Politecnico di Torino

Politecnico di Torino

Computer Engineering

A.Y. 2024/2025

Graduation Session October 2025

Accelerating Inter-Host Communication Between Microservices with RDMA/eBPF

Supervisors:

Prof. Fulvio Risso

Dott. Davide Miola

Candidate:

Luca Menozzi

Abstract

As the cloud-native paradigm increases in popularity and quickly becomes the de-facto standard for datacenter software development, monolithic applications are split into functionally distinct microservices. Consequently, east-west network traffic becomes critical to ensure system correctness. Traditionally, node-to-node communication relies on the TCP/IP stack, which provides reliable, ordered message delivery between applications. Recently, technologies such as Remote Direct Memory Access (RDMA) have emerged, offering equivalent reliability while supporting higher data rates and lower latency.

This thesis presents a TCP-to-RDMA proxy that bridges these technologies. The proxy transparently intercepts outbound application traffic at the socket level and leverages RDMA to transport it reliably, effectively bypassing the TCP/IP stack while maintaining full application compatibility. Achieving full transparency without kernel modifications is challenging; our solution exploits the eBPF framework, which enables in-kernel programmability and allows network traffic to be dynamically monitored and redirected. This approach ensures portability, low overhead, and seamless integration with existing applications.

In addition to outlining the overall architecture, this work discusses the main design choices and the implementation of the proxy, focusing on how eBPF was leveraged to ensure transparency and flexibility, and how RDMA was exploited to achieve higher throughput. We also conducted a preliminary performance evaluation: although the results are not yet fully satisfactory, they already provide useful insights into the system's behavior and point to clear directions for improvement. Along the way, we encountered unexpected issues with the SKMSG hook. While this was not part of the initial objectives, it turned out to be an interesting and valuable finding that sheds light on the limitations of current kernel mechanisms.

Looking ahead, we believe that this line of work opens the door for a new generation of datacenter networking. By bridging legacy application interfaces with modern transport technologies in a fully transparent way, it becomes possible to rethink the role of the operating system in distributed environments, reduce the reliance on traditional network stacks, and unlock new opportunities for performance and scalability.

Acknowledgements

I would like to thank Professor Fulvio Risso for his support and guidance throughout this thesis work.

I am especially grateful to Dr. Davide Miola for his constant availability, insightful suggestions, and technical expertise. His support was fundamental in shaping and improving the quality of this work.

A special thanks to the Lab9 team for their warm welcome and continued support during this experience.

Finally, I'm grateful to everyone I've bothered while asking for dedicated NICs to run RDMA — thank you for your patience!

Table of Contents

List of Figures

Glossary

RDMA
: Remote Direct Memory Access

eBPF
: extended Berkeley Packet Filter

SKMSG
: SocKet MeSsaGe

NIC
: Network Interface Card

DMA
: Direct Memory Access

IRQ
: Interrupt Request

RSS
: Receive Side Scaling

GRO
: Generic Receive Offload

GSO
: Generic Segmentation Offload

XDP
: eXpress Data Path

JIT

Just-In-Time (Compilation)

TCP

Transmission Control Protocol

RNIC

RDMA-capable Network Interface Card

QP

Queue Pair

SQ

Send Queue

RQ

Receive Queue

WR

Work Request

WQEs

Work Queue Elements

SGE

Scatter-Gather Element

CQ

Completion Queue

MR

Memory Region

L-KEY

Local Key

R-KEY

Remote Key

PD

Protection Domain

Chapter 1

Introduction

In recent years, the way applications are designed and developed has changed. In the past, the trend was to develop large, monolithic applications that typically ran on a single machine; today, there is a shift towards a new paradigm that is much more modular. In this new paradigm, the once centralized application is divided into many smaller, independent components that interact with each other. These components are called microservices.

Today, microservices have become the standard approach to *cloud-native* application development. Since each service is designed to focus on only a small part of the overall problem, the ability to communicate is crucial to the proper functioning of the entire application.

This communication has always been made possible by the `TCP/IP` stack, which segments data into many small packets, sends them, and receives them on the other side of the connection. However, in recent years, new technologies have emerged that offer equally valid alternatives for data transfer, such as *Remote Direct Memory Access* (`RDMA`).

`RDMA` is a new technology that aims to transfer data directly from the sender's memory to the receiver's memory. With `RDMA`, a program specifies the local source and remote destination, and the data is copied directly to the destination memory. This results in extremely low latency and high throughput.

The objective of this thesis is to develop a mechanism to use `RDMA` as the transport medium instead of `TCP`, while keeping everything completely transparent to the applications running on top. To make this work without touching the applications themselves, we need to:

(i) intercept the data being sent so it can be carried over `RDMA`, and

(ii) once it arrives, hand it over to the destination application exactly as it expects.

For this reason, we introduced a unit that works as a transparent bridge between

the application and RDMA: eBPF. With eBPF, developers can inject code directly into the kernel, which opens up a lot of new possibilities.

1.1 Goals

This thesis explores an experimental approach to communication in distributed applications, proposing an alternative to the traditional TCP/IP stack.

In standard TCP-based communication, applications rely on sockets. The idea here is to intercept data after it has been sent to the application's socket but before it enters the TCP layer. This interception can be achieved using the eBPF framework. The captured data is then transferred to the destination peer via RDMA and, on the receiving side, re-injected into a socket. The entire process is designed to remain fully transparent to the applications. The proposed architecture thus replaces TCP with RDMA, aiming to take advantage of the latter's improved throughput and latency for microservice-based applications.

The objectives of this work are twofold: first, to design an architecture capable of intercepting data at the socket layer (before it reaches the TCP stack) using SK_MSG and redirecting it through RDMA; and second, to evaluate both the practicality and effectiveness of this architecture.

Chapter 2

Background

2.1 Linux network stack

2.1.1 Overview

Every time a device communicates on a network, the information is divided into small units called *packets*. A packet is physically just a block of data, but it is logically layered and structured, with each layer adding content.

At the bottom layer is the Ethernet header, which specifies the hardware addresses and payload type. Above that is the IP header, which carries the source and destination IP addresses, and then the transport layer, which provides all the information necessary for the packet to be transmitted correctly over the network.

Together, these headers serve to specify where a packet comes from, where it is going, and how it should be handled once it is received.

Ingress: the path into the system

When a packet is received on a Linux-based system, it first arrives at the Network Interface Card (NIC). The NIC checks the destination MAC address and discards irrelevant frames (unless in *promiscuous mode*), moreover, it also verifies the Ethernet checksum, dropping corrupted frames avoiding wasting CPU time. If valid, the NIC transfers the packet into main memory via Direct Memory Access (DMA) and signals the CPU with an interrupt. Since this interrupt is generated by the NIC, it is a hardware interrupt and will likely interrupt the CPU immediately, potentially preempting currently running tasks.

Given that a non-preemptible interrupt cannot be interrupted while running, a long-running handler would prevent other interrupts to be managed. Best practice to solve this problem is to keep hardware interrupt work minimal, deferring most processing to a software interrupt (softirq).

This separation is known as the Top Half (urgent tasks) and Bottom Half (deferred tasks) model:

- **Top Half Handler (Hard IRQ):** Runs immediately when the interrupt is raised in order to handle time-critical work that cannot be delayed (eg copy the packet and freeing memory in the NIC). During execution, interrupts and preemption are disabled, so the handler must be as brief as possible.
- **Bottom Half Handler (Soft IRQ):** This handles tasks that take more time and can be deferred without issue. During this phase, interrupts are enabled, and preemption is disabled; Hence soft IRQs can be interrupted by higher-priority hardware interrupts, which makes them more flexible than top halves. If the workload in the bottom half becomes too heavy, it can be split into smaller, more manageable tasks. Through Soft IRQ, the packet will complete its trip inside the network stack.

Once the packet is in a buffer (thus the Hard IRQ ended) it is wrapped in a data structure called `sk_buff` (socket buffer). The `sk_buff` contains fields that assist in further packet processing. Initially, only the primary fields are allocated, then other fields can be populated by different components as needed. One main advantages of this structure is to allow headers to be "removed" or "re-added" simply by adjusting pointers, avoiding expensive memory copies (Figure 2.1).

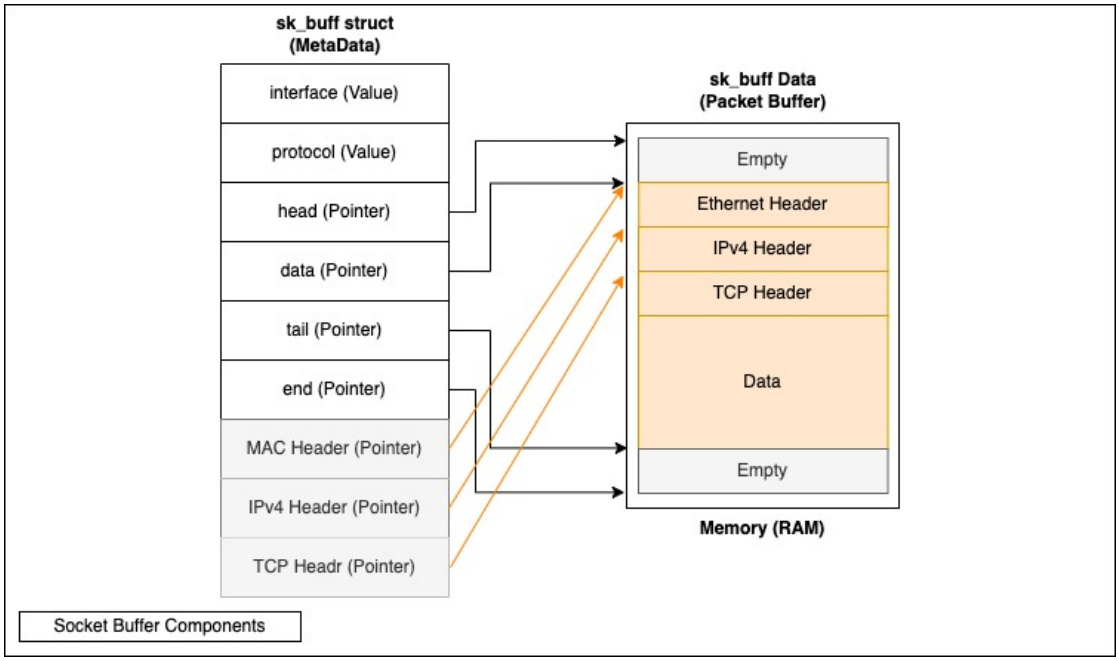

Figure 2.1: The `sk_buff` structure

After the `sk_buff` has been initialized, the packet travels up the full network stack for processing: the Ethernet header is removed, the IP header is validated, firewall rules may be applied, and routing decisions are made.

If the packet is not destined for the host, it may be forwarded; otherwise, it moves to the TCP stack, where it is parsed and delivered to the application through the appropriate socket. Finally, when the application calls `read()`, it receives the contents of the packet.

Egress: the path out

The reverse journey begins when an application calls the `write()` system call on a socket.

At this point, the kernel allocates a socket buffer, fills it with the payload, and constructs the necessary headers. The TCP layer sets its parameters, such as ports and sequence numbers, the IP layer adds the source and destination addresses, and finally the Ethernet layer appends the MAC addresses. Once all the required fields are in place, the packet is queued onto the NIC driver.

The NIC then fetches the packet via DMA, computes the final checksum, and transmits it onto the network.

Is important to note that, from the application's perspective, this entire process is completely transparent; It simply sees the `write()` call return.

2.1.2 Network stack short-cutting

While the basic process is straightforward, modern throughput demands require optimization. General-purpose CPUs are not the most efficient platform for implementing network functions. Consequently, processing tasks related to the network stack can consume a substantial portion of the CPU's overall resources, reducing the cycles available for application logic. To address this challenge, Linux incorporates various offloading techniques aimed at enhancing the efficiency of the software network stack and accelerating packet processing. These include both hardware- and software-based solutions.

Key techniques include:

- **Checksum offloading:** Letting the NIC compute and verify checksums instead of the CPU.
- **RSS (Receive-side scaling):** Distributing traffic across CPU cores (The NIC must be compatible)
- **NAPI:** Switching from interrupts to polling under heavy load, reducing interrupt overhead.

- **Generic Receive Offload (GRO):** Merging small packets into larger ones for efficiency.
- **Generic Segmentation Offload (GSO):** Keeping data aggregated and splitting only at transmission, often in hardware.

Another way to accelerate traffic is by working directly within the Linux network stack. Not all packets need to pass through every stage of processing; some well-known packets can safely skip certain steps since the system already knows how to handle them.

Two important techniques are worth highlighting:

- **Traffic Control (TC):** Flower is one of the TC filters (or *classifiers*) available in Linux. It matches packets based on specific flow information (inspired by OpenFlow) and can apply various actions such as *drop*, *police*, or *mirred*. The most relevant action here is *mirred* (mirror/redirect), which allows packets to be mirrored or redirected. This makes it possible to intercept packets based on their flow and forward them directly to another part of the Linux networking stack, effectively skipping unnecessary processing steps.
- **Flowtables:** Flowtables enable the automatic offloading of established flows. They act as the filtering component of the Linux firewall, invoked at specific points in the stack when packets pass through. Information about flows is stored in the flowtable, which can then be used to accelerate operations such as NAT by avoiding repeated processing.

Another important approach to consider is the use of eBPF. As will be explained later (Section 2.2), eBPF is a native Linux framework that allows users to inject code directly into the kernel, avoiding the need for direct kernel modifications, which can quickly become complex and difficult to maintain.

An eBPF program can be attached to various hook points within the kernel, including points located directly inside the networking stack. These programs can perform a wide variety of packet processing tasks, though with some restrictions on what is otherwise "almost arbitrary" processing. Once a packet is processed, an eBPF program has several options: pass, drop or redirect.

The most important hook point inside the linux network stack are:

- **XDP (eXpress Data Path) [1]:** The earliest hook in the NIC driver's RX path. It is extremely fast and well-suited for dropping, redirecting, or forwarding packets before they reach the kernel stack. Typical use cases include DDoS mitigation and load balancing.
- **TC (Traffic Control):** Ingress and egress hooks that operate on `sk_buffs`. TC is more flexible than XDP, supporting rich packet manipulation, shaping,

and policy enforcement. It is widely used in container networking solutions (e.g., *Cilium* [2]).

- **SockOps (Socket Operations):** Hooks into TCP connection lifecycle events, enabling custom congestion control, connection steering, and advanced in-kernel load balancing mechanisms.
- **Cgroup Hooks:** Attach to traffic within specific control groups, making it possible to enforce per-container or per-tenant network policies. They are particularly useful in Kubernetes environments for traffic redirection and isolation.

In general, eBPF can be applied in two ways: either to *shortcut* parts of the Linux networking stack or to *cooperate* with it. When bypassing portions of the stack, eBPF can deliver significant performance improvements. On the other hand, it can also be used to enhance existing functionality by working alongside the kernel. For instance, eBPF can take over firewalling tasks, removing the need for traditional iptables rules. In this role, it effectively acts as a Netfilter module, simplifying firewall management and improving flexibility.

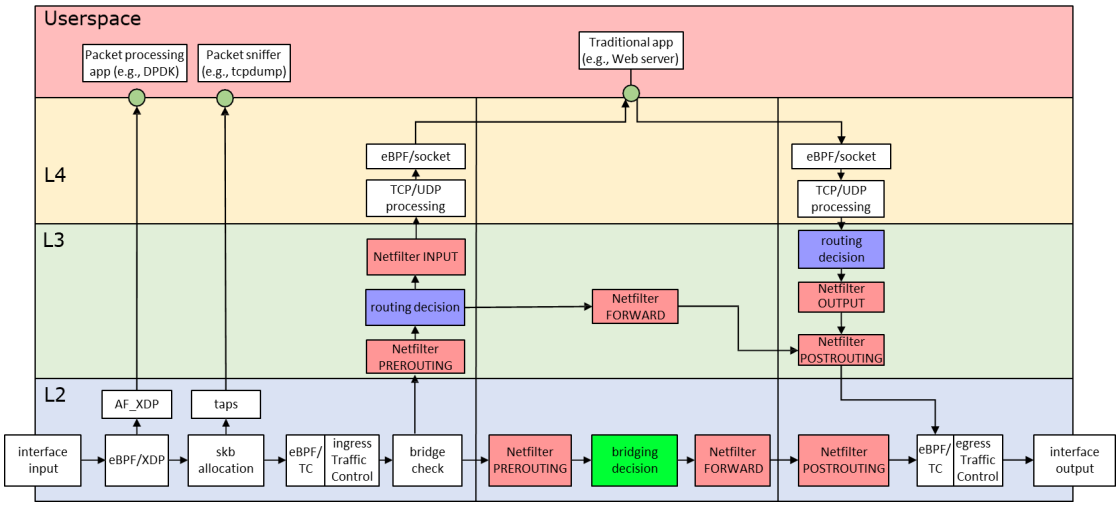

Figure 2.2: Linux network stack with most important eBPF hook

Although the Linux kernel is already heavily optimized, eBPF can often outperform standard packet processing in these scenarios.

2.1.3 Direct memory access

Data movements (either memory-to-memory, device-to-memory, or vice-versa) normally require an active CPU intervention: the data should be read from the

source address and then written to the destination address. It can thus be noted that this process may become highly inefficient since the CPU cannot perform other computations during the transfer.

To address this inefficiency, modern computer architectures employ a special-purpose component called a *Direct Memory-Access (DMA) controller*, which allows the CPU to offload much of this work to hardware.

The process begins when the host system writes a *DMA command block* into memory. This block contains a pointer to the source of the data, a pointer to its destination, and also the number of the bytes to be transferred. In more complex scenarios, a single command block can specify multiple, noncontiguous memory regions in what is called a *scatter–gather* transfer.

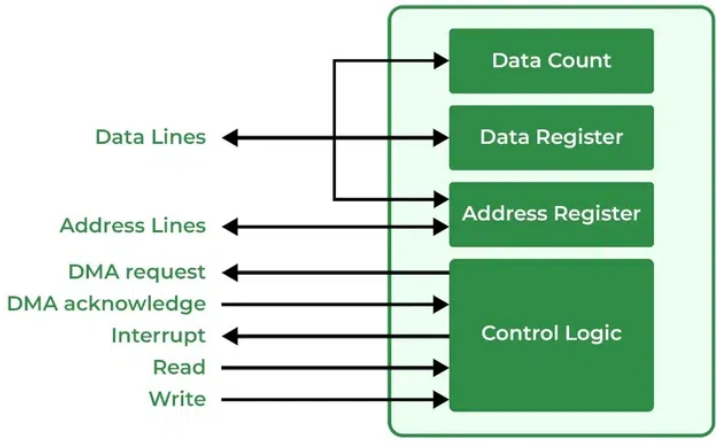

Figure 2.3: DMA Command Block

Once the CPU hands off the command block to the DMA controller, it is free to perform other tasks while the DMA controller takes over the memory bus and carries out the data transfer autonomously.

Usually, it is better to set DMA target inside kernel memory rather than user space. If user-space memory were used directly, there is a risk that the user could modify the data mid-transfer, potentially resulting in lost or corrupted data. To make DMA-transferred data accessible to user-space threads, an additional copy operation—from kernel memory to user memory—is required. This *double buffering* introduces inefficiency, which has been solved in modern operating systems by adopting *memory-mapped I/O*, allowing devices to interact directly with user-space memory without extra copying.

