-16-1-2104, N00014-16-1-2711, and N00014-18-1-2022.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Since the conceptual model of TCP/IP was first introduced many decades ago, many commodity operating systems still use a fundamentally similar TCP/IP network stack. However, the traditional network stack has several limitations. In particular, the network stack in commodity monolithic-kernel operating systems (OSes), such as Linux, is fault intolerant, and this raises reliability concerns. As a network stack resides in the kernel space shared by other system components, any fault (e.g., deadlocks, memory access violation, and interrupt handling failure) occurring in the network stack affects the entire system to the point of mandatory rebooting, which could be critical in some cases. For cloud providers, for instance, a fault occurring in a server may cause significant profit loss because server recovery may entail terminating all the guest machines. Moreover, the traditional network stack has a large code base, and this generally leads to a large attack surface. Therefore, the network stack in the monolithic kernel (which is the trusted computing base (TCB) of the system) also raises a security concern.

Multiserver OSes instead run the network stack on dedicated servers, which are isolated from the remaining system. Instead of running all the system components in the same kernel space, a multiserver OS runs its system components, such as system servers, in user mode (as user processes), with their own address space. Only core kernel features (e.g., memory management, scheduling, and inter-process communication) are kept in the kernel. This isolation of the system components addresses the reliability concern because any fault occurring in them can be isolated such that it does not affect the remaining system. Furthermore, the device drivers and system components are separated from the kernel, which minimizes the code size and additionally reduces the attack surface of the kernel. Thus, the security concern is addressed because the overall size of the TCB is significantly reduced.

In this thesis, we propose the design and implementation of a network server in LibrettOS [43], which is a research prototype OS being developed at Virginia Tech.[1] LibrettOS uses rumprun unikernels and the Xen hypervisor. Our network server is a service domain providing a layer 2 (L2) frame forwarding service for application domains. Similar to other network servers in multiserver OSes, our network server supports failure recovery. For failure recovery, the client state should be kept for recovering connections even when the network server crashes. Therefore, we place the TCP/IP stack in the application domains while our network server simply forwards L2 frames to the NIC driver.

Because LibrettOS is already based on rumprun unikernels, we also used it for the network server. Rumprun is a unikernelized version of the rump kernel running in a virtualized environment, such as a Xen hypervisor or kernel-based virtual machine (KVM). Our network server can leverage the existing device drivers from NetBSD (BSD, Berkeley Software Distribution), whereas other unikernels typically require a tremendous effort to port the existing

[1]The proposed network server design is also included in [43].

device drivers owing to their limited device driver support. Our network server can benefit from native features of unikernels; one such benefit is the short booting time, and the other is the small attack surface. The short booting time of rumprun allows a rapid recovery of our network server. In addition, the small TCB size of rumprun provides security benefits, as previously explained.

For the existing multiserver OSes, our network server is potentially more secure than other solutions. Our network server runs in the hardware virtualization (HVM) mode, which naturally protects our system from the Meltdown vulnerability [27] because Meltdown is ineffective in a completely virtualization environment [54]. The Meltdown vulnerability has been recently discovered in many modern superscalar processors such that commodity monolothic kernels (Linux, Windows, and MacOS) and microkernel OSes require extra fixes.

To have a better performance compared to those of monolithic OSes, the network servers should avoid additional layers. An alternative option, the Xen backend/frontend network driver model, introduces an additional indirection between the network domains and consumers (application domains). To support the Intel 10GbE NIC used in our experiments, we had to introduce glue code for the NetBSD's ixgbe drivers.

1.1 Thesis Contributions

As a part of the design and implementation of the network server, this thesis makes the following contributions:

1. This thesis contributes to the design and provides the implementation of a network server (Chapter 4) to demonstrate network isolation and failure recovery. The network

server enables strong isolation of the OS components and failure recovery to achieve security and reliability, which the traditional monolithic OSes lack.

2. This thesis introduces the rump kernel glue code for the 10 GbE ixgbe drivers (Chapter 5) in the rump kernel. This enables support for high-end network hardware.

1.2 Thesis Organization

The remainder of this thesis is organized as follows:

Chapter 2 provides the background information for this thesis. Section 2.1 presents a brief explanation of various OS models. The next section, Section 2.2, introduces anykernel, rump kernel, and rumprun. Next, Sections 2.3 and 2.4 cover hypervisors. Last, Sections 2.5 and 2.6 briefly describe single-root input output virtualization (SR-IOV) and LibrettOS.

Chapter 3 presents the related work, i.e., previous works on microkernel/multiserver OSes and library OS designs. Then it comapares LibrettOS to other OSes.

Chapter 4 addresses the design and implementation of the network server. Sections 4.1 and 4.2 present the architecture and initialization process of the network server. Next, Section 4.3 explains the initialization process of the application domains. Section 4.4 explains the protocol for connecting the network server to the application domains. Next, Sections 4.5 and 4.6 describe the design details of the network server components. Section 4.7 describes the network packet path in the network server. Lastly, Section 4.8 compares the network server to the Xen network backend/frontend drivers.

Chapter 5 describes the rump kernel glue code to support the Intel 10GbE NIC in the

network server. In particular, Section 5.1 describes required effort. Following Section 5.2 describes glue code for the ixgbe NIC. Finally, Section 5.3 introduces ifconfig, which is a system tool that we ported for the network interface configuration.

Chapter 6 presents failure recovery of the network server. Sections 6.1 and 6.2 introduce the protocol for reconnecting the restored network server to the existing application domains. Next, Sections 6.3 and 6.4 pertain to the port binding mechanism for failure recovery.

Chapter 7 shows the experimental evaluations of the network server and other commodity OSes (Linux and NetBSD). We use several micro- and macro-benchmarks such as NetPIPE, Apache benchmark for the Nginx HTTP server, and Redis benchmark. Furthermore, we demonstrate and evaluate failure recovery of the network server based on an experimental scenario.

Finally, **Chapter** 8 concludes the thesis and discusses future work.

Chapter 2

Background

This chapter provides the background information necessary for the remaining thesis. Section 2.1 first describes various OS models such as monolithic OS, microkernels, library OSes, and unikernels. Next, Section 2.2 covers the anykernel concept, rump kernel, NetBSD, and rumprun. Section 2.3 pertains to hypervisors. Furthermore, Section 2.4 discusses Xen, which is a widely used hypervisor. Finally, Section 2.6 introduces LibrettOS [43], an OS that is currently being developed at Virginia Tech. LibrettOS combines the concepts of a library OS, a multiserver OS, and unikernels.

2.1 Operating System Models

This section briefly presents an overview of various OS models.

2.1.1 Monolithic Operating Systems

Widely used OSes, such as Linux, run all their components in the kernel at the highest privilege level. The CPU privilege level divides the virtual address space of each application into two sections; one is the kernel space (highest privilege level) and the other is the user space (lowest privilege level). Applications in the user mode request OS services via system calls. A system call is a software trap from the applications into the OS to perform a privileged operation. When a system call is invoked, a mode switch from the user mode to the kernel mode occurs. In general, monolithic kernels have higher performance than microkernels because all the OS components run in the same address space such that communication between the OS components is not required. In addition, communication between the system components is fast because all the procedures and variables of the kernel are visible throughout the kernel space in which the entire kernel runs [5].

This design, however, raises security and reliability concerns because of its weak isolation. For example, The Meltdown [27] vulnerability is introduced in the monolithic kernel because of this weakness. Another downside of a monolithic kernel is that any faults occurring in it can cause the entire system to crash such that a hard reboot is required.

2.1.2 Microkernels

A microkernel is an OS design that reduces the TCB by running the OS components in the user mode and isolating them in separate address spaces [1, 12, 14, 26]. Microkernels keep only the core functionalities such as scheduling, memory management, and inter-process communication (IPC) within the kernel. Other OS components such as the network stack, storage stack, and device drivers are isolated into separate processes running in the user

mode. The components in the user mode communicate with each other through IPC.

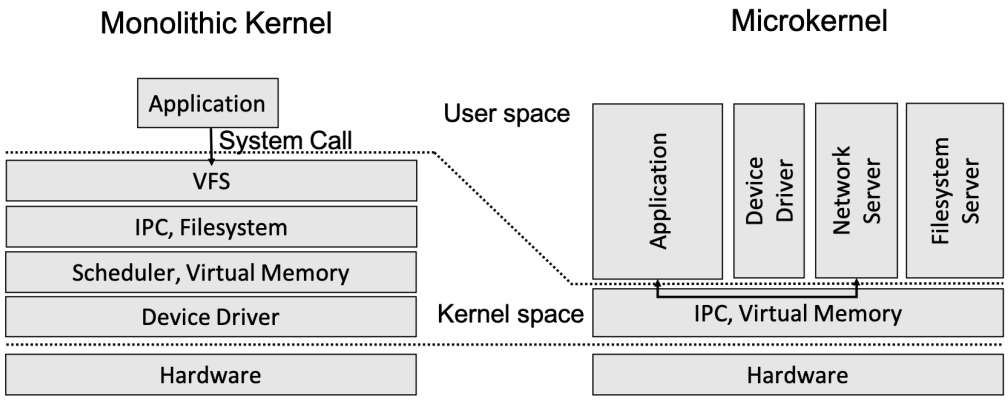

Figure 2.1: Microkernels

Unlike monolithic kernels, stronger security and reliability are provided by microkernels [11, 15, 24, 43]. Kernel components run in separate user address spaces such that the faults occurring in the components are isolated (Figure 2.1). Only failed components need to be restarted, i.e., they do not affect the entire system. Even though microkernels may have a slower performance than monolithic kernels, they are leveraged in many cases where high reliability is required (e.g., real-time, industrial, and military applications) owing to their high modularity and fault tolerance [5, 46]. Furthermore, the small size and modularity of kernel components make formal verification possible, as in the seL4 microkernel [24].

2.1.3 Multiserver and Library Operating Systems

Multiserver OSes are microkernel-based OSes, which are composed of multiple server tasks [4, 17, 45]. They isolate the device drivers and network/storage stacks from each other and the kernel by running them in separate user-space servers (processes) [10, 13, 14, 26, 37, 43].

A library OS was initially proposed in the exokernel [18] model. A library OS design eliminates the system call separation layer between the applications and kernel components,

which is typical in monolithic OSes. Instead, system calls are substituted with regular function calls, and the functionality of the kernel is moved into applications in the form of a library, bringing applications closer to the hardware and allowing better management of the resources as based on their own needs [5]. The concept of a library OS is often promoted for performance reasons and OS flexibility, but subsequent works have exhibited the benefits of library OSes in terms of security [40, 48, 49] because of the strong isolation provided by the reduced interactions between the application and privileged layer [43]. Although microkernels also move the kernel components into the user space, they are shared by all the applications in the system. Therefore, IPC is required for applications to request the OS services. Nonetheless, library OSes face challenges in sharing hardware resources across different applications.

