École Polytechnique Fédérale de Lausanne

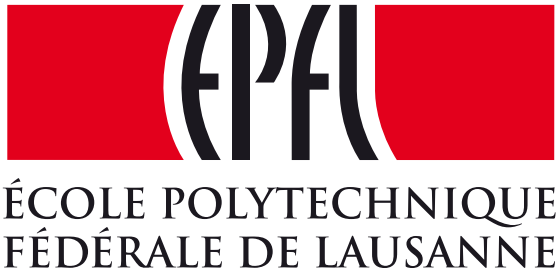
Master Thesis

A Zero Copy Key-Value Store in Rust

July 8, 2018

Author
Ogier Bouvier
ogier.bouvier@epfl.ch

Supervisors
Prof. Edouard Bugnion
DCSL | EPFL
edouard.bugnion@epfl.ch
Marios Kogias
DCSL | EPFL
marios.kogias@epfl.ch



Acknowledgements

- Marios Kogias, for his invaluable help during the project and proof-reading of this paper
- Prof. Dr. Edouard Bugnion, for his design suggestions
- Dr. George Prekas, for his advice on DPDK internals and debugging
- Adrien Ghosn for his help in proof reading
- Margaret Escandari, for her helpfulness and cheeriness

Abstract

To provide a key-value store with consistent latency independent from keys and values sizes we need to eliminate copies from the key-value store's datapath while still ensuring reliable delivery of responses to requests. We leverage several feature of the Rust programming language to achieve zero-copy reliable delivery of key-value store responses through the R2P2 protocol while providing a much higher level of safety than these type of systems usually do.

Contents

Contents

1 Introduction

Applications such as websearch, e-commerce, and social networking have strict tail-latency service level objectives (SLOs) to maintain users' attention [9]. While modern networking infrastructure allows for network transfers at 40 and 100Gbps, and round-trip latencies in a handful of microseconds [19], existing operating systems and application software can prevent applications from achieving close to hardware performance [11]. Thus, both academia and industry have focused on reducing the overheads coming from deep software stacks and inefficient system implementation, so that applications can perform as close to the hardware limits as possible.

Dataplane operating systems are such an approach that aims to optimize throughput and latency for certain types of applications, such as key-value stores. Dataplane operating trade-off generality for performance, since they can afford to simplify their networking stack, given that they serve one specific application. Systems such as IX [11] and Arrakis [22] showed that re-using design principles well-known from middleboxes, can significantly improve the application performance, both in terms of throughput, tail-latency and latency stability.

However, the above systems try to maintain compatibility with existing applications and only allowed minimal changes to the POSIX API. Moreover, the boundaries between the application and networking stack are distinct and the networking stack and operating system are agnostic to the application running on top of them. This design decision, though, prevented aggressive optimisations that would be possible in the case of co-designing the application with the networking stack. For example, the way TCP is exposed to applications through POSIX sockets, inherently implies copies between the application and the OS. These redundant copies can affect the tail-latency, especially in the presence of large values, and could potentially be eliminated with careful engineering of the application and the networking stack.

In this thesis, we attempt to co-design and implement a key-value store with the networking stack that depends on a reliable transport, while at the same time completely eliminating copies both on the receive and transmit path. To do so, we leverage Rust [15], a new systems programming language, that guarantees memory safety through an explicit memory onwernship model. By using Rust we can reason at compile time about the memory ownership that can change between the application and the networking stack.

We implemented a key-value store based on the Redis [5] API on top of Intel's DPDK [1] and serves requests over R2P2, a reliable transport protocol specifically designed for remote procedure calls (RPCs). Our key-value store depends on a lock-free store that eliminates data copies between the application and the networking stack. The key-value pairs are stored inside the networked buffers as they arrive to the server after a `set` operation, and the access to these network buffers is managed by Rust's explicit ownership model, thus preventing any race conditions.
The evaluation, while still incomplete, shows that Rust offers acceptable performance and considering the advantages it offers when dealing with highly concurrent software we think the performance penalty is well worth it.
The rest of this document is organised as follows:

- We will first explore a little background on the topic in chapter 2
- We will then explain how we intend to design the system in chapter 3
- After that we will see how the abstract design translates into a concrete Rust implementation in chapter 4
- Finally we will evaluate how our system performs compared to other similar systems in chapter 5.
- And to conclude we will mention what further work could be done to improve on the current system in chapter 6

2 Background

We start by describing the necessary background and related work to motivate our work. Given that our approach aims at co-designing a key-value store with the networking stack underneath, and do so based on specific programming language concepts, we start by describing dataplane operating systems, transport protocols for RPCs and then we focus on Rust.

2.1 Network Kernel-bypassing

Today's high end NICs support speeds that make traditional kernel networking too slow to fully exploit their capacity. The overhead of context switches and copying is just too much when treating millions of packets par second. Moreover the Linux kernel TCP stack is too general for specific applications, having to handle so many quirks of TCP makes it slower than could be achieve with a specifically designed networking stack. To address these problems, a technique called kernel bypassing was introduced allowing user space programs to speak directly to the NIC through the PCIe bus thereby eliminating the kernel overhead. These systems significantly improve throughput and tail-latency, because they reduced the per-packet system overheads.

The advances in dataplane systems were boosted by the advent of userspace packet processing frameworks. Intel's DPDK [1] and netmap [24] allow userspace programs to have access to the network rings, thus access directly the Ethernet frames, and on top implement any necessary functionality. While netmap is interrupt based, DPDK implements operates in a polling mode. Thus, dataplane systems use in a tight run-to-completion loop. Every received packet runs through network processing, then the application is called through application callbacks, and finally the responses are sent before processing the next packet.

Representative systems are IX [11], Arrakis [22], Reflex [16], mOS [14].

Applications that have benefited from dataplane operating systems include end-host applications with stringent latency requirements. Key-value stores are a big part of most of today's web services (Facebook [20], LinkedIn [6] and Amazon [2] notably makes heavy use of key-value stores). Applying kernel bypass techniques to key-value store could bring huge performance improvements, by allowing a codesign of the networking stack and the user application for our specific purpose. MICA [18] is an example of this, an holistic

approach to key-value stores.
However dataplane systems also come with drawbacks. They offer no safety guarantees. Since everything usually runs in the same userspace process, it is easy to corrupt the networking stack from the user application. Research efforts to offer safety guarantees to dataplanes depend either on hardware isolation and virtualization, such as IX, or on programming language features, such as NetBricks [21]. NetBricks is a Rust framework built on top of DPDK, aiming to provide isolated software defined network functions while eliminating the need for virtualization. It builds upon DPDK to provide a convenient Rust interface and provide safety guarantees close to that of virtualization with a much lower performance penalty.