Although temporarily seizing the memory bus can momentarily prevent the CPU to access the main memory (but not the cache), the overall effect is positive: offloading data transfer tasks to a DMA controller usually accelerates total system

throughput.

Another feature that has been introduced is *Direct Virtual Memory Access (DVMA)*, which allows a system to perform DMA operations using virtual memory addresses, rather than being limited to physical addresses.

In conclusion, the introduction of DMA significantly reduces CPU overhead and improves overall system performance. DMA controllers are now standard in all computing devices, from smartphones to servers.

2.2 eBPF

The *extended Berkeley Packet Filter (eBPF)* is a very important advancement in operating system design, originating from the Linux kernel.

eBPF allows to execute sandboxed programs directly within privileged contexts such as the kernel itself. This ability provides a safe and efficient mechanism to enhance kernel functionality without the need to alter the kernel source code or load new kernel modules.

The kernel enjoys a privileged position due to its unique oversight of the entire system, which allows it to monitor, regulate, and enforce the behavior of all processes. However, this centrality has also imposed limitations on modifications and the addition of new features. In fact, its central position in the system has made it particularly difficult to manage since any change must pass extremely high standards of stability and security, and as a result, the pace of innovation at the kernel level has often lagged behind that of user space technologies.

eBPF changed this balance. By enabling sandboxed programs to execute safely within the kernel, it allows developers to extend kernel behavior dynamically and without risk to system integrity.

The scripts injected in the kernel are subject to rigorous verification and benefit from the performance of a Just-In-Time (JIT) compiler, enabling eBPF programs to run both safely and with near-native efficiency.

2.2.1 Key features of eBPF

Event-driven execution and hooking

eBPF programs are inherently event-driven, executing only when the kernel or an application triggers a specific hook point. These hooks are predefined for a variety of common scenarios, including: system calls, function entry and exit points, kernel tracepoints, network events, and more.

In case we are looking to hook the program to a particular position where a predefined hook does not exist, it is still possible to create a kernel probe (*kprobe*) or a user probe (*uprobe*), allowing for extensive observability of both the kernel

and user-space applications. However, these probes offer limited customization compared to other hooking mechanisms.

Maps and Helper

A crucial aspect of eBPF programs is their ability to store state and share collected information across different execution contexts. To achieve this, eBPF provides *maps*. Maps are kernel-managed data structures that allow programs to store and retrieve data efficiently. These maps can be accessed both from within eBPF programs and from user-space applications (through system calls), facilitating communication between kernel and user space.

Maps enable effective communication and coordination across multiple eBPF programs, even when they are running on different CPU cores. This shared approach enhances scalability and allows programs to maintain persistent state without the need to embed the data directly in the code. In this model, the eBPF code remains stateless, while the maps hold all program state and configuration data. This separation of data from code improved both the efficiency by avoiding unnecessary data copying and also the flexibility.

eBPF supports multiple map types, often with both shared and per-CPU variations. Common examples include:

- Hash tables and arrays
- LRU (Least Recently Used) caches
- Ring buffers
- Stack traces
- LPM (Longest Prefix Match) tables

Within the eBPF framework, programs are deliberately restricted from accessing arbitrary kernel memory or calling kernel functions directly. Instead, any interaction with data outside the program's context must go through eBPF helper functions. These helpers provide a controlled and privilege-aware access, ensuring that a program can read and modify only the data it is allowed to handle.

Loader and verification architecture

Because eBPF code executes directly within the kernel, security becomes a fundamental concern: any flaw, bug, or malicious behavior could compromise the stability of the entire system. To guard against this, eBPF relies on two main mechanisms: a *sandbox* and a *verifier*.

The sandbox provides an isolated execution environment where eBPF programs run safely. It enforces strict boundaries, preventing any unauthorized or invalid memory access that might destabilize the kernel. This design ensures that even though eBPF code is highly integrated into kernel operations, it cannot directly interfere with or damage critical system resources.

The verifier acts as a safeguard that examines every eBPF program before it is executed in the kernel. Its role is to guarantee safety and correctness by performing a series of checks, including:

- **Guaranteed termination:** Programs must always run to completion. Loops are only permitted if the verifier can prove that an exit condition is guaranteed to be reached.
- **Memory safety:** Programs may not access uninitialized variables or read-/write memory outside of allowed bounds.
- **Size limits:** Programs must fit within system-defined size constraints; excessively large programs are rejected.
- **Finite complexity:** The verifier evaluates all possible execution paths, ensuring that analysis completes within the configured complexity limits.

Through these mechanisms, the verifier prevents unsafe or unstable code from reaching kernel space, ensuring that only well-formed and secure eBPF programs are admitted into the kernel.

Once a program successfully passes verification, it undergoes a hardening process that depends on whether it was loaded by a privileged or unprivileged process. Key hardening measures include:

- **Program execution protection:** Kernel memory storing eBPF programs is marked read-only. Any attempt to modify the program, whether due to a kernel bug or malicious action, will result in a kernel crash rather than continued execution of a corrupted program.
- **Mitigation against speculative execution attacks:** To mitigate Spectre, eBPF programs mask memory accesses to redirect speculative instructions safely, the verifier inspects paths reachable only during speculation, and the JIT compiler emits Retpolines when tail calls cannot be turned into direct calls.
- **Constant blinding:** All constants in eBPF programs are blinded to prevent JIT-spraying attacks, which could otherwise allow an attacker to execute injected code in the presence of another kernel vulnerability.

Once an eBPF program has been verified as safe and the appropriate hook point has been identified, it can be loaded into the Linux kernel using the `bpf` system call, typically via one of the available eBPF framework libraries.

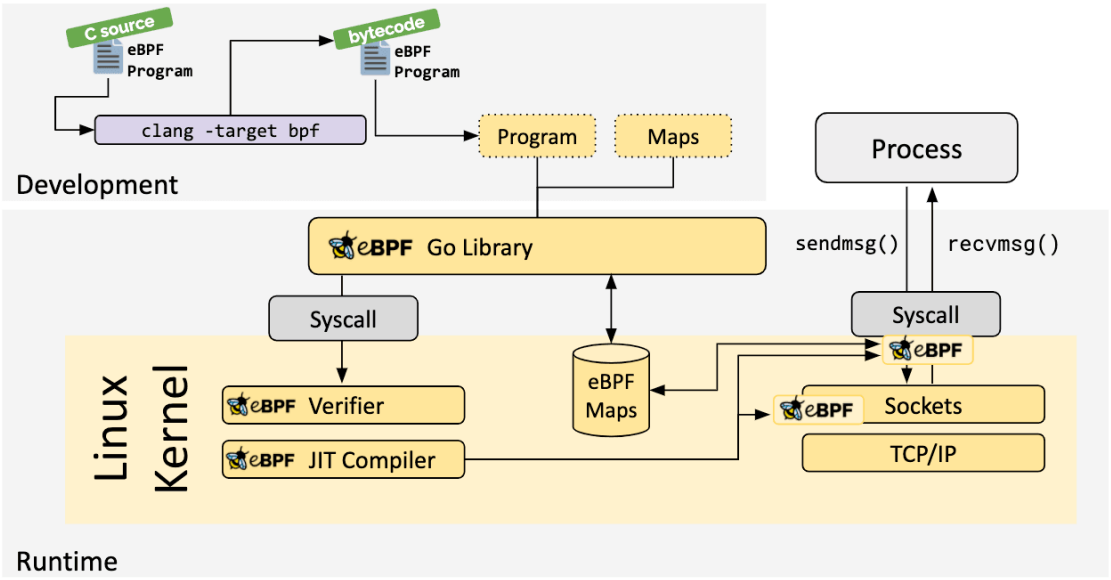

Figure 2.4: eBPF Framework

2.3 RDMA

RDMA technologies build on concepts from traditional networking but differ significantly from standard IP networks. The key distinction is that RDMA offers a messaging service enabling applications to directly access the virtual memory of remote computers. This has diverse applications, including: inter-process communication (IPC), communication with remote servers, or interaction with storage devices via Upper Layer Protocols such as iSER, SRP, SMB, Samba, Lustre, and ZFS.

RDMA achieves low latency by bypassing the OS network stack and avoiding unnecessary memory copies. It also reduces CPU utilization, alleviates memory bandwidth bottlenecks, and supports high-bandwidth usage. By providing channel-based I/O, RDMA allows applications to directly read and write remote virtual memory.

In contrast, traditional socket-based networks rely on the operating system to mediate data transfers via an API. RDMA uses the OS only to establish the communication channel, after which applications can exchange messages directly. These messages may be RDMA Read, RDMA Write, or Send/Receive operations.

The core idea of Remote Direct Memory Access (RDMA) is to extend the concept of DMA (Section 2.1.3) across the network. With RDMA, one computer can directly access the memory of another computer, bypassing the CPUs, caches, and operating systems of both machines. This capability enables highly efficient data transfers between systems, which is particularly beneficial in high-performance computing and data-intensive applications.

As mentioned, the key advantages of RDMA includes:

- **Zero-copy data transfer:** RDMA eliminates the need to copy data between buffers, thereby reducing latency and improving performance.
- **CPU bypass:** Memory operations do not consume CPU resources, allowing the processor to handle other tasks concurrently.
- **Operating system bypass:** By avoiding system calls, RDMA reduces the overhead associated with kernel involvement (context switch), leading to faster data movement.
- **Low latency:** By avoiding unnecessary processing steps, RDMA minimizes response times for memory access.
- **High bandwidth:** RDMA maximizes data transfer speeds by minimizing bottlenecks in the communication path.

Overall, RDMA provides a powerful mechanism for high-speed, low-latency communication between computers, making it a very popular tool in modern computing environments.

2.3.1 Data flow over TCP and RDMA

In a traditional Transmission Control Protocol (TCP) communication, data transfer involves several intermediate steps:

1. The sending application copies data to the kernel
2. TCP stack handles packetization and other processing tasks
3. The network interface card (NIC) transmits the packets over the network
4. On the receiving side, the NIC receives the incoming packets and forwards them back through the TCP stack
5. The kernel reconstructs the original data
6. Finally, the kernel delivers the data to the receiving application

Each of these steps requires CPU cycles, which can increase latency and reduce the system efficiency.

RDMA modifies this process by eliminating many of the intermediate steps. In an RDMA-enabled system, an RDMA-capable NIC (RNIC) communicates directly with the application, bypassing the operating system kernel and the traditional networking stack. This technique, commonly referred to as *kernel bypass*, substantially reduces CPU overhead and allows data to move between applications more quickly and efficiently, resulting in lower latency and improved performance.

This approach is made possible by the use of shared memory. During RDMA configuration, among other things, a shared memory region is registered and pinned, allowing the RNIC to access it directly. The memory physically resides in user space but is made accessible to the RNIC through kernel-managed mappings. This mechanism allows processes using RDMA to bypass system calls entirely.

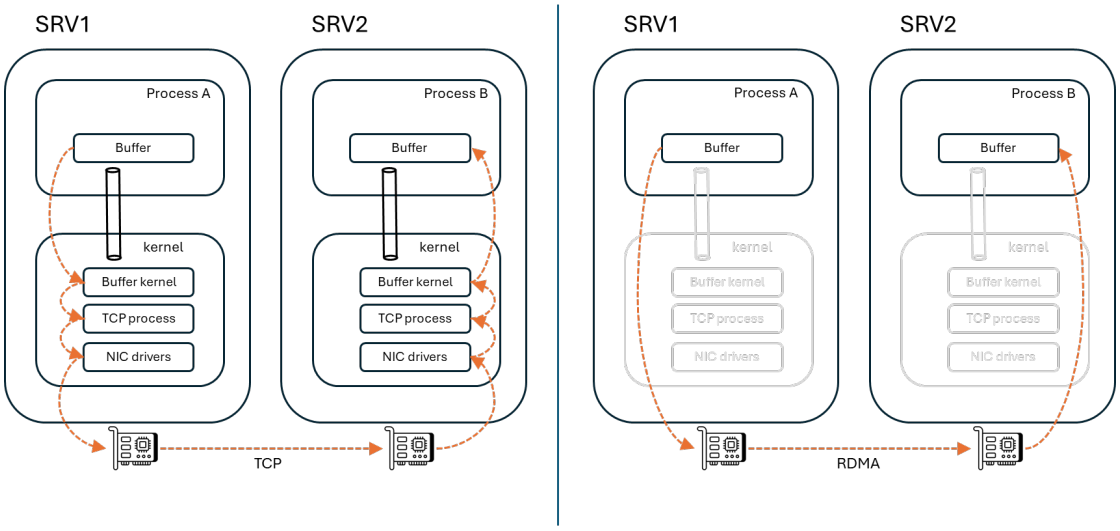

Figure 2.5: TCP (left) vs RDMA (right) architecture

2.3.2 Operation modes

Remote Direct Memory Access (RDMA) supports two primary categories of operations, each offering distinct approaches for transferring data between machines.

Channel semantics (Send/Receive Model)

In the channel semantics model, data transmission requires active participation from both the sender and the receiver. This model includes the following operations:

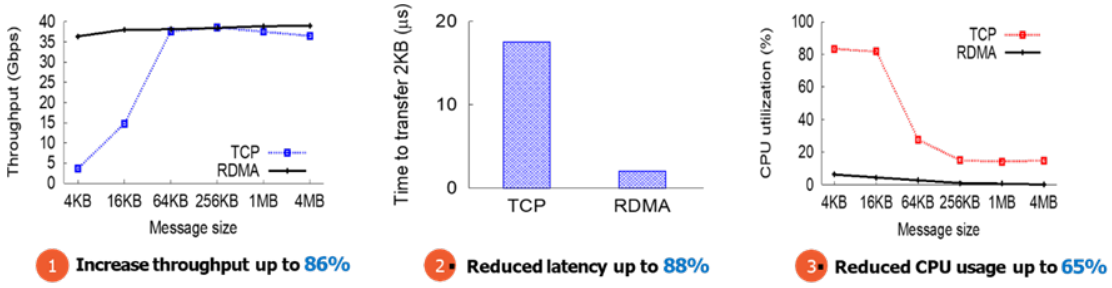

Figure 2.6: RDMA performance [3]

- **Send Operation:** The sender initiates the transfer by sending data to the receiver without precise knowledge of the memory location where the data will be stored on the receiving side
- **Receive Operation:** To handle this incoming data, the receiver must have a corresponding receive operation prepared in advance, used to specify where the data will be inserted inside the memory.

Memory semantics (RDMA Read/Write – CPU and Kernel Bypass)

The memory semantics model enables RDMA to transfer data directly between the memory spaces of two machines, bypassing the CPU and the operating system kernel entirely. The available operations are:

- **Read:** Data is read directly from remote memory into local memory.
- **Write:** Data is written directly to a specified location in remote memory.
- **Atomic Operations:** These operations perform read-modify-write sequences on remote memory atomically, ensuring data consistency in environments with concurrent access.

Overall, the memory semantics mode leverages the full potential of RDMA by minimizing overhead from the traditional networking stack and providing efficient low-latency data transfers and a CPU usage reduction.

2.3.3 Transports layer

Remote Direct Memory Access (RDMA) can be implemented over different transport protocols, each of which handles Layers 2, 3, and 4 of the network stack differently. The three main RDMA transport protocols are:

- **RoCE** (RDMA over Converged Ethernet)

- **iWARP** (Internet Wide Area RDMA Protocol)
- **InfiniBand**

While all three share a common user API (RDMA Verbs), they differ in their physical and link layers.

In modern data centers, RoCE is widely used to enable RDMA over Ethernet. It achieves this by replacing the link and physical layers of InfiniBand while keeping the upper layers unchanged. On an Ethernet network, RoCE traffic is identified using the EtherType `0x8915`, allowing high-speed, low-latency communication without the overhead of traditional TCP/IP networking.

RoCEv1 operates only at Layer 2, limiting it to a single Ethernet broadcast domain. RoCEv2 adds UDP encapsulation, which allows RDMA traffic to traverse Layer 3 networks, making it scalable and routable over both IPv4 and IPv6. Despite these advantages, RoCE depends on a lossless network, requiring careful configuration of congestion control mechanisms in data center networks.

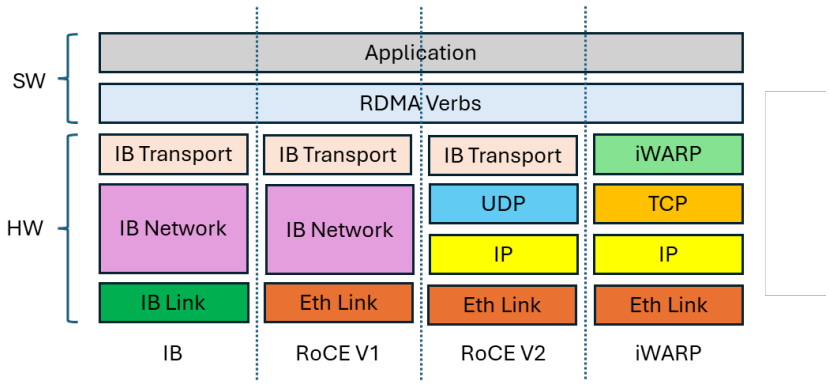

Figure 2.7: RDMA transport layer

2.3.4 RDMA objects

When an application interacts with RDMA, it communicates with an RNIC using a specialized "*verbs*" API. To efficiently manage data transfers, RDMA relies on several key objects. Although there are many components involved, the most important are queue pairs (QP), completion queues (CQ), and memory regions (MR).

Queue pairs (QP)

At the core of RDMA communication lies the concept of the *Queue Pair* (QP), which serves as a dedicated communication channel between two applications.

Whenever an application intends to send or receive data using RDMA, it performs these operations through a QP.

Unlike traditional networking, which relies heavily on CPU interrupts and kernel involvement for sending and receiving data, RDMA allows applications to interact directly with memory by posting operations to these queues. This direct memory access eliminates much of the overhead associated with conventional network communication.

Each queue pair consists of two primary components:

- **Send Queue (SQ):** This queue is used for outgoing operations such as Read, Write, and Send.
- **Receive Queue (RQ):** This queue handles incoming data from a remote application. Operation like receive are posted on this queue.

Queue Pairs (QPs) accept only specific types of data to be posted. In fact, within each QP, operations are described using *Work Queue Elements* (WQEs). A WQE, also referred to as a *Work Request* (WR), serves as a descriptor for a particular RDMA operation and contains essential information such as:

- `Opcode`: the type of operation (e.g., Read, Write, Send);
- `Remote_addr`: pointer to the location in the remote memory where the data will be placed.
- `R_key`: remote key of the Memory Region registered in the other side that will be used to store the data.

In addition, a Work Request (WR) must be linked to the local source of the data that will be transmitted. This is specified through the Scatter-Gather field, which is used to provide an SGE object. The SGE contains:

- `Address`: pointer to the local buffer that store the data to transfer
- `Size`: Size of the data that will be read from the address and sent.
- `L_key`: local key of a Memory Region that was registered and that store the buffer

Each Work Request (WR) can contain one or more Scatter-Gather Entries (SGEs), with the maximum number determined by the RNIC.

WRs can also be linked together to form a batch, reducing the number of calls required for posting operations.