2.1.4 Unikernels

Cloud providers deploy the virtual machines of their customers with fully functioning OSes, such as Linux, even though most customers only run a single application within virtual machines [30]. Such OSes have numerous device drivers and features. Most of them are

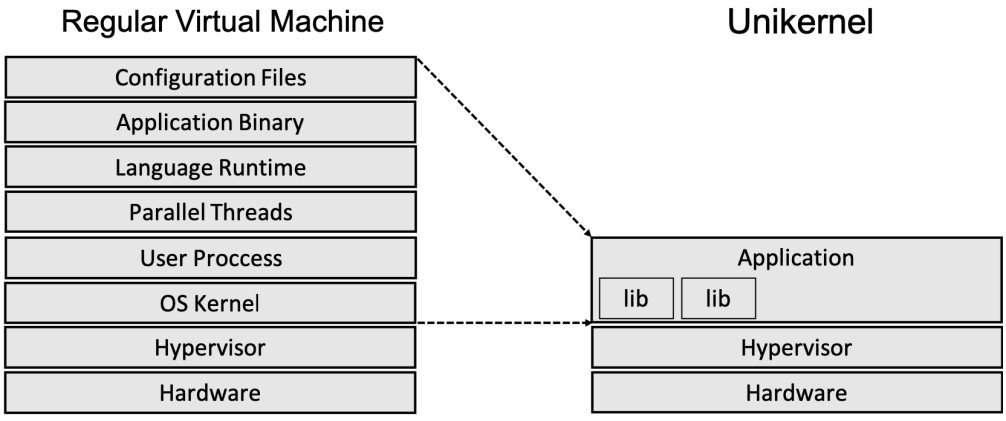

Figure 2.2: Unikernels

not used by the customers but can lead potentially to a large attack surface and poor performance, such as a long booting time [28].

In comparison, unikernels [6, 21, 23, 25, 29, 56], which are specialized library OSes for a single application, are single-address, fixed-purpose, and lightweight virtual machines containing single-application images compiled only with libraries and kernel features required for the application (Figure 2.2), such that they have a significantly smaller image size. Their smaller size reduces the attack surface and expedites the boot process [28]. Unikernels are types of library OSes in which the system calls are replaced by regular function calls.

2.2 anykernel, rump kernel, NetBSD, and rumprun

In the "anykernel" concept introduced by the NetBSD community [19], the driver code base from the monolithic NetBSD kernel can be extracted and integrated into any OS model without code changes [20]. Rump kernels are the first implementations of the anykernel introduced and made prominent by Antti Kantee [19] and the NetBSD community [34]. A rump kernel specifically replaces the system calls to function calls, i.e., converts the NetBSD components to a library. NetBSD is an open source Unix-like monolithic operating system derived from BSD [34]. Unlike other BSD systems, its device drivers and core components are included in the anykernel components.

As described in Figure 2.3 (the figure is reproduced here from [43] under fair use and shown for completeness), a special rump kernel glue layer enables the execution of anykernels outside of the monolithic NetBSD kernel.

Rumprun is a unikernelized version of a rump kernel running in a virtualized environment,

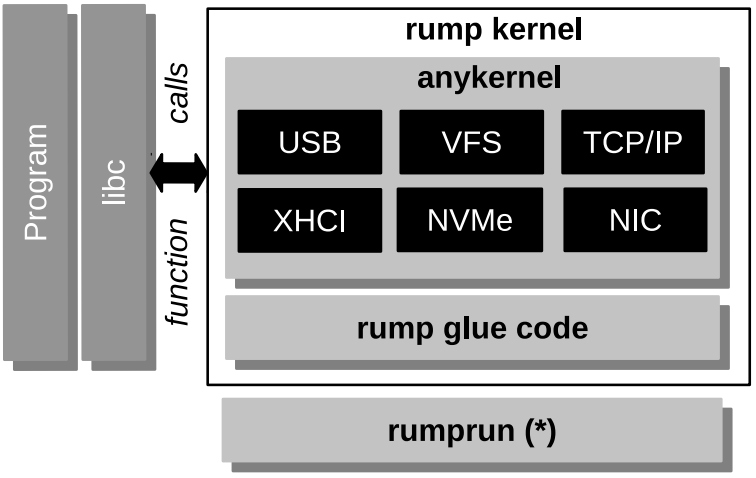

Figure 2.3: Rump software stack

such as the Xen [2] hypervisor, kernel-based virtual machine (KVM) [22], or a bare metal.

The network server is based on rumprun because it can leverage the large collection of the device drivers of NetBSD, including the 10 GbE ixgbe NIC driver.

2.3 Hypervisors

A hypervisor or virtual machine monitor (VMM) is a software enabling other OSes to run on virtual machines on the same host machine. The OSes running on the virtual machines are called guest OSes, and they are isolated by and unaware of the virtual machines because the hypervisor virtualizes the host machine resources such as processors and memory. Similar to microkernels, hypervisors can be used for driver isolation in virtual machines [8].

There are two types of hypervisors: type-1 and type-2. The type-1 hypervisor is called a bare metal hypervisor, which runs directly on the hardware of the host to control the hardware and manage the guest OSes. The most well-known type-1 hypervisor is Xen. The type-2

hypervisor, in contrast, runs on a conventional OS. In the type-2 hypervisor, a guest OS runs as a process on the host operating system. For example, KVM and VirtualBox are type-2 hypervisors [31].

2.4 Xen

Xen is a well-known type-1 hypervisor first introduced and developed by the Computer Laboratory at University of Cambridge. Their team introduced paravirtualization, which requires modification of the guest kernel but achieves performance improvement [2]. Guest virtual machines are called "domains" in Xen. When Xen boots, Dom0 is started first. Dom0 is a privileged domain that manages other domains (DomU). In addition, Dom0 runs the Xen management tool stacks and device drivers to control the hardware resources. As Dom0 contains essential components for Xen, Xen is not usable without Dom0 [51].

Hardware-assisted virtualization or the HVM mode is one of guest types supported by Xen. HVM requires virtualization extensions (e.g., Intel VT or AMD-V) from the host CPU [50], and hardware devices such as BIOS, IDE disk controllers, VGA graphic adapters, USB controllers, and network adapters are emulated by QEMU. In particular, the HVM is not affected by the Meltdown vulnerability [54] because virtual machines are isolated by full virtualization.

2.4.1 Xen Hypercalls and Events

There are two mechanisms for control interactions between Xen and the domains. The first one is through hypercalls, which are synchronous calls from the domains to Xen. The second one is through event channels, which are asynchronous notifications delivered from

the domains to Xen [2]. A hypercall is a software trap from the domains into the hypervisor to perform a privileged operation, similar to the system calls in the conventional OSes. One example when domains use hypercalls is when they update their page tables. An event channel is a signaling mechanism that is used in a path from the Xen hypervisor to the domains [53]. Events can also be sent from one domain to another through virtual interrupts (VIRQs).

2.4.2 Xen Grant table

A grant table provides a generic mechanism for memory sharing between domains. Each domain has its own grant table that is shared with Xen. Domains can notify Xen what permissions other domains have on their pages through the grant table [52]. Grant references are integers and used by the domains to map the pages of the granting domains. Shared pages via grant tables are used by the backend/frontend drivers of Xen for block and network I/O.

2.4.3 Xen Split Driver Model

Domains perform block and network I/O based on the split driver model, which is comprised of a backend driver in Dom0 (a privileged domain that has access to the hardware) and frontend driver in DomU (other domains except Dom0). The backend and frontend drivers communicate through the Xen I/O ring buffers in the inter-domain shared pages. A sending end places the data into the ring buffer and sends a VIRQ through the Xen event channel to the receiving end. Dom0 has actual device drivers and communicates with hardware. Therefore, the outgoing packets of DomU are sent from the network frontend driver to the backend driver through the ring buffer, and the Dom0 forwards them to the actual NIC

driver. Because Dom0 is normally running Linux, guest OSes can benefit from the large collection of device drivers of Linux. An existing rumprun only provides a simple frontend driver for the network I/O, i.e., no backend driver is available for rumprun.

2.4.4 Bridging and Network Address Translation (NAT)

Bridging is a common configuration for Xen networking. A guest has its own IP address used by the outside network and is connected to a network by using the network adapter on Dom0. In this configuration, Dom0 acts as a switch and forwards Ethernet frames based on the MAC addresses.

In the network address translation (NAT) configuration, however, a guest does not have an IP address on the external network; instead, a separate private network is set up and each guest is connected to the private network with its own inteernal IP address. Therefore, Dom0 performs address translation for the incoming/outgoing packets of the private network.

2.4.5 Xen PCI Passthrough and Input-Output Memory Management Unit (IOMMU)

The Xen PCI passthrough allows domains to access directly the physical devices with full control [55]. This has several benefits for performance and security. First, DomUs can bypass Dom0 and access the NIC hardware directly. Second, the input-output memory management unit (IOMMU) hardware support makes the PCI passthrough safe by remapping the interrupts and I/O addresses used by the devices. The IOMMU isolates the host machine memory such that the devices can only access the memory to which they are assigned (Figure 2.4).

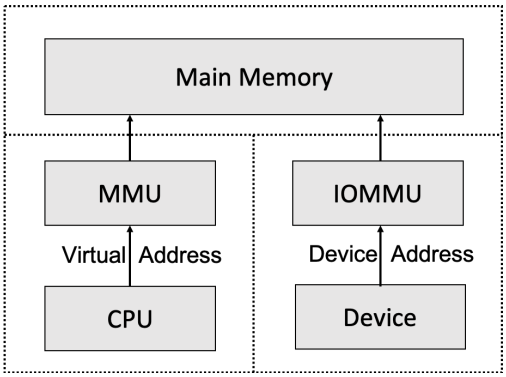

Figure 2.4: Input-Output Memory Management Unit

2.5 Single-Root Input-Output Virtualization (SR-IOV)

SR-IOV is a hardware support for virtualizing a single physical device into several virtual device instances shared across isolated entities such as virtual machines or processes. SR-IOV introduces the concepts of physical functions (PFs) and virtual functions (VFs). PFs are fully featured PCI functions and used by an OS to create and delete VFs; VFs are lightweight functions that lack configuration resources. Without having multiple real devices, VFs of a single device can be directly accessed by virtual machines. SR-IOV is widely used for high-end network adapters (NIC), and VFs represent logically separate devices with their own MAC/IP addresses and transmit/receive (TX/RX) buffers. Each PF and VF is assigned a unique PCI express requester ID (RID). This allows the IOMMU to differentiate between different traffic streams and apply memory and interrupt translations between PFs and VFs [47]. When using the PCI passthrough, a domain can directly access VFs such that the traffic streams are delivered directly to the appropriate domain.

