Inter-Process Communication in a Safe Kernel

Dan Appel
University of California, Irvine

September 1, 2020

Abstract

Traditional operating systems follow a monolithic design, executing all kernel subsystems in a shared address space, thereby achieving good performance at the cost of isolation and security. Microkernels have improved on this design by separating operating system components into individual address spaces, but historically have been prohibitively expensive due to the high cost of switching address spaces. RedLeaf is a new operating system which relies on the safety of the Rust programming language as opposed to hardware mechanisms for isolation. RedLeaf runs all operating system kernel subsystems in the same hardware address space, and instead achieves isolation through the combination of language safety and special communication primitives. The result is that communication overhead is on the order of dozens of cycles, comparable to a regular function call. However, even in a safe kernel, cross-domain communication requires careful design choices to maintain isolation in the case that domains crash. This thesis describes these design choices, and introduces the consequent concepts of the *shared heap* and *remote references* (*RRef*), which build on Rust's safety model to provide zero-copy communication that ensures safety and isolation even in the face of crashing subsystems.

1 Introduction

Background Most modern operating systems are implemented as monolithic kernels. Monolithic kernels are rich in features and include device drivers, with user-level programs living in individual virtual address spaces. Although they have good performance, monolithic kernels have poor fault tolerance, with a single panicking device driver having the potential to bring down the entire system. Even worse, a vulnerability in any part of a monolithic kernel can result in a full take over of the machine. Microkernels are a response to monolithic kernels with better security and fault isolation, achieved by shifting as much of the operating system functionality out of the kernel as possible. This modularity greatly increases fault tolerance and security, but comes at the cost of expensive address space crossings of isolated subsystems. Modern microkernels such as the L4 family have improved the inter-process communication (IPC) performance, and are widely adopted in production, but still suffer a significant overhead. The state of the art seL4 microkernel introduces a minimum overhead of 1260 cycles on a 3.4GHz x86 CPU [1], which is still prohibitive for modern I/O intensive workloads that require millions of domain switches per second. With the recent emergence of low-cost memory-safe programming languages there is now room to improve performance of domain switches by relying on memory safety guarantees instead of hardware isolation mechanisms.

Rust Rust is a new systems programming language with a focus on compile-time memory safety achieved through a sub-structural type system [**?**]. This means that regular Rust pointers have a statically defined lifetime, which prevents a whole suite of memory bugs such as use-after-free, dangling pointers, and double-free. Further, in *safe* Rust (a strict subset of the language) pointer casts and raw-pointer dereferencing are prohibited at compile time. By enforcing safe Rust guarantees, it is possible to statically ensure that a bad actor's program will not manipulate external memory, eliminating the need for virtual address spaces.

Redleaf RedLeaf [**?**] is a novel microkernel written from scratch in Rust. RedLeaf relies on Rust's memory safety instead of hardware mechanisms for memory isolation. Programs are organized into domains, which share a single address space but are otherwise isolated. This design eliminates the expensive address space crossings plaguing microkernels, achieving incredible cross-domain communication performance than without sacrificing safety or isolation.

Isolation While Rust enforces memory safety on a per-domain basis, it is not enough to ensure total isolation during cross-domain communication. When a domain panics, the kernel unloads it from memory to free up resources. If this happens in the middle of a cross-domain call, Rust's memory safety guarantees are violated, as any external pointers into the dead domain are now dangling. In order to achieve total domain isolation, we introduce the following invariants: domains do not share heaps, and domains do not expose pointers from their respective heaps. With these guarantees, a domain can be safely unloaded from memory without any external dangling pointers. For cross-domain communication, domains are to exchange pointers from a single shared heap. To enforce this, we introduce the mechanism of *remote reference*s (*RRef*s), references to shared memory. *RRef*s enable domains to create and exchange shared heap objects in a zero-copy manner, while preserving safety in the face of domain crashes.

This thesis describes the implementation of remote references and the shared heap, the mechanisms behind RedLeaf's zero-copy and fault-tolerant IPC.