An application may maintain multiple Queue Pairs (QPs) at the same time, enabling parallel processing of different data streams. This is especially useful in high-performance computing (HPC) clusters, where many applications often communicate concurrently. By creating multiple QPs, an application can efficiently manage separate traffic channels instead of relying on a single queue for all communication.

By leveraging multiple QPs and carefully managing WQEs, RDMA systems can handle large-scale parallel workloads efficiently with minimal CPU involvement. This architecture makes RDMA particularly well suited for high-speed, low-latency networking in demanding computational environments.

Completion queue (CQ)

In RDMA, once an application posts a work request to a QP, it needs a mechanism to determine when the operation has completed. This role is fulfilled by *Completion Queues* (CQs), which serve as a notification system for finished work requests.

A CQ allows an application to avoid constantly checking whether a work request, represented by a Work Queue Element (WQE), has completed. Instead, the application can monitor the CQ and be notified when the RNIC signals that an operation is done. This reduces unnecessary CPU usage and allows more efficient handling of network operations.

- SQ and RQ must be associated with a CQ. However, this is not necessarily a strict one-to-one mapping: a single CQ can handle completions from multiple SQs and RQs.
- When an operation completes, the RNIC writes a *completion entry* into the associated CQ.
- The application then processes the CQ to determine which operations have finished and take appropriate action.

There are two primary methods by which an application can process completions from a CQ:

- **Polling Mode:** The application continuously checks the CQ for completed operations. This approach delivers low latency and high throughput but is CPU-intensive, as the processor constantly consumes cycles to monitor the queue.
- **Interrupt-Based Mode:** The RNIC interrupts the CPU only when a completion occurs, reducing CPU usage. However, this method introduces slightly higher latency, since the CPU must handle the interrupt before processing the completed operation.

By carefully selecting between polling and interrupt modes, applications can optimize performance according to their requirements. For ultra-low latency tasks, such as high-frequency trading, polling is often preferred. For workloads where power efficiency is more critical, interrupt-based processing is typically the better choice.

Memory region (MR)

A key aspect of RDMA high performance is its ability to read from and write directly to remote memory. To safely control access to memory areas, RDMA uses the concept of *Memory Regions* (MRs). Before an application can perform RDMA operations on a particular memory area, it must first register that region with the RDMA-capable network interface card (RNIC). This registration process effectively "pins" the memory, ensuring that the operating system does not move it during data transfers, which could otherwise lead to errors.

When a memory region is registered, the RNIC provides two keys that govern access to the memory:

- **L-KEY (Local Key):** Used by the local application to access the registered memory.
- **R-KEY (Remote Key):** Used by a remote application to read from or write to this memory region.

Whenever a Work Queue Element (WQE) is posted, it specifies the memory locations involved in the operation through a SGE list. SGEs allow applications to read from or write to multiple non-contiguous memory regions in a single operation. This capability makes RDMA highly efficient when handling complex data structures or large datasets.

Protection domains (PD)

To ensure secure and controlled access to memory and other RDMA resources, RDMA introduces the concept of *Protection Domains* (PDs). A PD acts as a security container that groups related RDMA objects—such as Queue Pairs (QPs) and Memory Regions (MRs)—together.

When an RDMA operation is initiated, the RNIC verifies that the QP and MR involved belong to the same PD. If they do not, the operation fails with an error. This enforcement guarantees that only authorized components can interact with each other, preventing unintended or malicious access to memory.

By logically separating RDMA resources into protection domains, multiple applications or processes can safely share the same system without interfering with one another, maintaining both security and operational integrity.

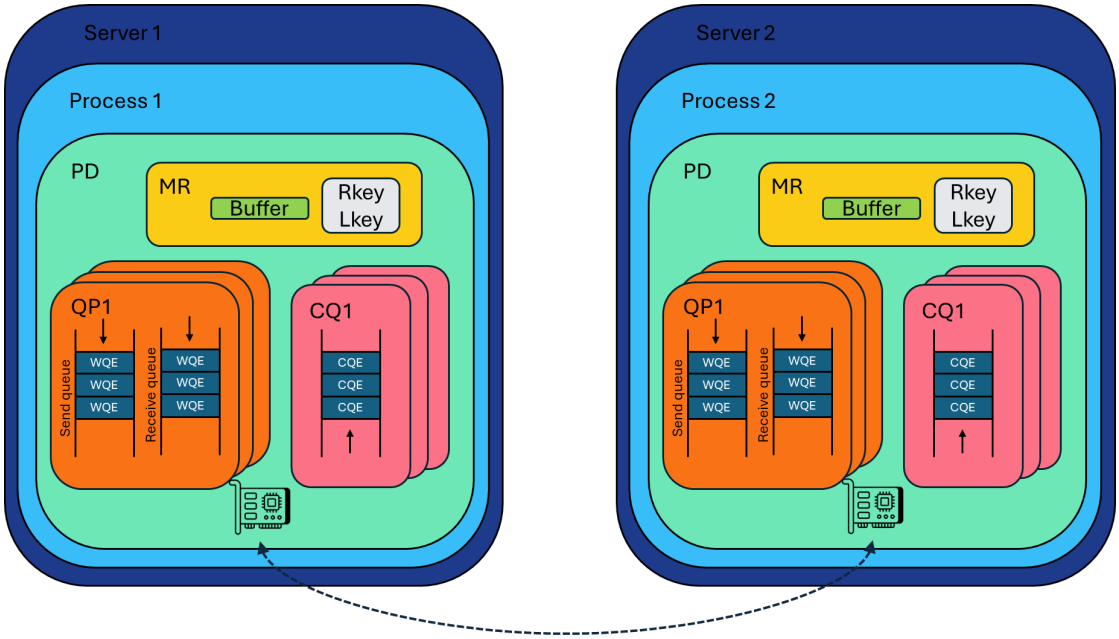

Figure 2.8: RDMA complete setup

Shared receive queue (SRQ)

In a standard RDMA setup, each Queue Pair (QP) maintains its own Receive Queue (RQ), which requires a dedicated set of memory buffers for incoming messages. While this design works well in many scenarios, it can be inefficient when multiple QPs handle similar types of messages, as memory buffers are duplicated unnecessarily.

To address this issue, RDMA provides the concept of a *Shared Receive Queue* (SRQ). An SRQ allows multiple QPs to share a single RQ, centralizing the management of incoming messages and memory buffers.

The primary benefits of using SRQs include:

- **Reduced memory consumption:** By sharing buffers across multiple QPs, SRQs eliminate the need to duplicate memory for each individual queue.

- **Simplified management:** With fewer receive buffers to maintain, applications can manage incoming data more efficiently and with less overhead.

By enabling multiple QPs to use a shared pool of receive buffers, SRQs improve memory utilization and simplify the overall management of incoming messages, making RDMA communication more efficient in high-performance environments.

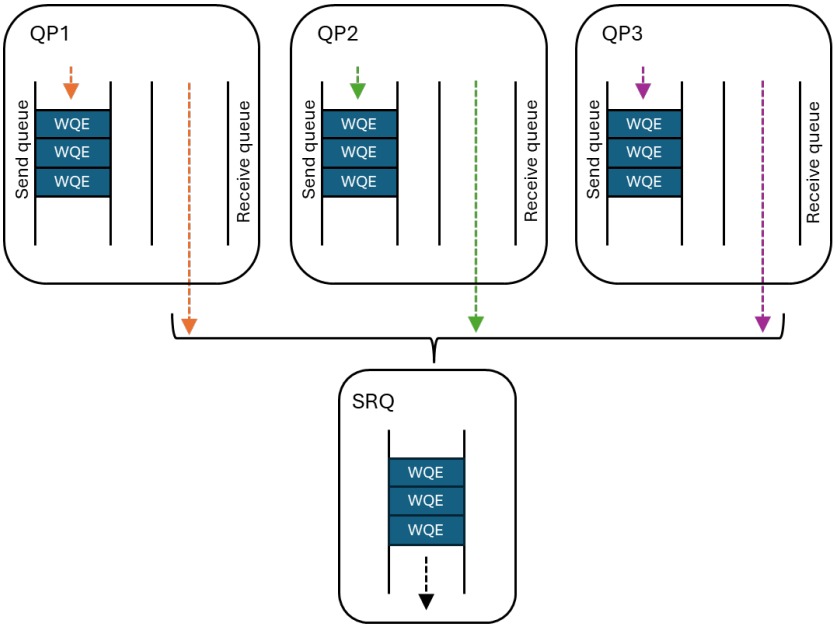

Figure 2.9: Shared Receive Queue

summary

Object	Purpose
QP	Handles RDMA operations (Read, Write, Send, Atomic). Contains SQ and RQ.
CQ	Notifies applications when an operation is complete. Supports polling and interrupt modes.
MR	Registered memory for RDMA transfers. Requires L-KEY and R-KEY.
PD	Group of RDMA objects to ensure valid access.
SRQ	Allows multiple QPs to share a single RQ for better memory efficiency.

Address vector (AV)

Another important component in RDMA is the Address Vector (AV). While it is hidden from the developer's point of view, it plays a crucial role in ensuring that packets are routed correctly. An AV describes the route from the local node to the remote node. In every UC or RC Queue Pair, an AV is included in the QP context, whereas in a UD QP, the AV must be specified for each posted Send Request. Address vectors are implemented using `struct ibv_ah`.

Global routing header (GRH)

The Global Routing Header (GRH) is used for routing between subnets. When using RoCE, the GRH handles routing within the subnet and is therefore mandatory.

For Unreliable Datagram (UD) Queue Pairs (QPs) with global routing, a GRH occupies the first 40 bytes of the receive buffer. This space stores global routing information, allowing the receiver to generate a correct Address Vector to be used as response path to the received packet. GRHs are implemented using `struct ibv_grh`.

2.3.5 Connection management in RDMA

RDMA supports various types of Queue Pairs (QPs) depending on the level of reliability required for data transmission. These connection types provide flexibility for different application needs:

- **Reliable Connection (RC):** Ensures ordered and reliable delivery of data, similar to TCP. This type is suitable for critical applications that require guaranteed data integrity.
- **Unreliable Connection (UC):** Offers similar functionality to RC but without retransmission. Lost packets are not recovered, which may be acceptable for applications that can tolerate occasional data loss.
- **Reliable Datagram (RD):** Combines the reliability of RC with datagram-based communication, providing ordered and reliable delivery in a datagram-oriented model.
- **Unreliable Datagram (UD):** Provides connectionless, UDP-like communication. UD only supports Send operations and does not allow Read or Write operations, making it suitable for lightweight messaging scenarios.

Typically, RDMA connections are established *out-of-band* (OOB), meaning that the connection setup occurs separately from the actual data transfer, allowing applications to use their preferred mechanisms, such as TCP sockets, to exchange the necessary setup information before RDMA operations begin.

2.3.6 RDMA verbs

RDMA verbs define the set of operations that an application can use to interact with an RDMA-capable Network Interface Card (RNIC). Conceptually, these verbs act as low-level instructions, directing the RNIC on how to carry out specific tasks,

similar to how application programming interfaces (APIs) provide structured access to software functions.

The operating system exposes RDMA verbs through a dedicated API; however, not all RNICs support the full range of verbs. The actual capabilities depend on the hardware and the corresponding driver implementation. One of the key advantages of RDMA is that the same set of verbs can be used across multiple wire protocols, including InfiniBand (IB), RDMA over Converged Ethernet (RoCE), and iWARP. This universality enables applications to run on different RDMA implementations with minimal changes.

In general, RDMA verbs can be categorized into two distinct groups:

- **Slow-Path Verbs (Control Operations):** These verbs are used for setting up and managing RDMA resources.
- **Fast-Path Verbs (Data Operations):** These verbs handle the actual transfer of data and other high-speed memory operations.

Slow-path verbs: managing RDMA resources

Slow-path verbs correspond to control operations that are primarily responsible for configuring RDMA resources, such as Queue Pairs (QP), Completion Queues (CQ), and Memory Regions (MR). Since these operations involve the allocation, modification, or deallocation of system resources, they typically require privileged access and interact with the RDMA driver in the operating system kernel.

RDMA Component	**Operations**	**Purpose**
Device	Open / Close Device	Initializes or shuts down an RNIC device.
Protection Domain (PD)	Allocate / Deallocate	Creates a logical grouping of RDMA objects (QPs, MRs, etc.).
Memory Region (MR)	Register / Deregister	Registers memory so the RNIC can access it.
Queue Pair (QP)	Create / Destroy / Modify	Establishes and configures communication between peers.
Completion Queue (CQ)	Create / Resize / Destroy	Manages work completion notifications.
Shared Receive Queue (SRQ)	Create / Resize / Destroy	Allows multiple QPs to share receive buffers.

Table 2.1: RDMA Components, Operations, and Purpose

Example Use Case: Before an application can send or receive data over RDMA, it must perform several initialization steps:

1. Register memory (MR) so that the RNIC can directly access it.
2. Create a queue pair (QP) to establish communication with a remote application.
3. Set up a completion queue (CQ) to track completed operations.

All of these steps rely on slow-path verbs because they involve managing resources at the OS or driver level.

Fast-path verbs: high-speed data operations

Once RDMA resources have been configured, applications require a mechanism for fast and efficient data transfer. Fast-path verbs provide this capability by allowing user-space applications to interact directly with the RNIC, bypassing the operating system kernel. This bypass is a key factor in RDMA high performance, as it significantly reduces both latency and CPU overhead during data transfers.

Operation Type	**Description**
Send Operation	Sends data to a remote QP. The receiver must have already posted a receive request.
Receive Operation	Receives incoming data. The application must post a receive request before the sender sends data.
RDMA Write	Writes data directly into a remote memory location. The remote application is not notified.
RDMA Read	Reads data from a remote memory location. The remote application is not notified.
Atomic Operations	Performs atomic memory updates, such as fetch-and-add or compare-and-swap.
Memory Operations	Includes binding memory windows, fast memory registration, and local invalidation.

Table 2.2: RDMA Operation Types and Descriptions

Send vs. RDMA Write:

- **Send/Receive:** Requires the receiver to post a receive request before the sender transmits data.
- **RDMA Write:** Does not require a matching receive request, as data is directly written into the target's memory.

Optimizing memory operations

Memory registration, a slow-path operation, can be expensive because it involves pinning physical memory pages. RDMA provides two key optimizations to mitigate this overhead:

- **Memory Windows (MW):** A Memory Window (MW) gives applications more flexible control over remote access to local memory. It ca be used when an application needs to grant and revoke remote access rights to a registered region dynamically, without incurring the performance penalty of deregistration/registration or reregistration or want to grant different remote access besed on the remote agent. Different MWs can overlap the same MR (event with different access permissions).
- **Fast Memory Registration (Fast Register MR):** allows an application to expose a set of contiguous memory locations to the network adapter using virtual addresses. Registered memory pages are pinned to preserve their physical–virtual mapping, and permissions such as local write, remote read/write, atomic, and bind are enforced. Each MR has a local key (`l_key`) for local access and a remote key (`r_key`) for remote RDMA operations. A memory buffer can be registered multiple times with different permissions, generating distinct keys for each registration.

2.3.7 Communication setup

The structure of a typical RDMA application can be summarized as follows:

1. **Get the device list:** Retrieve the list of available InfiniBand (IB) devices on the local host. Each device includes a name and a GUID (e.g. `mlx4_1`).
2. **Open the requested device:** Iterate over the device list, select a device by name or GUID, and open it.
3. **Allocate a Protection Domain (PD):** A PD restricts which components (AH, QP, MR, MW, SRQ) can interact with each other.
4. **Register a memory region (MR):** Only registered memory can be used in RDMA operations. Set memory permissions and obtain local and remote keys (`lkey`/`rkey`) to refer to this buffer.
5. **Create a Completion Queue (CQ):** The CQ holds completed Work Requests (WRs).
6. **Create a Queue Pair (QP):** Creating a QP also creates an associated Send Queue and Receive Queue.

7. **Bring up a QP:** Transition the QP through several states until it reaches Ready To Send (RTS), providing the information necessary for sending and receiving data.
8. **Post Work Requests and poll for completion:** Use the QP for communication operations.
9. **Cleanup:** Destroy resources in reverse order of creation: delete QP, delete CQ, deregister MR, deallocate PD, close device

2.3.8 Congestion control mechanisms

Effective congestion control is essential for maintaining low-latency, high-bandwidth performance in RoCE networks. Common mechanisms include:

- **PFC (Priority Flow Control) - Layer 2:** Pauses specific traffic flows during congestion using PAUSE frames. Prevents packet loss but may cause head-of-line blocking; careful tuning is required.
- **DCQCN (Data Center Quantized Congestion Notification) - Layer 3:** Uses ECN to signal congestion and dynamically adjusts traffic rates. Allows critical traffic to continue and is effective for NVMe-oF storage systems.
- **RoCC (Robust Congestion Control) - Switch-Based:** Monitors switch queue sizes and adjusts flow rates for fairness and efficiency, preventing buffer overflows at the switch level.

Chapter 3

Architecture

3.1 Overview

To address the challenge of replacing TCP with RDMA, an application has been developed. Its core functionality is based on intercepting messages traveling over sockets. This operation is not possible using conventional approaches, so it is necessary to operate inside the kernel. Since working directly in the kernel is typically complex and reduces portability, we rely on the eBPF framework (Section 2.2). This allows us to inject programs directly into the kernel and capture data from sockets in a safe and portable way.

The application is composed of two interacting components that cooperate to guarantee the correct functioning of the system. The first component (eBPF traffic interceptor) is implemented with eBPF and runs inside the kernel. Its main task is to intercept messages generated by the original application and forward them to the second component (and vice-versa). The second component (TCP/RDMA proxy) is then responsible for delivering these messages to the remote endpoint using RDMA.

On the remote side, where the same application is running, the data is received through RDMA by the TCP/RDMA proxy unit and is sent to the kernel. Once again, eBPF is employed to redirect the messages to their intended final destination, thereby completing the communication workflow. It is important to emphasize that both the original sender and the original receiver remain completely unaware of the underlying architecture.

3.2 Kernelspace

Kernelspace is the domain of eBPF program. A crucial aspect concerns the communication between the two modules. At startup, the application loads the

eBPF program into the kernel and creates a pool of spare sockets prepared to receive data. These sockets serve as a bridge between kernelspace and userspace: the eBPF program intercepts data from the original sockets and redirects it to one of the available proxy sockets. Symmetrically, to ensure correct operation in the reverse direction, eBPF also manages the redirection of messages from the proxy sockets back to the original destination sockets. This process is explained in Figure 3.1

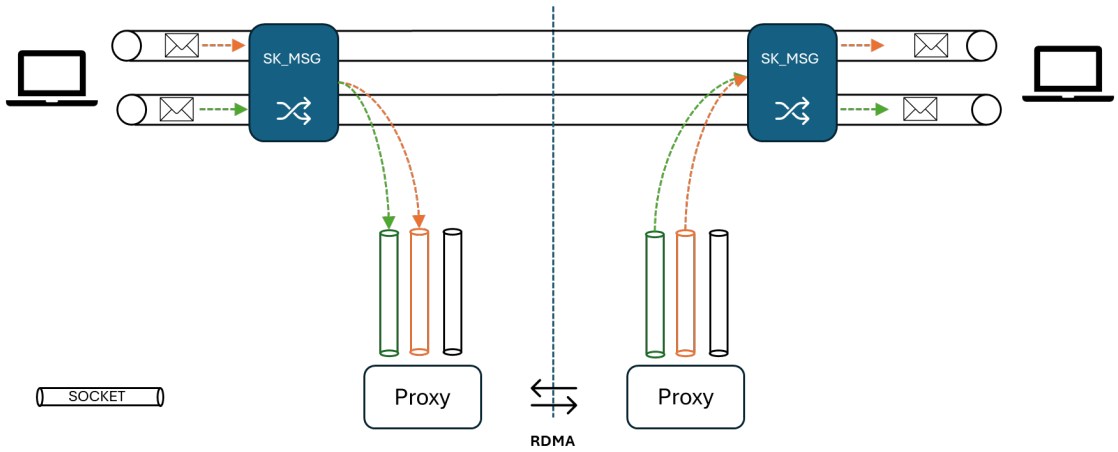

Figure 3.1: Kernelspace and userspace communication

Inside the eBPF traffic interceptor unit, three hooks drive the mechanism:

- `SOCKOPS`: Invoked during various socket operations, of which socket creation is the most relevant for our use case. Upon creation, `sockops` associates the new socket with one from the pool of sockets previously allocated and loaded by the userspace module.

