# Porting ixy.rs to Redox

Simon Ellmann, Paul Emmerich*
*Chair of Network Architectures and Services, Department of Informatics
Technical University of Munich, Germany
Email: ellmann@in.tum.de, emmericp@net.in.tum.de

***Abstract*—Drivers are traditionally written in C and make up a huge part of every operating system. 66% of all code in the Linux kernel is driver code, and the current number of drivers and their complexity are still increasing. However, this complexity comes at a price of decreasing readability of the code and greater vulnerability. Some of the problems related to drivers, especially concerning their safety and security, can be mitigated by running drivers in user space. This is especially beneficial on modern microkernel architectures where the operating system can seamlessly interact with user space drivers. A good example for such an operating system is Redox, a Unix-like OS written in Rust.**

**In this paper, we implement the first 10 Gbit/s user space network driver for Redox by porting an existing Linux implementation, ixy.rs. We evaluate the driver's overall structure, integration into the operating system and performance and compare it to the original implementation and Redox's other network drivers. We show that our driver is many times faster than Redox's other drivers although it uses less unsafe code. Our code is available as free and open source under the AGPL-3.0 license at https://github.com/ackxolotl/ixgbed.**

***Index Terms*—Rust, User Space Driver, Redox, Performance, Microkernel**

## 1. Introduction

Up until the 1940s, computers could only perform series of single tasks. Today well known features like scheduling, memory management and multitasking did not exist in operating systems – as far as programs were run in operating systems at all – until the 1960s when hardware abstraction became prevalent. In 1969, the development of Unix started, an operating system containing most of its functionality inside a big kernel, thus forming one of the first monolithic kernel architectures. The development of Unix set a variety of standards for today's operating systems. Popular operating systems like Windows, macOS and Linux-based ones like Debian or Ubuntu are still built on monolithic or hybrid kernel architectures.

Although monolithic kernels are very common and might be easier to implement, they are considered obsolete by researchers since the 1990s (see the Tanenbaum–Torvalds debate). This is due to various flaws in their architecture: Programming mistakes in the kernel can take down the whole system or corrupt other processes since every piece of software in the kernel is executed with full privileges, development of new software is tedious because common libraries and debuggers are missing and maintenance of the kernel can be challenging if its complexity is rapidly increasing (e.g. like the Linux kernel).

A more temporary approach to operating system design are microkernels. Unlike monolithic kernels, microkernels try to minimize the amount of software running in kernel space by moving almost all applications to user space. This includes (but is not limited to) the following services running in user space: Drivers, file system and inter-process communication. Keeping the kernel small yields various advantages: Kernel components traditionally written in C can be rewritten in other programming languages, execution of this software can be easily debugged in user space and faults in user space daemons have no impact on the overall system.

So why is all this relevant? In 2019, Cutler et al. evaluated security bugs leading to arbitrary code execution in the Linux kernel [1]. Of the 65 bugs published in the CVE database in 2017 with patches available, 40 were memory bugs that could have been prevented by using a memory-safe language like Go or Rust. Of these 40 memory bugs, 39 were located in device drivers. Since 66% of the code in the Linux kernel is driver code [2], the findings of Cutler et al. reveal that drivers offer a large attack surface and many possibilities for improvement.

Rewriting drivers in memory-safe programming languages can mitigate many safety and security faults. An example for such a driver is ixy.rs [3], a rewrite of the simple user space network driver ixy [4] in Rust for Linux. Unfortunately, Linux is not particularly suitable for user space networking due to its monolithic kernel design: The OS network stack cannot be used by user space drivers and memory allocation for the PCIe device is only possible by using a quirk in Linux. However, there are other operating systems based on microkernels like Redox [5], a Unix-like operating system written in Rust, that implement full network functionality in user space.