2.6 LibrettOS

LibrettOS is a new OS developed by Virginia Tech that fuses multiserver and library OS designs for better isolation, failure recovery ability, and performance while still using the existing NetBSD drivers and software [43]. As LibrettOS is built on rumprun, the large col-

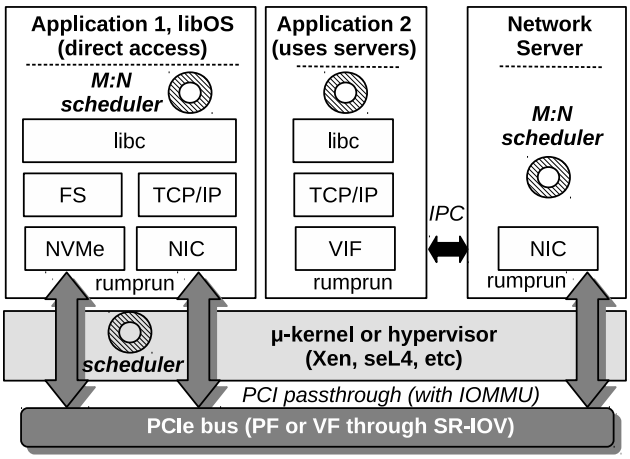

Figure 2.5: LibrettOS design

lection of drivers of NetBSD can be also leveraged in LibrettOS. As we see in Section 2.1.3, the critical limitation of a library OS of an insufficient driver support, can be solved. LibrettOS can work in two modes; direct access mode and multiserver mode. The work presented in this thesis specifically implements a network server for the multiserver OS mode. Figure 2.5 shows the design of LibrettOS, and the figure is reproduced here from [43] under fair use and is shown for completeness.

2.6.1 MPMC Ring Buffer

A ring buffer is a bounded first-in first-out (FIFO) queue that uses a single, fixed-size buffer. For better scalability, the network server uses a lock-free multi-producer multi-consumer

(MPMC) ring buffer from LibrettOS [43], which was already implemented there for its scheduler.

Chapter 3

Related Work

The work presented in this thesis is concerned with three OS research topics: 1) isolation of the system software components for security and tolerance; 2) bypassing the OS layers for applications to access directly devices for performance; and 3) application-specialized OS components for security and attack-surface reduction.

This chapter introduces related work. The multiserver OS design that uses a microkernel (Section 3.1) attempts to move the core OS functionality into separate user processes. Library OSes and unikernels bring applications closer to devices for performance (Section 3.2).

3.1 Microkernel and Multiserver OS design

Mach [1] and L4 [26] use microkernel-based designs in which the core functionality, such as task scheduling and message-based IPC, is implemented in the kernel, whereas most of the

other components are isolated as user processes. seL4 [24] is a formally verified OS that uses a microkernel-based design. To support POSIX, typically a multiserver design is used where each server performs some specific task. SawMill is a multiserver OS design that attempts to decompose the functionalities of a monolithic kernel, such as network and file systems, into separate servers running on top of a microkernel [10]. Minix 3 is a highly reliable, UNIX-compatible multiserver OS with performance degradation of 5% to 10%. It runs most core components as user-mode processes and compartmentalizes device drivers such that the size of the TCB is significantly reduced [14]. VirtuOS is a fault-tolerant OS design that provides isolation of the kernel components by running them in virtualized service domains on top of the Xen hypervisor. Service domains such as networking or storage domains are created from existing Linux kernels [38]. QNX is a unix-like real-time microkernel initially used for embedded systems. It is one of the first commercially successful microkernels. However, drivers need to be ported to QNX.

3.2 Library OS and Unikernels

IX [3] and Arrakis [39] adopt a library OS design. These OS designs present techniques to bypass the OS kernel for the I/O operations and directly access the I/O device while also retaining isolation capabilities.

IX is a data plane OS that separates the management and scheduling functions of the kernel (control plane) from the network processing (data plane) with hardware virtualization support [3]. Although IX leverages the DPDK [16] library to access NICs, it implements its own custom API instead of POSIX API. Arrakis attempts to remove the kernel from the I/O data path without undermining the process isolation. Arrakis leverages hardware features

(e.g., SR-IOV, IOMMU) to deliver the I/O directly to a customized user-level library. As opposed to the traditional operating system (e.g., Linux), it mediates all the I/O operations to enforce the process isolation and resource limits [39]. Arrakis supports POSIX on top of the Barrelfish OS [7], which has a limited device driver support.

Unikernels [6, 21, 23, 25, 29, 56] are cloud-specialized library OSes and have emerged in recent years. LibrettOS has two modes; one is the direct access mode (library OS) and the other is the multiserver OS mode. LibrettOS as a library OS can fully leverage the existing device drivers and applications of NetBSD without significant modification. LibrettOS runs all the components (e.g., the network server and application domains) as unikernels. The work presented in this thesis implements the network server of LibrettOS.

Table 3.1: Comparison with other OSes

Feature	LibrettOS	MINIX 3	QNX	Linux	NetBSD
Paradigms	LibOS& Multiserver	Multiserver	Multiserver	Monolithic	Monolithic
API	POSIX/ BSD	POSIX	POSIX	POSIX/ Linux	POSIX/ BSD
Driver Base	Large	Limited	Limited	Large	Large
Failure Recovery	Yes	Yes	Yes	No	No
TCP/IP	Application	TCP/IP Server	TCP/IP Server	Kernel	Kernel

3.3 Summary

Table 3.1 summarizes the OS design features and compares the network server of LibrettOS to those of the existing multiserver OSes introduced in this chapter. Minix 3 and QNX also

support failure recovery like LibrettOS. The network server in LibrettOS places TCP/IP in the applications such that the client state can be retained even if the network server crashes. However, Minix 3 and QNX put TCP/IP stack in a TCP/IP server.

Chapter 4

Network Server

This chapter addresses the design and implementation of our network server in LibrettOS. First, Section 4.1 briefly describes the network server architecture. Sections 4.2 and 4.3 describe the initialization processes of the network server and application domains. Next, Section 4.4 presents the connection routine of the network server to an application domain. Following this, Section 4.5 explores the hypercalls used by our prototype. Furthermore, Section 4.6 discusses the design and implementation of the IPC between the network server and application domains. Section 4.7 covers the packet forwarding. Last, Section 4.8 compares the network server to Xen's split driver model.

Using a network server in LibrettOS enables safe and secure device sharing while using the existing drivers and software. Furthermore, the network server supports failure recovery, which monolithic OSes lack. The failure recovery allows the network server to contain any faults occurring including memory access violations, deadlocks, or interrupt handling routine failures.

4.1 Network Server Architecture

To retain network connections while performing failure recovery, we place the TCP/IP stack directly in the application address space [43]. Applications talk to the network server through the IPC. The IPC channel of the network server comprises of TX/RX ring buffers and a virtual interrupt (VIRQ). The TX/RX ring buffers relay L2 network frames. We use mbuf as the transfer unit in the IPC. The mbuf is a data structure used as a basic unit of the memory management in the kernel IPC subsystem [9]. A network frame may span multiple mbufs arranged into one mbuf chain (linked list), which allows adding or trimming network headers with little overhead. The sending end places a frame in one mbuf chain and sends it through the ring buffers. The receiving end restores the mbuf chain and processes it. An application uses a virtual network interface (VIF) for sending and receiving frames. The network server has a handler that forwards mbufs to the network subsystem (and eventually NIC driver) per each application. Figure 4.1 shows the architecture of the network server, and the figure is reproduced here from [43] under fair use and shown for completeness.

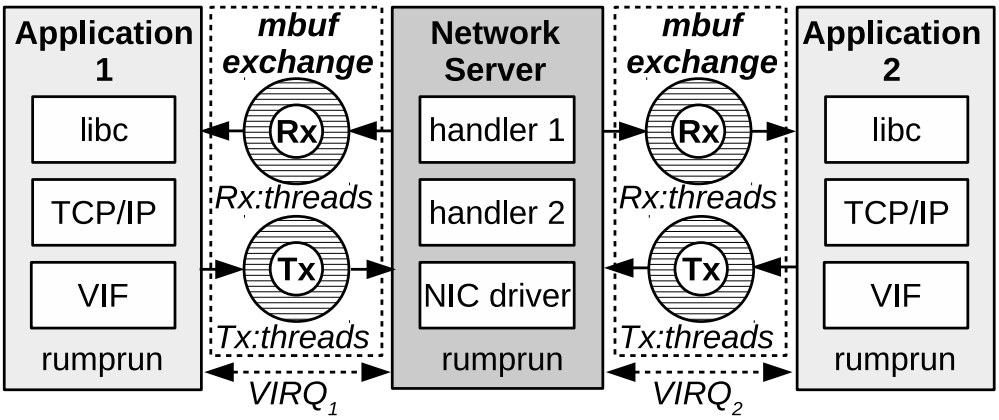

Figure 4.1: Network Server

A TCP state preserved inside an application allows the application to recover from any network server failures (Chapter 6). Although the network server operates at the L2 net-

work stack, knowledge of the higher-level protocols is required for routing across different applications.

We perform routing based on TCP/UDP ports per applications and TCP/UDP ports are stored in a portmap table. Because the network server loses all the information when it crashes, we place the port map table in Xen. (Section 6.4). When an application requests socket binding, a requested port is passed through a distinct port-bind hypercall. If a port is available and successfully allocated, the hypercall returns the port number. Port binding does not occur frequently, and therefore, hypercalls do not incur substantial cost. A port map table is a read-only mapped to the network server for faster access when routing inbound traffic.

When a frame arrives through NIC, the network server extracts the port number and protocol type from the frame. The network server directly reads the port map information from the port map table and determines the target application to which it forwards the frame. The network server can check if the other side is active or not by reading an atomic counter variable. If an application domain is inactive (the counter is 0), the network server wakes it up by sending a VIRQ. The counter prevents unnecessary interrupts to the active application domains. A similar counter also exists on the application domain side for the opposite traffic flow.

4.2 Network Server Initialization

When the network server is launched, it first starts initialization. In its initialization stage, the network server determines whether any application domain exists or not by a hypercall. If application domains are running, the reconnection routine should start (Section 6.1). If

not, the network server starts the regular routine.

The network server registers its information, which application domains can use. The information for the connections, such as the domain id (domid) of the network server and grant table references for the ring buffers, is stored in Xen by a particular hypercall. Then the network server allocates a welcome VIRQ, which is used when the application domains send an initial notification to the network server.