- `SK_MSG`: Triggered whenever a message is sent through a socket. Its role is to redirect messages: if the message originates from the original application, it is forwarded to the proxy; conversely, if the proxy sends the message, it is redirected back to the original application.

- `TCP_DESTROY`: Invoked upon socket closure. This hook releases the association previously established by the `sockops` hook. it is attached to the trace point: *tracepoint/tcp/tcp_destroy_sock*

By combining these hooks, the system can detect sockets that need to be intercepted and redirect their messages directly to the proxy, enabling efficient kernel-to-user space communication.

3.3 Userspace

The user-space program constitutes the second major component driving the mechanism. Its primary responsibilities are to establish RDMA connections between peers that wish to communicate and to manage the transmission and reception of messages.

To enhance performance, the TCP/RDMA proxy unit leverages multi-threading, allowing different tasks to be carried out concurrently. In particular, it relies on six dedicated background threads, each serving a specific purpose:

- **Server thread**: Runs in the background, waiting for incoming connection requests. When a request arrives, it accepts the connection and initiates the entire `RDMA` handshake procedure required before data transmission can begin. Once the handshake is complete, the thread returns to waiting for new requests.

- **Writer thread**: Waits for data arriving from the proxy socket (i.e., data redirected by the `SK_MSG` program). When new data is received, the Writer places it into a dedicated buffer allocated within the `RDMA` memory region (Section 2.3.4).

- **Flush thread**: Works in tandem with the Writer to complete the data transmission pipeline. It continuously monitors how much data has been written into the buffer, and once a certain threshold (or timeout) is reached, it sends the data using an `RDMA WRITE` operation over one of the queue pairs (QP (Section 2.3.4)) allocated for the connection.

- **Reader thread**: Complements the Writer. It waits for new data to arrive in the buffer on the receiving side. When data is available, the Reader passes it back into the kernel, where the `SK_MSG` program redirects it to the correct destination socket.

- **Index thread**: Completes the work of the reader threads by updating the remote index, allowing the sender to determine how much space is available in the buffer.

- **Notification thread**: Balances efficiency and responsiveness by dynamically switching between two modes: *send/receive* (notification-based, low CPU cost) and *read/write* (polling-based, high responsiveness). Under light traffic, notifications are preferred; when traffic increases, the system switches to polling. Idle periods revert back to notifications. The Notification thread is responsible for listening to these events and deciding when to switch modes.

This way, the TCP/RDMA proxy unit ensures efficient and scalable management of communication within userspace, balancing responsiveness with resource utilization.

3.4 Application Logic

3.4.1 New Connection

When a new socket is created between two applications, the `SOCKOPS` subsystem begins its operation. It first checks whether the newly created socket should be intercepted. If the socket is not meant to be intercepted, it is ignored. Otherwise, it is added to the list of intercepted sockets. Once the socket has been added to this list, `SOCKOPS` notifies the userspace about the creation of the new socket adding the related information into the shared ring buffer.

Upon receiving the notification, in the userspace the function associated with the ring buffer is triggered. At this point, only the client node (identified by the `BPF_SOCK_OPS_ACTIVE_ESTABLISHED_CB` callback) is responsible for initiating the RDMA connection with the other peer and will act as a client.

The RDMA setup phase proceeds over TCP, during which the two peers exchange all required RDMA configuration parameters (such as `QPN`, `LID`, `r_key`, and `remote_addr`). This exchange ensures that both endpoints possess the necessary information to instantiate their RDMA contexts.

Once the RDMA context has been successfully created, direct RDMA-based communication can commence, enabling low-latency and efficient data transfer between peers.

3.4.2 Message Trip

1: Kernelspace sender

Since the socket has been added to the intercepted socket map, the messages travelling on it will trigger the execution of the hook `SK_MSG`. This hook plays a central role in message redirection. Upon interception, the hook examines the destination port to determine whether the message should be forwarded to the proxy or delivered directly to the original application. Depending on the communication direction, the hook performs a lookup in a dedicated map to identify the correct forwarding path. Once the appropriate association is identified, the message is redirected along the corresponding socket path using the `bpf_msg_redirect()` helper. This ensures that messages are transparently redirected without requiring changes in the application logic.

2: Userspace

We now examine in detail how the architecture transfers data from one side to the other. The process unfolds through a coordinated sequence of steps, involving multiple threads within the TCP/RDMA proxy unit:

1. When a message is redirected to one of the proxy sockets by the `SK_MSG` hook, the **writer thread**—which is blocked on a `select()` call over the corresponding file descriptor—is triggered.

2. Upon notification, the writer thread retrieves the incoming data and encapsulates it into a message object. This message is then stored in the ring buffer (inside the MR), making it ready for transmission through RDMA.

3. Once the message has been placed into the ring buffer, responsibility shifts to the **flush thread**. Running periodically, the flush thread is waiting for data to transfer to be placed into the RDMA buffer. When some data is found, it performs an RDMA write operation to transfer the content directly into the remote buffer. This mechanism bypasses both the CPU and kernel on the sending and receiving sides, ensuring high efficiency.

4. Assuming this is the first communication performed by the proxy after startup, the receiving peer operates in `send/recv` mode rather than through active polling. As a result, it must be explicitly notified whenever new data becomes available. To handle this, the flush thread sends a notification to the remote side.

5. On the receiving end, the **notification thread** is awakened from its sleep upon detecting the incoming notification via the RDMA `send/recv` mechanism. After parsing the message, it signals the **reader thread** that new data is ready to be consumed.

6. The **reader thread** starts and begins consuming the data stored in the buffer. Each message is parsed, the corresponding proxy socket is identified, and the message is then forwarded to the kernel through that socket.

7. Finally, once the reader thread has processed all the incoming messages, the system updates the read index on the remote side. This step is essential, as it marks portions of the shared buffer as consumed, thereby freeing up space for the sender to store subsequent messages.

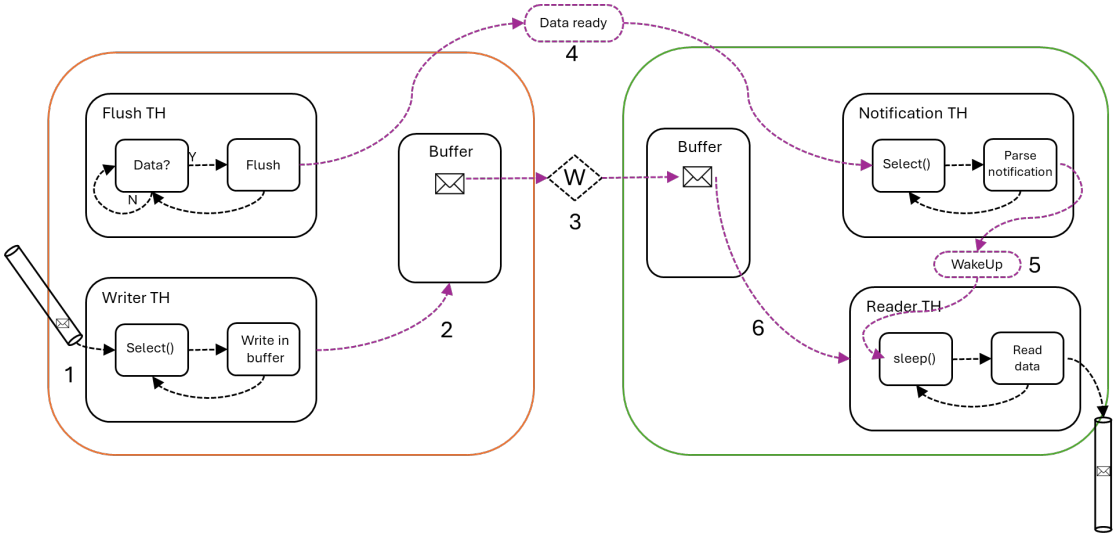

Figure 3.2: Message trip inside the application

3: Kernelspace receiver

On the receiving side, the process is symmetrical. When eBPF (specifically the `SK_MSG` hook) detects a message arriving from a proxy socket (hence sent by a reader thread), it begins to process it. Since the intercepted message originates from a proxy socket, the hook consults the association map to identify the corresponding destination socket, namely the actual application socket.

After locating the correct association, the message is redirected to the intended receiver socket, ensuring that the data reaches the target application transparently. In this way, the redirection mechanism remains invisible to the application layer while maintaining control and flexibility at the system level.

3.4.3 Event-Driven and Polling Modes

As discussed in the background chapter (Section 2.3.6), RDMA supports two communication modes: *send/receive* and *read/write*. Besides their differences in resource usage and implementation, the most relevant aspect here is the way they handle signaling. The *send/receive* mode can generate notifications, while the *read/write* mode simply updates memory and requires polling to detect changes.

To take advantage of both approaches, the application adopts a hybrid strategy that combines event-driven notifications with polling. At startup, the system operates in *send/receive* mode. In this phase, the sender issues a notification after writing data, and the receiver, which is blocked on a `select()`, is awakened to process it. This notification-based mechanism is efficient in terms of CPU usage,

making it well-suited for periods of low traffic.

When traffic increases, however, the overhead of handling frequent notifications becomes significant. To address this, the receiver monitors the number of notifications arriving in a given time window. If the threshold is exceeded, the system switches to polling mode. In this mode, the receiver actively checks the buffer waiting for new data to consume written by the sender, enabling faster response times at the cost of higher CPU usage.

The system does not remain in polling mode indefinitely. If no new updates are detected for a certain period, the receiver transitions back to *send/receive* mode. In this way, the application balances efficiency and performance: notifications handle low-traffic scenarios with minimal overhead, while polling ensures responsiveness under heavy load. Note that the data are always transferred using write operation, the only difference is how to notify the receivers that there are new data to consume.

The application has been developed to operate effectively under this trade-off. However, in order to maximize throughput and minimize latency, power consumption reduction has been relegated to a secondary priority.

Nonetheless, this functionality is still present in the final software, although it remains unused.

Chapter 4

Implementation

4.1 Memory region and messages layout

4.1.1 Socket in user and kernelspace

The entire project is based on communication between kernelspace and userspace in the context of socket handling. In user space, a socket is identified using a file descriptor. In kernel space, however, a socket is represented solely by its corresponding `struct`. The file descriptor itself serves as an index in the process's file descriptor table, linking the descriptor to the underlying socket.

Since this application relies on interaction between kernelspace and userspace, a common approach was needed to bridge this gap. To address this limitation, a dedicated structure called `sock_id_t` was introduced. This structure stores the information required to uniquely identify a socket: source IP, destination IP, source port, and destination port.

```
1 typedef struct sock_id sock_id_t;
2 struct sock_id {
3   __u32 sip;   // stored in NET order
4   __u32 dip;   // stored in NET order
5   __u16 sport; // stored in HOST byte
6   __u16 dport; // stored in HOST byte
7 };
```

Listing 4.1: `sock_id_t` struct

It is also important to note that when two peers open a socket, they will see the source and destination inverted because each side views the connection from its own perspective. From the kernel's point of view, the "source" is always the local endpoint, and the "destination" is the remote endpoint. Therefore, what one peer considers the source IP and port, the other peer sees as the destination IP

and port, and vice versa.

For example, if Peer A connects from `10.0.0.1:5000` to Peer B at `10.0.0.2:80` Peer A sees:

- Source: `10.0.0.1:5000`
- Destination: `10.0.0.2:80`

While Peer B sees:

- Source: `10.0.0.2:80`
- Destination: `10.0.0.1:5000`

This inversion is crucial when using the `sock_id_t` structure, as it ensures that a socket can be uniquely identified regardless of whether the perspective is local or remote. By storing both source and destination IPs and ports, the structure provides a consistent representation of the connection that can be recognized from either side.

From now, `sock_id` will be used as a reference to this struct.

4.1.2 Message structure

Since each message that needs to be transferred is sent via sockets, the receiver must know not only the data but also which socket originated the message. Each message consists of several fields:

- `seq_number_header`: A number that uniquely identifies each message and strictly increases with every new message. It is used by the receiver to detect newly arrived messages.
- `msg_flags`: Intended to annotate the message with additional information. This feature is not yet implemented but serves as a placeholder for future extensions.
- `original_sk_id`: Specifies the `sock_id` of the original socket responsible for the communication.
- `msg_size`: Indicates the size of the message in bytes.
- `number_of_slots`: Denotes how many slots in the ring buffer the message occupies (default is 1).
- `msg`: An array of bytes that stores the actual message payload.

- `seq_number_tail`: Similar in purpose to `seq_number_header`, this field ensures that the entire message has been internally transferred, not just its first part.

```
typedef struct {
    uint32_t seq_number_head;
    uint32_t msg_flags;
    struct sock_id original_sk_id;      // id of the socket
    uint32_t msg_size;
    uint32_t number_of_slots;
    char msg[Config::MAX_PAYLOAD_SIZE]; // message
    uint32_t seq_number_tail;
} rdma_msg_t;
```

Listing 4.2: Message data structure

The `seq_number_tail` is a critical component because, once data is posted over RDMA, the segmentation of the message during transmission is not known in advance. By examining the `seq_number_tail`, the receiver can determine when a message has been completely transferred. Only after confirming that the tail sequence matches the head sequence and that both correspond to the actual sequence number can the receiver safely consider the message fully received and forward it to its final destination.

4.1.3 Memory design

This application uses RDMA to exchange data between peers. To enable this, a memory region (MR) must be allocated by both peers. As explained in a previous chapter (Section 2.3.4), an MR is a section of memory that supports remote operations such as read and write. Naturally, this memory must be designed according to specific criteria to ensure proper communication. In practice, messages are written to the local memory region of one peer and then copied to the remote memory region using RDMA write operations.

To preserve message order, a ring buffer structure is used to manage data transfer. The ring buffer size can be specified in the configuration, and a small portion of the buffer is always left empty to distinguish between full and empty states.

The memory region is a contiguous block of memory, logically divided into three main parts: the `Notification area`, `ringbuffer1`, and `ringbuffer2`. This double-ring-buffer design allows both peers to write and read simultaneously. Specifically, `ringbuffer1` is used by the client to write and the server to read, while `ringbuffer2` is used in the opposite direction (server writes, client reads).

A notification space is also included to manage the send/receive mechanism of RDMA (Section 2.3.6), allowing the system to move from polling-based to event-driven operation for better resource management.

Each ring buffer is composed of the following elements:

- `rdma_flag_t`: indicates the current status of the peer. For instance, it can be used to know if the peer is in polling or notification mode.
- `remote_read_index`: tracks the position reached by the remote peer when reading, indicating which spaces are free for reuse without overwriting data.
- `local_write_index`: tracks the position reached by the sender when writing data.
- `local_read_index`: tracks the position reached by the receiver when writing data.
- `buffer`: the array of `rdma_msg_t` elements that stores the actual messages.

The transmitter inserts messages into the buffer, each with an increasing sequence number. The receiver waits for the expected sequence number and parses the message. Upon completing parsing, it is crucial that the receiver updates the `remote_read_index` to notify the transmitter that the data have been consumed and the space can be reused. Without this update, the transmitter may stall, as it would perceive the buffer as full.

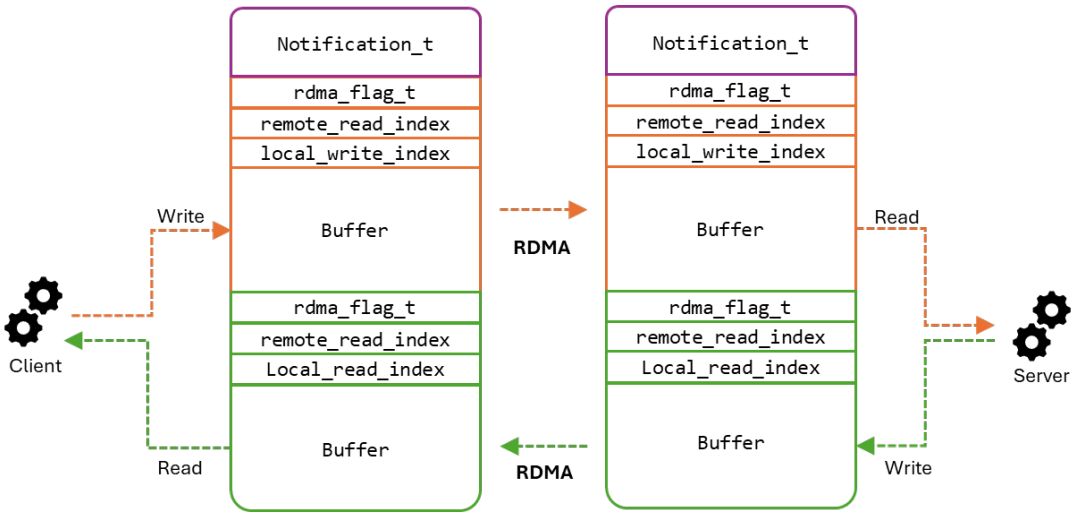

Figure 4.1: Memory layout schema

```
typedef struct {
    rdma_flag_t flags;
    uint32_t local_read_index;
    std::atomic<uint32_t> remote_read_index;
    uint32_t local_write_index;
    rdma_msg_t data[Config::MAX_MSG_BUFFER];
} rdma_ringbuffer_t;
```

Listing 4.3: Ringbuffer structure

As discussed earlier, the *notification area* is responsible for storing the notifications exchanged between peers. This area is further divided into two parts: one dedicated to messages sent from the server to the client, and the other dedicated to messages sent from the client to the server. A notification itself is represented by a simple code. When a peer receives a notification, it retrieves the corresponding code from the notification area and performs the appropriate action based on its meaning.

```
enum class CommunicationCode : int32_t {
    RDMA_DATA_READY = 10,
    RDMA_CLOSE_CONTEXT = 5,
    NONE = -1
};

typedef struct {
    CommunicationCode code; // code of the notification
} notification_data_t;

typedef struct {
    notification_data_t from_server; // notification from server
    notification_data_t from_client; // notification from client
} notification_t;
```

Listing 4.4: Notification structure

4.2 KernelSpace: eBPF

This component is designed to operate within the eBPF framework, running in kernel mode. It is composed of three main blocks: `SOCKOPS`, `SK_MSG`, and `TCP_DESTROY`, and it utilizes various maps to store data in a persistent way and share it with userspace.