In this paper, we try to combine the advantages of a user space network driver and an operating system based on a microkernel, both written in a memory-safe programming language, by porting ixy.rs to Redox. The common denominator of ixy.rs and Redox is Rust [6], a novel programming language illustrated in Section 2. Section 3 introduces Redox, while the following Section is about ixy. In Section 5 we evaluate ixy on Redox, Section 6 presents related work to the inclined reader. We summarize our results in Section 7 and present opportunities for future work in the area of ixy.rs on Redox.

The main contribution of this paper is the first 10 Gbit/s network driver on Redox [7].


doi: 10.2313/NET-2019-10-1_07

## 2. Rust

Rust is a relatively new systems programming language focusing on memory- and thread-safety. Its first stable version was released in May 2015. While Rust provides zero-cost abstractions like C++ and is also syntactically similar, its main selling point is memory safety due to its novel ownership system [6].

### 2.1. Type System

Rust is statically typed, i.e. the types of all variables and functions are checked at compile time. Functions have to be annotated by programmers explicitly, types of variables can be inferred in most cases by the Rust compiler. The type system provides "traits", i.e. interfaces that can be implemented by multiple types similar to type classes in Haskell, and generic parameters to allow for inheritance and ad hoc polymorphism.

### 2.2. Memory Management

Rust's core feature is its unique ownership system which enforces Rust's guarantees of memory safety and data-race freedom. While many programming languages make use of garbage collectors, Rust ensures at compile time that memory is allocated, handled and freed correctly. This yields two major advantages compared to garbage collection:

1) Memory handling, especially cleanup of resources, is deterministic.
2) There are no performance issues for real-time applications caused by garbage collection.

Unlike in C or C++, it is not possible in (safe) Rust to build a program leading to undefined behaviour by freeing memory twice, accessing dangling pointers or other operations violating memory safety due to the additional rules that Rust enforces on memory handling. Since Rust verifies memory safety at compile time and not at runtime, there is no size or performance overhead in compiled programs [8].

### 2.3. Ownership

The ownership system of Rust ensures that every value in Rust has a unique owner and that the scope or lifetime of a value depends on the scope/lifetime of its owner, i.e. if the owner of a value goes out of scope the value is freed (similar to *Resource Acquisition Is Initialization* (RAII) from C++) [6]. Ownership can be transferred between variables, values are either copied or moved in memory depending on whether the value is stored on the stack (and it is thus cheap to copy the value) or it is stored on the heap. Where a value is placed in memory usually depends on whether the size of a value is known at compile time or not. Values can be passed to functions by immutable or mutable references, or by value. As long as there is a reference to a value, the value cannot be moved (to inhibit dangling pointers). There can be multiple immutable references to a value or a single mutable reference. While a mutable reference to a value exists, i.e. the value is borrowed mutably, the value can only be modified through that reference and not through its owner to prevent data races.

### 2.4. Safe and Unsafe

The ownership system of Rust is very powerful. However, static analysis is quite conservative and still subject to limited decision capabilities. There are valid programs that are rejected by the compiler when the compiler is unable to decide whether the code upholds the required guarantees. This is always the case for programs that

- call foreign functions (e.g. from `libc`),
- dereference raw pointers,
- access and modify mutable static variables or
- call unsafe functions or implement unsafe traits.

These features can be used inside an `unsafe` block. In unsafe code the developer has to ensure that the program obeys the memory guarantees of Rust. Unsafe code in Rust is nothing unusual, e.g. many parts of the Rust standard library make use of unsafe code. Nevertheless, by verifying parameters before and return values after unsafe code, developers ensure that the unsafe operations are actually safe, thus forming safe wrappers around unsafe code.

## 3. Redox

Redox is an operating system written in Rust. It was published in 2015 by Jeremias Soller, is actively maintained since then and has received over 2,000 contributions by more than 70 developers. Similar to the Rust programming language, Redox focuses on safety, reliability and eventually performance [5]. To achieve these goals, the Redox developers opted for a microkernel architecture similar to MINIX [9]. Redox's developers try to "generalize various concepts from other systems, to get one unified design" [10], namely concepts from Plan 9 [11], Linux and BSD.