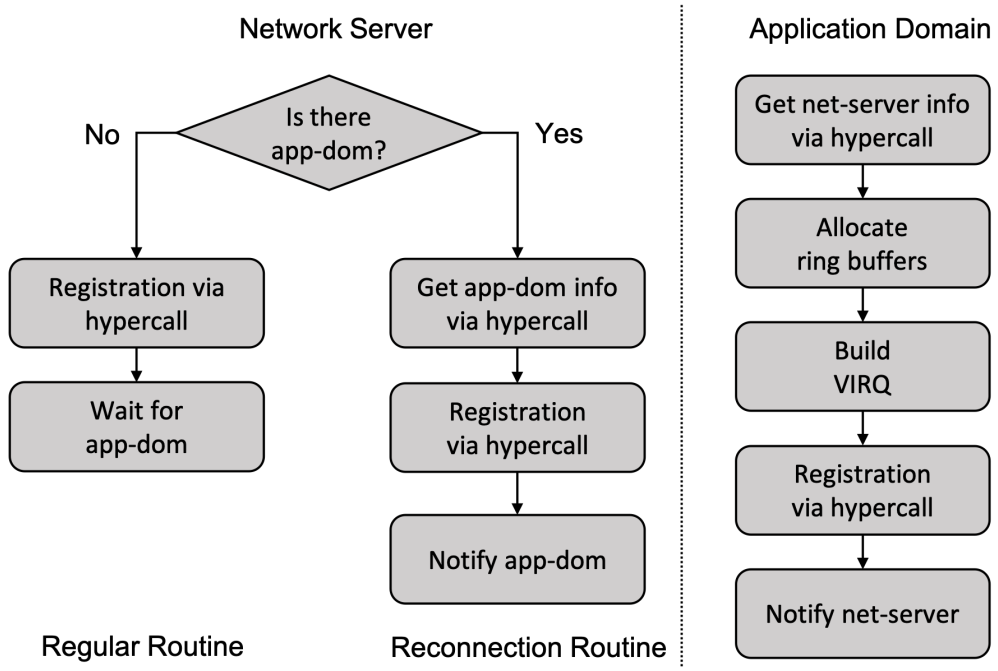

Figure 4.2: Initialization process of the network server and the application domain

The notification invokes the network server to connect to the application domains (Section 4.4). Because the network server starts before the application domains, it does not know the application domains domids. As Xen event channel requires a remote domid for binding a VIRQ port, we create a unique magic value for the domid. If a VIRQ is allocated with a remote domid parameter of DOMID_BACKEND, any domain with a random domid can bind to the VIRQ. We modify the Xen event channel mechanism to support this be-

havior. A connection handler is bound to the welcome VIRQ, which invokes the connection routine when the VIRQ comes from the application domain. In case the application domains are already running when the network server is launched, the reconnection routine should start (Section 6.1). Figure 4.2 shows the network server initialization process.

4.3 Application Domain Initialization

An application domain initializes its ring buffers and event channel VIRQs for frame forwarding prior to connecting to the network domain. First, the network server registration information is obtained by a hypercall. With this information, an application domain grants some continuous range of pages to the network server. An application domain passes only the grant table references of the two pages that contain grant references to other pages. The network server maps these two pages and other pages with the grant references obtained from these first two pages. After granting pages, the application domain initializes ring buffers. Then, the receiver thread and reconnector thread are created. The receiver thread is woken up by the frame forwarding VIRQ. It dequeues the mbufs from the TX/RX ring buffers.

The reconnector thread is in charge of reconnecting the network server during failure recovery. A new network server instance sends a VIRQ to the application domains to wake up the reconnector thread up (Section 6.2.1). The reconnector thread will perform a normal initialization and also destroy the old ring buffers because new ring buffers have to be allocated for the new network server.

Next, the application domain will allocate the reconnection VIRQ. The existing port bound to the welcome port of the network server is not available, and therefore, a new port should be allocated. Similarly for allocating a welcome VIRQ in the network server, the DO-

MID_BACKEND is passed as a parameter in the allocation function because the new network server may have a random domid.

After all the data structures and event channel ports are initialized, the application domain registers its information via a hypercall. Then the application domain sends a VIRQ to the network server to invoke the network server connection routine. Figure 4.2 shows the application domain initialization process.

4.4 Network Server Connection Routine

The network server connection routine is invoked by a VIRQ through the welcome port (Figure 4.3). The network server connection routine starts after the application domains are

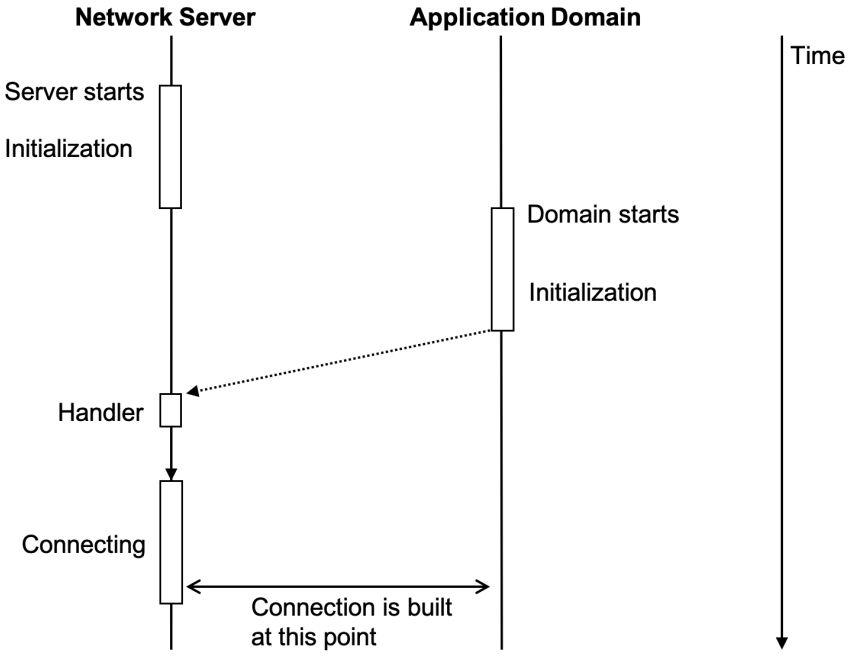

Figure 4.3: Network Server Connection Routine

initialized. The information of the application domains is obtained by a hypercall. Next, it

maps two pages from the application domains with grant table references. Through these shared pages, the network server obtains the grant table references for the TX/RX ring buffers (Section 4.6.1). The network server builds the event channel for the main VIRQ used in frame forwarding.

Then, the network server receiver thread is created, which is similar to that of the application domain. To connect multiple application domains, the network server allocates a new welcome port for the new application domains. If it is in the reconnection routine (Section 6.1), the network server does not allocate a new welcome port.

4.5 Hypercalls

We use hypercalls for the initial stage of connecting the network server and application domains. The information for building this connection is stored in Xen.

Table 4.1: Hypercalls for connecting the network server to application domains

Hypercall	Network Server	Application Domain
REGISTER	store net-server information	
REGISTER APP		store app-dom information
QUERY		query net-server information
FETCH/RECONNECT	fetch app-dom information	
PORT BIND		register requested port
CLEANUP	cleanup all information	

In our prototype, we have implemented six hypercalls (Table 4.1). The hypercalls, except PORT_BIND, are protected by a spin lock because they are not called on the performance critical path. Because port binding is simple and only changes the port map table, we implement it using compare-and-swap (CAS).

4.6 IPC

4.6.1 TX/RX Ring Buffer

Ring buffers contain fixed-size (9162-byte) elements to support jumbo frames. The sending end copies the frames into the TX ring buffer, and the receiving end copies them from the ring buffer and inserts them in its network stack.

4.6.2 Virtual Interrupt

VIRQs are an essential part of the IPC. We implement VIRQs using the Xen event channel subsystem. If the receiving side is inactive when the sending side is relaying mbufs, it must be notified by a VIRQ. As a stream of frames is generally transmitted without delays, the number of the VIRQ interrupts is typically small. The sending end knows that the receiving end is active by reading an atomic thread counter variable in a read-only shared page [43]. Therefore, reducing the interrupts by reading the atomic thread counter variable can achieve performance improvement. When scheduling VIRQ worker threads, race conditions on the receiving end can also be prevented by the same variable.

4.6.3 Welcome port

The welcome port is an event channel port used for the VIRQs from an application domain to the network server when the very first connection is built. After the connection to an application domain is built, the network server destroys and creates a new welcome port for other application domains because once the port is bound, it cannot be reused. The

newly allocated welcome port is stored in Xen through a hypercall, and a new application domain that wants to connect to the network server can also retrieve it via this hypercall. The welcome port is bound to the application domain hello port.

4.6.4 Main port

The main port is used for a VIRQ in frame forwarding when working with ring buffers. As explained in Section 4.6, the sending end sends the VIRQ through the main port if the receiving end is inactive. An event channel of the main port is built in the connection routine of the network server (Section 4.4) and kept valid until the application domain is terminated or the network server reboots because of faults. In case of failure recovery (Chapter 6), a new event channel of the main port should be built.

4.6.5 Reconnection port

The reconnection port is an event channel port created by application domains. It is used as a VIRQ for reconnecting the network server and application domains for the failure recovery of the network server (Chapter 6). In the failure recovery, the application domains stay active and they should be notified by the network server regarding the network server reboots. In the initialization of the network server (Section 4.2), the network server can obtain knowledge about the application domains running through a hypercall. Then, the network server binds the reconnection ports of each application domain and sends a notification (Figure 4.4).

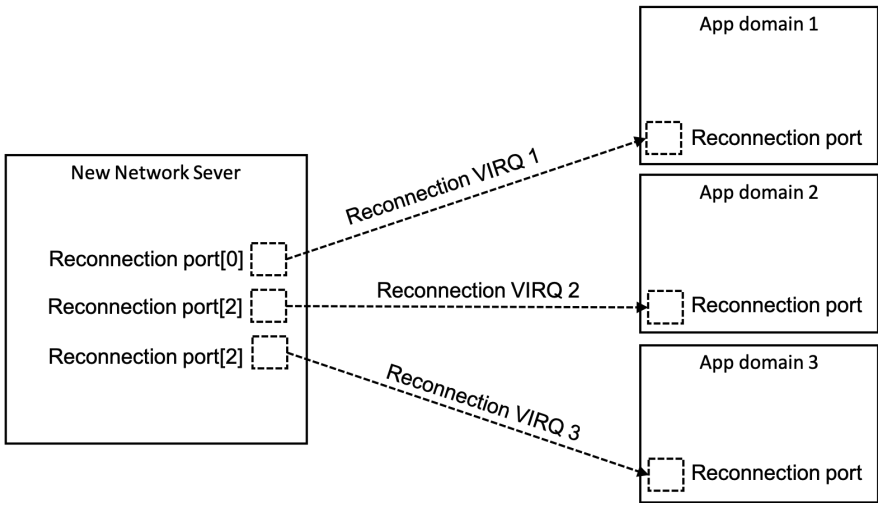

Figure 4.4: Reconnection Port

4.6.6 Receiver Thread

We implement the receiver thread on both the network server and applications. The receiver thread is a thread in charge of receiving the frames from the sending end, and after its creation it sleeps until the corresponding VIRQ arrives. The main VIRQ handler for frame forwarding wakes up the receiver thread. The receiver thread sets the thread counter variable such that it helps the sending ends to know the receiving end is active (Figure 4.5). Therefore, the sending ends does not send the VIRQ and allows avoid handling the VIRQ. This improves the performance because the receiver thread rapidly batches pending frames in the ring buffer without being interrupted.