2.2 Transport protocols for RPCs

The choice of transport protocol is critical to performance when designing an efficient RPC protocol. Widely used RPC protocols such as Thrift [25] and gRPC [3] depend on TCP and HTTP2 equivalently as their underlying RPC transport.
A reliable transport protocol is a necessary basis for higher-level application abstractions, and reduces the complexity of the application code. TCP is the defacto standard reliable transport protocol that runs on commodity hardware. However ensuring reliability usually involves creating copies of the data sent through the network to be able to retransmit it if needed while not sharing data with the user program. In the mean time, these application data could be altered. Thus copies guarantee this data integrity. Traditional reliable networking protocols (such as the Linux kernel's TCP stack) make heavy use of copies to avoid this problem. However this makes the latency dependent on the size of data therefore degrading performance, especially when the payloads grow in size. The question is then can we provide a reliable RPC protocol while avoiding copies altogether?
Although there are alternative reliable transport protocols both in literature and research, all of them suffer from the redundant copies. Such protocols are Reliable UDP [7], QUIC [4], and R2P2. R2P2, the Request Response Pair Protocol, is a highly scalable RPC protocol, that provides load-balanced delivery of multi-packet requests and responses on top of UDP, and is designed and implemented at DCSL in EPFL.
R2P2 exposes a minimal request-response API and we decided to use that to expose our proposed zero-copy key-value store.

2.3 Rust

Rust is a systems programming language from Mozilla Research, stemming from Mozilla's dissatisfaction with C++. As of today several parts of Firefox's rendering engine, Servo, have been rewritten in Rust after the language was deemed mature enough to do so. Rust is engineered to provide a lot of safety guarantees, notably memory and concurrency safety through its type system thereby eliminating a lot of common problems in systems programming.
The strong Rust guarantees also speed up development significantly by eliminating whole

categories of bugs. Most importantly, Rust's type system eliminates use-after-free, double-free, invalid pointers, memory leaks and data-races. Overall, the time writing code is increasing since the compiler will often refuse straight out to compile code that potentially violates its guarantees. The counter-part to this is that the time spent debugging is very much reduced. This is particularly useful when tracking down data race bugs which are notoriously hard to diagnose. Those kind of bugs are usually found much later in the development cycle and are usually quite hard to reproduce since they depend on a certain scheduling of the execution.

We make the claim that Rust [15] is a better suited language than C for kernel bypass networked applications. The safety guarantees Rust brings are very well suited for this kind of applications, while the performance loss (if any) is negligible. As an additional benefit Rust allows code to be much more concise and abstract while drastically cutting down on the memory and concurrency bug tracking time. Its memory management model will also prove invaluable when ensuring integrity of data in between the networking stack and the user application.

As a proof of concept we implement a key-value store on top of DPDK in order to prove that it is doable as well as reasonable, performance-wise, to use Rust in this kind of scenario.

2.3.1 Memory safety

One of the most important feature of Rust is its guarantee of memory safety, providing both the speed of manual memory management and the safety of garbage collected languages.

To provide this feature, Rust introduces a new concept called lifetimes and makes raw pointers second class citizens. These raw pointers (such as that of C) can't be dereferenced since the compiler can't guarantee that they are valid. Pointers are hard to validate and verify statically mostly because of pointer arithmetic. References, on the other hand, are guaranteed to always point to a valid memory location. Like the references in C++, references in Rust do not allow pointer arithmetic, they can only be created from a variable, which ensures that the referenced memory is valid.

Lifetimes are the central part of Rust's memory safety guarantees. They work in conjunction with references to ensure that references are used correctly (i.e. not after the value they refer to has been destroyed), since they allow the compiler to tell when a reference might outlive the value it refers to. The combination of lifetimes and references thus eliminate the dangling pointers problem entirely. Lifetimes are Rust's way of eliminating the need for garbage collection while not requiring users to manually manage memory. A lifetime gets attached to each variable created, the compiler then analyzes the control flow of the program to ensure that references or values derived from that first value (such as reference to a struct member) are not used after the value is destroyed.

However Rust is a at the core a systems programming language so we need some way to bypass the safety restrictions it provides. This is where the **unsafe** [13] keyword comes in. This keyword allows programmers to bypass many of the safety checks of

Rust [17] [23]. It is particularly useful when dealing with low level details, as is the case when sending packets in a kernel bypass fashion.

2.3.2 Concurrency safety

Rust's type system also provides protection against data-races, a common problem when using C usually solved through atomic operations and locks. While it is possible to solve this kind of problems in C, it requires careful design and is an error-prone task. Rust therefore allows quicker and safer development of concurrent software. This safety does not come for free though. In order to guarantee that there is no data-race anywhere, Rust only allows either one mutable reference or any number of immutable reference to any given variable. By doing that, the compiler can ensure that the variable will only be read concurrently or written exclusively.
Even though we speak of concurrency safety, Rust does not provide protection against deadlocks. This is far too complex a task for static analysis in a compiler since it requires costly call graph analysis and sometimes produces false positives [26].
However, the Rust standard library provides safe ways to mutate shared variables. As an example the RwLock standard library structure allows creating mutable reference to a value from an immutable RwLock instance.

2.3.3 Performance

Being a systems programming language, performance is critical to Rust. All the aforementioned guarantees are enforced statically so they do not incur any performance penalty. Removing pointer aliasing from the language also allows the compiler to do many optimizations that are not feasible safely in C or C++. As to the performance of the generated assembly code, Rust utilizes the LLVM compiler infrastructure and thus benefits from all the work that was done there. Rust also features what the Rust developers call 0-cost abstraction. Idiomatic Rust makes heavy use of closures, and that would incur, in some languages, a slight degradation of performance. But, in Rust, closure can be, most of the time, inlined thus making as if the programmer wrote the concrete code while allowing the interface to remain abstract.
In this regard, Rust combines the best of both worlds, high-level readable and abstract code while also having performance close to that of other systems programming languages.

3 Design

3.1 Goals

The goal here is to provide a zero-copy, in-memory, eventually consistent key-value store using kernel bypassing to support high throughput as well as low latency. The zero-copy property will allow our latency to remain the same regardless of the sizes of keys and values, though we can't say the same for the throughput since we will be limited by the capacity of the NIC in term of raw bandwidth.

Providing low latency and high throughput requires every component of the system to be fast and zero-copy. We therefore need a fast networking stack and obviously a high-performance and concurrent hashmap to store key-value entries as well as an efficient way to interface with the network interface controller that will send our packets on the wire. The choice of the networking protocol to transmit requests and responses is also quite important since it will be the central part of the system and will determine how much processing is necessary to receive and send packets.

But creating a fast system is only part of the goal here. The networking code should be re-usable in order to facilitate further development of safe high-performance networking applications in Rust. Providing a convenient and clear API to the networking stack is therefore also a design goal. That means the API should follow idiomatic Rust interface design principles and leverage all the language features necessary to make it re-usable and generic.

The design of the system as a whole, as well as how each component interface with the others is shown in figure 3.1.

3.2 Local store

The central component of any key-value store is an efficient datastructure to store and retrieve values by their keys. Since we also aim to provide true zero-copy semantics, this datastructure should also avoid copying data around. Moreover, it should allow values to be retrieved and used concurrently with minimum overhead. A way to make values live outside of the store should also be provided in order to permit said values to be held in the networking stack after having been removed from the store.