2 RedLeaf Architecture

Domains The RedLeaf microkernel handles a narrow set of tasks and delegates the rest to user-level *domains* (Figure 1). The microkernel performs scheduling and domain loading, and provides an interface for allocating shared memory and handling interrupts. Device drivers, the file system, and all other programs are implemented as user-level domains running atop the kernel.

Virtual Memory Safe Rust enforces several aspects of memory safety that are relevant to RedLeaf. Notably, raw pointers, roughly equivalent to regular C pointers, cannot be dereferenced or converted into safe pointers. Consequently, safe Rust programs can only access memory pointing to the result of an allocation or a system call — they cannot make pointers from scratch. RedLeaf enforces safe Rust in untrusted domains and isolates each domain's heap, which guarantees that a domain will not access external memory. This alleviates the need for hardware virtual address spaces for isolation, and so all domains can run in the same address space.

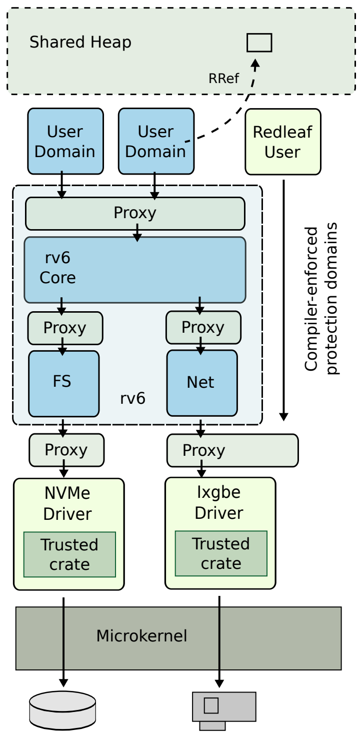


Figure 1: RedLeaf architecture [?]

Inter-Process Communication We define inter-process communication (IPC) to be communication between any two domains. As opposed to a message-passing architecture (typical for microkernels) or privileged system calls (typical for monolithic kernels), the RedLeaf IPC interface is exposed as a set of capabilities [?] implemented as Rust trait objects. Because all domains live in the same address space, communication can be zero-copy. Arguments in IPC calls can either be small values on the stack, passed by value, or larger objects which are passed by reference. Without careful design, passing objects by reference can be an attack vector.

Attack Model RedLeaf relies on the safety of Rust for enforcing isolation across domains. IPC can invalidate Rust safety, breaking isolation which allows for attacks. With the invariant that each domain has its own heap, sharing objects by reference allows for external pointers into a domain's heap. If a domain is unloaded from memory while another domain references its heap, that reference becomes a dangling pointer, breaking Rust's memory safety model. This can be weaponized: a domain can purposefully panic in the middle of an IPC call to bring down another domain with it, breaking isolation. Rather than restricting IPC, our solution revolves around a *shared heap*, which ensures that memory shared by domains is always valid.

Shared Heap The shared heap is a special region of memory which keeps track of ownership. When a domain requests memory from the shared heap, it marks itself as the owner of the pointer. When the domain shares this pointer with another domain, the pointer's owner changes, which is recorded by the shared heap. This way, each domain is totally isolated, and the RedLeaf kernel can safely unload a domain and free any shared heap memory owned by the domain.

Remote References The shared heap provides memory that is safe to use for IPC. In order to enforce proper use of the shared heap, we introduce remote references (*RRef*s). *RRef*s allocate memory on the shared heap and keep track of domain ownership. *RRef*s are statically guaranteed to point to valid memory on the shared heap, enforcing the memory safety guarantees of Rust. This allows domains to safely communicate via IPC with *RRef*'s as arguments, even in the case that one of the domains panics.

Proxies In order to handle faults and track *RRef* domain ownership, we introduce proxies. Proxies are transparent domains that interpose communication between each pair of domains. For each cross-domain call, proxies record *RRef* ownership and call-stack information. In the case that a domain panics in the middle of an IPC call, execution winds back to the proxy using the call-stack information, isolating the fault. The proxy is generated by the RedLeaf IDL [?], and introduces a small overhead to ensure total domain isolation.