### 3.1. Everything is a URL

Redox generalizes Unix's "everything is a file" with its concept of "everything is a URL" [10], i.e. URLs, schemes and resources are used as communication primitives between applications. URLs model segregated virtual file systems that can be arbitrarily structured. They consist of two parts separated by a colon, the scheme (e.g. `file`) and a reference part (e.g. `/usr/bin/ping`). URLs identify resources like genuine files in the filesystem, websites, hardware devices and other primitives. Schemes are created by the kernel or user space daemons. They are registered by opening the name of the scheme in the root scheme (which defaults to empty), i.e. to create the `file` scheme a process has to open `:file` with the `CREATE` flag. Accesses to a URL are processed by the scheme-registrar which returns a handle to the requested resource, e.g. a file descriptor. Resources behave either file- or socket-like, i.e. reads and writes are buffer- or stream-oriented.


doi: 10.2313/NET-2019-10-1_07

### 3.2. Drivers in Redox

As is to be expected with a microkernel architecture, drivers in Redox operate as user space daemons. PCI drivers are launched on boot by Redox's PCI driver manager, `pcid` which in turn is launched by Redox's init process. `pcid` parses a configuration file associating PCI device vendors and classes with Redox's drivers and their command line parameters, i.e. name of the device, location of Base Address Registers (BARs), etc. Drivers have to implement various functions like `open`, `read`, `write`, `close`, etc. to communicate with other applications via Redox's URL API. Network drivers have to register the `network` scheme.

Communication between other user space programs and the driver is handled via socket-like resources.

## 4. ixy

ixy is a light-weight user space network driver written for educational purposes [4]. It is a custom reimplementation of Intel's ixgbe driver for 10 Gbit NICs. ixy's architecture is inspired by DPDK [12] and Snabb [13]. ixy does not rely on a kernel module (like Snabb) and features memory management of DMA buffers with custom pools, polling instead of an interrupt-driven design and an API that supports batch operations for receiving and transmitting packets (like DPDK). ixy was originally written in C by P. Emmerich et al. in 2017 but has been ported to more than ten other programming languages including Go, Haskell, Python and Rust (also known as ixy.rs).

We will describe the architectural design and the implementation of ixy on Redox in the following subsections. To understand how ixy and similar drivers work it is necessary to understand how the driver and the device communicate with each other. There are two communication channels for PCIe devices: The driver can access the device's configuration registers (BARs) to control the device and the device can access main memory via direct memory access (DMA) to read and write packet data and packet status information [4].

### 4.1. Memory Management

While ixy makes use of custom memory pools which are a reasonable choice for user space drivers on Linux due to missing tools for allocating and managing DMA memory in user space and to gain high performance, ixy.rs on Redox does not use custom memory pools for two reasons:

1) Redox provides an API for handling DMA memory in user space.
2) Performance of the driver is restricted due to its interrupt-driven design and the fact that packets cannot be processed in batches. These performance barriers cannot be mitigated by custom memory pools.

ixy.rs on Redox allocates all DMA memory via Redox's syscall API. Accesses to the device's registers (BARs) happen via memory mapped IO: The device is mapped into the memory space of the driver, read and write operations on the registers lead to data transfer on the PCIe bus.

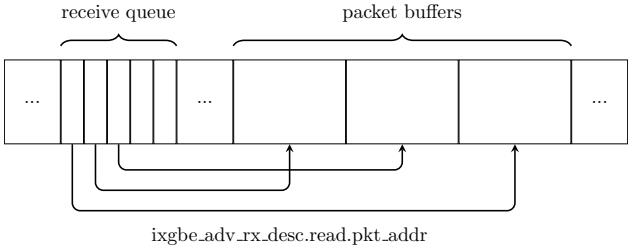


Figure 1: DMA memory containing receive queue with descriptors pointing to packet buffers.