With all these constraints in mind, we go with the design shown in figure 3.2. The top

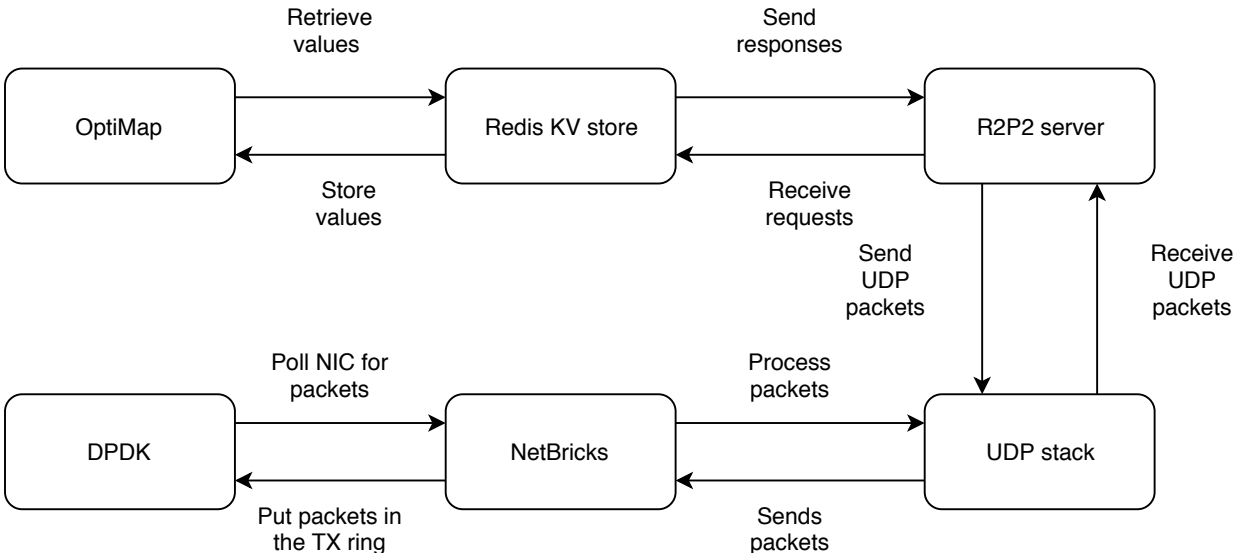


Figure 3.1: System design overview

level table is a contiguous array of atomic pointers to overflow buckets. And in turn each overflow bucket entry is a pointer to a key-value pair structure allowing fast, easy and zero-copy retrieval of values.

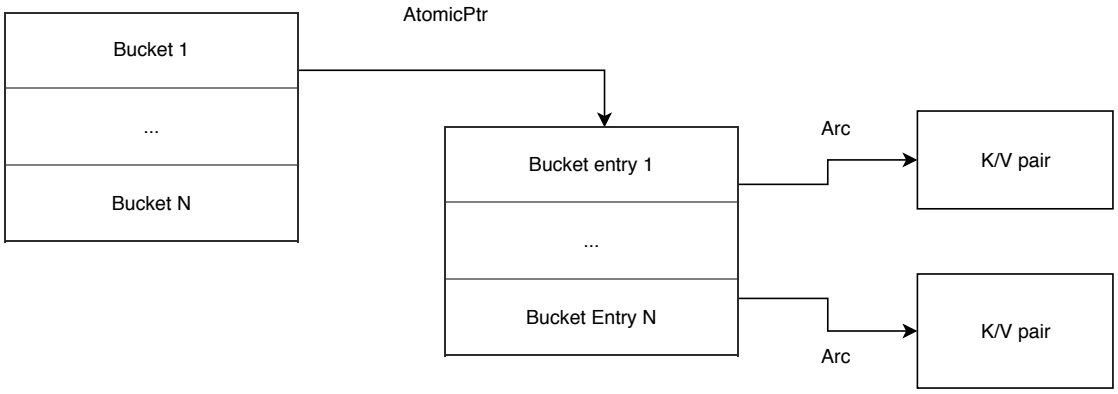


Figure 3.2: Hashmap overview

To satisfy the constraints, each pointer inside each overflow bucket, as well as the pointers to the overflow buckets themselves are atomically reference counted. Since our key-value store will use many threads working concurrently to satisfy requests an atomic reference count is necessary to avoid race conditions when updating the reference count. But more importantly an atomic reference count allows values to be removed from the store while still being used elsewhere, as shown in figure 3.3.
Let's walk through how a **concurrent insert** will work with a concurrent retrieval of the same key. First both threads will attempt to fetch the current value of the bucket, bringing its reference count to 3. Once that is done the GET thread will extract the

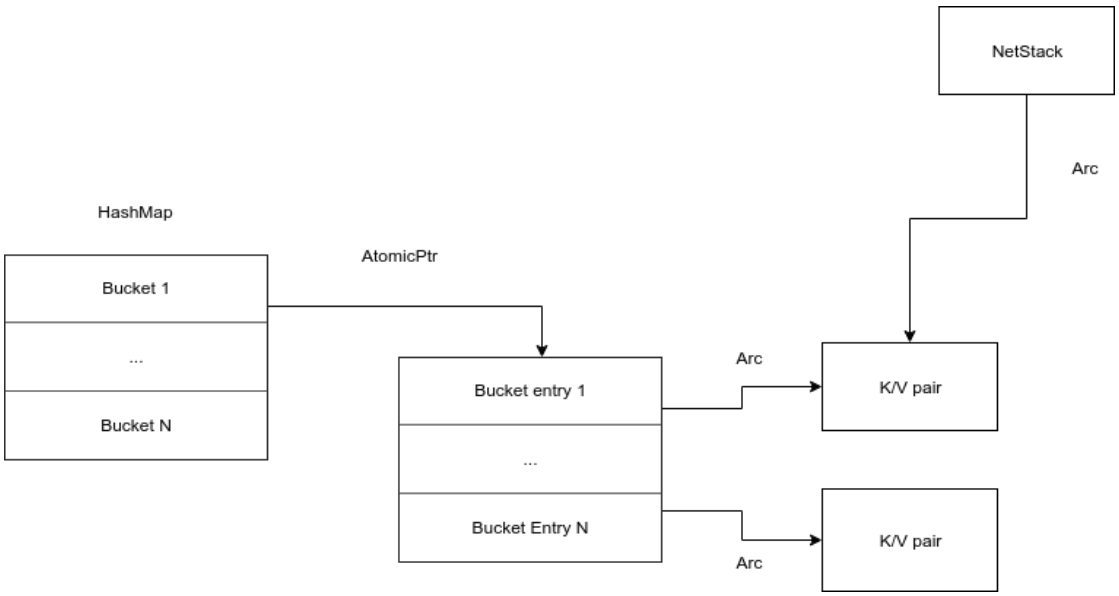


Figure 3.3: Value sharing

proper key from the bucket and send it. In the meantime the PUT thread will copy the content of the same bucket, append or replace the correct key inside the bucket then attempt to swap atomically the bucket with the value currently stored (the one the GET thread is using). For the sake of simplicity, let's say no one has modified the bucket in the meantime, the swap therefore succeeds. The PUT thread then drops its reference to the old bucket, but the GET thread still has a reference to it so it stays allocated until the GET thread is done with its retrieval. Every subsequent GET request after this point will then access the new version of the bucket created by the GET thread.

A **key removal** follows the same process as an insertion except that instead of adding a key to the bucket we remove it before swapping it back with the old version.

This lock-free design allows a fast, concurrent and zero-copy insertion of values inside the hashmap. However in the case where the workload is write-heavy, an optimistic concurrency control scheme will result in a lot of wasted work as each thread will need to build the new version of the bucket multiple times. The most expensive operation for an insert is actually the atomic compare and swap. Since a typical workload on key-value stores implies a majority of reads compared to writes (the Facebook USR load is only 0.2% writes [8]), this won't be a problem in our target use case.