3 Implementation

Remote references (*RRef*s) are implemented in a trusted crate, exposing the types RRef<T>, RRefArray<T>, and RRefDeque<T>. Further data structures can be built on top of these types in the future. This section covers the implementation details of the *RRef* crate, along with the shared heap component powering it.

3.1 Shared Heap

RRef's are special pointers to the shared heap. They reference shared memory safe to use for IPC, and update ownership of the memory based on what domain they are in. The shared heap exposes methods to allocate and deallocate *raw* pointers to memory. Dereferencing raw pointers requires unsafe code [**?**], limiting their use to trusted RedLeaf components. The *RRef* crate is one of these trusted components and is built on top of the shared heap capability.

Listing 1 shows the interface for the shared heap capability. The alloc(layout:drop_fn:) method returns three pointers for managing state of the shared heap memory. This state is abstracted by the RRef<T> type, which in turn is managed by the proxy domain during domain crossings.

The shared heap maintains a registry containing information about every allocation, along with its destructor method. Records are inserted or deleted from the registry upon allocation or deallocation, and serve as a source of truth for the kernel for cleaning up shared memory when a domain panics.

```rust
pub struct SharedHeapAllocation {
    pub domain_id_pointer: *mut u64,
    pub borrow_count_pointer: *mut u64,
    pub value_pointer: *mut u8
}

pub trait Heap {
    unsafe fn alloc(
        &self,
        layout: Layout, // size and alignment
        drop_fn: extern fn(*mut u8) -> () // kernel-invoked cleanup method
    ) -> SharedHeapAllocation;
    unsafe fn dealloc(&self, value_pointer: *mut u8);
}
```

Listing 1: Shared Heap capability interface.

3.2 RRef

3.2.1 Rust pointer types

```rust
// Manipulating a regular heap pointer
let mut box_ptr: Box<u64> = Box::new(10u64);
*box_ptr = 50;
// Manipulating a remote reference
let mut rref_ptr: RRef<u64> = RRef::new(10u64);
*rref_ptr = 50;
```

Listing 3: Working with *RRef*.

The RRef<T> pointer type was designed to be as Rust-native as possible, and is modeled after existing types in the standard library. Unlike C, pointer types in Rust are distinct at the type level. In C, a pointer can reference the heap, the stack, a memory mapped value, or nothing at all (null). In Rust, each of those cases are represented by different types.

```
pub trait BDev: Send + Sync {
    fn read(&self, block: u32, data: &mut [u8; BSIZE]) -> RpcResult<()>;
    fn write(&self, block: u32, data: &[u8; BSIZE]) -> RpcResult<()>;
}
```

(a) Interface with regular references.

```
pub trait BDev: Send + Sync {
    fn read(&self, block: u32, data: RRef<[u8; BSIZE]>) -> RpcResult<RRef<[u8; BSIZE]>>;
    fn write(&self, block: u32, data: &RRef<[u8; BSIZE]>) -> RpcResult<()>;
}
```

(b) Interface with remote references.

Listing 2: Comparison of block device driver interface with and without *RRef*.

Rust's most primitive pointer type is `Box<T>`, which points to a value on the heap. `Box::new(10u64)` creates a pointer to a 64-bit unsigned integer on the heap. More complex pointer types are built on top of `Box<T>`. With this in mind, the base `RRef<T>` type is modeled after `Box<T>`. `RRef::new(10u64)` is semantically very similar to `Box::new(10u64)`. However, rather than allocating on the domain's heap, `RRef::new(10u64)` allocates on the *shared* heap. Just like for `Box<T>`, more complex types like `RRefArray<T>` and `RRefDeque<T>` are built on top of this base `RRef<T>` type.

Working with a `RRef<T>` is very similar to working with standard library pointers. Compare the block device driver interfaces shown in Listing 2a (regular Rust references) and Listing 2b (remote references). The `data` argument, passed by reference, is wrapped in a `RRef<T>` to enforce use of the shared heap. The mutable borrow (`&mut`) is converted to move-in-move-out semantics, for reasons covered in Section 3.2.6. On the call site, the syntax for dereferencing and manipulating `RRef<T>` is essentially identical to the `Box<T>` type (Listing 3).