In order to redirect messages to userspace, the application creates multiple sockets (proxy sockets), which are then pushed to the eBPF program into a specific map. When a new socket is created, it is associated with one of the sockets exposed by the application in userspace granting the possibility to redirect messages.

4.2.1 Maps

As discussed in Section 2.2, eBPF does not allow data to be stored directly within the program itself; instead, it requires the use of specific data structures provided by the kernel, called maps (see Section 2.2.1). Our eBPF program leverages several different maps to facilitate message redirection:

- `new_sk`: Implemented as a `BPF_MAP_TYPE_RINGBUF`, this map is used to notifies userspace whenever a new socket is intercepted. Each notification includes the socket data (source IP, destination IP, source port, and destination port) as well as the operation type, which is used to indicate whether the peer will act as a client or server in subsequent phases.
- `intercepted_sockets`: A `BPF_MAP_TYPE_SOCKHASH` map that stores all sockets whose messages are to be intercepted by `SK_MSG`. Only sockets inserted into this map will have their messages intercepted.
- `free_sockets`: A `BPF_MAP_TYPE_QUEUE` map that maintains a list of available sockets. During the startup phase, userspace populates this map with all pre-created sockets (proxy sockets) that will be used to receive messages.
- `socket_association`: A `BPF_MAP_TYPE_HASH` map that tracks associations between proxy sockets and application sockets. It is used by `SK_MSG` to determine the destination of each intercepted message. Each entry is a key-value pair where both the key and value are `sock_id` structures (including source IP, destination IP, source port, and destination port). Whenever a new association is inserted, a corresponding reverse entry is also added to handle message directionality correctly.
- `target_port`: A `BPF_MAP_TYPE_HASH` map storing ports that need to be intercepted. When a new socket is created, its destination and source ports are checked against this map. The map uses a key-value structure where the key is the port number and the value is an integer indicating whether interception is required.
- `target_ip`: A `BPF_MAP_TYPE_HASH` map storing IP addresses that need to be intercepted. Each time a new socket is created, its destination IP is compared against this map. Similar to `target_port`, the key is the IP address and the value indicates whether interception is required.
- `server_port`: A `BPF_MAP_TYPE_HASH` map storing the server port information. The key is always 0, and the value is the port number where the proxy is listening. This map is used by `SK_MSG` to determine the direction of the message flow.

4.2.2 SOCKOPS

Socket operations (SOCKOPS) programs are attached to cGroups and are triggered by various socket lifecycle events. This allows the program to modify connection settings or record the existence of a socket as needed.

In our application, the SOCKOPS program intercepts sockets during their creation, responding to specific events of interest. For the purposes of this system, we focus on two particular events, denoted as BPF_SOCK_OPS_PASSIVE_ESTABLISHED_CB and BPF_SOCK_OPS_ACTIVE_ESTABLISHED_CB. Due to the client-server nature of socket creation, one side will receive event BPF_SOCK_OPS_ACTIVE_ESTABLISHED_CB while the other side will receive BPF_SOCK_OPS_PASSIVE_ESTABLISHED_CB.

All other operations that may be triggered during the lifetime of the socket will be ignored, as we are currently not concerned with their management.

Upon intercepting a socket, the SOCKOPS program performs several actions:

1. **Determine if the socket is a target:** The program checks whether the socket should be intercepted by consulting the target_port and target_ip maps. If the socket does not belong to the target application, it is ignored.

2. **Add the socket to intercepted sockets:** The socket is added to the intercepted_sockets map, enabling SK_MSG to intercept its messages.

3. **Retrieve a free proxy socket:** one of the available proxy sockets is obtained from the free_sockets map, and will be used to forward data to userspace. At this stage, it is crucial that the userspace allocates a sufficient number of sockets to handle the traffic. This value can be manually tuned through the application's configuration. If no sockets are available in the queue, the socket is skipped.

4. **Create socket associations:** To track the new association, two new entries are added into the socket_association map: one linking the original socket to the proxy socket, and the other linking the proxy socket back to the original application socket. This mapping allows SK_MSG to correctly correlate messages between the original and proxy sockets.

5. **Notify userspace:** Before completing its operation, SOCKOPS must inform the userspace component about the new socket. This is done using the new_sk map, where the new entry contains the sock_id_t (composed by source and destination IPs and source and destination ports) corresponding to the new socket and the operation that was intercepted.

```
sockops(app_sk) {
    switch (operation) {
    case BPF_SOCK_OPS_PASSIVE_ESTABLISHED_CB: // server
    case BPF_SOCK_OPS_ACTIVE_ESTABLISHED_CB:  // client
        if(SOCKET_NOT_TARGET) return;

        // Add the socket to the map for SK_MSG
        add(intercepted_sockets, app_sk);

        // Get a free proxy sk
        proxy_sk = pop(free_sockets)

        // add the association
        add(socket_association, proxy_sk -> app_sk);
        add(socket_association, app_sk -> proxy_sk);

        // Notify the user space about the new socket
        ring_buff_add(operation, new_sk);
    default
        break; // ignore other operations
    }
}

```

Listing 4.5: `SOCKOPS` hook pseudo code

4.2.3 `SK_MSG`

The `SK_MSG` program is responsible for redirecting messages and operates on the sockets stored in the `intercepted_sockets` map. Upon receiving a message, `SK_MSG` follows a specific sequence of actions:

1. **Determine the message direction:** Messages can flow either from the application to the proxy or from the proxy to the application. To determine the direction, `SK_MSG` checks the source port. If the source port matches the server port (stored in the `server_port` map), the message comes from the proxy and should be redirected to the application. Otherwise, the message originates from the application and must be sent to the proxy socket.

2. **Retrieve the destination socket:** Once the direction is known, `SK_MSG` looks up the destination socket in the `socket_association` map. Using the current socket as a key, it identifies the corresponding socket to which the message should be sent.

3. **Redirect the message:** Finally, `SK_MSG`, uses the `bpf_msg_redirect_hash` helper to forward the message to the correct destination socket.

Note that all proxy sockets created at boot time are already inserted into the `intercepted_sockets` map. This is necessary to allow the `SK_MSG` hook to intercept messages sent by the proxy (i.e., messages directed to the end application).

```
sk_msg(source_sk) {
    if(app_sk.dest_port == server_port){
        // directed to proxy -> direct it to app
        app_sk = get(socket_association, source_sk);
        bpf_msg_redirect_hash(app_sk);
    } else {
        // directed to app -> direct it to proxy
        proxy_sk = get(socket_association, source_sk);
        bpf_msg_redirect_hashproxy_sk;
    }
}
```

Listing 4.6: `SK_MSG` hook pseudo code

4.2.4 `TCP_DESTROY`

The `TCP_DESTROY` program represents the final component of the eBPF system. It handles the last phase of a socket's lifecycle: the socket closure event. Its role is to notify the application to stop propagating messages and release any allocated resources. This hook is triggered whenever a socket is closed and operates as follows:

1. **Identify the corresponding proxy socket:** The function checks the `socket_association` map to locate the proxy socket previously associated with the socket being closed.

2. **Remove association entries:** Both entries linking the application socket to the proxy, and vice versa, are removed from the `socket_association` map.

3. **Return the proxy socket to the pool:** The proxy socket is returned to the `free_sockets` map, making it available for reuse.

4. **Notify userspace:** Userspace is informed of the socket closure by adding a new entry to the `new_sk` map, containing the `sock_id_t` of the closed socket. This allows userspace to perform any necessary cleanup or follow-up actions.

```
1  tcp_destroy(source_sk) {
2      // Delete the entries in the association table
3      associated_sk = get(socket_association, source_sk);
4      delete(socket_association, source_sk);
5      delete(socket_association, associated_sk);
6
7      // Mark the sk as free
8      push(free_socket, associated_sk);
9
10     // Notify the user space about the socket deleted
11     ring_buff_add(source_sk, SK_DELETED);
12 }
```

Listing 4.7: `TCP_DESTROY` hook pseudo code

4.3 Userspace

Once a message has been forwarded by the eBPF program into one of the available proxy sockets, the userspace component of the application reads the message from that socket and transfers it to the remote peer using RDMA. This userspace component is further organized into two main parts: `SocketManager` and `RdmaManager`.

The `SocketManager` component is responsible for handling all the proxy sockets used to forward messages from `SK_MSG` to user space. Its main role is to create these sockets and store them in the `free_sockets` map (queue). This design allows the eBPF program to pop sockets on demand and link them to the original socket from which the messages were generated. Since sockets are represented differently in kernel and user space, `SocketManager` also maintains a mapping between `sock_id` identifiers (used in kernel space) and file descriptors (used in user space).

The unit responsible for managing data transfer in RDMA is divided into two components: `RdmaContext` and `RdmaManager`. The `RdmaContext` represents a connection between two peers and is used for point-to-point communication. Consequently, in a network of N nodes, $\frac{N \times (N-1)}{2}$ contexts are required to enable full connectivity among all nodes. In contrast, the `RdmaManager` is a per-peer object that oversees all created contexts (Figure 4.2).

A connection between two peers is established only when necessary, namely at the time of socket creation. As explained in Section 4.2.2, `SOCKOPS` intercepts the creation of new sockets and performs several actions, one of which is notifying user space of the new socket via a ring buffer. Each message in this buffer includes the `sock_id` of the new socket along with the intercepted operation. Among these operations, the relevant ones are `BPF_SOCK_OPS_PASSIVE_ESTABLISHED_CB` and `BPF_SOCK_OPS_ACTIVE_ESTABLISHED_CB`, which are triggered on the server (passive) and client (active) sides, respectively.

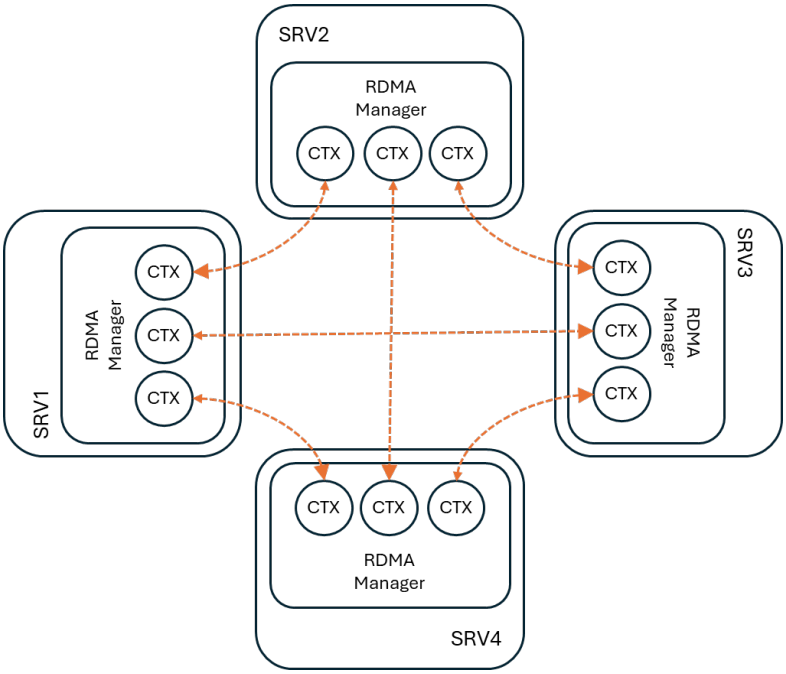

Figure 4.2: Layout of RDMA connections between peers

4.3.1 Background threads

As mentioned in the previous chapter, to achieve higher throughput and parallelism, the application runs multiple independent threads in parallel.

Writer thread

The *Writer Thread* (WT) is responsible for reading data from a socket and writing it into the memory region (MR). In practice, there can be multiple writer threads, up to the number of sockets that must be monitored, hence the number of proxy socket. The general behavior of a WT is to continuously check for new data to consume.

Given one or more file descriptors associated with sockets, the WT performs a `select()` call to wait for events. When new data is available, the operating system wakes the thread, allowing it to proceed with its work.

The first step consists of checking the RDMA buffer for free slots where new data can be written. The availability of slots is determined by the difference between two indices: `local_write_index` that specifies the position up to which data has already been written locally and `remote_read_index`: indicates the position reached by the remote peer in consuming the data.

It is crucial to ensure that unconsumed data is not overwritten. For this reason, if no free space is available, the WT remains in a busy-waiting loop until space is released.

Once free slots are found, the thread moves to the second phase: invoking `recv()` on the target file descriptor to retrieve the incoming data. The system call directly writes the received data into the buffer, by using the pointer of a free slot as the destination. This ensures that the data is copied only once, thereby minimizing overhead.

The WT continues to perform `recv()` calls until either (i) the available buffer space is exhausted, or (ii) the sockets have no more data to transmit. In the case where buffer space runs out but the sockets still hold data, the thread returns to the waiting state until new space is freed.

For each message, the WT does not perform a single `recv`, but multiple ones. This approach helps optimize buffer usage by reducing the number of RDMA posts and increasing the size of each post. This is possible because the proxy sockets operate in non-blocking mode, so a `recv` call may return even when no data has been read.

For each message, the WT also writes some additional metadata:

- Sets the message size.
- Sets the current sequence number and increments the sequence counter to prepare for subsequent messages.
- Sets the `sock_id_t` associated with the socket that originated the message.
- Sets the number of slots that the message occupies.
- Sets the message flags.

This additional information is necessary to construct a header for the message, which is used on the receiving side to correctly parse the message during the reception phase.

Since multiple writer threads may coexist, access to shared resources in the memory region must be synchronized. For this purpose, a `mutex` is employed to guarantee mutual exclusion and prevent race conditions.

The work of the WT does not end with writing the message; there is still another operation to perform: signaling the data to the Flush thread. The system uses multiple queues, one for each employed QP. The WT pushes the index of the message that has been written into one of the queue available (selected in round robin), and later the FT pops from the same queue to perform the flush of the message.

The behavior of `WT` is illustrated through pseudocode in Listing 4.8.

```
1  WriterThread(fd) {
2      // wait for data to consume
3      select(fd);
4      // wait for available space
5      while(1) {
6          space = remote_read_index - local_write_index
7          if(space > 1)   break;
8      }
9      // write the data inside the buffer
10     while(space > 0) {
11         msg = buffer[local_write_index];
12         msg.size = recv(fd, msg, MAX_MSG_SZ);
13         msgs_idx_to_flush_queue.push(local_write_index);
14         local_write_index++;
15         space--;
16     }
17 }
```

Listing 4.8: Writer thread pseudo code

Flush thread

The *Flush Thread* (FT) works in coordination with the Writer Thread (WT) to complete data transmission. As mentioned earlier, dedicated queues store the indices of messages ready to be flushed. The FT continuously monitors these queues by popping elements.

The FT does not operate directly on the messages. RDMA communication relies on *Work Requests* (WRs), which specify the operations to be performed (in this case, indicating where to write the incoming data). The FT is responsible for preparing the WRs to be posted.

Each FT maintains a pre-allocated array of WRs to minimize runtime allocations. For each popped message, the FT creates the corresponding WR and stores it in the local array, ready to be flushed.

Once the number of WRs in the local queue reaches a threshold defined in the configuration, the FT posts the corresponding WRs to the Network Interface Card (NIC).

If the number of WRs does not reach the threshold within a specified time window, the FT still flushes the WRs that have been collected, ensuring that no data remains unprocessed indefinitely.

Once an FT has prepared the WRs into the queue, the posting process does not begin immediately. Before the data can be sent to the Queue Pair (QP), the FT retrieves the corresponding WRs and links them together. This aggregation is crucial because the `ibv_post_send()` call is relatively expensive. By linking multiple

WRs and posting them in a single call, the NIC can process them sequentially and efficiently.

Typically, multiple FTs are employed, with each thread associated with a specific Completion Queue (CQ). This association avoids race conditions and increases parallelism, allowing multiple FTs to operate concurrently.

The behavior of `FT` is illustrated through pseudocode in Listing 4.9.

```
1  FlushThread(){
2      j=0;
3      while(true){
4          idx = busy_idx.pop();
5          j++;
6          wr_to_post.push(wrs[idx])
7          if(j > MAX_WR_PER_CQ ||
8              time.now() - last_flush > TIME_FLUSH)
9          break;
10     }
11     post_wr(wr_to_post, QP[i])
12 }
```

Listing 4.9: Flush thread pseudo code

Since RDMA post operations are asynchronous, it is necessary to verify that they have been completed correctly. To this end, RDMA introduces the concept of a *Completion Queue* (CQ) (Section 2.3.4). Each Queue Pair (QP) is associated with a CQ, which can be queried to determine whether a given Work Request (WR) has been completed.

This check is performed through the function `ibv_poll_cq()`, which returns the number of completed events. As `ibv_poll_cq()` is a single call, it must be executed within a loop until the required number of completions has been obtained.

Because polling introduces some overhead, the best practice is not to poll the CQ for every single posted WR. Instead, completions are checked after many batches of WRs have been posted. The usual approach is to link multiple WRs together and mark only the last one with the flag `IBV_SEND_SIGNALED`. This flag ensures that only the final WR in the batch generates a completion event, which can later be retrieved from the CQ.

The Flush Thread (FT) maintains a counter of posted WRs. Since each batch contributes exactly one signaled WR, the FT can decide when to poll the CQ by comparing this counter against a predefined threshold. Once the threshold is reached, the CQ is polled to retrieve the completion event corresponding to the earlier WRs.

This mechanism serves a dual purpose: it reduces overhead by limiting the number of CQ polls, and it prevents the QP from becoming overloaded. If completion events are not retrieved in time, it is common for post operations to fail due to the

lack of available space for enqueuing new jobs.

The possibility of having multiple independent Flush Threads (FTs) is enabled by two key design elements. First, each message includes a *sequence number*, which allows the receiver to determine whether a given slot in the memory buffer contains a message ready to be consumed. Second, when creating a Work Request (WR), the WT explicitly specifies the target location in the remote memory where the data should be stored. These two mechanisms together ensure that multiple FTs can operate concurrently without conflicts. Each FT can post WRs to the NIC independently, knowing that the sequence numbers will maintain message ordering and that the remote memory addresses prevent data from being overwritten or misrouted.

Reader thread

The *Reader Thread* is responsible for parsing messages on the receiving side. As previously explained, the Writer Thread (WT) writes data into the local buffer, and the Flush Thread (FT) transfers it into the remote buffer. The Reader Thread then reads data from the remote buffer and forwards it to the destination socket.

The Reader Thread operates in a loop that iterates over the ring buffer. For each index, it checks the `seq_number_head` and `seq_number_tail` against its local sequence counters. If the sequence numbers match, it indicates that the message has been fully transferred and is ready to be parsed.

During the forwarding phase, an additional challenge arises: the message contains only the `sock_id` of the sender, which is not directly useful for sending data to the correct socket. To address this, the Reader Thread performs two important steps:

1. Swap the `sock_id` to indicate the correct direction.

2. Perform a lookup for the file descriptor of the proxy socket associated with the sender's `sock_id`.

Since this lookup is relatively expensive, the Reader Thread maintains a hash map to cache previously resolved associations, improving efficiency for subsequent messages.

Once the file descriptor is determined, the sending phase begins. To minimize system calls and context switches, the Reader Thread does not immediately call `send()`. Instead, it uses the `sendmmsg()` system call, which allows sending multiple buffers from different memory locations in a single operation. The rationale is as follows: messages often arrive in bursts (previously batched by the FT). The Reader Thread accumulates pointers to messages in an `iovec` structure. While waiting for the next burst, the thread can send the accumulated data to the destination socket improving throughput.