If we consider all the properties that the resulting hashmap has, we realize that it uses an optimistic multi-version concurrency control scheme. Optimistic, since each thread works on its own before attempting to synchronize with other threads, and multi-version since each bucket can have multiple versions in use at any given point.

On a related note, Rust's concurrency guarantees ensure that sharing key-value pairs extracted from the hashmap amongst threads isn't a problem. Indeed since each thread will have an immutable reference to the data, the compiler will enforce that these threads

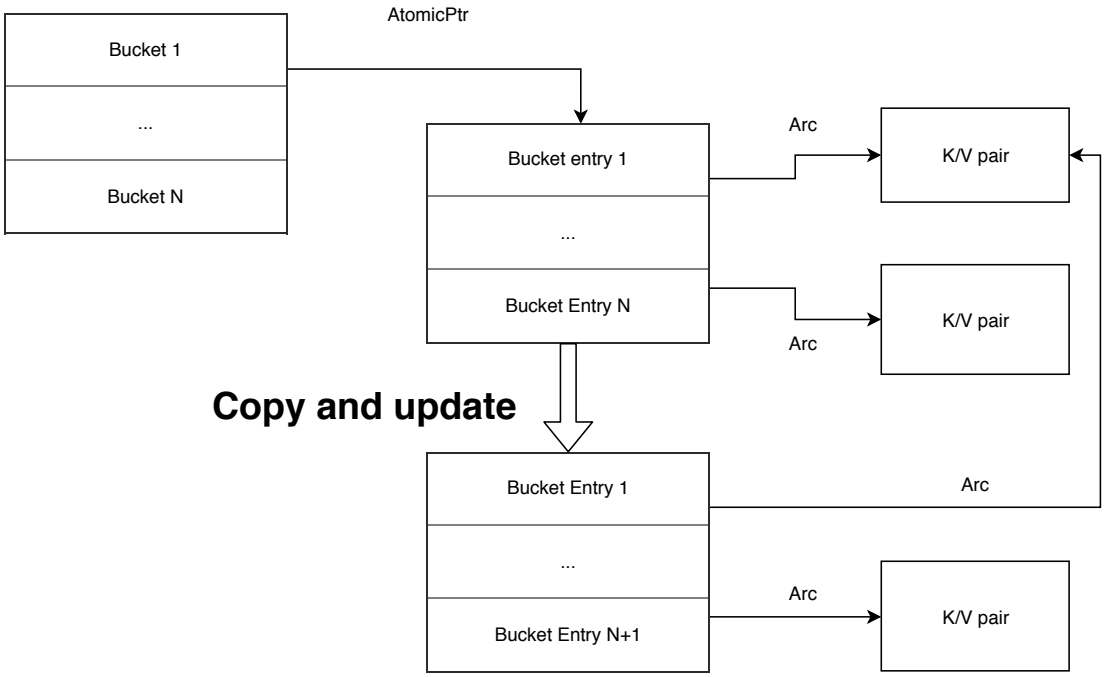


Figure 3.4: Value insertion

don't modify it at compile time.

3.3 Key-value store

The key-value store is what would be considered as the user application. As such it will provide an example of how to use the system. It is the last layer of the system and should not rely on any details of the lower layers of the networking stack. Its role in the system while be to process individual R2P2 request and parse them using the application layer protocol, and also to respond accordingly.

It will also have to handle the critical task of storing the key-value pairs in the store in a manner that allows zero copy insertion and retrieval. Great care should therefore but put toward the design of this part of the system.

3.4 R2P2 stack

As stated in the design goals the R2P2 stack should provide an easy-to-use, reliable and zero-copy interface to user applications. This R2P2 stack is the interface between the user application itself (the key-value store in our case) and the UDP networking stack and handles request parsing, response sending and retransmissions when needed.

The R2P2 stack will be implemented on top of the UDP stack described in section 3.5. The API design of the R2P2 stack is therefore constrained by the design of the UDP stack. We thus make use of callback in the R2P2 stack. The R2P2 stack itself interfaces with the UDP stack by registered a callback invoked on each UDP packet. Since R2P2