3.2.2 Structure

```
pub struct RRef<T> where T: 'static + RRefable {
    value_pointer: *mut T,
    domain_id_pointer: *mut u64,
    borrow_count_pointer: *mut u64
}
```

Listing 4: `RRef` struct definition

An *RRef* is a combination of three pointers to values on the shared heap. The syntax used for these pointers, `*mut`, is a *raw pointer* [**?**]. Raw pointers are part of the unsafe subset of Rust, and offer very few memory guarantees. With `RRef<T>`, these pointers always come directly from the trusted shared heap, so we can assume them to be valid.

The `value_pointer` contains the actual object. The type `T` is generic and constrained to `RRefable` (more in Section 3.2.5), which means the actual type, layout, and size is resolved at compile time. `RRef::new([1usize,2,3])` will request memory to store an array of three integers, and copy the array over to the shared heap (this is the only copy in *RRef*s life cycle).

The `domain_id_pointer` keeps track of the current owner of the domain. *RRef*s have a single owner, and, except in the case of read-only borrows (Section 3.2.6) are deallocated automatically if their current owner dies. When a *RRef* is passed from one domain to another, its owner gets changed in the trusted proxy guarding each domain.

Finally, the `borrow_count_pointer` counts how many domains are borrowing this object. This information is used to allow immutable borrows without leaking memory; Section 3.2.6 goes into more detail.

3.2.3 Initialization

By far the most intricate and unsafe portion of the *RRef* crate is the RRef::new initialization method. RRef::new requests memory from the shared heap and performs a bytewise copy of the RRefable object (more on this in Section 3.2.5). This method is crucial to get right because it performs raw memory manipulation which can undermine Rust's memory guarantees.

```
 1 impl<T: RRefable> RRef<T> {
 2     fn new(value: T) -> RRef<T> {
 3         // compile-time size and alignment info
 4         let layout = Layout::new::<T>();
 5         // clean up function in case owner domain panics
 6         let drop_fn = unsafe {
 7             transmute::<extern fn(*mut u8) -> ()>(drop_t::<T>)
 8         };
 9         // request memory (as seen in Listing 1)
10         let allocation = unsafe {
11             HEAP.alloc(layout, drop_fn)
12         };
13         // reinterpret these bytes as T
14         let value_pointer = allocation.value_pointer as *mut T;
15         unsafe {
16             // initialize ownership and value
17             *allocation.domain_id_pointer = CRATE_DOMAIN_ID;
18             *allocation.borrow_count_pointer = 0;
19             core::ptr::write(value_pointer, value);
20         }
21         RRef {
22             domain_id_pointer,
23             borrow_count_pointer,
24             value_pointer
25         }
26     }
27 }
```

Listing 5: RRef::new initialization method

Listing 5 shows the full initialization code. On line 11, we request memory from the shared heap and get three *raw* pointers. RedLeaf control the initialization of the HEAP global for each domain, so we consider it trusted. Therefore, we can assume that any pointers it returns are going to be valid and have the expected layout. Rust is able to compute the layout (size and alignment) of a generic type at compile time, seen on line 4. <T: RRefable> ensures that the generic type contains no references, so it is safe to simply allocate bytes and cast them to T (line 14).

The remainder of the RRef<T> implementation wraps operations on the three raw pointers to simplify working with shared heap memory.

3.2.4 Dereferencing

Rust has builtin traits for dereferencing, Deref [?] and DerefMut [?]. These traits enable several levels of syntactic sugar which are particularly useful for smart pointers. First, they allow for the asterisk dereferencing operator (*ptr), which extracts a mutable or immutable reference from the smart pointer. Second, the compiler implicitly implements the methods of type T for any type which implements Deref<Target=T>. This means that, for example, RRef::new(10u64).checked_add(1) is valid code which automatically dereferences RRef<u64> to &u64 and invokes its respective checked_add method.