### 4.2. Receiving Packets

NICs provide multiple ring buffers to receive and transmit packets. Incoming traffic can be split with filters if multiple queues are configured [4]. For the sake of simplicity, ixy on Redox uses only one receive and one transmit queue. Receive and transmit queues work in a similar way: Every queue is a ring buffer filled with descriptors that point to the memory address of the corresponding packet and contain status information about the packet, i.e. size of a received packet or an indicator whether a packet queued for transmittion has been sent out by the NIC yet. The structure of a receive queue is illustrated in Figure 1. Transmit and receive queues are managed by the driver and the device on a rotating basis. The device indicates its current position in the ring via the head pointer, the driver via the tail pointer [4]. Both pointers can be accessed through the BARs of the device.

Before receiving packets, the driver has to initialize the descriptors in the receive queue with physical addresses. For every incoming packet, the NIC writes the packet's data to the memory address given in the descriptor, updates the descriptor and increases the queue's head pointer.

Whether new packets have arrived can be checked by reading the head pointer. This is what the e1000 driver and rtl8186 driver of Redox do [14]. Since accessing the head pointer of the queue incurs a PCIe round trip, a better way to check for new packets is to read the descriptor status field from DMA memory that is effectively kept in the CPU cache [4].

With its custom memory pools, ixy on Linux maintains a stack of free buffers and tracks which buffers are currently in use by the device and the driver. When reading a packet, the corresponding buffer is passed to the user application and the physical address of the descriptor is updated to an unusued buffer from the free stack. Unfortunately, ixy on Redox cannot pass its memory to other user space applications and thus has to copy all received data. However, this obviates the need for a free stack and simplifies the driver: All buffers can be immidiately reused after copying the packet data, no addresses in the descriptors have to be changed.

### 4.3. Transmitting Packets

Transmitting packets works similarly to receiving packets but is more complicated as packets are sent


doi: 10.2313/NET-2019-10-1_07

asynchronously for performance reasons. The transmit functions consists of two parts: verifying if packets from previous calls have been sent out and putting the current packet in the transmit queue. The first part is usually called cleaning and is executed for the first time when every descriptor of the transmit queue has been used once, i.e. the transmit functions keeps track of used descriptors and runs its cleaning part when the counter of free descriptors equals zero. Cleaning works as follows: The status flag of the descriptor after the last cleaned descriptor is checked. If the descriptor is done, i.e. the packet has been sent out, the next descriptor is checked and the clean index and the counter of free descriptors are increased by one. If the descriptor is not done yet or the whole queue was cleaned, cleaning is finished and the packet to be sent is put into the transmit queue by copying the packet's data to the descriptor's buffer, updating the descriptor (e.g. setting the packet size) and increasing the tail pointer of the transmit queue.

### 4.4. Interrupts

The official ixy driver works in poll-mode only [4] and does not support interrupts yet. This is not a technical restriction but a performance decision since Linux offers full interrupt support in user space. However, T. Zwickl has implemented interrupt-handling for the original ixy driver [15]. Based on his work we have added support for MSI-X interrupts to ixy on Redox as well. Unfortunately, Redox does not feature MSI-X interrupts yet (which will hopefully change in the future).

### 4.5. Offloading Features

Although NICs of the ixgbe family support various offloading features, and frameworks like DPDK make use of these features, ixy only enables CRC checksum offloading to keep the driver's complexity low [4].

## 5. Evaluation

Redox is still in an experimental development state and subject to major changes. No stable version has been released yet. It is possible to boot Redox on real hardware but this requires a hard disk with no partition table [10]. For development purposes it is preferable to run Redox in a virtual machine, e.g. in QEMU which we used to conduct some performance measurements. The results of the following subsections confirm that Redox and its components (e.g. its other network drivers) are still in a very premature development state. However, our implementation is a valuable contribution to Redox from both the performance and the operational safety point of view.