After the receiver thread consumes all the pending frames in the ring buffer, it resets the thread counter variable so that the sending end can have knowledge about the status of the receiving end. The receiver thread sleeps again until the VIRQ arrives.

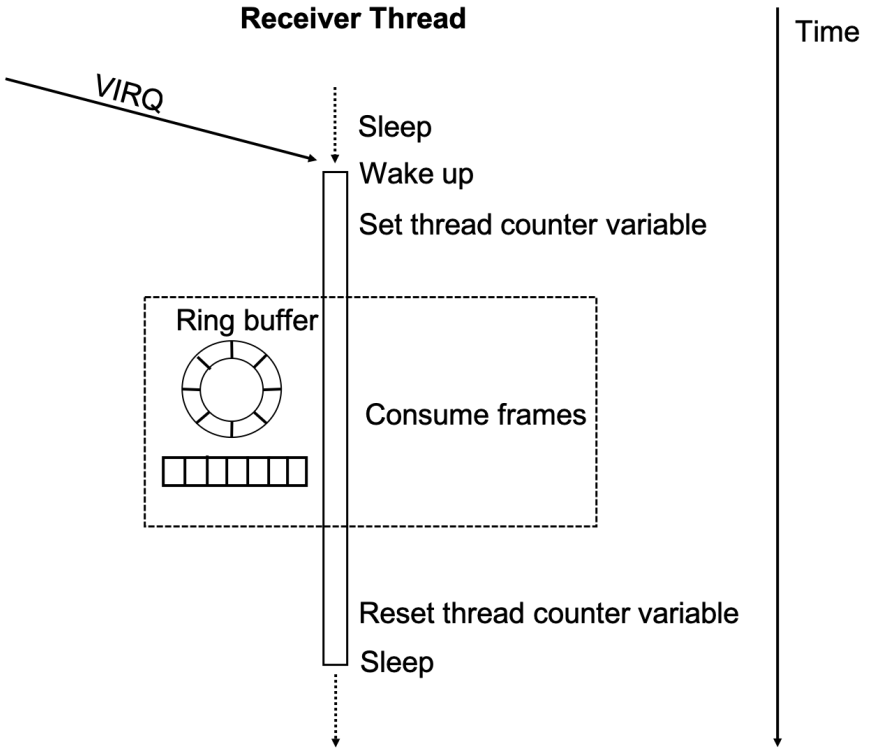

Figure 4.5: Receiver Thread

4.7 Sending and Receiving Network Frames

The network server is a dedicated service domain for networking, which sends and receives frames through the ixg interface created by the 10GbE ixgbe NIC driver. It can directly access NIC via the Xen PCI passthrough (Chapter 5) and receive the frames coming through the ixg interface. When receiving, it extracts a protocol and the layer 3 port from the frame, identifies the corresponding target domain from the port map table with the protocol and port, and sends frames to the target via the IPC. When sending the frames coming from the application domain, the network server simply forwards the frames through the ixg interface. It ensures that the application domain does not fake the IP and MAC addresses.

4.8 Comparison with the Xen Split Driver Model

An alternative solution is the netfront/netback model used by Xen (Figure 4.6). The network backend (netback) driver typically runs in Dom0 and accesses a physical NIC device. The network frontend (netfront) driver is used by DomUs as a virtual NIC to establish network connectivity with the netback through the I/O ring buffer.

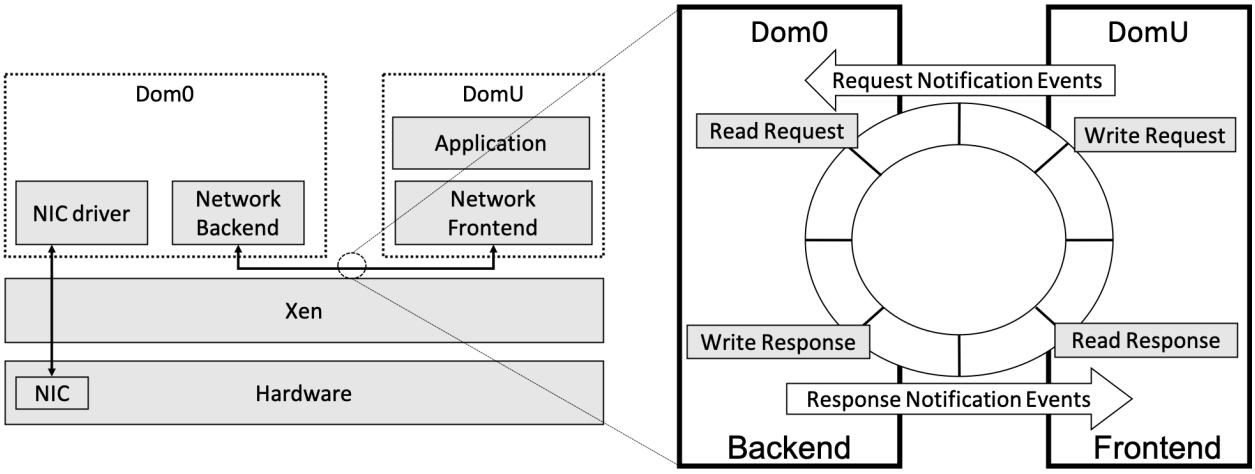

Figure 4.6: Xen split-driver model

In this model, the transmitted packets from NIC must traverse through the network stack in Dom0. In addition, all the traffic needs to go through the network address translation or bridging because each application (virtual machine instance, DomU) has its own IP address [43].

In contrast, our approach allows the packets to mostly bypass the network stack on the network server side. In addition, each application has the same IP address but a different port number such that the packets can be demultiplexed like in traditional OSes. We evaluate the Xen split-driver model and compare it to our approach of the network server in Section 7.1.

Chapter 5

Rump Kernel Glue Code

This chapter describes the glue code needed for the network server to enable support for the Intel 10GbE hardware in rumprun. Section 5.1 pertains to the modification for the 10GbE ixgbe NIC. The following Section 5.2 presents the glue code that we added. Finally, Section 5.3 covers ifconfig for configuring the ixgbe NIC exposed to the network server.

We use the Xen PCI passthrough to access NIC in the network server. The above-mentioned NIC driver is already available in NetBSD but it requires slight adaption in rumprun.

5.1 10GbE ixgbe NIC

The network server requires directs access to NIC. We choose 10GbE Intel x520-2, which is a high-end NIC device. We build the ixgbe drivers as a library (Section 5.2.1). In addition, we port ifconfig [35] to configure the ixg network interface of the ixgbe driver (Section 5.3).

5.2 Glue Code

This section describes the rump glue code for 10GbE ixgbe NIC.

5.2.1 Libpci_ixgbe

To support ixgbe in rump kernels, a configuration file (PCI_IXGBE.ioconf) and Makefile are required. Below are the configuration file and Makefile for libpci_ixgbe.

Listing 5.1: PCI_IXGBE.ioconf for libpci_ixgbe

```
1         ioconfpci_ixgbe
2
3         include "conf/files"
4         include "dev/pci/files.pci"
5         include "rump/dev/files.rump"
6
7         pseudo-root pci*
8
9         ixg* at pci? dev ? function ?
10        ixv* at pci? dev ? function ?
```

Listing 5.2: Makefile for libpci_ixgbe

```
1         RUMPTOP=${TOPRUMP}
2
3         .PATH:  ${RUMPTOP}/../dev/pci
4         .PATH:  ${RUMPTOP}/../dev/pci/ixgbe
5
6         LIB=    rumpdev_pci_ixgbe
7         COMMENT=ixgbe driver
8
9         IOCONF= PCI_IXGBE.ioconf
10        RUMP_COMPONENT=ioconf
11
12        SRCS+= ixgbe.c
13        SRCS+= ixgbe_phy.c
14        SRCS+= ixgbe_vf.c
15        SRCS+= ixgbe_mbx.c
16        SRCS+= ixgbe_osdep.c
17        SRCS+= if_sriov.c
18        SRCS+= ixgbe_api.c
19        SRCS+= if_bypass.c
20        SRCS+= if_fdir.c
21        SRCS+= ixgbe_common.c
```

```
22          SRCS+= ixgbe_netbsd.c
23          SRCS+= ixgbe_82599.c
24          SRCS+= ixgbe_netmap.c
25          SRCS+= ixv.c
26          SRCS+= ix_txrx.c
27          SRCS+= ixgbe_x550.c
28          SRCS+= ixgbe_x540.c
29          SRCS+= ixgbe_82598.c
30
31          CPPFLAGS+= -I${RUMPTOP}/librump/rumpkern
32
33          .include "${RUMPTOP}/Makefile.rump"
34          .include <bsd.lib.mk>
35          .include <bsd.klinks.mk>
```

5.3 Ifconfig

ifconfig is a system administration utility for network interface configuration. We ported it to rumprun from NetBSD. We use ifconfig to set the IP address, netmask, and maximum transmission unit (MTU) for the ixg network interface before an application starts. To configure the network interface, we need to pass parameters to ifconfig, which is done when creating rumprun binaries.

Chapter 6

Failure Recovery

This chapter describes the failure recovery supported by the network server. Section 6.1 covers the reconnection routine on the network server side. Next, Section 6.2 presents the application domain reconnection routine. Section 6.3 discusses the port binding, and Section 6.4 describes the port map table.

The design of the network server allows failure recovery. Failure recovery is one of the important advantages gained by adapting the multiserver OS design. The drawbacks, however, in the monolithic kernel design, can affect the entire system such that the system has to be rebooted [38].

After the network server reboots owing to any faults, it has to reconnect to all application domains such that they are not affected by the faults. An application domain does not have knowledge that the network server rebooted. The new network server should notify the application domains that it is a new network server and new connection is needed. The

notified application domains destroy the old connection and prepare for the new connection (Section 6.1) with the new network server.