Listing 6 shows the implementation of the DerefMut traits for RRef<T>. The code is deceptively simple, dereferencing the raw pointer and creating a regular mutable reference to the pointer's value. Behind the scenes, Rust also inserts lifetimes [?] linking &mut T to &mut self. This ensures that a dereferenced &mut T will not outlive the RRef<T> container. Further, this allows the compiler to enforce the Rust model of "one mutable reference *or* many immutable references" [?] for RRef<T>.

```
impl<T: RRefable> DerefMut for RRef<T> {
    fn deref_mut(&mut self) -> &mut T {
        unsafe { &mut *self.value_pointer }
    }
}
```

Listing 6: RRef DerefMut implementation

3.2.5 RRefable

RRef<T> ensures that the memory for T lies on the shared heap. However, what if T is a data structure with another reference to memory outside the shared heap? This reference could become a dangling pointer during IPC if it points to memory on a dead domain's heap. Pointers in safe Rust are guaranteed to be valid, so this would break Rust safety and as such be an attack vector. For this reason we restrict RRef<T> to contain exclusively "copy" types *or* other references to memory on the shared heap. Following these rules, RRef<usize>, RRef<[1,2,3]>, and RRef<RRef<usize>> are all valid, while RRef<&str> should fail to compile.

We could enforce this in the RedLeaf IDL. However, with Rust's rich trait type system we can do most of this validation at compile time [1]. To this extent, the T in RRef<T> is restricted by the RRefable trait.

The RRefable trait definition is shown in Listing 7. As an *auto trait* [?], RRefable is implemented implicitly for every type that fits the definition. Rather than enumerate all the possible types that RRef<T> can contain, RRefable instead provides *negative* implementations for all reference types. The Rust compiler recursively checks auto trait definitions, so even though Box<T> does not directly match *mut T, *const T, &T, or &mut T, it is still not RRefable because it contains a reference somewhere in its tree of definitions (specifically, Box<T> is backed by Unique<T>, which itself is backed by *const T).

```
#![feature(optin_builtin_traits)]
#![feature(negative_impls)]
pub unsafe auto trait RRefable {}
impl<T> !RRefable for *mut T {}
impl<T> !RRefable for *const T {}
impl<T> !RRefable for &T {}
impl<T> !RRefable for &mut T {}
```

Listing 7: RRefable trait definition

3.2.6 Borrowing

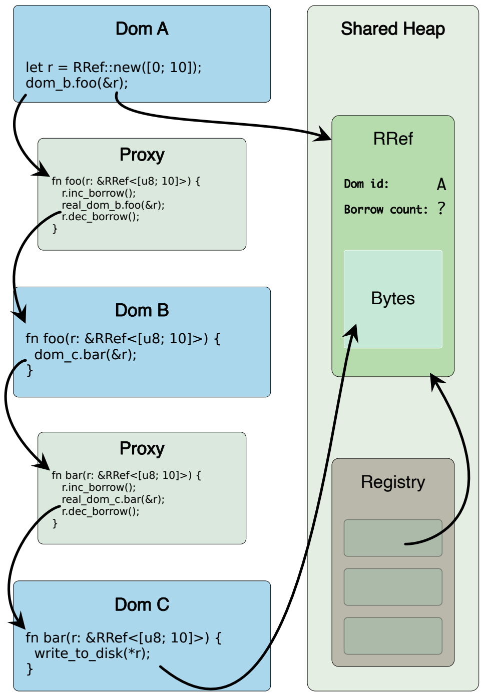
Earlier, Listing 2b showed the interface for a typical block device driver. Compared to non-RRef code (Listing 2a), the primary difference is the read(block:data:) method which takes and returns the data buffer (move-in-move-out) instead of mutably borrowing it. Move-in-move-out is more cumbersome, but unfortunately, mutable borrows cannot be supported in IPC. Consider the scenario where the block device driver domain partially writes to the buffer, and then panics. The buffer is left in a corrupted state and should not be returned to the caller domain. This alone prevents their use in IPC. On the other hand, *immutable* borrows pose no such threat.