### 5.1. Performance

We wrote a simple packet forwarder and packet generator application called `rheinfall`[1] to assess the transmit capabilities of our driver. All measurements were performed on commit `bccd1ca` of our implementation.

1. https://github.com/ackxolotl/rheinfall

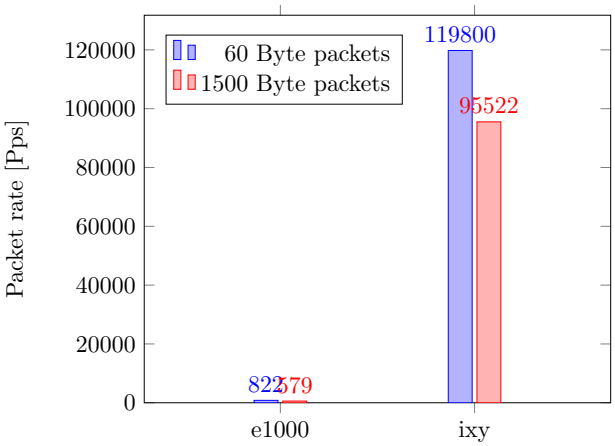


Figure 2: Packet transmit rates measured on AMD Ryzen 7 1800X with Intel 82574L (e1000) and Intel 10G X550T (ixy).

We run `rheinfall` on an AMD Ryzen 7 1800X at 3.6 GHz with Redox 0.5.0 in QEMU with KVM using an Intel X550T NIC via PCIe passthrough through the AMD IOMMU. `rheinfall` bypasses Redox's network stack `smoltcp` by accessing the driver directly, i.e. receiving and sending raw ethernet frames from/to `network`, the scheme registered by the driver. Our measurements show that ixy on Redox can transmit up to 120,000 packets per second with a size of 60 Bytes or up to 100,000 packets per second with a size of 1,500 Bytes which is equivalent to a transmit rate of about 1.1 Gbit/s.

Reasons for this rather poor performance are probably OS-dependent: a not extensively optimized kernel, sluggish interprocess-communication, many context switches for large queues of packets and the fact that data has to be copied multiple times from the application to the driver to the NIC and the other way around. Nevertheless, compared to Redox's two other network drivers, the e1000 and rtl8168 driver, this is a very reasonable packet rate. Figure 2 shows the transmit capabilities of Redox's two Intel network drivers, the e1000 and our implementation. The transmit rates of Redox's e1000 and rtl8168 driver are many times smaller than ixy's (around 90 to 150 [!] packets per second on emulated hardware or up to 1,000 packets on real hardware) due to the fact that – unlike in ixy – their transmit functions block until a packet has been sent out. This simplifies the transmit function but also has a massive impact on performance.

### 5.2. Safety

| Driver | NIC Speed | Code [Lines] | Unsafe [Lines] | % Unsafe |
|---|---|---|---|---|
| Our implementation | 10 Gbit/s | 901 | 68 | 7.5% |
| e1000 | 1 Gbit/s | 421 | 117 | 27.7% |
| rtl8168 | 1 Gbit/s | 399 | 114 | 28.6% |

TABLE 1: Unsafe code in different Redox drivers, counted with `cloc`.

Another point to note is the different amount of unsafe code in our implementation and the other two network drivers in Redox shown in table 1. Unlike the e1000 and


doi: 10.2313/NET-2019-10-1_07

rtl8168 driver, our implementation provides safe functions to read and write the device's registers by asserting that the memory address of a register is indeed inside of the mapped memory region. This optimization alone leads to a few hundred lines less unsafe code.