6.1 Network Server Reconnection Routine

The reconnection routine of the network server starts with obtaining the existing application domain information. After the new network server reboots, it collects the application domains information via a hypercall (Figure 4.1) in its initialization stage (Section 4.2). If there are existing application domains, the new network server starts the reconnection routine. For reconnecting the application domains, the network server needs to notify them. As described in Section 4.4, an application domain allocates a reconnection port in its initialization. The new network server registers its information in Xen through the REGISTER hypercall (Figure 4.1). The port number is stored in Xen and passed to the new network server via the hypercall so that the network server can bind it. The network server sends a notification (VIRQ) to the existing application domains. After sending the notification, the new network server waits for a VIRQ from the application domains. The VIRQ handler for the reconnection routine is practically the same as that for the regular connection routine (Section 4.4).

The process of reconnecting the network server to the application domains is the same as that of the connection routine. However, the network server does not allocate a new welcome port in the reconnection routine.

6.2 Application Domain Reconnection Routine

On the application domain side, the incoming VIRQ from the new network server wakes up the reconnector thread (Section 6.2.1), which is created in the application domain initialization stage (Section 4.3). The reconnector thread terminates old ring buffers and unmaps

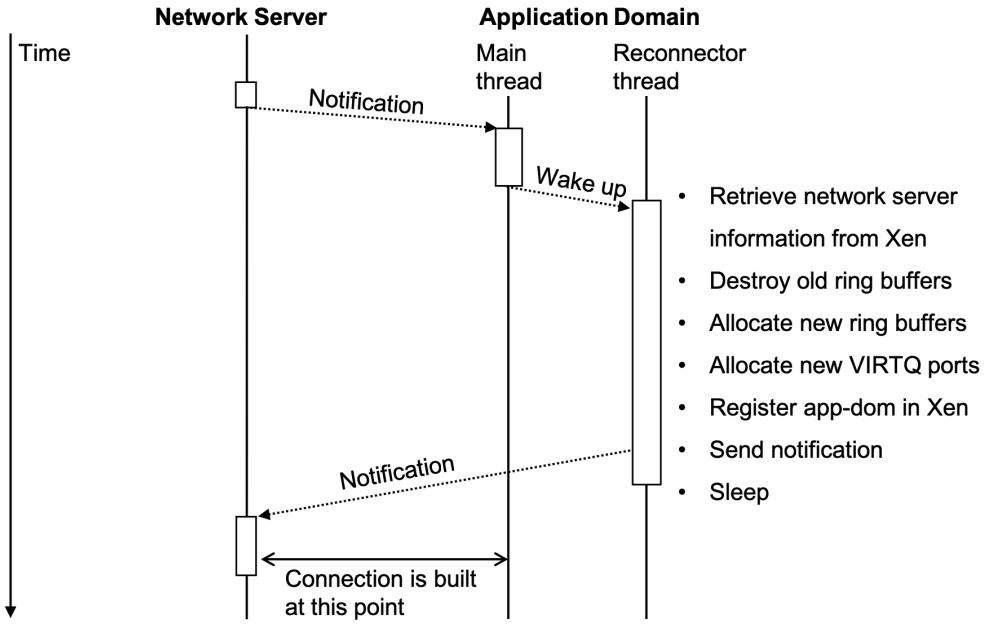

Figure 6.1: Reconnector Thread

the pages shared with the old network server. Then it obtains the information of the new network server via a hypercall. With the information, the reconnector thread initializes new ring buffers and grants pages to the new network server, similar to the regular connection routine. In addition, the application domain allocates a new reconnection port as well for future failure recovery. Last, it sends back the VIRQ that the new network server sent.

6.2.1 Reconnector Thread

The reconnector thread (Figure 6.1) is a dedicated thread for reconnection. Because the application domain should be stable during failure recovery, the reconnector thread wakes up by the VIRQ from the new network server and reconnects to the new network server. When the reconnector thread is woken up by the reconnection VIRQ from the new network server, it first obtains the network server information such as domid, reconnection VIRQ port, frame forwarding VIRQ port, and grant table references for ring buffers by making a hypercall. Then it destroys old ring buffers because ring buffers must be reinitialized in the new network server. After destroying the ring buffers, new ring buffers are created on the newly allocated shared pages. A new VIRQ port is also allocated for the frame forwarding VIRQ. In addition, the reconnector thread creates a new reconnection port for future reconnection to a new network server with DOMID_BACKEND domid. Then it registers its information of the grant table reference of the pages, including the new ring buffers, VIRQ ports, and application domain id, in Xen. Finally, the reconnector thread sends a notification to the network server to start the reconnection routine.

6.3 Port Binding

The process of port binding has to be designed differently for failure recovery. Without failure recovery of the network server, an application domain simply requests the port binding to the network server and the network server maintains the port map table. However, the port map data in the network server can be lost if any fault occurs in the network server, which then has to be rebooted. To prevent the port map data from being lost, we retain the port map table (Figure 6.2) in Xen. Keeping the port map table there allows the new network

server to access the port map data after crashing. Port binding in an application registers

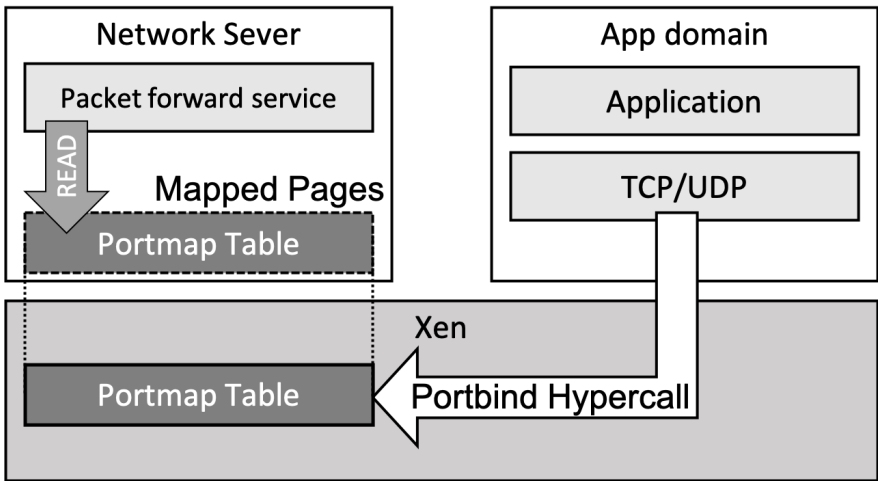

Figure 6.2: Port Map Table

ports in the port map table through a hypercall. When the application binds a socket, it calls the hypercall to request port binding to Xen. The requested port binding can be accepted or rejected if the port is already bound by others.

6.4 Portmap Table

For the transport layer protocol, the port map table is a 64K port-domid map. The port map tables for TCP and UDP are allocated in the shared pages in Xen. The shared pages are allocated when Xen boots up, and they are read-only mapped to the network server. The only approach for manipulating the table is through a hypercall by an application domain. The network server reads the port map table when it forwards the incoming frames from NIC to the corresponding target application domain. Operations on the port map tables are atomic for concurrent access from multiple threads.

6.4.1 Xen Shared Memory

The network server should be able to read the port map table in Xen for frame forwarding. We map the pages in Xen to the network server with read-only permissions. When Xen boots up, it allocates 128 pages. Then, the network server is launched, and it maps these pages by a hypercall. When receiving network packets, the network server directly accesses the port map table and obtains a target application domain to forward the frames.

6.4.2 Reconnect Multiple Application Domains

It is possible that there are multiple application domains running when the network server recovers from a failure. Therefore, the network server has to support reconnecting multiple application domains. The reconnection routine is invoked by a VIRQ, and the connection process is in interrupt context. There is a possibility that the connection processes are executed simultaneously. To reconnect appropriately, the connection process must be multiprocessor safe (MP-safe). Hypercalls in the connection process do not have issues because we use a spinlock to serialize them. For port binding, all the operations to the port map table are atomic, i.e., MP-safe as well.

Chapter 7

Evaluation

This chapter describes the results of the experimental evaluation. The objective of the evaluation is to answer the following questions:

- What are the major contributors to the performance overhead in the network server compared to NetBSD and Linux?
- Does then the network server provide better latency and throughput for real-world cloud applications?
- Is the network server capable of failure recovery, and how does it affect the applications?

We evaluate the network server with a series of micro-benchmarks and macro-benchmarks and compare it against NetBSD, which is our baseline, because rumprun is based on the NetBSD code. We also include Linux as it is the most commonly used server OS [32].

Section 7.1 first describes the NetPIPE results. Section 7.2 covers the NGINX http server.

Following this, Section 7.3 briefly focuses on Redis. Finally, Section 7.4 introduces the evaluation of the failure recovery. Our experimental set-up is presented in Table 7.1.

Table 7.1: Experimental setup

Processor	2x Intel Xeon Silver 4114, 2.20GHz
Number of cores	10 per processor, per NUMA node
HyperThreading	OFF (2 per core)
TurboBoost	OFF
L1/L2 cache	64 KB / 1024 KB per core
L3 cache	14080 KB
Main Memory	100 GB
Network	Intel x520-2 10 GbE NICs (82599ES chipset)
Storage	Intel DC P3700 NVMe 400 GB

We use Xen 4.10.1 and Ubuntu LTS 16.04 with Linux 4.13 as Xen's Dom0. The same version of Linux is run on the bare-metal machine to evaluate Linux. For testing NetBSD, the latest version of NetBSD (NetBSD-8.0 [33]) with the NET_MPSAFE feature is used. The same NetBSD code base is used for the network server. The MTU size of 9000 is set in all the experiments for better 10GbE performance.

7.1 NetPIPE

NetPIPE is a well-known ping pong benchmark that sends a fixed-size message to other servers and measures the latency and bandwidth of a single flow. We evaluate Linux, the network server, NetBSD, and the original rumprun with Xen split drivers. We run Linux on the client side for all the cases. NetPIPE is configured to send 10000 messages from 64 bytes to 512 KB. Figure 7.1 shows the throughput for different message sizes.