Once the data have been successfully sent to the destination socket, the Reader Thread (RT) updates the `local_read_index` of the ring buffer. This update is a local operation, but it must also be propagated to the remote side to ensure that the sender knows which slots have been consumed.

The responsibility of communicating these updates to the remote peer is delegated to a separate thread. This design separates the concerns of reading and forwarding messages from the task of synchronizing buffer indices, reducing the overhead of the RT.

```
ReaderThread() {
    for(int i=0; i<MSG_PER_BUFFER; ++i) {
        *msg = &buffer[i];
        // Wait for the message to come for entire
        while(msg.header != seq_n || msg.tail != seq_n) {}
        seq_n++;
        // retrieve the associated fd from the cached map
        int fd = lookup(fds, msg.origin_sk_id);
        // send data to the socket
        send(fd, msg.data);
    }
}
```

Listing 4.10: Reader thread pseudo code

Index thread

The *Index Thread* (IT) is responsible for propagating updates of the read index to the remote peer. As explained earlier, the Reader Thread (RT) advances the `local_read_index` whenever data has been consumed. The IT detects these changes and reflects them by updating the `remote_read_index`. It then issues an RDMA operation to synchronize the new index value with the other peer. To prevent interference with the Flush Threads (FTs), the IT is assigned a dedicated Queue Pair (QP). This separation ensures that index synchronization traffic remains isolated from data transmission traffic, thereby avoiding contention and preserving the overall performance of the system.

```
IndexThread() {
    idx = local_read_index
    while(true) {
        if(idx == local_read_index) continue;
        remote_read_index = local_read_indesx;
        idx = local_read_index;
        RDMA_post(&remote_read_index, QP_idx);
    }
}
```

Listing 4.11: Index thread pseudo code

Server thread

When an `RdmaManager` instance is initialized, it launches a dedicated thread that acts as a server to accept incoming RDMA connections. Its role is to wait for client requests and establish the corresponding `RdmaContext` performing the RDMA handshake. When a client attempts to connect, the context is instructed to process the request. At this stage, the connection is established, and the context becomes bound to the client. To record this relationship, the server adds the context to its collection of active contexts.

In addition, once the first client is connected, the ST also starts certain background threads. In particular, it launches the *Notification Thread*, which is created only once and shared across contexts.

After completing the handling of a connection, the ST resumes its main loop, preparing new contexts and waiting for subsequent client connections.

RDMA handshake The RDMA setup is complex and is divided into two phases: pre-handshake and post-handshake. The pre-handshake phase, which can be executed locally by each peer, involves the following steps:

1. **Open the RDMA device and create a protection domain:** The device represents the network interface capable of performing RDMA operations, while the protection domain defines the scope of memory and queue pair (QP) resources that can access each other. This ensures isolation and protection of communication resources.

2. **Establish completion queues (CQs) and a completion channel:** RDMA operations are asynchronous. Multiple send CQs are typically used for load balancing, while a dedicated receive CQ tracks incoming messages. The completion channel allows event-driven notifications of work completions, improving efficiency compared to constant polling.

3. **Optionally create a shared receive queue (SRQ):** An SRQ allows multiple QPs to share the same pool of receive buffers, reducing memory duplication and improving scalability. If unsupported, individual receive queues can be used for each QP.

4. **Allocate and register memory:** All memory involved in RDMA communication must be registered with the device, providing a memory region

key (rkey). This allows remote peers to read or write directly without CPU intervention.

5. **Create and initialize queue pairs (QPs):** Each QP consists of a send queue and a receive queue, and multiple QPs are created to increase parallelism. QPs are initialized with associated CQs, access permissions, and optionally linked to the SRQ. They are then moved to the *INIT* state, preparing them for handshake transitions.

6. **Gather local connection information:** Each RDMA-capable port has identifiers such as the local identifier (LID) and global identifier (GID). These identifiers, along with QP numbers, memory addresses, and memory keys, constitute the local connection information necessary for the remote peer to establish a valid RDMA connection.

This pre-handshake setup ensures that the local system is fully prepared for RDMA communication, enabling low-latency, high-throughput transfers once the handshake with the remote peer is complete.

After completing the first phase, we must proceed to the second one. Unlike the initial step, this second phase cannot be completed locally; it requires the exchange of information between peers. Since the RDMA connection is not yet established, such information must be communicated using an out-of-band (OOB) mechanism.

For the application under development, TCP has been chosen as the transport channel for this exchange. In particular, the handshake requires the following data to be communicated:

- `qp_num`: Each Queue Pair (QP) is assigned a unique identifier. During the handshake, peers exchange QP numbers so that each side knows which QP to target when posting work requests.

- `rq_psn`: The Receive Queue Packet Sequence Number (PSN). Every QP keeps track of PSNs to ensure reliable connections (RC). Exchanging initial PSNs guarantees that both peers agree on the starting point of the sequence, thus avoiding the misinterpretation of old or retransmitted packets as valid.

- `rkey`: Extracted from the registered memory region, this key grants access to remote memory for both read and write operations.

- `addr`: The starting address of the remote memory region.

- `gid`: The Global Identifier (128-bit) used to identify the correct port or interface for RDMA communication. Since the implementation relies on RoCEv2, the GID is mapped to an IP address, enabling RDMA to operate over routed networks.

```
1 struct conn_info {
2     uint32_t qp_num[Config::QP_N];
3     uint32_t rq_psn[Config::QP_N];
4     uint32_t rkey;
5     uint64_t addr;
6     union ibv_gid gid;
7 } __attribute__((packed));
```

Listing 4.12: RDMA handshake data

Once the peers have exchanged these data, the process advances to the final phase of the RDMA handshake: moving the QPs into the *Ready to Receive* (RTR) state. RDMA QPs must transition through a series of well-defined states before becoming fully operational. The key states are as follows:

- **RESET**: The initial state of a newly created QP, with empty queues.
- **INIT**: Basic information has been configured, and the QP is ready to accept postings to its receive queue.
- **RTR (Ready to Receive)**: Remote addressing information is set for connected QPs. In this state, the QP can now receive incoming packets.
- **RTS (Ready to Send)**: Timeout and retry parameters are configured, allowing the QP to send packets.

At the conclusion of this procedure, both peers have their QPs configured and are fully prepared to use RDMA.

Notification thread

The *Notification Thread* (NT) implements the event-driven portion of RDMA (Section 3.4.3) communication, its role is to process completion events from multiple RDMA contexts.

The NT operates in a loop, scanning all active contexts to identify the file descriptors associated with completion channels that are waiting for notifications (e.g., RDMA send/recv approach). Once the scan is complete, the NT blocks on a `select()` call over these descriptors. The call returns when one or more file descriptors become ready, indicating that new events are available.

For each file descriptor that becomes ready, the Notification Thread (NT) follows a series of steps to handle the event. First, it determines which `RdmaContext` the descriptor belongs to, ensuring that the event is properly associated with the right connection. It then retrieves the corresponding completion event from the queue and immediately acknowledges it, confirming that the event has been received and processed.

Next, the NT re-arms the completion queue so that it will continue to generate notifications for future events.

To ensure that RDMA remains able to receive subsequent notifications, the NT also posts a receive Work Request (WR) onto the receive queue. This queue may be implemented either as a Shared Receive Queue (SRQ) or as a dedicated Receive Queue (RQ).

Finally, once the work completion has been retrieved, the NT processes the event by interpreting the incoming message or notification and dispatching it to the appropriate handler. In this way, the NT serves as the central dispatcher for asynchronous RDMA events across all contexts.

Since the RDMA part of the application is almost entirely based on `WRITE` operations (which are posted on the send queue of a QP), this thread is the only one that actually makes use of the receive queue by posting the work requests required to handle incoming sends.

Given the limited use of the receive queue and the fact that the application employs many QPs, a Shared Receive Queue (SRQ) (Section 2.3.4) is adopted to reduce memory consumption and improve efficiency. However, if for any reason (e.g., NIC incompatibility) an SRQ cannot be used, the application falls back to the classic approach: it reverts to using the standard receive queue, with only the one associated with QP index 0 serving as the SRQ.

Chapter 5

Results

5.1 Test and Debug Tools

During this work, alongside the main application, three additional applications were developed, all related to the main one. These applications proved very useful for performance testing and for breaking down the larger problem into smaller, more manageable parts, allowing for more detailed and tailored debugging.

In particular, three auxiliary applications were developed:

- **Socket app**: This application consists of two main components, a client and a server. The core functionality is that they create a TCP socket and exchange a specified amount of data in gigabytes to test throughput, latency and data integrity.

- **eBPF app**: Extracted from the main application, this app acts as a lightweight proxy that emulates the main application locally. The eBPF script is very similar to the main one but lacks RDMA support. It is primarily used to leverage the `SK_MSG` feature locally and is particularly useful for testing purposes.

- **RDMA app**: This application is used to fine-tune RDMA performance. It consists of a server and a client that establish an RDMA connection and exchange data. It is essential when developing and optimizing the core RDMA transport method.

In conclusion, the Socket app is intended solely for testing purposes, while the other two applications can be considered as modular parts of the main application.

5.1.1 Configuration

All application parameters can be manually tuned by modifying the `Config.hpp` file. This file contains all the constants used to configure the entire setup. The most important parameters are related to RDMA communication, such as the number of queue pairs (QPs), the maximum number of messages that can reside in a buffer, and the maximum number of work requests (WRs) that can be posted in a single batch.

These parameters provide fine-grained control over the application, allowing it to be adapted to specific scenarios—for example, to optimize either latency or throughput.

```
1  ...
2  inline static const int MAX_MSG_BUFFER = (256);
3  inline static const int MAX_PAYLOAD_SIZE = (128 * 1024)
4  inline static const int QP_N = 2 + 1;
5  inline static const int DEFAULT_QP_IDX = 0;
6  inline static const int N_OF_QUEUES = QP_N - 1;
7  inline static const int POLL_CQ_AFTER_WR = 64;
8  inline static const int MAX_WR_PER_POST_PER_QP = 32;
9  inline static const int FLUSH_INTERVAL_NS = 150;
10 inline static int IOVS_BATCH_SIZE = 32;
11 inline static int N_RETRY_WRITE_MSG = 30;
12 ...
```

5.1.2 Test devices

All testing and development has been performed using Intel E810 and Nvidia ConnectX-7 100 Gbps NICs, with the RoCE v2 RDMA transport.

5.2 SK_MSG

Let us now take into account some numerical results. These tests were carried out using the auxiliary applications described in Section 5.1, in particular the Socket app and the eBPF app.

The idea was to leverage the `SK_MSG` feature locally to transfer data. The complete setup is illustrated in Figure 5.1.

In this setup, the client and the server establish a TCP socket. The client begins sending messages over the socket, while the server is intentionally slowed down by introducing a timeout in each loop, preventing it from receiving at full speed.

The message path can be summarized as follows:

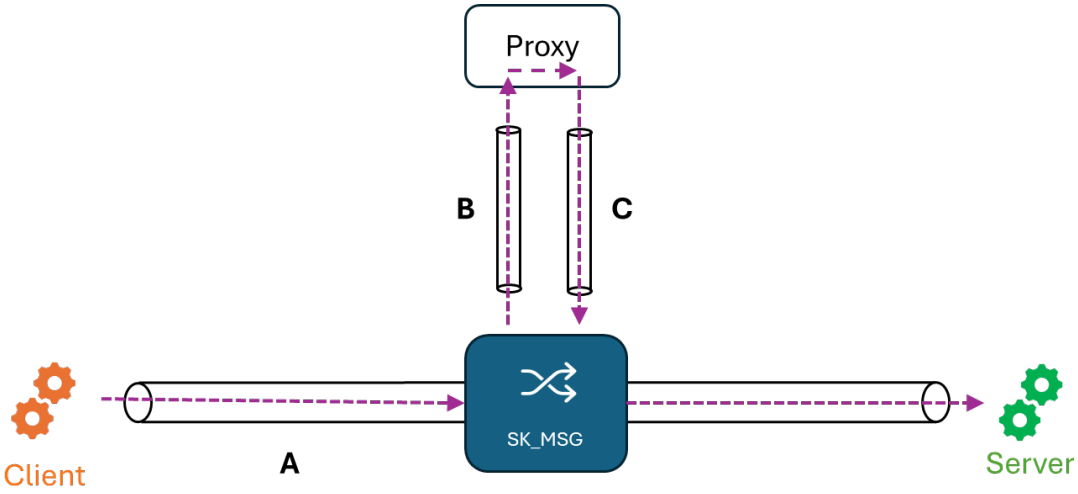

Figure 5.1: SK_MSG test schema

1. The client sends a message through socket A.
2. `SK_MSG` intercepts the message and redirects it to socket B, which connects to the proxy.
3. The proxy reads the message from socket B and forwards it to socket C, again passing through `SK_MSG`.
4. `SK_MSG` intercepts the message once more and redirects it back to socket A, finally delivering it to the server.

5.2.1 Considerations on backpressure behavior

The first phase of development focused on creating the eBPF script. During early testing, we observed a concerning issue related to the eBPF `SK_MSG` hook.

In a typical TCP socket communication, it is the responsibility of TCP to handle situations where the receiver is slow and cannot consume the data sent by the transmitter in a timely manner. This mechanism is known as *backpressure*. TCP backpressure is crucial for maintaining network stability and efficiency, as it prevents the transmitter from overwhelming the receiver by sending more data than it can handle.

Backpressure occurs when a receiving node (e.g., a server) cannot process incoming data at the rate it is being sent by the sending node (e.g., a client). This can happen due to various factors, such as limited processing capabilities, buffer overflows, or temporary network congestion.

From our tests, it emerged that *SK_MSG* fails to adhere to the expected backpressure behavior. In the case of a slow receiver, all data accumulates in the socket until it is consumed. `SK_MSG` continuously allocates kernel memory to store incoming data. This process continues until a substantial amount of memory (around 40 GB) has been allocated, at which point the receiver encounters the error: `ENOMEM`, which indicates a condition of "Not Enough Memory". This can potentially break applications that are not designed to handle this error code.

Memory usage test

As shown in Figure 5.2, memory usage rapidly peaks at approximately 50 GB within a period of roughly ten seconds. Afterwards, the graph exhibits a series of rises and falls: a decrease indicates that the sender is being blocked with the `ENOMEM` error, whereas an increase reflects ongoing transmission by the sender.

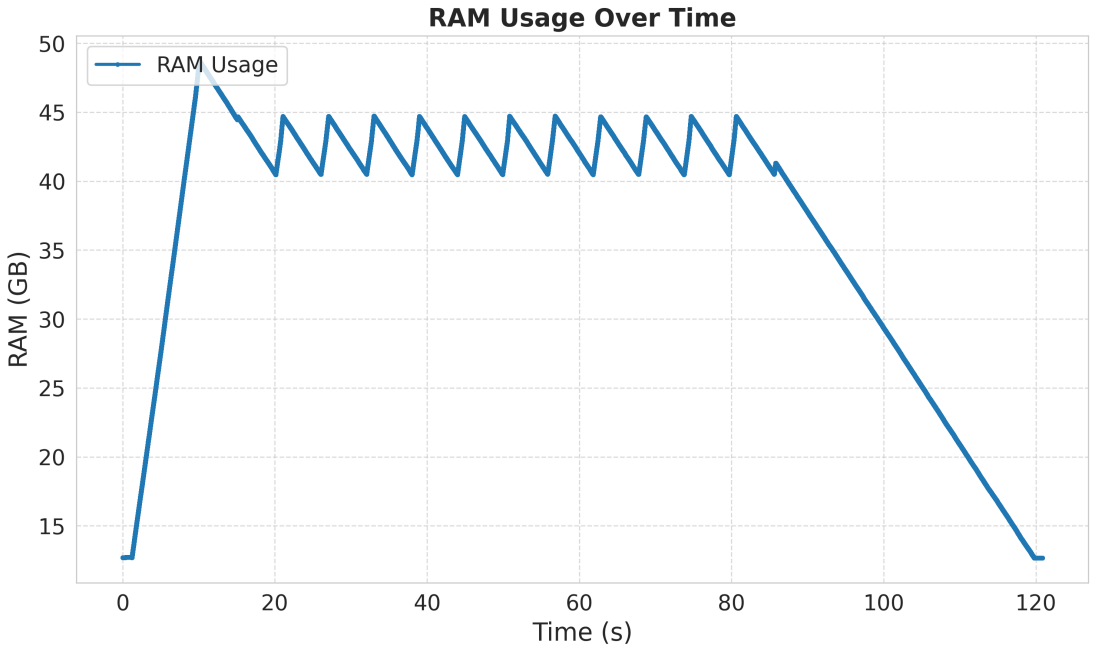

Figure 5.2: RAM usage over time while transferring 100 GB of data with `SK_MSG` steering and slow receiver.

Notably, once the peak is reached, the kernel halts further memory allocation and returns the `ENOMEM` error to the `send()` system call invoked by the sender. Subsequently, the consumer gradually processes the data, leading to a steady decline in memory usage until it returns to the baseline of around 5 GB.

This graph clearly illustrates the lack of backpressure in `SK_MSG`: the system is

not limited by the receiver's processing capacity, but only by the kernel, which stops the process when memory usage becomes excessive. We believe that this behavior may be causing inefficiencies in the system that limit the maximum performance improvement achievable by our RDMA-accelerated transport; future work should thus focus on investigating alternative socket-level traffic steering techniques for the project.

5.2.2 Performance

Let us now analyze the `SK_MSG` hook in more detail and the overhead it introduces. According to the main design (Figure 3.1), data needs to traverse the kernel twice, and consequently pass through the `SK_MSG` hook twice as well. These two steps occur respectively at the first peer and at the second one.

As shown in Figure 5.1, we can leverage this design to measure the average performance of the system.

Figure 5.3 presents the results of an `iperf3` test executed locally, with both the client and server running on the same machine. The figure displays two graphs: one showing the throughput of the baseline local test, and the other showing the same test performed with the `SK_MSG` hook enabled.