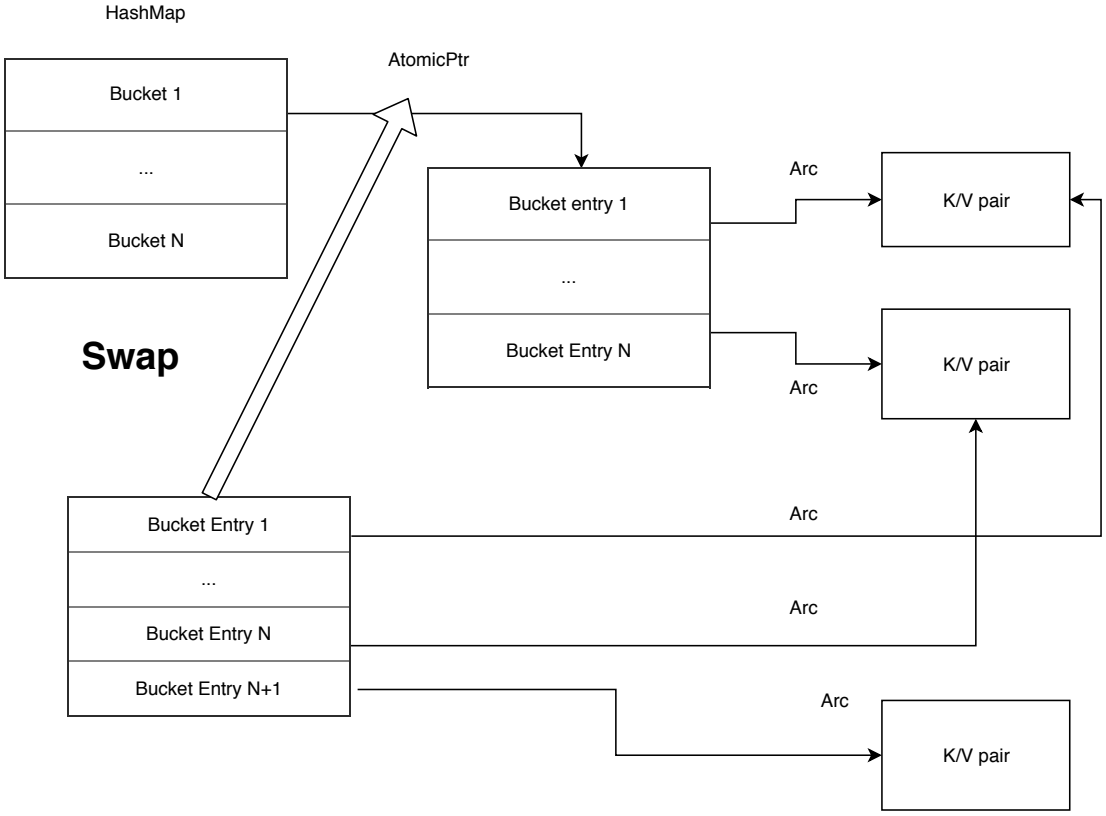


Figure 3.5: Bucket swapping

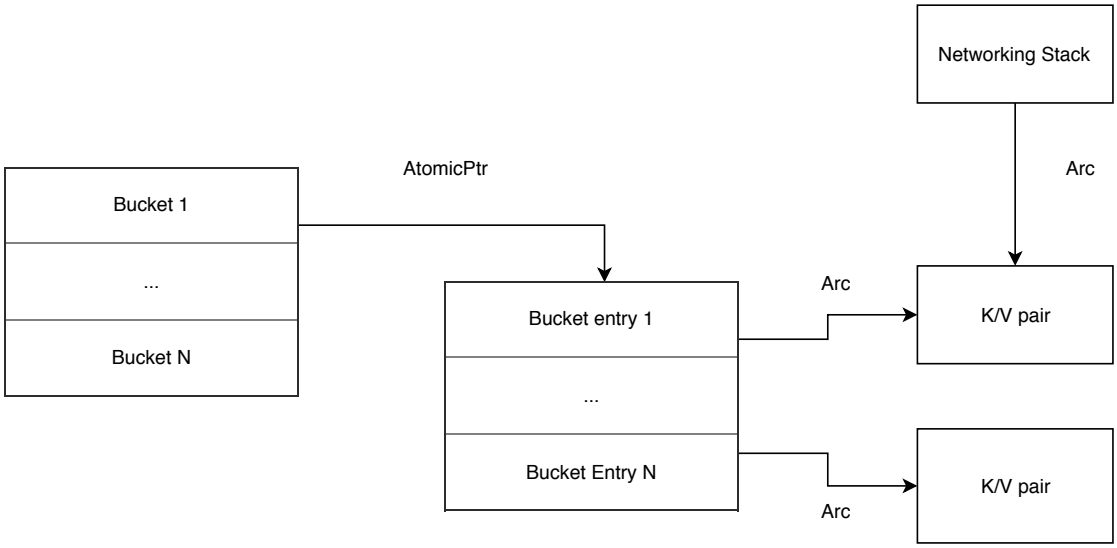


Figure 3.6: Concurrent value removal

provides multi-packet requests, the R2P2 should handle reconstruction of requests. Since the user application will only care about handling complete requests the R2P2 stack will store packets and group by them by request and finally invoke the user provided callback once the whole request has been received and parsed. This callback should construct the response from the request. The R2P2 stack then handles transmitting the response to the client using the UDP stack.

The R2P2 layer is also responsible for the reliable delivery of responses. It should then hold on to responses produced by the user callback until they are acknowledged by the clients. In order to avoid duplicating work, and thereby reducing both latency and throughput, it should also identify when a request has already been processed by the user application and avoid processing this request again.

3.5 UDP stack

The core of this project is not the low level details of interfacing with a NIC or even the kernel bypass. We will thus use an existing framework to do that for us and build our networking on top of said framework.

It is worth mentioning the Linux kernel socket flag for zero copy. Though it does not provide guaranteed zero copy sending/receiving a port of nginx using it saw a 9% increase in performance with minimal code changes [12]. The choice was made not to use it, since it would eliminate one of the most interesting aspect of the project, the ability to have a safe userspace networking stack in the same address space as the user program.

As for the local store, the UDP stack should uphold the zero-copy property of the system. That means it should interface with the kernel-bypassing framework and extract data from that framework while not copying anything.

Kernel bypassing frameworks follow what is known as a run to completion model, the

driver polls the NIC for incoming packets [10] and hands them over to the user application in batches. This will influence the API design of the UDP stack and forbid the classic socket style API typically found in kernel networking stacks. Since we aim to provide a convenient and safe Rust API to the UDP stack we will need to come up with an alternative design.
The run to completion model thus prompts a callback based API design. We thus have a callback oriented UDP stack. Since the framework polls the NIC for packets on its own this is appropriate. When the user application creates a new socket on the stack, it will provide a closure. Once a packet is received, its headers will be parsed and the UDP stack will dispatch to the appropriate socket registered earlier by the user application. Of course the callback provided by the user application should be reentrant since it will potentially be invoked from many threads concurrently.

4 Implementation

We now discuss how our abstract design translates into a concrete Rust implementation, including how it ties into the Rust standard library and precisely how each component interfaces with the others.

4.1 Overview

Each packet is first received by DPDK from the NIC. The packets are then processed through the NetBricks pipeline to be handed over to the UDP stack. The UDP stack then filters and matches packets to a given socket. Each socket has an associated callback registered by the user. In our case, the socket are created by the R2P2 server which uses them to receive R2P2 requests from the network. Once every packet in a request has been received, the R2P2 server then invokes its own user callback, this time called the Request Callback. This callback is registered by the last layer in the system, the actual key-value store. The key-value store uses R2P2 to satisfy requests in the Redis protocol. This is also where our hashmap comes in (though it is used all across the system), to store the key value pairs.

4.2 OptiMap

OptiMap is the name for the highly concurrent hashmap, described in section 3.2, and we now will see how exactly the design translates into a concrete Rust implementation.

4.2.1 AtomicBox

As stated in section 3.2, we need some way to atomically swap values while also atomically reference counting them. Unfortunately the rust standard library only provides a mean to atomically swap raw pointers using the **AtomicPtr** construct. This effectively forces us to make use of raw pointers, thus not benefiting from any of Rust's memory safety guarantee and potentially leading to memory corruption or leaks. Even though we are forced to use unsafe, Rust still provides strong safety guarantees and debugging time is still very much reduced since we know for sure that memory corruption problems are located inside **unsafe** code blocks.

Listing 4.1: The AtomicBox structure

```
pub struct AtomicBox<T: Sized>
{
    ptr: AtomicPtr<T>,
}
```

This leads us to the **AtomicBox** abstraction, a safe wrapper around **AtomicPtr**, that we will use to build our OMVCC hashmap. Creating safe wrapper around unsafe code is a very common practice in Rust, it allows the potential memory problems to be confined to a particular datastructure while users of the wrapper don't have to worry about the unsafety of the underlying code.
The AtomicBox makes use of X86 CAS instructions to improve on Arc. Arc (**A**tomically **R**eference **C**ounted smart pointers) provides an atomic reference count and AtomicBox provides an atomic reference count as well the possibility to atomically swap that value. Since every value is atomically reference counted it will stay allocated as long as any reference to it still exists while allowing new requests to fetch the newer value to do their work. Updates are made using the value atomically fetched at that point in time, creating a copy, modifying it as appropriate and then swapping it back with the old one. If the swap succeeds the old value's reference count is decreased (and dropped if we had the last reference), effectively providing an optimistic multi-version concurrency control. If the swap fails, i.e. someone already swapped it with another value, we repeat the same process until the swap is successful.
To build the AtomicBox abstraction we make use of the capability of Arc to be transformed into raw pointers, thus allowing a user to control exactly how long the memory on the heap lives, including the value outliving the AtomicBox instance. The reference can then be transformed back into an Arc, although this requires using **unsafe** since we turn arbitrary pointers into Arc instances therefore introducing a risk of double-free or invalid memory accesses.