The network server and Linux achieve a throughput of 6.9 Gbps and 7.9 Gbps with 512 KB,

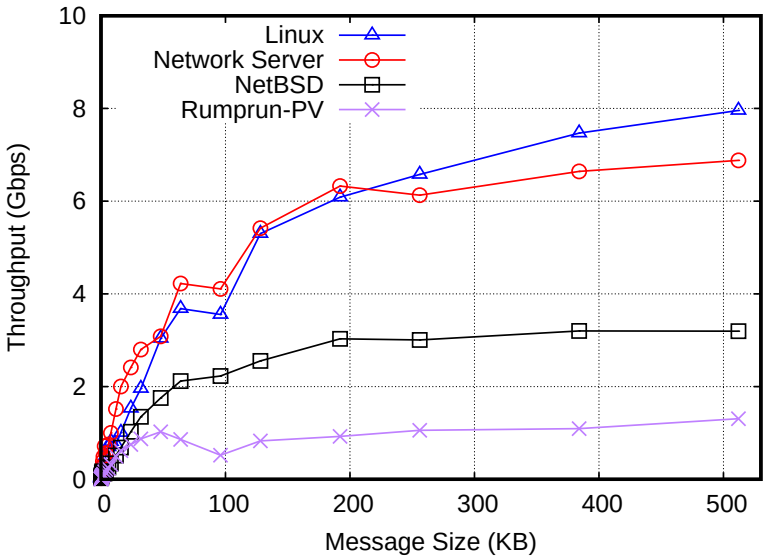

Figure 7.1: NetPIPE

respectively, and have a one-way latency of 20.8 μs and 23.9 μs, respectively, with 64-byte messages. In contrast, NetBSD can only reach 3.2 Gbps in this test while having a latency of 20.7 μs. Last, the original rumprun with the Xen split drivers can only reach 1.3 Gbps with 512 KB messages and has a latency of 75.1 μs.

The network server outperforms NetBSD for all the message sizes. Interestingly, the network server is even faster than Linux for smaller messages (less than 200 KB). We attribute this performance improvement to the cost of system calls for smaller messages [44] because the application domain replaces all the system calls with regular function calls and there is only a single address space such that the mode switch that the traditional monolithic OS requires does not occur.

7.2 Nginx HTTP server

We choose a well-known web server, Nginx [36], and the Apache benchmark as network-bound macro-benchmarks. We set the Apache benchmark to send 10000 requests with different concurrencies on the client side.

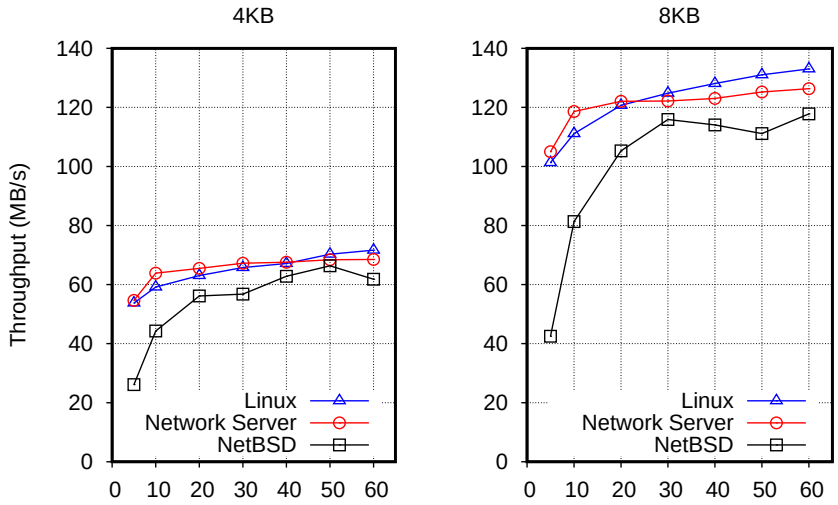

Figure 7.2: Nginx HTTP Server with file size 4KB and 8KB

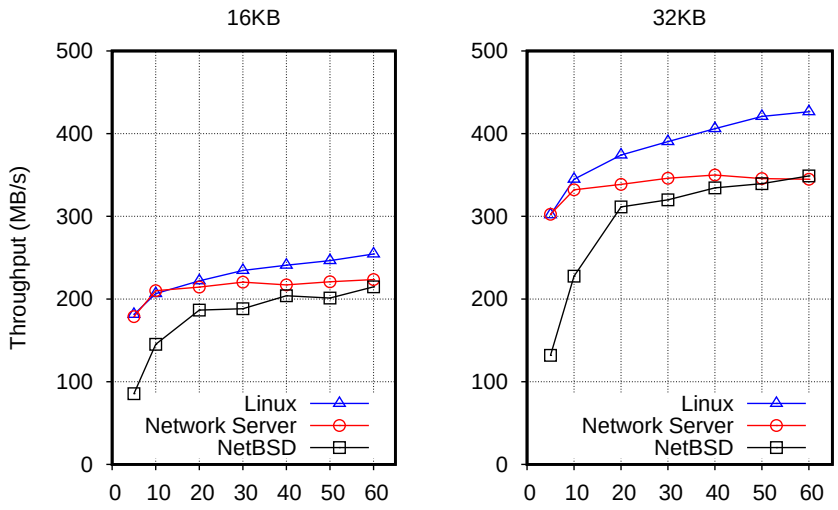

Figure 7.3: Nginx HTTP Server with file size 16KB and 32KB

Figures 7.2, 7.3, 7.4 describe the throughput for different concurrency levels and file sizes.

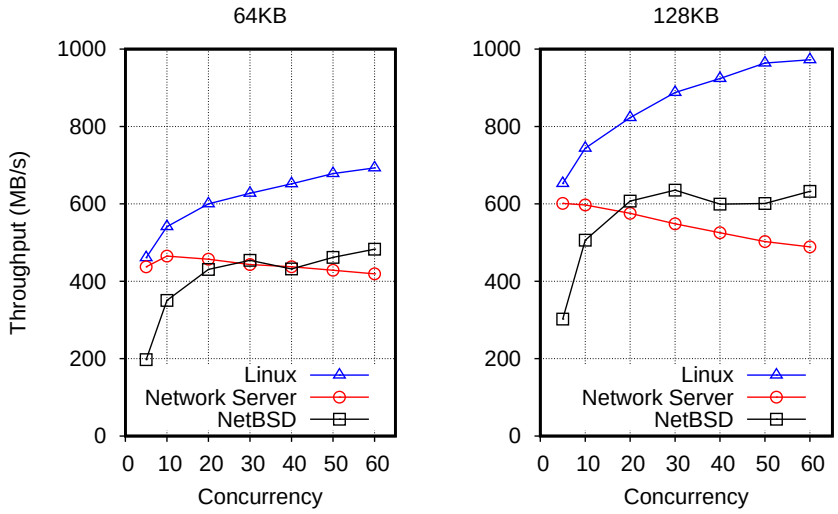

Figure 7.4: Nginx HTTP Server with file size 64KB and 128KB

The network server outperforms NetBSD and even Linux for small file sizes, and is particularly 21% faster than NetBSD for 8 KB/50. The network server is faster than NetBSD in all other settings.

7.3 Redis

Redis is an in-memory key-value store used by many cloud applications [42]. We run Redis-server 4.0.9 and benchmark 1000000 SET/GET operations with the Redis benchmark [41].

The pipeline mode of the Redis benchmark sends requests without waiting for responses. The pipeline size is configured to 1000, and the number of concurrent connections is set from 20 to 280. Figure 7.5 shows the results. The throughput of NetBSD is worse than that of Linux owing to a better-optimized Linux network stack. The network server, however, outperforms NetBSD because it removes the system call layer.

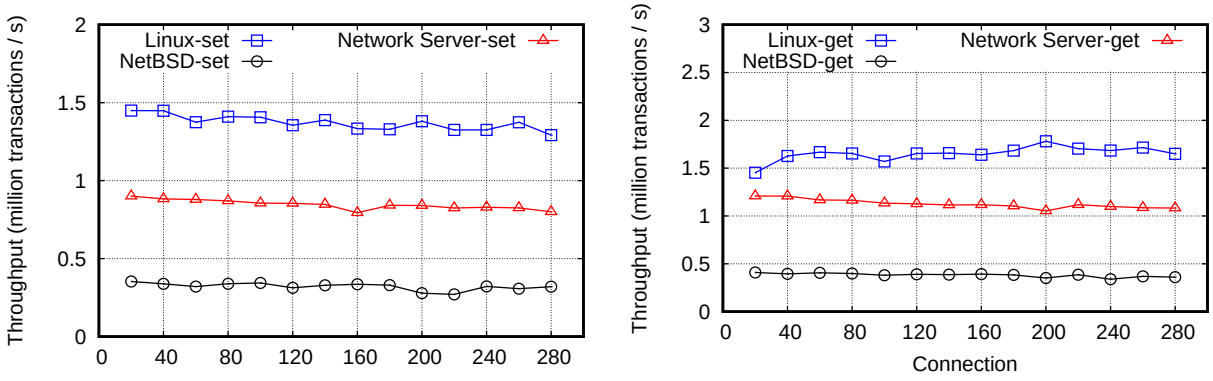

Figure 7.5: Redis benchmark of SET/GET operations

7.4 Failure Recovery

The network server can be recovered from any faults occurring in service domains, including memory access violations, deadlocks, or interrupt handling routine failures. Failure recovery is one of the most important benefits achieved by adopting a multiserver OS design. However, any failure can affect the entire system in the monolithic kernel design such that the system has to be rebooted. We designed an experiment running an application that has two concurrent jobs: one uses networking (Nginx) and the other performs file I/O. We want to demonstrate that even though Nginx is unable to talk to NIC owing to a fault in the network server, the file I/O job is not affected.

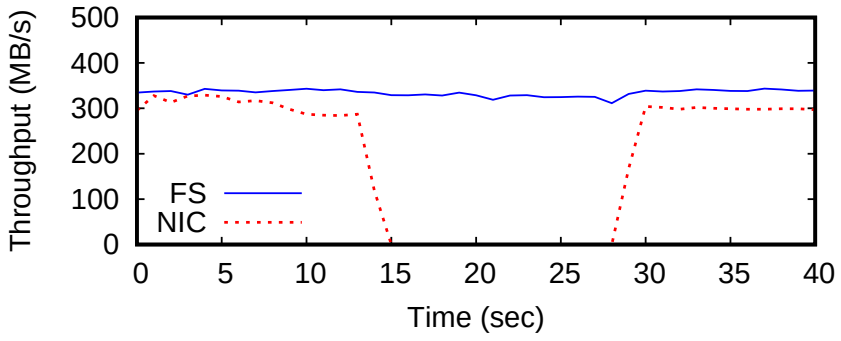

Figure 7.6: Failure Recovery

Figure 7.6 displays the network and file I/O transfer speeds. As Figure 7.6 exhibits, the file I/O transfer speeds are consistent throughout the entire test and the network transfer speeds are restored after the network server is rebooted.

Chapter 8

Conclusion

Owing to the reliability and security concerns of the traditional monolithic kernel, multiserver OS designs have emerged. Strong isolation of the network server in a multiserver OS provides better robustness and security and reduces the TCB size. Moreover, the failure recovery capability of the network server enhances the reliability such that a failed network server can be restored without affecting the remaining system. In this thesis, we proposed a network server for LibrettOS [43], a research prototype OS being developed at Virginia Tech. Our network server is a service domain running on rumprun and providing an L2 frame forwarding service to application domains. It can fully leverage existing device drivers from NetBSD with very little (if any) modification required.