[1]The RRefable trait does not account for function pointers. For this reason, we further validate *RRef* types with the RedLeaf IDL.

The motivation for supporting immutable borrows for *RRef*s is twofold. First, immutable borrows elegantly express that a value is only read, whereas move-in-move-out values change ownership and as such can be modified. Further, and more importantly, in the case that a domain *moves* a remote reference to another domain, which happens to panic, that *RRef* is permanently gone. In the case of an immutable borrow, that reference should still be valid since it was never modified.

We keep track of ownership during an immutable borrow with a simple counter. The purpose of the borrow counter is to clean up a remote reference in the case that a domain panics. A proxy lies in between each domain crossing and manages the borrow counter. It increments the counter when descending into another domain, and decrements it upon return (successful or unsuccessful). The *RRef* is only deallocated when the borrow counter hits zero *and* its owner domain is dead.

Consider the scenario in Figure 2. When `R` is borrowed by domain B, the counter increments to one. If at this point domain A panics, the kernel will check the shared heap registry and find that the owner of `R` panicked, but it has a non-zero borrow count and as such is still in use. Once `R` returns back to domain B's proxy, its borrow counter is reduced back to zero. Since the owner domain (A) is now dead, and the borrow counter is zero, this reference is deallocated.



```
let r = RRef::new([0; 10]);
dom_b.foo(&r);
```

```
fn foo(r: &RRef<[u8; 10]>) {
   r.inc_borrow();
   real_dom_b.foo(&r);
   r.dec_borrow();
}
```

```
fn foo(r: &RRef<[u8; 10]>) {
  dom_c.bar(&r);
}
```

```
fn bar(r: &RRef<[u8; 10]>) {
   r.inc_borrow();
   real_dom_c.bar(&r);
   r.dec_borrow();
}
```

```
fn bar(r: &RRef<[u8; 10]>) {
  write_to_disk(*r);
}
```

Figure 2: RRef immutably borrowed across 3 domains.

In the case that domains B or C panic before `R` returns to domain A, the intermediate proxies handle the unwinding and safely return the reference back to domain A. Because the reference was immutably borrowed, Rust guarantees that it was never modified, and as such remains valid for further use.

3.3 RRefArray

`RRefArray` is the first data structure built on top of the base `RRef` type. While it is possible to represent an array of remote references with just `RRef<[T; N]>`, it quickly becomes unwieldy. To move an element out of the array, it has to be wrapped in an `Option` to represent the empty slot. Further, the element must also be wrapped in its own `RRef`, since its owner can change when it is removed from the array. To manage this ownership, proper accessors have to be generated by the RedLeaf IDL [**?**]. `RRefArray` simplifies this use case, and serves as the base for further data structures like `RRefDeque` (Section 3.4).

Listing 8 shows that an `RRefArray` is mostly a wrapper around `RRef<[Option<RRef<T>>; N]>`. Elements can be pulled out and reinserted, and their ownership is tracked. When an element is inserted into the array, its owner changes to `0` to represent that it is owned by another *RRef* (this avoids double-free). Because this code lives in the trusted `RRef` crate, it is allowed to use privileged methods to update ownership which would otherwise have to be generated by the IDL.

3.4 RRefDeque

`RRefDeque` builds on `RRefArray` in order to provide a deque (double ended queue) data structure. The deque data structure supports push and pop operations on both the head and tail of the queue, and is included in Rust's standard library as `VecDeque` [**?**]. `RRefDeque` aims to be a drop-in replacement for `VecDeque`.