Listing 4.2: AtomicBox public interface

```
pub fn replace_with<F>(&self, f: F) where F: Fn(Arc<T>) -> T;

pub fn get(&self) -> Arc<T>;
```

The AtomicBox abstraction allows us to implement the optimistic multi-version concurrency control scheme. Indeed, the AtomicBox allows users to extract the value it contains and use independently from the AtomicBox. We can thus extract values from the AtomicBox and keep them even when they are removed from the AtomicPtr as shown in figure 4.2, allowing for an a multi-version concurrency control scheme.
The public interface provided by AtomicBox aims to be simple, fast and safe, hiding the unsafe details of the actual operation. We therefore provide an idiomatic way to update the value as well as a clean way to fetch it. Updates are handled through closures, the user passes a closure to the AtomicBox which will apply this closure to the current value and then atomically replace the old value with the new one. In order to allow the value contained in the AtomicBox to outlive it, when getting the value we actually provide

an Arc<T> to the user making the allocation of the value itself independent from the allocation of the AtomicBox.
Since they are to be shared amongst threads, all the hashmap components are allocated on the heap, thus justifying our use of Arcs. As all elements are of small size (most of them are one or two pointers big), we make the claim that memory fragmentation is not gonna be an issue in our case. Indeed most of the allocation with respect to size will be coming directly from DPDK allocating memory buffers from Linux's **hugepages** and thus won't affect the heap.

4.2.2 HashMap

We now use the AtomicBox abstraction described in section 4.2.1 to build the concurrent hashmap.
We will use the AtomicBox abstraction to store the buckets of the hashmap. So the top level table of the hashmap is a contiguous array of AtomicBoxes that stores arrays of key-value entries.

Listing 4.3: AtomicBucket

```
struct AtomicTable<K, V, S>
where
    K: PartialEq + Hash,
    S: BuildHasher,
{
    buckets: Vec<AtomicBucket<K, V>>,
    hash_builder: S,
}

struct AtomicBucket<K, V> {
    value: Arc<AtomicBox<Vec<Arc<BucketEntry<K, V>>>>>,
}
```

The **BucketEntry** structure is then a plain structure that just contains the key and the value, also shown in figure 4.3. It also provides convenient access to both the key and the value to simplify handling the two as one coherent entity.
OptiMap also handles values escaping from its scope. We therefore provide an abstraction named **ValueHolder** that allows values to outlive the hashmap, as shown in figure 4.4. This ValueHolder structure also allows convenient access to the value it contains through Rust primitives.

Listing 4.4: ValueHolder construct

```
/// A struct used to hold the bucket for as long as the value is needed
pub struct ValueHolder<K, V> {
    bucket: Arc<BucketEntry<K, V>>
}

impl<K, V> Deref for ValueHolder<K, V> {
    type Target = V;
```

```
    fn deref(&self) -> &Self::Target {
        self.bucket.value_ref()
    }
}
```

ValueHolder makes use of the Rust trait **Deref**. The Deref trait is used in Rust to make values behave as if they were references. So from the user's perspective the ValueHolder actually is just a reference to the value that can be used independently from the OptiMap.

4.3 Key value store

The key-value store uses a simplified version of the Redis protocol. For the sake of simplicity it only supports SET and GET requests. Its two most important task will be to interface with the local store in a zero copy manner, which proved to be a non-trivial task, and to handle incoming Redis request.

The requirements for the actual hashmap value and key datastructures are mostly to play nice with the Rust memory management model and manage the lifetime of DPDK mbufs. Indeed since we use segmented packets, we need the headers to be freed once the request has been satisfied but the payload itself must be kept as long as either the hashmap or the networking stack hold a reference to it. As the payload can be sent concurrently from different threads we need some sort of atomic reference counting on the mbufs, fortunately DPDK already provides such a thing, we will see how we tie it to the Rust type system in section 4.6.

Listing 4.5: KeyEntry

```
struct RedisKeyEntry {
    pkt: Vec<CrossPacket>,
    offset: usize,
    len: usize,
}

impl Deref for RedisKEntry {
    type Target = [u8];

    fn deref(&self) -> &Self::Target {
        unsafe {
            slice::from_raw_parts(self.pkt.get_payload(self.offset),
                                  self.len)
        }
    }
}
```

The role of the key-value store is therefore to transform the keys and values contained inside packet payloads in a way that allows them to be inserted and retrieved from the store while avoiding copies. To do this the key-value store wraps the payload in a KeyEntry or ValueEntry) structure that allows the payload to be used as if it was a plain Rust value. We use again the Deref trait as well as an offset to offer easy access to the key

(or value) contained inside the payload. Both the KeyEntry and the ValueEntry share the same reference to the payload, thereby preventing any modification of said payload. This design also makes the KeyEntry usable as an array of unsigned 8 bits integers, making hashing and comparisons easy (which is always helpful in a hashmap).

4.4 R2P2 server

We use a slightly modified version of the R2P2 protocol that adds active acknowledgement from clients in order to reduce latency when packets are dropped. For the sake of simplicity we also do not handle packets that are delivered out of order, mostly to avoid resizing buffers used to store the request while it is not completely received.
The R2P2 stack is built on top of the aforementioned UDP stack. The R2P2 server registers a packet callback and then dispatches the packet to the correct request after parsing the headers. Once a request has been received entirely, the R2P2 server uses callbacks registered by the user application to handle requests. This callback returns a R2P2Response structure that is then sent by the R2P2 server through the UDP stack.
The R2P2 server handles reliable delivery through client acknowledgement. The response returned by the user callback is stored in the R2P2 server while waiting for the acknowledgement and retransmitted if need be. This avoids having to duplicate work in case some packets get lost when transmitting the response.

4.5 UDP stack

The UDP stack ties in to NetBricks by registering a pipeline on all available cores and using each pipeline to dispatch packets to the correct socket. Each socket is stored in a hashmap in the stack by its port. Upon receiving a packet the stack filters in multiple steps the packets dropping invalid ones and routing the others to the correct socket. The corresponding callback is then called on the packets one by one.

Listing 4.6: UDP stack public interface

```
pub fn bind<F>(&self,
               addr: Ipv4Addr,
               port: u16,
               read_cb: F) -> Arc<UdpSocket>
where F: Fn(CrossPacket, Ipv4Addr, u16) + 'static;

pub fn send(&self, pkt: &CrossPacket,
            addr: &Ipv4Addr,
            port: u16);
```

The bind method shown in figure 4.6 allows users to bind socket on a given UdpStack. The user provides the IP address to listen on as well as the UDP port. In order to receive incoming packets the user also provides a closure that will be called every time a packet is read from the NIC by the UdpStack. The stack will actually transfer ownership of the packets received to the callback, thereby permitting the zero copy networking stack we are aiming for.

The send method on the other hand does not need to transfer ownership of the packets to the UDP stack, because it does not need to keep the packets after sending them. Therefore it only borrows the Packet the user wants to send. After which it creates a segmented DPDK packet to allow sending the same packet concurrently to different destinations. To avoid flooding the same TX ring, send() uses a simple round-robin approach to select the output queue.
As mentioned in section 3.5, the user provided callback should be re-entrant to ensure integrity. We once again leverage Rust features to ensure that users won't be able to provide a non re-entrant packet handling callback. Indeed, Rust has three types of closure: `Fn`, `FnMut` and `FnOnce`.
`Fn` is the type of closure that do not modify the variables they capture. Therefore they can be safely shared between threads since they won't modify any variables outside of their scope. `FnMut` on the other hand are closures that modify their environment therefore not being thread-safe, these closures are prevented by the Rust compiler from crossing thread boundaries. And lastly `FnOnce` are even more restrictive. They are the type of closures that consume their environment, i.e. they transfer ownership of their captured values to other functions. That means an `FnOnce` closure can only be run once (as the name implies) since the Rust compiler will prevent them from using values they do not own anymore.