With our proposed network server, we gained knowledge that isolation of the network stack without degrading the performance is possible. However, we also learned that the multiserver OS generally has a slower performance than the monolithic kernel because of the overhead of the IPC (between system servers) introduced in the multiserver OS [5]. Our experimental

evaluation with the NetPIPE and Apache benchmark demonstrated that the network server outperformed NetBSD in terms of the latency and throughput. We claim this is a result of adopting unikernels and carefully leveraging the Xen PCI passthrough. Last, our proposed network server proved that failure recovery can be achieved in a short time, which bodes well for future work.

8.1 Future Work

There is additional work that can be conducted on the network server. For performance improvement, we can implement zero-copying in the packet forwarding path. In the current implementation of the network server, we make multiple copies of the packets to/from the ring buffers, which are used for the IPC between the network server and applications. To accomplish zero-copying, modification on the TCP/IP stack in the rump kernel is required. We can also extend our network server to run on seL4, which is secured and formally verified. Rumprun can already run on seL4, but additional work such as page sharing and IPC is needed.

Bibliography

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. 1986. Mach: A new kernel foundation for UNIX development. (1986).

[2] Paul Barham, Boris Dragovic, Keir Fraser, and et al. 2003. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*. Bolton Landing, NY, USA, 164–177.

[3] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A protected dataplane operating system for high throughput and low latency. In *Proceedings of the 11th USENIX Symposium on Operating System Design and Implementation (OSDI)*. USENIX.

[4] T. Bushnell. 1996. Towards a new strategy for OS design. http://www.gnu.org/software/hurd/hurd-paper.html.

[5] Daniel J Chiba. 2018. *Optimizing Boot Times and Enhancing Binary Compatibility for Unikernels*. Master's thesis. Virginia Polytechnic Institute and State University.

[6] Cloudozer LLP. 2017. LING/Erlang on Xen website. http://erlangonxen.org/. Online, accessed 11/20/2017.

[7] ETH Systems Group. 2018. The Barrelfish Operating System. http://www.barrelfish.org/ Online, accessed 2018/11/13.

[8] Keir Fraser, H. Steven, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. 2004. Safe hardware access with the Xen virtual machine monitor. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for the on-demand IT InfraStructure (OASIS'04)*.

[9] FreeBSD Kernel Developer's Manual. 2018. MBUF(9). https://www.freebsd.org/cgi/man.cgi?query=mbuf&sektion=9&manpath=FreeBSD+6.0-stable Online, accessed 2018/10/22.

[10] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin Elphinstone, Volkmar Uhlig, Jonathon E. Tidswell, Luke Deller, and Lars Reuther. 2000. The SawMill Multiserver Approach. In *Proceedings of the 9th ACM SIGOPS European Workshop*. Kolding Denmark, 109–114. https://os.inf.tu-dresden.de/~reuther/publications/sigops00.pdf

[11] Cristiano Giuffrida, Anton Kuijsten, and Andrew S Tanenbaum. 2012. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization.. In *USENIX Security Symposium*. 475–490.

[12] Per Brinch Hansen. 1970. The nucleus of a multiprogramming system. *Commun. ACM* 13, 4 (1970), 238–241.

[13] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. 2006. The Architecture of a Fault-Resilient Operating System. In *Proceedings of 12th ASCI Conference (ASCI'06)*. Lommel, Belgium, 74–81.

[14] Jorrit N Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S Tanenbaum.

2006. Reorganizing UNIX for reliability. In *Asia-Pacific Conference on Advances in Computer Systems Architecture*. Springer, 81–94.

[15] Jorrit N Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S Tanenbaum. 2009. Fault isolation for device drivers. In *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*. IEEE, 33–42.

[16] Intel Corporation. 2018. Data Plane Development Kit (DPDK). http://www.intel.com/content/www/us/en/communications/data-plane-development-kit.html

[17] Jakub Jermář. 2010. Developing a Multiserver Operating System. http://www.helenos.org/doc/slides/2010-02-03-Jermar-Multiserver.pdf Online, accessed 2018/11/06.

[18] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. 1997. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP '97)*. ACM, New York, NY, USA, 52–65. https://doi.org/10.1145/268998.266644

[19] Antti Kantee. 2012. Flexible Operating System Internals: The Design and Implementation of the Anykernel and Rump Kernels. In *Ph.D. thesis, Department of Computer Science and Engineering, Aalto University*.

[20] Antti Kantee. 2016. *The design and implementation of the anykernel and rump kernels* (second ed.). Self-publishing.

[21] Antti Kantee and Justin Cormack. 2014. Rump Kernels No OS? No Problem! *USENIX; login: magazine* (2014).

[22] Avi Kivity. 2007. KVM: the Linux virtual machine monitor. In *OLS '07: The 2007 Ottawa Linux Symposium*. 225–230.

[23] Avi Kivity, Dor Laor Glauber Costa, and Pekka Enberg. 2014. OS v - Optimizing the Operating System for Virtual Machines. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC'14)*. 61.

[24] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 207–220. https://doi.org/10.1145/1629575.1629596

[25] Stefan Lankes, Simon Pickartz, and Jens Breitbart. 2016. HermitCore: a unikernel for extreme scale computing. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS 2016)*. ACM.

[26] Jochen Liedtke. 1996. Toward real microkernels. *Commun. ACM* 39, 9 (1996), 70–77.

[27] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *ArXiv e-prints* (Jan. 2018). arXiv:1801.01207

[28] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, Jon Crowcroft, and Ian Leslie. 2015. Jitsu: Just-In-Time Summoning of Unikernels. In *Proceedings of the 2015 USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*. 559–573.

[29] A Madhavapeddy, R Mortier, C Rotsos, DJ Scott, B Singh, T Gazagnaire, S Smith, S Hand, and J Crowcroft. 2013. Unikernels: library operating systems for the cloud.. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. ACM, 461–472.

[30] Anil Madhavapeddy and David J. Scott. 2013. Unikernels: The Rise of the Virtual Library Operating System. *acmqueue* 11 (2013), 30. Issue 11.

[31] Dirk Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.* 2014, 239, Article 2 (March 2014). http://dl.acm.org/citation.cfm?id=2600239.2600241

[32] Moe Long. 2018. 12 Best Linux Server Operating Systems and Who Should Use Them. https://www.makeuseof.com/tag/best-linux-server-operating-systems/ Online, accessed 2018/11/13.

[33] NetBSD Contributors. 2018. NetBSD 8.0. https://www.netbsd.org/releases/formal-8/NetBSD-8.0.html Online, accessed 2018/09/30.

[34] NetBSD Contributors. 2018. The NetBSD Project. http://netbsd.org/ Online, accessed 2018/09/30.

[35] NetBSD Manual. 2018. ifconfig(8). http://netbsd.gw.com/cgi-bin/man-cgi?ifconfig+8.amd64+NetBSD-8.0 Online, accessed 2018/10/25.

[36] NGINX Contributors. 2018. Nginx website. http://nginx.org/ Online, accessed 2018/09/30.

[37] Ruslan Nikolaev. 2013. *Design and Implementation of the VirtuOS Operating System*. Ph.D. Dissertation. Virginia Polytechnic Institute and State University.

[38] Ruslan Nikolaev and Godmar Back. 2013. VirtuOS: an operating system with kernel virtualization. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 116–132.

[39] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2014. Arrakis: the operating system is the control plane. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*. USENIX Association, 1–16.

[40] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. 2011. Rethinking the Library OS from the Top Down. *SIGARCH Comput. Archit. News* 39, 1 (March 2011), 291–304. https://doi.org/10.1145/1961295.1950399

[41] Redis Contributors. 2018. How fast is Redis? https://redis.io/topics/benchmarks Online, accessed 2018/11/08.

[42] Redis Contributors. 2018. Redis. https://redis.io/ Online, accessed 2018/09/20.

[43] Ruslan Nikolaev, Mincheol Sung, Pierre Olivier, and Binoy Ravindran. 2019. LibrettOS: Fusing Multiserver and Library OS Paradigms for Isolation and Performance. Under submission.

[44] Livio Soares and Michael Stumm. 2010. FlexSC: flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design & Implementation (OSDI'10)*. Vancouver, BC, Canada, 1–8. http://dl.acm.org/citation.cfm?id=1924943.1924946

[45] J. Mark Stevenson and Daniel P. Julin. 1995. Mach-US: UNIX on generic OS object servers. In *Proceedings of the USENIX 1995 Technical Conference (TCON'95)*. New Orleans, Louisiana, 119–130. http://dl.acm.org/citation.cfm?id=1267411.1267421

[46] Andrew S. Tanenbaum and Herbert Bos. 2014. *Modern Operating Systems* (fourth ed.). Prentice Hall Press, Upper Saddle River, NJ.

[47] Ted Hudek. 2018. Overview of Single Root I/O Virtualization (SR-IOV). https://docs.microsoft.com/en-us/windows-hardware/drivers/network/overview-of-single-root-i-o-virtualization--sr-iov- Online, accessed 2018/11/13.

[48] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E Porter. 2014. Cooperation and security isolation of library OSes for multi-process applications. In *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 9.

[49] Chia-Che Tsai, Donald E Porter, and Mona Vij. 2017. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC)*.

[50] Xen Project Software Overview. 2018. Hypercall. https://wiki.xen.org/wiki/Xen_Project_Software_Overview#HVM_and_its_variants_.28x86.29 Online, accessed 2018/11/21.

[51] Xen project wiki. 2018. Dom0. https://wiki.xenproject.org/wiki/Dom0 Online, accessed 2018/11/13.

[52] Xen project wiki. 2018. Grant Table. https://wiki.xenproject.org/wiki/Grant_Table Online, accessed 2018/11/05.

[53] Xen project wiki. 2018. Hypercall. https://wiki.xenproject.org/wiki/Hypercall Online, accessed 2018/11/05.

[54] Xen project wiki. 2018. Respond to Meltdown and Spectre. https://wiki.xenproject.org/wiki/Respond_to_Meltdown_and_Spectre Online, accessed 2018/11/19.

[55] Xen project wiki. 2018. Xen PCI Passthrough. https://wiki.xenproject.org/wiki/Xen_PCI_Passthrough Online, accessed 2018/10/25.

[56] Yiming Zhang, Jon Crowcroft, Dongsheng Li, Chengfen Zhang, Huiba Li, Yaozheng Wang, Kai Yu, Yongqiang Xiong, and Guihai Chen. 2018. KylinX: A Dynamic Library Operating System for Simplified and Efficient Cloud Virtualization. In *Proceedings of the 2018 USENIX Annual Technical Conference*.