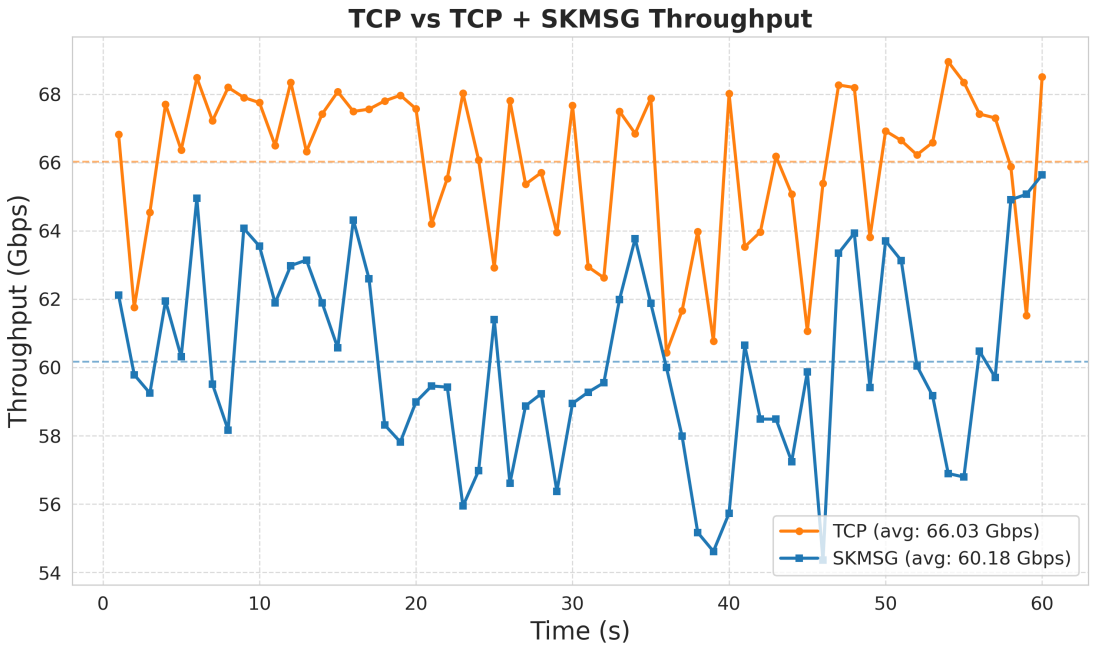

Figure 5.3: Local iperf3 test with and without `SK_MSG` using a single stream

From this comparison, we can clearly observe a performance penalty introduced

by `SK_MSG`, amounting to approximately 5 Gbit/s, or around 10% slower throughput. This penalty is not negligible, but it is important to consider that data incurs an additional copy when employing the `SK_MSG` data path:

1. The proxy reads the data from the socket (received via `SK_MSG`) using the `recv()` system call.

2. The proxy writes the data to another socket (directed through `SK_MSG`) using the `send()` system call.

The overhead of `SK_MSG` becomes even more evident in Figures 5.4 and 5.5, where `iperf3` is executed with three parallel streams (`-P 3`).

In the baseline case without `SK_MSG`, all three streams align at roughly 60 Gbps. However, when `skmsg` is enabled, the average throughput drops to around 45 Gbps. This represents a 15 Gbit/s decrease, or approximately 30% of the total throughput.

While the behavior in the single-stream case can be justified by the additional data copies, the multi-stream result is more surprising.

Since the proxy uses a multi-threaded approach, creating a new thread for each newly established stream, the throughput per stream should theoretically remain similar to that observed in the single-stream scenario. However, the aggregate performance drops significantly when multiple streams are active.

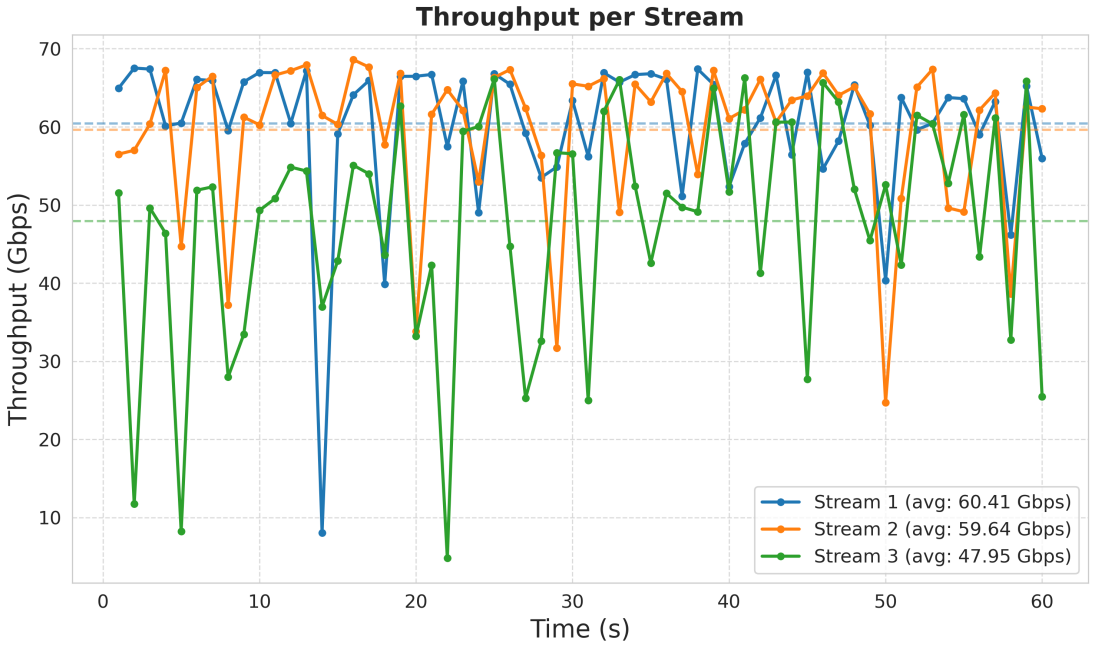

Figure 5.4: Local iperf3 test with three streams

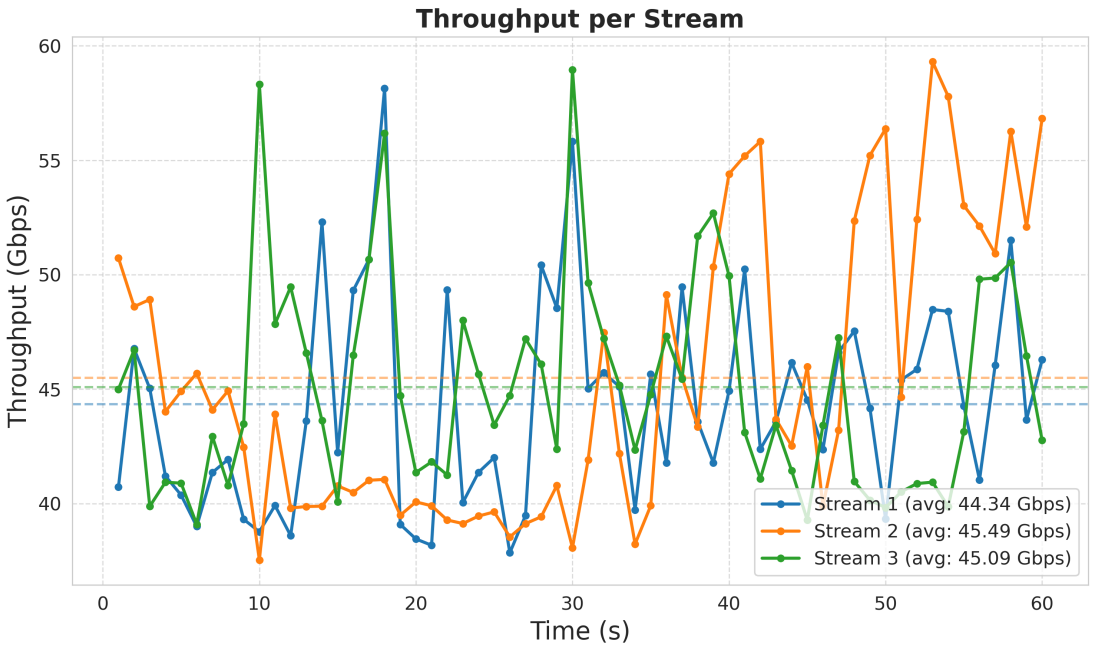

Figure 5.5: Local iperf3 test with three streams using `SK_MSG` hook

5.3 Application Performance

We now turn our attention to the overall system performance. The following analysis is conducted by considering the complete application architecture, illustrated in Figure 5.6.

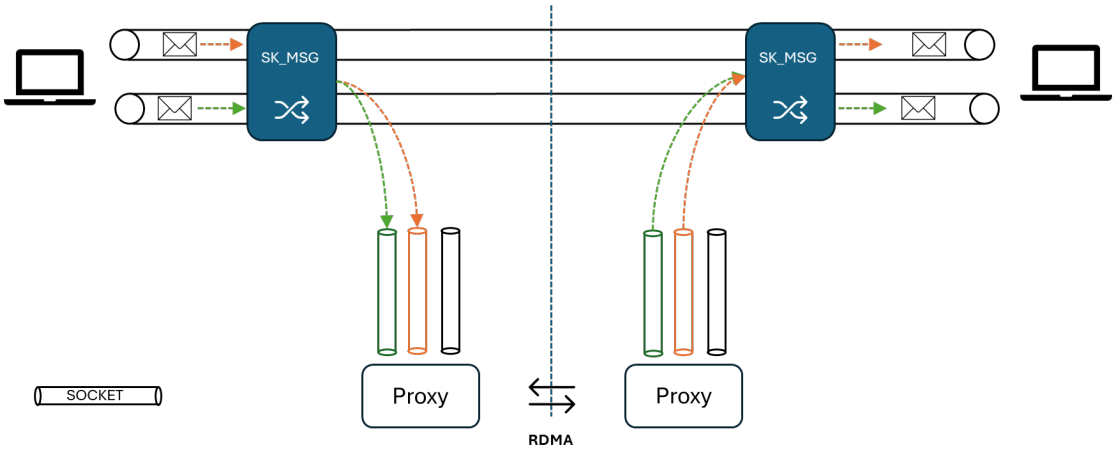

Figure 5.6: Application structure

5.3.1 Latency

Since latency is one of RDMA's major strengths, we expect it to outperform TCP in this regard. The test was conducted using the socket application described in Section 5.1. The first run was performed on a raw connection between two servers – i.e., a standard TCP socket connection – while the second was done using our application as transport method.

The results reported in Figure 5.7 show an average TCP latency of 54.64 µs compared to an average RDMA latency of **48.81 µs**. This represents an improvement of about **6 µs**, which corresponds to roughly **12%**.

It is worth noting that the graph begins displaying values only after 400 packets. This choice was made because the first 300 packets were required to trigger the Linux NAPI mechanism, effectively switching the kernel from an event-driven mode to a polling-based approach, which leads to lower latency. Before this transition, the average TCP latency was around 200 µs. In contrast, our application is able to maintain consistently low latency across all circumstances.

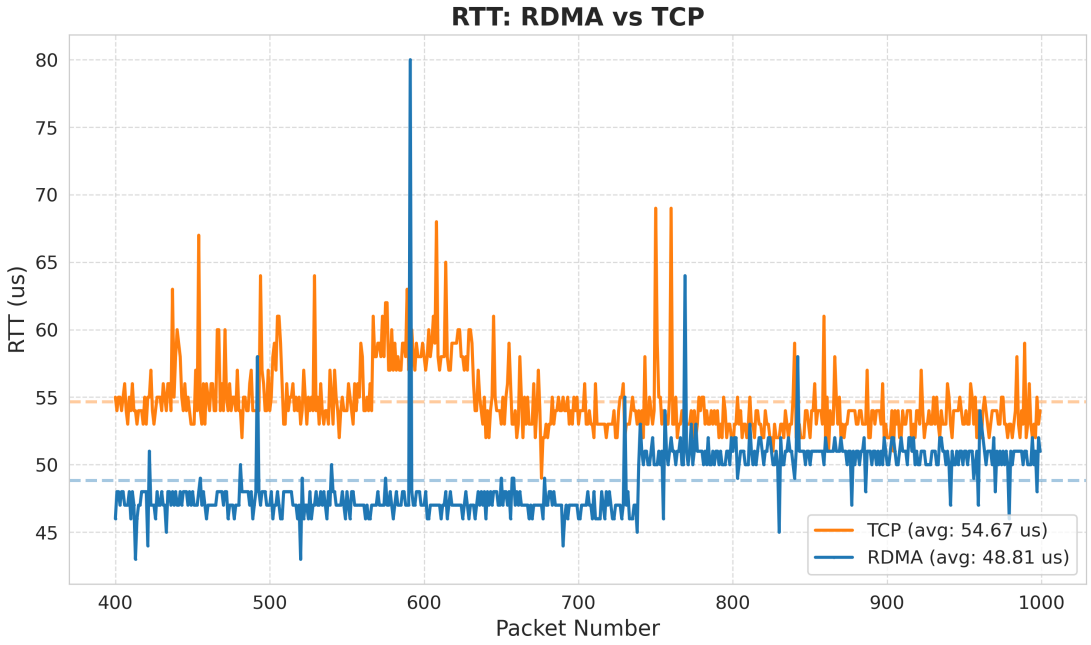

Figure 5.7: TCP vs RDMA latency.

It is also important to emphasize that this experiment was conducted after carefully tuning the application for optimal latency. The following optimizations were applied:

- **Flush Thread (FT) code** (Listing 4.3.1): The FT continuously polls the

queue for work requests (WRs). Once a predefined threshold is reached, it flushes the WRs. To avoid application stalls, a timeout mechanism is employed in addition to queue polling. The timeout value is a critical parameter: a shorter timeout improves latency but can reduce throughput. In this test, the timeout was set to 20 ns to prioritize latency.

- **Write Thread (WT)** (Listing 4.8): The WT was optimized to minimize unused portions of message payloads by performing multiple read operation. For latency optimization, the WT performs only a single read iteration from the socket, rather than multiple retries, in order to reduce overhead.

- **Maximum message payload size**: To further reduce latency, the maximum payload size was decreased from 128 kB to 256 B to reduce the overhead.

On the other hand, to maximize throughput, higher values of the time threshold and maximum payload size are preferable, as they allow the thread to start with more work, thereby exploiting batching and optimizing resource utilization.

5.3.2 Resource utilization

Let us now compare resource utilization between our new approach and the classical TCP implementation. The analysis focuses only on CPU usage, since the application does not impose significant memory demands.

Two tests were conducted while monitoring CPU consumption: (i) a standard `iperf3` benchmark between nodes, and (ii) the same benchmark executed with our application running in the middle.

This comparison is not straightforward, as our application relies heavily on parallelism and multithreading to achieve high performance, whereas the standard TCP path is simpler. For this reason, we chose overall CPU utilization as the unit of measurement. Moreover, these tests have been conducted on the transmitter side, but the situation is largely symmetrical on the receiving side.

The figures present two graphs: one showing CPU usage in kernel mode and the other showing CPU usage in user mode. The Y-axis represents the percentage of CPU utilization, where 100% corresponds to a single fully utilized core.

Figure 5.8 illustrates the CPU usage during a simple `iperf3` test. It can be observed that CPU consumption in user space is negligible, while kernel space accounts for nearly a full core, with an approximate **80%** utilization.

On the other hand, Figure 5.9 illustrates the CPU utilization of our application. In this case, CPU consumption is significantly higher than in the previous experiment, reaching a peak of **500%** of the total available capacity, corresponding to five fully utilized cores.

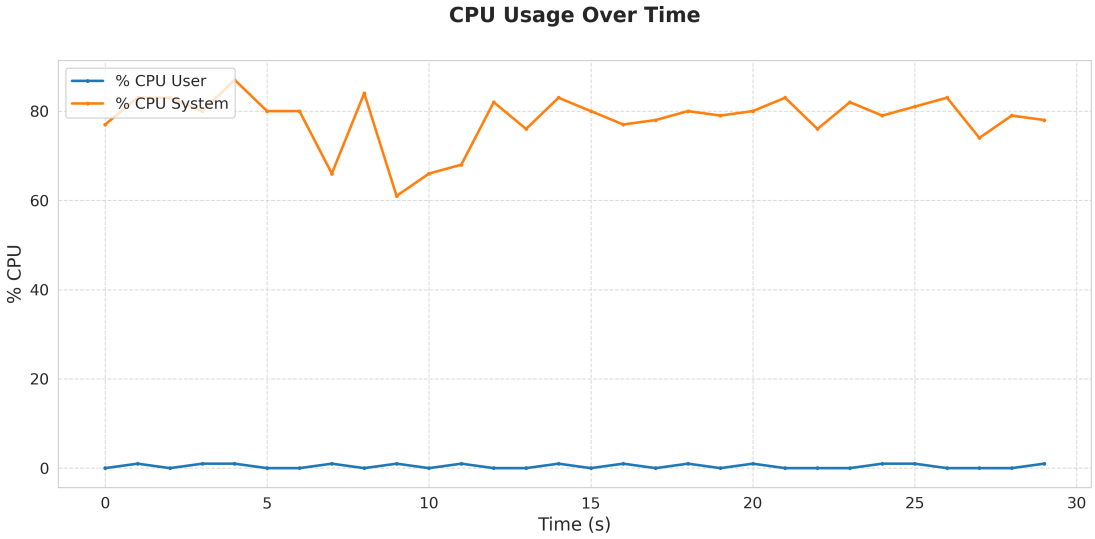

Figure 5.8: CPU utilization over time with iperf3 test between two nodes.

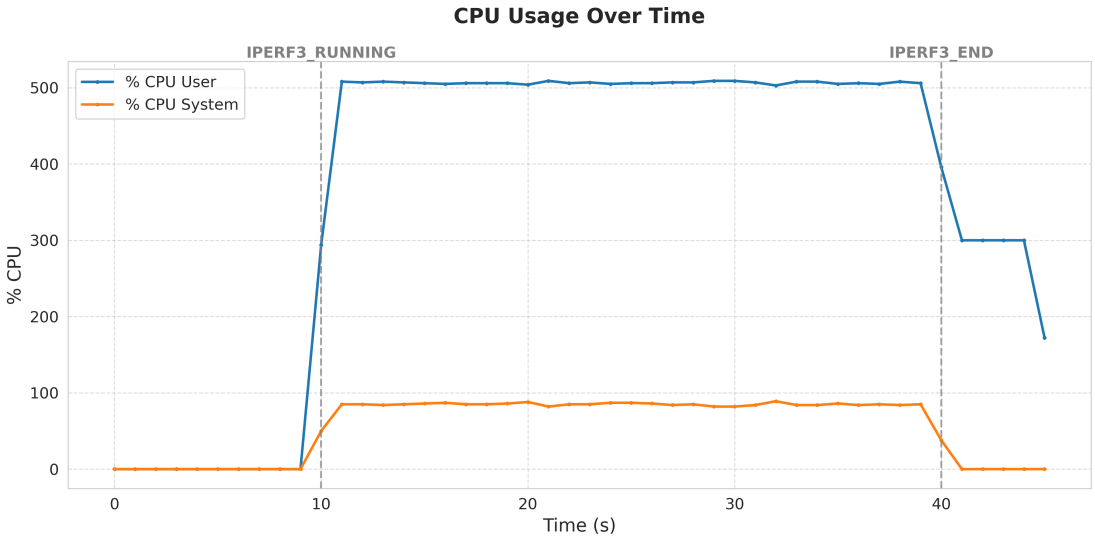

Figure 5.9: CPU utilization over time while running our application.

By taking a closer look at the application graph, we can identify three main phases:

1. **Application startup (no load):** No contexts are created, so the application runs with minimal resource usage.

2. **`iperf3` execution:** A context is created, and the application becomes active. During this phase, five threads operate in polling mode: `ReadT`, `WriteT`, two `FlushT` threads, and `UpdateIdxT`.

3. **After `iperf3` completion:** The writer and reader threads enter a sleep state, waiting for new data. However, the flush and index threads continue polling, which results in noticeable CPU usage.

Clearly, this behavior is not optimal, and further optimizations are needed to reduce overhead.

Additionally, we observe that kernel CPU usage remains around 100%, corresponding to a fully utilized core, which is consistent with the TCP baseline from the `iperf3` test. In contrast, our application demonstrates effective exploitation of the RDMA paradigm while remaining entirely in user space.