## 6. Related Work

As early as 1993, researchers proposed to move network software traditionally implemented in kernel space to user space. Exemplary for these efforts is the work of Chandramohan A. Thekkath, Thu D. Nguyen, et al. [16], in which they suggested to rewrite transport protocols as user-level libraries. Their work already includes a multitude of observations on different kernel designs and the resulting advantages and disadvantages for software. They claim that it is possible to implement protocols in a highly performant and secure way in user space.

Another scientific paper is "The Case for Writing Network Drivers in High-Level Programming Languages", in which P. Emmerich, S. Ellmann et al. present a network driver written in various high-level programming languages [2]. They propose to rewrite drivers instead of the whole operating system in memory-safe languages.

## 7. Conclusion and Future Work

Many bugs in current operating systems are located in driver code. By moving driver code to user space and using high-level languages (preferably memory-safe ones like Go and Rust), many safety and security related bugs can be mitigated. However, this requires a different kernel design. Modern microkernel architectures like the Redox kernel provide such a design. They form a safe alternative to the deprecated monolithic kernel design of the Linux kernel. In line with the proposal of P. Emmerich, S. Ellmann et al. to port drivers to high-level languages and user space, we present the first 10 Gbit/s user space network driver on Redox written in Rust. Our evaluation shows that the driver is a great contribution to Redox. However, there is still a huge potential and need for optimization on part of the operating system.

Future work on the driver might include implementing the use of multiple receive and transmit queues, further reducing the amount of unsafe code or enabling more hardware offloading features like VLAN tag offloading. Furthermore, depending on the development status of Redox, more detailed performance measurements could be performed, possibly using a high-speed packet generator like MoonGen [17].

## References


[1] C. Cutler, M. F. Kaashoek, and R. T. Morris, "The benefits and costs of writing a posix kernel in a high-level language," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 89–105.

[2] P. Emmerich, S. Ellmann, F. Bonk, A. Egger, T. Günzel, A. Obada, M. Stadlmeier, S. Voit, S. Huber, and G. Carle, "The Case for Writing Network Drivers in High-Level Programming Languages," 2019.

[3] S. Ellmann, "Writing Network Drivers in Rust," 2018.

[4] P. Emmerich, M. Pudelko, S. Bauer, and G. Carle, "User Space Network Drivers," in *Proceedings of the Applied Networking Research Workshop*. ACM, 2018, pp. 91–93.

[5] "Redox," https://www.redox-os.org/, accessed: 2019-06-23.

[6] S. Klabnik and C. Nichols, *The Rust Programming Language*. No Starch Press, 2018.

[7] S. Ellmann, "Ixgbe user space driver for Redox," https://github.com/ackxolotl/ixgbed, 2019, accessed: 2019-06-24.

[8] J. Blandy and J. Orendorff, *Programming Rust: Fast, Safe Systems Development*. " O'Reilly Media, Inc.", 2017.

[9] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "Minix 3: A highly reliable, self-repairing operating system," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 3, pp. 80–89, 2006.

[10] "The Redox Operating System," https://doc.redox-os.org/book/, accessed: 2019-06-23.

[11] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom, "Plan 9 from bell labs," *Computing systems*, vol. 8, no. 2, pp. 221–254, 1995.

[12] "DPDK Website," https://www.dpdk.org/, accessed: 2019-06-23.

[13] L. Gorrie *et al.*, "Snabb: Simple and fast packet networking."

[14] "Redox OS Drivers," https://gitlab.redox-os.org/redox-os/drivers, accessed: 2019-06-23.

[15] "Interrupt Handling in Ixy," https://github.com/tzwickl/ixy/tree/vfio-interrupt/, accessed: 2019-06-23.

[16] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. D. Lazowska, "Implementing network protocols at user level," *IEEE/ACM Transactions on Networking*, vol. 1, no. 5, pp. 554–565, 1993.

[17] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "Moongen: A scriptable high-speed packet generator," in *Proceedings of the 2015 Internet Measurement Conference*. ACM, 2015, pp. 275–287.





doi: 10.2313/NET-2019-10-1_07