4.6 NetBricks

```
ReceiveBatch::new(queue.clone())
    .parse::<MacHeader>()
    .parse::<IpHeader>()
    .metadata(box move |pkt|{
        // keep the source ip around for later replies
        pkt.get_header().src()
    })
    .parse::<UdpHeader>()
    .map(box move |pkt| {
        let src_addr = Ipv4Addr::from(*pkt.read_metadata());
        let src_port = pkt.get_header().dst_port();

        stack.sockets.get(&src_port).map(|sock| {
            sock.deliver(&pkt, src_addr, src_port)
        });
    })
    .compose()
```

Figure 4.1: UDP packet pipeline

In order to adapt NetBricks to end-host networking, the fork comes with a few modifications in packet management. The first issue we need to address is the inability of packets to cross thread boundaries. Indeed since NetBricks is geared towards network functions the packets are only meant to be processed by the user defined pipeline and be freed as soon as they have gone through the pipeline. To remedy this we create the

CrossPacket abstraction which can be created from the standard NetBricks Packet and is able to cross thread boundaries by being immutable. But we still need to be able to send packets through NetBricks and NetBricks does not know how to handle CrossPackets. It is therefore necessary to be able to convert a CrossPacket to a Packet as well as the reverse.

Listing 4.7: CrossPacket abstraction

```
pub struct CrossPacket {
    payload: *const MBuf,
}
```

The CrossPacket abstraction is how we tie together the details of DPDK memory management and the Rust type system. NetBricks' **Packet**s are, by default, mutable meaning the Rust compiler will not allow them to cross thread boundaries in order to prevent data races. However since the pointer to the mbuf is immutable this poses a problem when sending the packet. Indeed DPDK stores headers in the same buffer as the payload, meaning we can't easily send the same payload concurrently to two different destination. The way to solve this is make use of DPDK's mbuf chaining, the first mbuf in the chain being local to the current thread (therefore being mutable) and the second one being the immutable payload shared between threads. This is the second notable modification we make to our NetBricks fork, the ability to chain mbufs to maintain the immutability of the packets we received from the network while also having the possibility to send them concurrently to different destinations.
One more thing that needs to be implemented in NetBricks is packets living longer than the packet processing pipeline. Since NetBricks is aimed at network function once a packet has gone through the packet processing pipeline it has no reason to stay allocated (as in the case of a firewall, when the decision has been made to forward the packet it's not needed anymore). But in our case packets are longer lived, we need to keep them in the store to answer queries later on. We then need a way to prevent deallocation of packets once they leave the pipeline. We make use of mbuf reference counts to do this. In our segmented packets the headers always have a reference count of one, indeed they are not needed once the packet has been sent successfully, thus we let DPDK free them once the packet have gone through the NIC. The payload on the other hand should not be deallocated after sending, so we set its reference count to at least 2, thereby preventing DPDK from freeing it. But we still need to free them once they are not needed anymore. The CrossPacket abstraction provides a convenient wrapper around mbufs that make the link between the static Rust analysis and the manual memory management required by C. This is done through the use two other standard Rust traits, **Drop** and **Clone**. Both of these are closely related to how Rust manages memory. The Drop trait indicate that an object needs to perform specific steps when being deallocated. In our case it is decreasing the reference count of the wrapped mbuf and freeing it if the reference count reaches 0. In essence this greatly simplifies the memory management since Rust automatically calls the Drop trait of an object when that object is about to be destroyed.

Listing 4.8: Drop and Clone trait

```
impl Clone for CrossPacket {
    fn clone(&self) -> Self {
        unsafe {
            mbuf_ref(self.payload as *mut MBuf);
        }

        CrossPacket::new(self.payload)
    }
}

impl Drop for CrossPacket {
    fn drop(&mut self) {
        unsafe {
            mbuf_free(self.payload as *mut MBuf);
        }
    }
}
```

The Clone trait is the reverse of the Drop trait, it allows to create copies of objects that can't be bitwise copied. In our case the Drop trait is critical in providing zero copy semantics for CrossPacket. Indeed we can implement the Drop trait by copying the wrapped mbuf pointer and increasing its reference count, thereby creating and easy to use packet duplication mechanism without actually copying their payloads.

5 Evaluation

We will now evaluate the performance of the system as well as the performance of the local store.

The server benchmarks are run on an Intel Xeon E5-2637 @ 3.50 Ghz machine with 64 GB of RAM, running Ubuntu. The machines have Intel 82599ES 10-Gigabit NICs. Since we make heavy use of experimental Rust APIs, we use the Rust nightly 1.28. NetBricks also requires Rust nightly and is built on top of DPDK 17.05.

5.1 Local store

We first evaluate how the local hashmap performance, especially when under a lot of load. We use UTF-8 string keys of 128 bytes with values of 64 bits to simulate only copying pointers to packets from the network. We generate a random workload for each of the 15 concurrent thread and start them simultaneously. To avoid scheduling overhead each thread is bound to a physical through the Linux core affinity API. Each workload is made of one million GET and 1 million PUT request, on a previously initialized map.

5.1.1 No collisions, balanced load

We first establish the optimal performance we can expect from the hashmap by reducing the contention on the buckets, thus reducing the amount of wasted work by threads trying to swap out the buckets unsuccessfully. To this end, we allocate four times more buckets than they are keys and using a uniform distribution when generating workload. The set of keys is pre-determined and no new keys are inserted throughout the test, meaning the size of each bucket will stay constant.

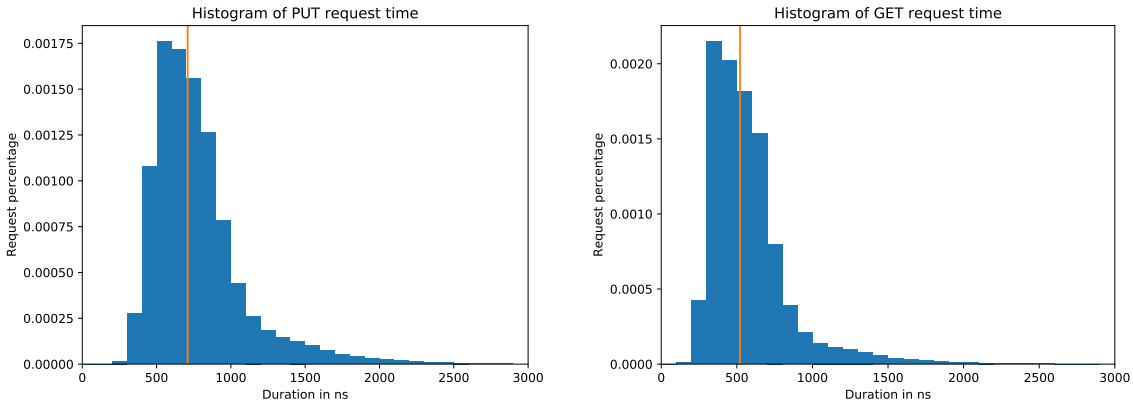


Figure 5.1: Distribution of request durations

The orange line is the median value of request durations. The median value for GET request time is 520ns and it is 700ns for PUT request in the case where we minimize collisions. Considering that there is a non-compressible 4.5 microseconds time to perform PCI transactions when sending packets the key-value store won't be the major latency inducing part of the system in the case where collisions are low.