In conclusion, in terms of resource utilization, the classical TCP approach is currently preferable.

5.3.3 Throughput

Let us now analyze the final pillar of this work: throughput. First, it is important to note that the setup used for testing can achieve a maximum baseline RDMA throughput of approximately **88 Gbit/s**.

Figure 5.10 shows the results of an `iperf3` test in which the data is proxied through our application. The results indicate that we are able to achieve an average throughput of approximately **73.43 Gbit/s**, with occasional peaks reaching 77 Gbit/s, which is a very satisfactory outcome.

In the same figure, we also report the results of an `iperf3` test performed using TCP, where the average throughput is about **47.86 Gbit/s**. Thus, our approach introduces an improvement of approximately 55% in single-stream throughput.

However, a limitation arises related to the `SK_MSG` hook, particularly when multiple streams are involved. As discussed in Section 5.2.1, `SK_MSG` does not implement any form of backpressure. Consequently, when multiple streams push data simultaneously at their maximum rate, `SK_MSG` redirects the messages to the proxy. In our application structure, each stream is associated with a dedicated thread, and messages are written into a shared buffer. To prevent synchronization issues, the buffer is protected by a mutex.

The problem emerges because the maximum throughput achievable with a single stream is, as observed, about 74 Gbit/s. One would expect a higher aggregate throughput with multiple streams. However, the buffer quickly saturates, while the remaining messages remain in the socket memory. Therefore, all data must be fully transferred before the throughput can be accurately measured, which limits the observed performance in multi-stream scenarios.

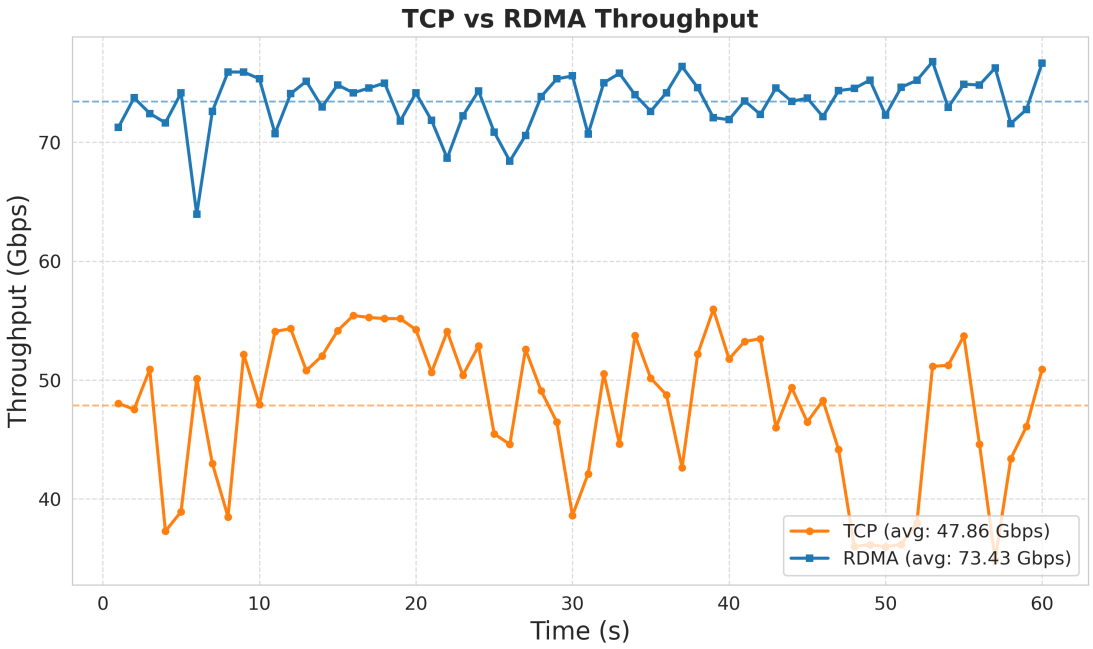

Figure 5.10: RDMA vs TCP iperf3 single-stream throughput comparison

5.3.4 Redis benchmark

To conclude, let us analyze a scenario closer to real-life conditions.

The last test was performed using the Redis benchmarking tool.

Redis [4] is an open-source, in-memory data structure store that can function as a database, cache, and message broker. It supports a wide range of data types, including strings, hashes, lists, sets, sorted sets, bitmaps, hyperloglogs, and geospatial indexes. Renowned for its high performance, Redis is widely used in scenarios that require rapid data retrieval, such as caching, session management, real-time analytics, and message queuing.

To evaluate latency between two servers in this a specific scenario, we use Redis's built-in benchmarking tool, `redis-benchmark`, which simulates multiple clients executing various commands to measure throughput and response times.

We present three graphs, each based on 1,000,000 requests of 2,048 B in size. The first graph corresponds to 10 parallel streams, the second to 30 streams, and the last to 40 streams. These graphs show the maximum operations per second achieved in each test with SET, GET, HSET, HGET, LPUSH and LPOP

Figure 5.11 represents the test performed with 10 parallel streams. In this case, we observe a significant performance increase, approximately double that of the baseline.

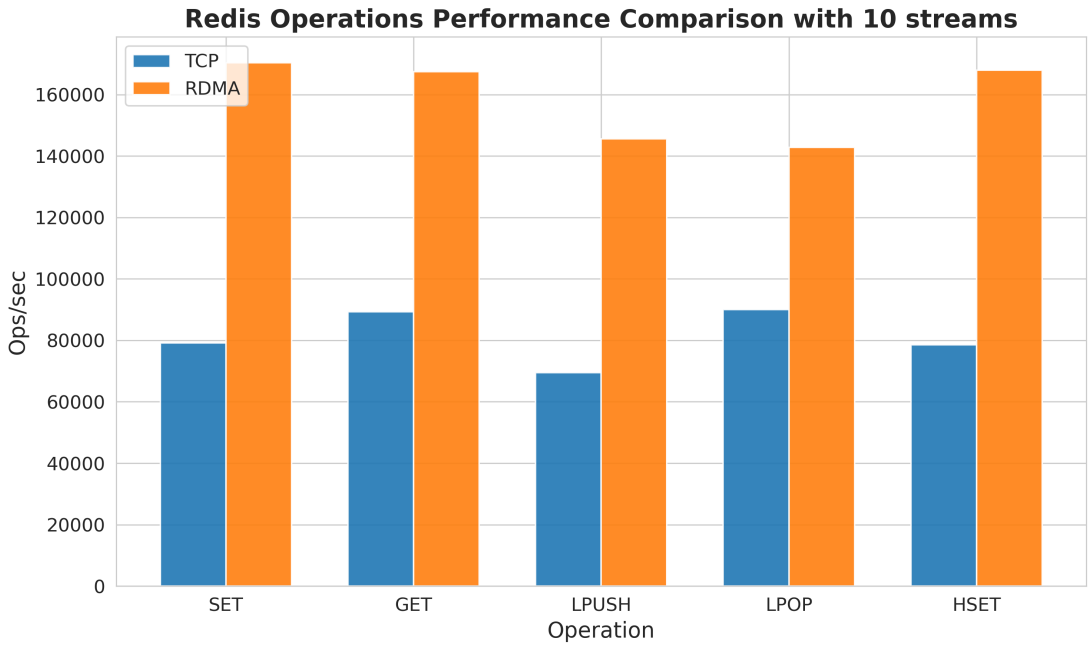

Figure 5.11: Redis benchmark with 10 parallel streams.

In the subsequent tests (Figures 5.12 and 5.13), we can observe the trade-off between TCP and RDMA. With 30 streams, RDMA still performs better; however, with 40 streams, TCP begins to gain performance. Further tests show that beyond 45 streams, the performance of our components drops significantly, whereas TCP demonstrates much better scalability.

This slowdown is partially due to the mutex protecting thread access to the buffer, which becomes a bottleneck under high concurrency.

In conclusion, these tests demonstrate that substantial performance gains are possible, but improvements in scalability are still required.

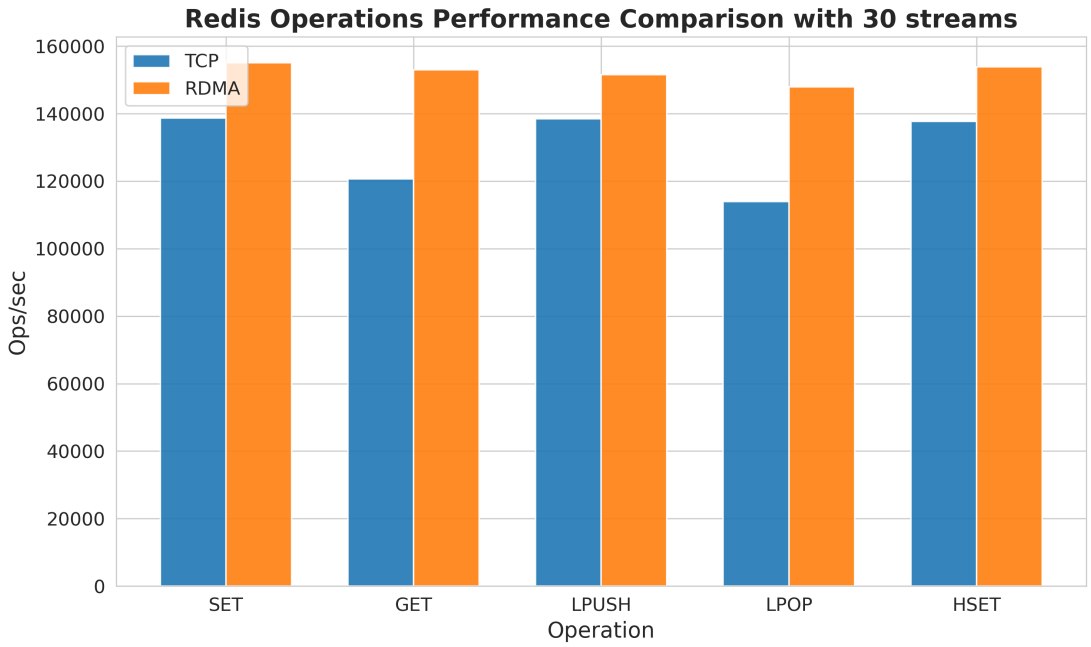

Figure 5.12: Redis benchmark with 30 parallel streams

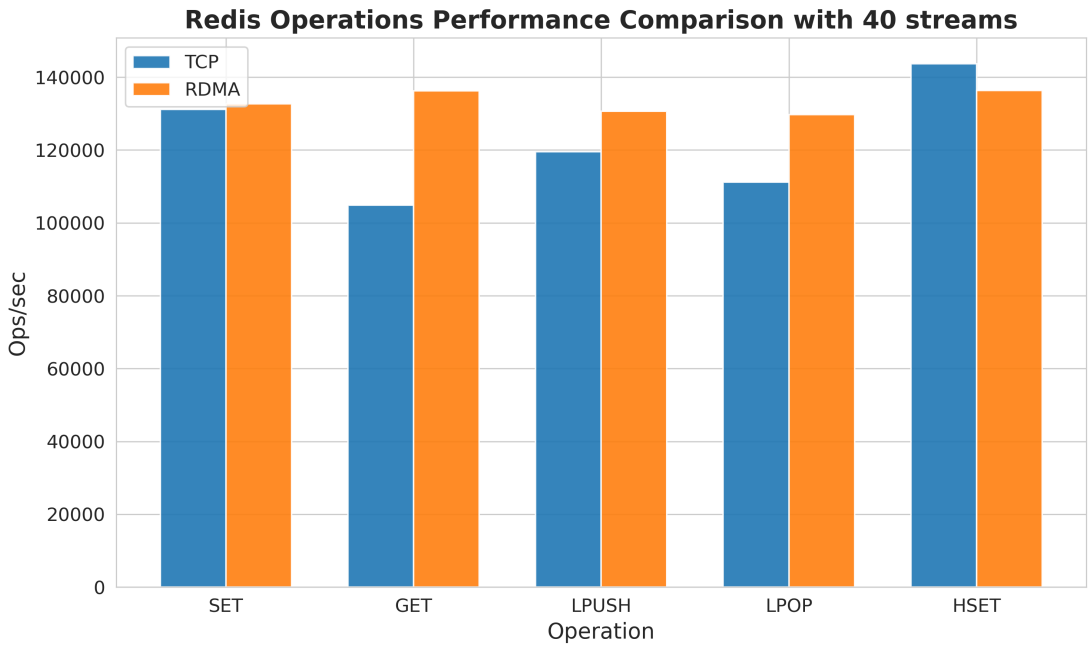

Figure 5.13: Redis benchmark with 40 parallel streams

Chapter 6

Conclusions

As the cloud-native paradigm continues to grow in popularity and rapidly establishes itself as the de-facto standard for datacenter software development, monolithic applications are increasingly decomposed into functionally distinct microservices. Consequently, east-west network traffic has become critical to ensuring system correctness. Traditionally, node-to-node communication relies on the TCP/IP stack, which provides reliable and ordered message delivery between applications. More recently, technologies such as Remote Direct Memory Access (RDMA) have emerged, offering comparable reliability while supporting higher throughput and lower latency.

RDMA generalizes the concept of Direct Memory Access (DMA) to the network scale, enabling one server to directly read from or write to the memory of another server. This technology provides extremely high throughput, low latency, and reduced resource consumption, as the workload is offloaded to the Network Interface Card (NIC), thereby freeing the CPU to handle other tasks. Despite these advantages, RDMA is still relatively new, and most applications are not designed to leverage its features. Furthermore, RDMA requires specialized hardware: servers must be equipped with RDMA-capable NICs, which can be costly and limit widespread adoption.

The objective of our work is to deliver a solution that allows any application to benefit from RDMA without requiring modifications. The guiding principle here is *transparency*: to ensure compatibility, applications must remain unaware of RDMA's presence. Our system intercepts the data produced by the sender, transfers it using RDMA, and delivers it to the receiver in a way that is indistinguishable from traditional communication.

Directly intercepting application data is not normally feasible, as such operations are restricted to kernel space. Modifying the kernel, however, would significantly increase complexity and reduce portability. To address this challenge, our solution leverages the eBPF framework, which allows the injection of custom code into the

kernel, enabling us to intercept traffic efficiently.

Finally, we chose sockets as the interception point since they have become the standard abstraction for remote communication in modern systems.

The application has been successfully developed and is capable of operating over RDMA, effectively replacing the traditional TCP stack. However, some limitations remain. Most of these issues are related to the eBPF component, particularly the `SK_MSG` hook used to intercept messages traveling through sockets.

As discussed during the evaluation (Section 5.2.1), the `SK_MSG` destination does not provide any form of backpressure. This implies that, in the case of a slow receiver, data continues to flow rapidly, leading to significant memory allocations inside the kernel. These allocations are not on the order of a few megabytes, but rather tens of gigabytes (up to 50 GB) of kernel memory. The only mechanism that slows down the transmitter is the kernel itself, which eventually returns an `ENOMEM` error when memory is exhausted.

In conclusion, although the `SK_MSG` hook functions as intended, it is not yet mature enough to support high-performance use cases of this kind. The framework requires further evolution and stabilization before it can be reliably deployed in production environments.

Despite these limitations, the results achieved are remarkable in terms of throughput and latency. Specifically, we reached a throughput of approximately 70 Gbit/s and a latency of 48.81 µs, representing a significant improvement compared to the baseline.

At the beginning of this work, the objectives were threefold: (i) reducing latency, (ii) increasing throughput, and (iii) reducing resource utilization. As development progressed, the resource utilization goal was partially abandoned in favor of maximizing latency reduction and throughput. Indeed, achieving the reported results required heavy reliance on parallelization (i.e., multithreading) and polling, which in turn resulted in high resource consumption. This aspect remains a key area for future improvement.

6.1 Future work & improvement opportunities

As demonstrated throughout this work, our approach is viable, but several aspects require further refinement and optimization:

- **Latency and throughput trade-off:** As discussed in the results chapter (Section 5.3.1), the tests were performed by carefully manually tuning the application to optimize for either latency or throughput. Such behavior is not acceptable in a production environment. A potential solution is to implement dynamic control of the flush timeout: decreasing it during low traffic to reduce

latency, and increasing it under heavy traffic to enable batching and improve throughput.

- `SK_MSG` **limitations:** We observed that the `SK_MSG` hook is not yet mature enough for this specific use case. Future work could either explore alternative methods for intercepting data or rely on forthcoming improvements to `SK_MSG`. Currently, it represents the most critical component of the entire design.
- **Optimization opportunities:** The application already minimizes data copying where possible. However, since much of the data handling occurs inside the kernel, we do not have full control over all internal operations. These steps could be further optimized to improve overall efficiency.
- **Resource usage:** As shown in the results section, the application is resource-intensive, primarily due to the polling performed by multiple threads. This behavior could be mitigated by introducing smarter resource management strategies, particularly during low-traffic periods.
- **Concurrency improvements:** As discussed in the results section, our application may encounter scalability issues due to the use of a single mutex to control access to the buffer used for message transfer. Solving this limitation would increasing the degree of parallelization, thereby improving performance in multi-process communication.

In conclusion, this work demonstrates that the transparent integration of RDMA into applications is a feasible and promising approach to enhance communication performance in cloud-native systems, while highlighting critical areas for further optimization and future development.

Bibliography

[1] Wikipedia contributors. *Express Data Path*. 2025. URL: https://en.wikipedia.org/wiki/Express_Data_Path (cit. on p. 6).

[2] Cilium Project. *Cilium: eBPF and Kubernetes Networking*. 2025. URL: https://cilium.io/ (cit. on p. 7).

[3] Erik Smith, Michal Kalderon, and Rohan Mehta. *Everything You Wanted to Know About RDMA But Were Too Proud to Ask*. 2025. URL: https://www.snia.org/sites/default/files/ESF/Everything-You-Wanted-to-Know-About-RDMA-But-Were-Too-Proud-to-Ask-Final%20v2.pdf (cit. on p. 15).

[4] Redis. *Redis — The Real-time Data Platform*. 2025. URL: https://redis.io/ (cit. on p. 65).

[5] Amr ElHusseiny. *Linux Networking Part 1: Kernel Net Stack*. 2022. URL: https://amrelhusseiny.github.io/blog/004_linux_0001_understanding_linux_networking/004_linux_0001_understanding_linux_networking_part_1/.

[6] Alok Prasad. *What is the relationship of DMA ring buffer and TX/RX ring for a network card?* 2019. URL: https://stackoverflow.com/questions/47450231/what-is-the-relationship-of-dma-ring-buffer-and-tx-rx-ring-for-a-network-card/59491902#59491902.

[7] NVIDIA Corporation. *RDMA Aware Networks Programming User Manual*. 2023. URL: https://docs.nvidia.com/rdma-aware-networks-programming-user-manual-1-7.pdf.

[8] GeeksforGeeks. *Direct Memory Access (DMA) Controller in Computer Architecture*. 2025. URL: https://www.geeksforgeeks.org/computer-organization-architecture/direct-memory-access-dma-controller-in-computer-architecture/.

[9] eBPF.io. *What is eBPF? An Introduction and Deep Dive into the eBPF Technology*. 2025. URL: https://ebpf.io/what-is-ebpf/.

[10] Arpit Kumar. *Understanding TCP Protocol and Backpressure*. 2023. URL: https://sumofbytes.com/blog/understanding-tcp-protocol-and-backpressure.