`RRefDeque`'s primary discrepancy is its fallible insertion operation. This stems from the fact that all shared heap allocations need to be of a fixed size, because the shared heap relies on compile time layout information of types. In the future, the shared heap could also support dynamically-sized data structures, but is currently limited by Rust language support [**?**]. As a result `RRefDeque` is backed by a fixed-size `RRefArray`

```
#![feature(const_generics)]
pub struct RRefArray<T: RRefable, const N: usize> where T: 'static {
    arr: RRef<[Option<RRef<T>>; N]>
}
impl<T: RRefable, const N: usize> RRefArray<T, N> {
    pub fn get(&mut self, index: usize) -> Option<RRef<T>> {
        let value = self.arr[index].take();
        if let Some(rref) = value.as_ref() {
            unsafe { rref.move_to_current() }; // mark as owned by this domain
        }
        value
    }
    pub fn set(&mut self, index: usize, value: RRef<T>) {
        unsafe { value.move_to(0); } // mark as child of another RRef
        self.arr[index].replace(value);
    }
    // ...
}
```

Listing 8: RRefArray struct definition

and can run out of memory during insertion and return an error. VecDeque, on the other hand, requests more memory when it nears capacity, allowing for infallible insertion operations.

Listing 9 covers the source code behind RRefDeque. RRefDeque builds on RRef and RRefArray, making its implementation trivial, without any manipulation with regards to domain ownership. This demonstrates that RRef is a scalable abstraction that can support most IPC needs.

```
#![feature(const_generics)]
pub struct RRefDeque<T: RRefable, const N: usize> where T: 'static {
    arr: RRefArray<T, N>,
    head: usize, // index of the next element that can be written
    tail: usize, // index of the first element that can be read
}
impl<T: RRefable, const N: usize> RRefDeque<T, N> {
    // ...
    pub fn push_back(&mut self, value: RRef<T>) -> Option<RRef<T>> {
        if self.arr.has(self.head) {
            return Some(value); // give the element back if full
        }
        self.arr.set(self.head, value);
        self.head = (self.head + 1) % N;
        return None;
    }
    pub fn pop_front(&mut self) -> Option<RRef<T>> {
        let value = self.arr.get(self.tail);
        if value.is_some() {
            self.tail = (self.tail + 1) % N;
        }
        return value;
    }
    // ...
}
```

Listing 9: RRefDeque struct definition

The primary use case for RRefDeque in RedLeaf was the submit_and_poll operation in the ixgbe network driver (Listing 10). submit_and_poll sends a packet queue to the network driver, which processes the packets and moves them to the "collected" queue. As covered in Section 3.2.6, a mutable borrow of the packet queues would be cleaner, but is prohibited.

```
fn submit_and_poll(
    &self,
    mut packets: RRefDeque<[u8; 1514], 32>,
    mut collect: RRefDeque<[u8; 1514], 32>,
    tx: bool,
    pkt_len: usize
) -> RpcResult<Result<(
    usize,
    RRefDeque<[u8; 1514], 32>,
    RRefDeque<[u8; 1514], 32>
)>>;
```

Listing 10: Ixgbe network driver submit_and_poll interface using RRefDeque

4 Performance

Operation	Cycles
seL4 [1]	1260
RedLeaf	42
RedLeaf (passing an *RRef*)	59

Table 1: Language-based cross-domain invocation vs hardware isolation mechanisms.

We ran benchmarks comparing RedLeaf's IPC to the state of the art on openly-available CloudLab [**?**] c220g2 servers configured with two Intel E5-2660 v3 10-core Haswell CPUs running at 2.60 GHz. Table 1 shows the results of these benchmarks, where RedLeaf is an order of magnitude faster than seL4. RedLeaf IPC performance is comparable to that of several regular function calls. The only overhead is keeping track of ownership at domain boundaries, which is relatively cheap. This is the result of relying almost exclusively on the programming language for safety and isolation.

5 Conclusions

Replacing hardware methods for isolation with language safety proves to improve performance significantly without sacrificing flexibility for cross domain communication. Rust's support for designing smart pointer types allows for remote references to behave like regular pointers (as seen in Listing 3) with the added benefit of isolation. Relying on the RedLeaf IDL to generate glue code in proxies makes this a scalable approach to ensuring isolation. Rust allows for a new wave of microkernel development that can achieve fine-grained isolation without suffering from hardware overhead.

Acknowledgments

This research is supported in part by the National Science Foundation under Grant Number 1840197.

References

[1] sel4 performance. `https://sel4.systems/About/Performance/`.