5.1.2 Artificial collision, balanced load

OMVCC scheme are known for performing badly under write heavy workloads. So we now consider the case of a skewed and write heavy workload and evaluate the performance of the hashmap in this scenario. In order to simulate a skewed workload, we reduce the number of buckets in the hashmap. This will create lots of contention between threads for each buckets since the chance of two keys mapping to the same bucket is increased by a factor of 4. The test setup is, again, the same as described in section 5.1.

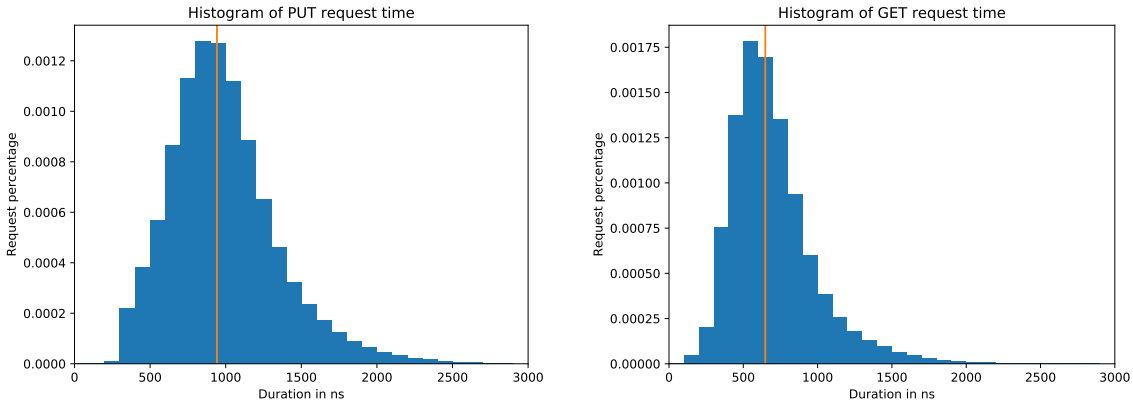


Figure 5.2: Distribution of request durations with collision

The goal of this test setup is to evaluate how the key-value store will perform in adverse conditions. As we can see the median has increased by around 200ns on PUT requests,

which is to be expected since there will be more attempts where work will need to be done more than once because of the contention on buckets.
We also notice a slight increase in GET times. This is most likely due to the atomic instructions used to fetch buckets that can cause stalling due to the necessity of synchronizing memory accesses across cores. However even when introducing high collision rate the hashmap still has reasonable performance.

5.2 Networked store

Though we hoped to have the performance evaluation of the system as a whole ready, a few unforeseen problems surfaced at the last minute. These problems (related to low-level DPDK implementation details) prevented us from providing a comprehensive performance evaluation of the whole system. Nonetheless we will try to have one ready for the defense.

6 Further work

We finish by mentioning what features are missing or could be useful in such a piece of software. We have provided a UDP stack on top of NetBricks, even though in real-life scenario of networking applications UDP is rarely used. A TCP stack on top of NetBricks would therefore be a considerable improvement.
As it stands destination MAC addresses have to be provided by the user application when sending packets. This is less than ideal since it means the user application still has to handle low level networking details. In our case we respond to a request, so we have both the source and destination MAC addresses at hand, but in some other potential applications (such as a sending packets to an arbitrary destination that has not sent a request) this could be a problem.
With an ARP and TCP stack we would be close to a full featured networking framework in Rust for kernel bypass. Such a system would provide both the speed and the convenience of traditional socket based networking while also providing better networking performance and the safety of the Rust programming language. A complete networking framework for kernel bypassing in Rust, would provide an easy way of to create safe, high throughput and low latency networking applications in Rust.

7 Conclusion

We hope to have provided good foundation on which to build further kernel bypass networking applications, as well as having provided enough evidence in favor of Rust for kernel bypass networking applications. Even though there is no comprehensive performance evaluation available at this time, each part of the system benchmarked as a single entity provided good performance. Most components are generic enough to be re-used in further work on the subject.

Bibliography

[1] The dataplane development kit. https://www.dpdk.org/.

[2] *Dynamo: Amazon's highly available Key-value Store.*

[3] grpc, a high performance rpc framework. https://grpc.io/.

[4] The quic transport protocol. https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/46403.pdf.

[5] Redis. https://redis.io/documentation.

[6] Voldemort, a distributed database. https://www.project-voldemort.com/voldemort/.

[7] Reliable udp protocol internet-draft. https://www.ietf.org/proceedings/44/I-D/draft-ietf-sigtran-reliable-udp-00.txt, 1999.

[8] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. *Workload analysis of a Large-Scale Key-Value store.* 2012.

[9] Speed matters. https://ai.googleblog.com/2009/06/speed-matters.html.

[10] The DPDK authors. *DPDK, Poll Mode Driver.*

[11] Adam Belay, George Prekas, Ana Klimovic, Samul Grossman, Christos Kozyrakis, and Edouard Bugnion. *IX: A Protected Dataplane Operating System for High Throughput and Low Latency.* 2014.

[12] Willem de Bruij. *sendmsg copy avoidance with MSG_ZEROCOP.*

[13] The Rust developers. The rustonomicon. https://doc.rust-lang.org/nomicon/.

[14] Muhammad Asim Jamshed, YoungGyoun Moon, Donghwi Kim, Dongsu Han, and KyoungSoo Park. mos: A reusable networking stack for flow monitoring middleboxes. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 113–129, Boston, MA, 2017. USENIX Association.

[15] Steve Klabnik and Carol Nichols. *The Rust book, 2nd edition.* 2017.

[16] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. *ReFlex, Remote Flash Local Flash*. 2017.

[17] Amit Levy, Michael P Andersen, Bradford Campbell, David Culler, Prabal Dutta, Branden Ghena, Philip Levis, and Pat Pannuto. *Ownership is Theft: Experiences Building an Embedded OS in Rust*.

[18] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. *MICA: A Holistic Approach to Fast In-Memory Key-Value Storage*. 2014.

[19] Maziar Manesh, Katerina Argyraki, Mihai Dobrescu, Norbert Egi, Kevin Fall, Gianluca Iannaccone, Eddie Kohler, and Sylvia Ratnasamy. *Evaluating the Suitability of Server Network Cards for Software Routers*.

[20] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, 2013. USENIX.

[21] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. *NetBricks: Taking the V out of NFV*. 2015.

[22] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, Broomfield, CO, 2014. USENIX Association.

[23] The Redox Project. Redox book. `https://doc.redox-os.org/book/`.

[24] Luigi Rizzo. netmap: A novel framework for fast packet i/o. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 101–112, Boston, MA, 2012. USENIX Association.

[25] Apache thrift. `https://thrift.apache.org/`.

[26] Amy Williams, William Thies, and Michael D. Ernst. *Static deadlock detection for Java libraries*.